ЮЖНЫЙ
ФЕДЕРАЛЬНЫЙ
УНИВЕРСИТЕТ

А.Г. Чефранов, Р.В. Троценко

ПРОЕКТИРОВАНИЕ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

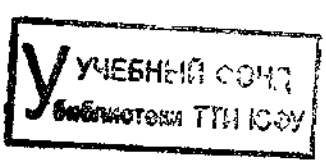
**ТЕХНОЛОГИЧЕСКИЙ ИНСТИТУТ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО ОБРАЗОВАТЕЛЬНОГО
УЧРЕЖДЕНИЯ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ» В Г. ТАГАНРОГЕ**

А.Г.Чефранов, Р.В.Троценко

ПРОЕКТИРОВАНИЕ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Учебное пособие

Рекомендовано УМО по образованию в области инновационных междисциплинарных образовательных программ в качестве учебного пособия по специальности "Математическое обеспечение и администрирование информационных систем" – 010503
(Решение Совета УМО на базе Санкт-Петербургского государственного университета от 11.11.2005 № 17/05)



Таганрог 2009

УДК 681.324 (076)

А.Г.Чефранов, Р.В.Троценко Проектирование систем реального времени. Учебное пособие. Изд. 2-е. - Таганрог: Изд-во ТРТУ, 2009. - 226 с.

Предназначено для студентов специальности 351500 "Математическое обеспечение и администрирование информационных систем", студентов других специальностей, изучающих курс "Системы реального времени", а также для профессионалов, интересующихся принципами построения и функционирования систем реального времени (СРВ). Рассмотрены: классификация СРВ; методы проектирования и описания СРВ, в том числе UML и ROOM; методы управления процессами, организации мультипрограммирования, синхронизации; проблема тупиков в централизованных и распределенных системах; описание средств программирования (Ада, Оккам, PVM); методы управления ресурсами СРВ: оперативной и внешней памятью, процессорами, приоритетное планирование задач, схемы наследования приоритетов; методы взаимодействия с внешними устройствами: прерывания, организация драйверов; описание SCADA-систем, автоматизирующих разработку и упрощающих использование СРВ; операционные системы реального времени; примеры СРВ.

Ил. 149. Библиогр.: 41 назв.

Печатается по решению редакционно-издательского совета Таганрогского государственного радиотехнического университета.

Рецензенты:

Терехов А.Н., заведующий кафедрой Системного программирования СПбГУ, профессор, доктор физ.-мат. наук.

Булычев Д.Ю., доцент кафедры Системного программирования СПбГУ, канд.физ.-мат. наук.

Оглавление

Глава 1. Введение

Системы реального времени (СРВ) предназначены для взаимодействия (в первую очередь, для управления) с реальными физическими объектами, и это взаимодействие должно протекать в реальном физическом времени, в тех временных масштабах, какие требуются объекту. Например, если говорить об управлении летательным аппаратом, движущимся со скоростью приблизительно 600 км/час на высоте 50 м, то временной масштаб для выработки и выполнения управляющих воздействий составляет приблизительно 0,01 с (за это время объект пролетит порядка 2 м). "

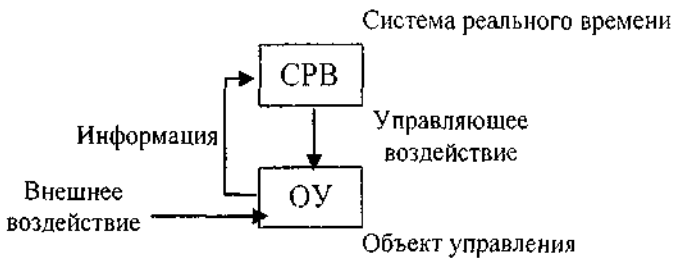


Рис.1.1. Укрупненная структура СРВ

Таким образом, ***системы реального времени*** - это вычислительные системы, предназначенные для работы в условиях жестких временных ограничений и имеющие развитые возможности взаимодействия с внешним миром для получения информации об объектах и передачи им выработанных управляющих воздействий. Такие системы должны обладать повышенной надежностью.

На рис. 1.1 приведена примерная структура СРВ. На ней имеется объект управления, от которого посредством датчиков поступает информация о его состоянии. Состояние может меняться как самим объектом управления, так и внешними воздействиями на объект. СРВ анализирует поступающую от объекта информацию и выдает управляющее воздействие, позволяющее скорректировать состояние объекта управления.

1.1. Классификация систем реального времени

Принято различать СРВ с мягкими (soft) требованиями (нарушение временных ограничений в некотором диапазоне не приводит к сбою системы), СРВ с жесткими (hard) требованиями (нарушение временных ограничений приводит к сбою системы) и СРВ с твердыми (firm) требованиями (нарушение временных ограни-

чений в определенном диапазоне не приводит к сбою системы с определенной вероятностью).

Примерами систем жесткого РВ являются бортовые системы управления, системы аварийной защиты, регистраторы аварийных событий. Примером системы мягкого РВ может быть сетевая среда. Если очередной переданный пакет не обработан, то это приведет к тайм-ауту на передающей стороне и повторной посылке (в зависимости от протокола, конечно). Данные при этом не теряются, но производительность сети снижается.

1.2. Пример системы реального времени

Приведенные ниже материалы основаны на [1].

Рассмотрим аналоговый PID-контроллер (PID - Proportional, Integral and Derivative) с одним входом и выходом.

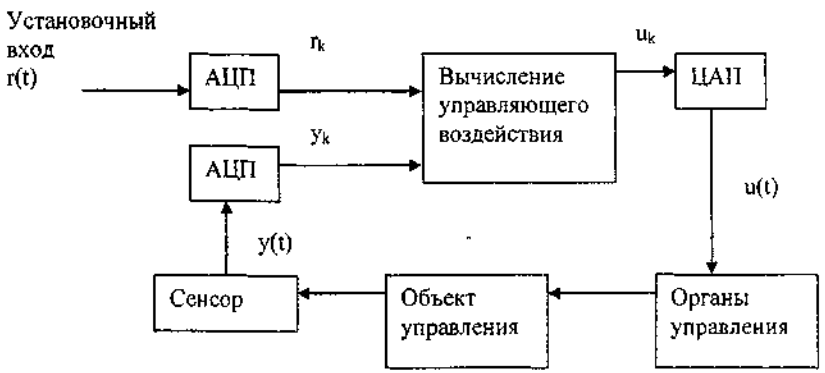


Рис. 1.2. Схема аналогового PID-контроллера

Схематическое изображение контроллера приведено на рис 1.2. Аналоговый сенсор, считывающий y(t), выдает измеренное состояние управляемого объекта в момент t. Пусть e(t) = r(t) - y(t) означает разность между желаемым состоянием r(t) и измеренным состоянием y(t) в момент t.

Выход контроллера u(t) состоит из 3-х членов, пропорциональных e(t), интеграла от e(t) и производной от e(t).

При цифровом управлении входы для блока "Вычисление управляющего воздействия" - дискретные значения y_k и r_k, k = 0,1,2,.., получаемые аналого-цифровым преобразователем (АЦП) дискретизацией y(t) и r(t) периодически каждые T единиц времени. Величина $e_k = r_k - y_k$ - это k-е значение ошибки e(t). Есть много путей дискретизации вычисления производной и интеграла e(t). Например, можно представить производную от e(t) для $(k-1)T \le t < kT$ как $(e_k - e_{k-1})/T$, и исполь-

зовать правило правых прямоугольников для численного интегрирования и представления непрерывного интеграла в дискретной форме:

$$\int_a^b f(x)dx = f(b)(b-a)$$

В результате имеем

$$u_k = \alpha e_k + \beta(e_k - e_{k-1})/T + \gamma \int_0^{kT} e(t)dt = \alpha e_k + \beta(e_k - e_{k-1})/T + \gamma T \sum_{i=1}^{k} e_i .$$

Рассматривая u_{k-1}, получаем

$$u_{k-1} = \alpha e_{k-1} + \beta(e_{k-1} - e_{k-2})/T + \gamma T \sum_{i=1}^{k-1} e_i .$$

Из двух последних уравнений имеем

$$u_k = u_{k-1} + ae_{k-2} + be_{k-1} + ce_k , \qquad (1.1)$$

где a,b,c - некоторые константы, их значения выбираются на этапе проектирования контроллера. Во время k-того периода дискретизации СРВ вычисляет выходное значение контроллера в соответствии с (1.1). Различные методы дискретизации ведут к различным выражениям для u_k, но все они достаточно просты для вычислений (10-20 машинных команд).

Из (1.1) можно увидеть, что в течение любого периода дискретизации (скажем, k-го), управляющий выход u_k зависит от текущего и предыдущего значений y_i, $i \le k$. Будущие значения y, для $i > k$ в свою очередь зависят от u_k. Такая система называется системой с обратными связями, или циклической системой с обратными связями. Ее можно реализовать в виде следующего бесконечного по времени цикла:

1. Установить таймер на прерывания с периодом T.
2. При каждом прерывании от таймера делать:
   2.1. Аналого-цифровое преобразование для получения y.
   2.2. Вычислить управляющий выход u.
   2.3. Цифро-аналоговое преобразование и выдача u.
3. Конец цикла.

Величина T времени между моментами измерения y(t) и r(t) - период дискретизации - имеет ключевое значение при проектировании контроллера. Поведение результирующего цифрового контроллера критически зависит от этого параметра. В идеале дискретный контроллер должен вести себя так же, как и аналоговый. Этого можно добиться, делая T малым. Но это ведет к более частым вычислениям управляющего воздействия и высоким требованиям к процессорному времени. Желательно, чтобы период дискретизации T был как можно больше, сохраняя при этом хорошее качество управления, т.е. нужен компромисс.

При выборе периода дискретизации надо учитывать два фактора. Первый - ожидаемая реактивность системы в целом (объекта управления и контроллера). Зачастую система управляется оператором (водитель, пилот). Оператор может выдать команду в любой момент, скажем в момент t. Соответствующее изменение входного воздействия читается и отрабатывается цифровым контроллером в следующий момент дискретизации. В худшем случае этот момент - t+T. Таким образом, дискретизация вносит задержку в системный отклик. Оператор будет ощущать систему как неповоротливую, если задержка будет больше 0,1 с. Поэтому период дискретизации должен быть меньше 0,1 с.

Второй фактор - динамическое поведение объекта управления. Рекомендуется использовать частоту дискретизации в два раза больше наивысшей частоты изменений в системе для того, чтобы иметь возможность восстановления таких сигналов (теорема Найквиста-Котельникова).

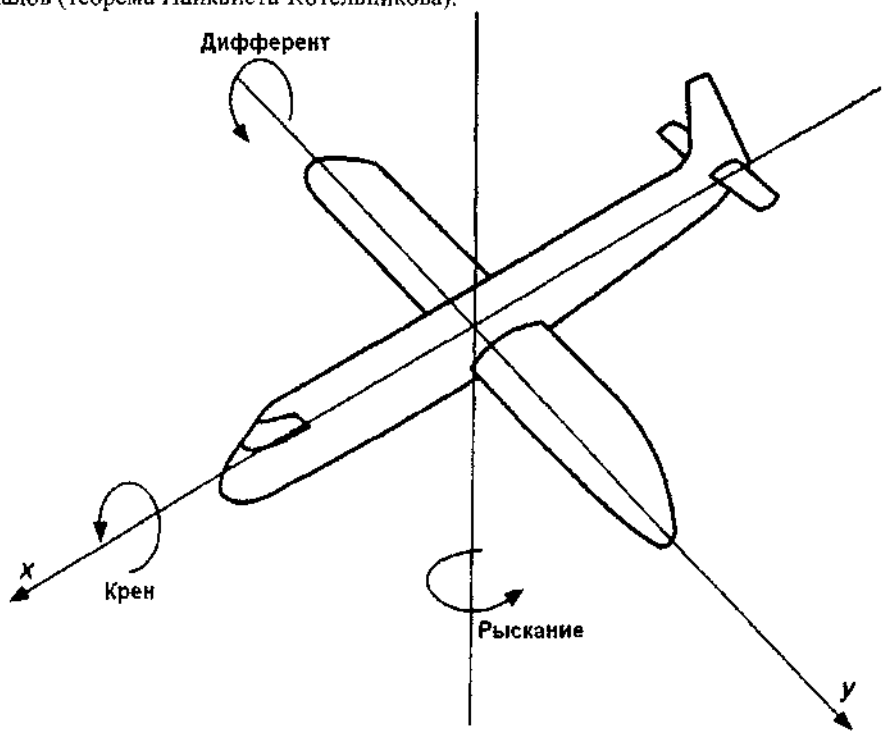


Рис. 1.3. Направление векторов скоростей вращения

Рассмотрим структуру программы управления скоростью полета самолета:
1. Делать следующее в каждом 1/180-секундном цикле (180 раз в секунду):

1.1. Проверка корректности данных с датчиков; при ошибке - реконфигурация системы.
1.2. Делать следующие задачи авионики, каждую один раз в 6 циклов (частота 30 Гц):
1.2.1. Ввод с клавиатуры и выбор режима.
1.2.2. Нормализация данных и преобразование координат.
1.2.3. Фиксация изменения входа.
1.3. Делать следующие вычисления, каждое один раз в 6 циклов (30 Гц):
1.3.1. Вычисление управляющего воздействия для дифферента - внешний цикл.
1.3.2. Вычисление управляющего воздействия для крена - внешний цикл.
1.3.3. Вычисление управляющего воздействия для рыскания и для коллективных скоростей - внешний цикл.
1.4. Делать каждое из следующих вычислений один раз в 2 цикла (90 Гц), используя выходы, полученные 30-Гц вычислениями и задачами авионики как входы:
1.4.1. Вычисление управляющего воздействия для дифферента - внутренний цикл.
1.4.2. Вычисление управляющего воздействия для крена и для коллективных скоростей - внутренний цикл.
1.5. Вычисление управляющего воздействия для рыскания - внутренний цикл, используя выходы 90-Гц вычислений законов управления как входы.
1.6. Выдача команд.
1.7. Выполнение встроенного теста.
1.8. Ждать начала следующего цикла.

Три скорости вдоль осей названы в алгоритме коллективными скоростями, скорости вращения вокруг осей показаны на рис. 1.3.

Этот многочастотный (многопериодический) контроллер управляет только динамикой полета. Система управления на борту самолета имеет также много других столь же критических подсистем: воздухозаборники, подача топлива, гидравлическая система, тормоза, антиледовое управление и т.д.

Во многих случаях датчики дают не значения переменных состояния, а только некоторые наблюдаемые характеристики объекта управления. Значения же переменных состояния должны вычисляться на их основе. В таких более сложных случаях контроллер может быть представлен следующей схемой:

1. Установить таймер на прерывания периодически с периодом T.
2. При каждом прерывании от таймера делать:
2.1. Дискретизировать показания датчиков.

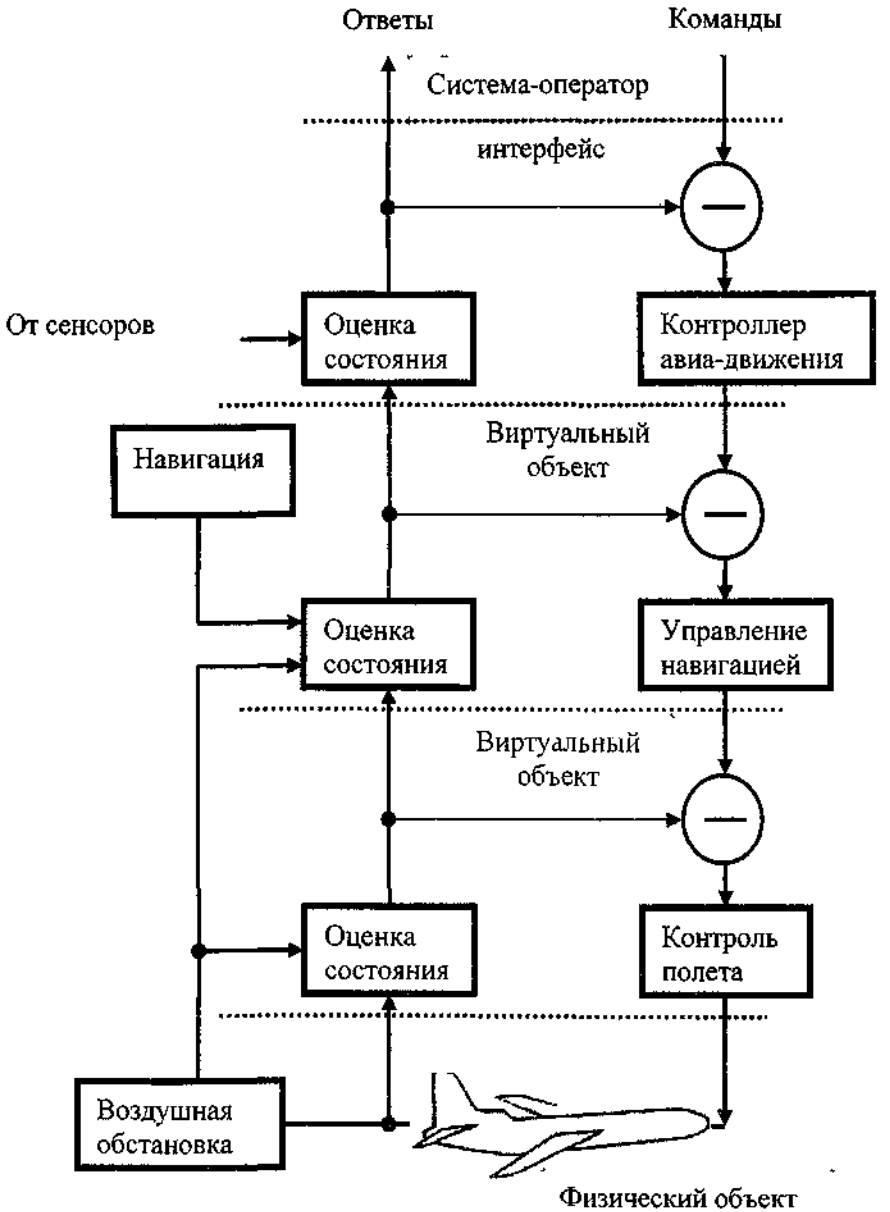


Рис 1 4 Иерархия системы управления полетом/авиадвижением

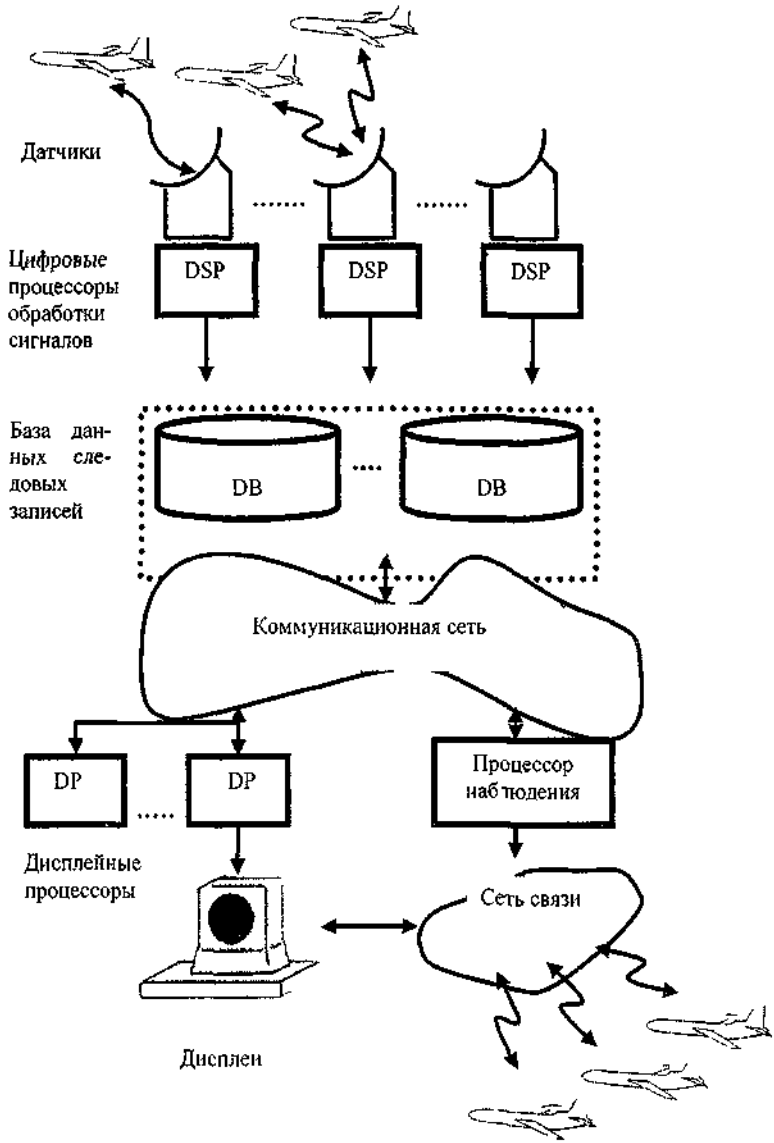


Рис. 1.5. Архитектура системы управления воздушным движением

2.2. Вычислить управляющее воздействие на основе измеренных значений и значений переменных состояния.

2.3 Преобразовать выходное значение в аналоговую форму

2.4. Вычислить и обновить значения параметров объекта управления и переменных состояния.

3. Конец цикла.

Обычно управление производится на многих уровнях иерархии.

Рис. 1.4 показывает уровни иерархии: управление динамикой полета, управление навигацией и управление воздушным движением.

Система управления воздушным движением (СУВД) находится на верхнем уровне. Она управляет потоками самолетов в каждый аэропорт назначения, указывая самолетам время прибытия в точки отметки - известные географические точки на пути к аэропорту назначения. Расстояние между такими точками обычно составляет 40-60 миль. Предполагается, что самолет прибудет в точку отметки в указанное время. В каждый момент полета назначенное время прибытия в очередную точку отметки является входом для бортовой системы управления навигацией. Бортовая система управления навигацией выбирает такую траекторию полета, чтобы самолет был в точке отметки в назначенное время. Скорость полета, радиус поворота, скорость взлета/посадки и тому подобное для обеспечения прибытия в нужное время являются входами для системы управления динамикой полета на нижнем уровне иерархии.

Вычисления контроллеров на нижнем уровне могут быть простыми и детерминированными, в то время как контроллеры высокого уровня взаимодействуют с оператором и являются более сложными и многовариантными (например, экспертная система). Период для контроллера низкого уровня находится в пределах от миллисекунд до секунд, периоды высокоуровневых контроллеров могут равняться минутам и даже часам.

Рис. 1.5 иллюстрирует архитектуру СУВД. СУВД отслеживает самолеты в своей зоне ответственности, генерирует и отображает информацию, необходимую операторам. Выходы СУВД включают назначение времен прибытия в точки отметки для каждого самолета. Как уже было сказано, эти выходы являются входами для бортовых систем управления навигацией. Таким образом, СУВД неявно управляет встроенными компонентами нижних уровней иерархии. В дополнение, СУВД обеспечивает голосовую и телеметрическую связь с бортовыми устройствами. Она собирает информацию о состоянии каждого самолета с помощью радаров. Такие радары опрашивают каждый самолет периодически. При опросе самолет отвечает посылкой СУВД значений своих переменных состояния: идентификатор, позиция, высота, заголовки и т.д. (эти переменные на рисунке обозначены как следовая запись, а текущая траектория самолета - его след). СУВД обрабатывает сообщения от самолета и хранит информацию о состояниях в базе данных. Эта информация читается и отображается дисплейными процессорами. В то же время, система наблюдения постоянно анализирует сценарий развития событий и выдает предупреждения операторам в случае потенциальной опасности (например, возможность столкновения). Снова упомянем, что системы, с которыми работают

операторы (клавиатура, дисплей), должны обеспечивать частоту реакции не меньше 10 Гц.

Из этого примера можно видеть, что система управления в целом не похожа на низкоуровневые контроллеры. В противоположность низкоуровневому контроллеру, чья нагрузка или полностью, или в значительной степени периодическая, системы управления высших уровней должны отвечать на спорадические (случайные) события и команды оператора. Они могут обрабатывать изображения и речь, делать запросы поиска и модификации в базах данных, моделировать различные сценарии развития событий и т.д. Требования к ресурсам и процессорному времени для таких систем могут быть большими и меняющимися.

1.3. Критерии эффективности систем реального времени

При проектировании СРВ, а также при сравнении существующих систем используются различные критерии их качества и эффективности, наиболее важные из которых :

- производительность;
- стоимость системы;
- стоимость единицы операции;
- объем, масса;
- энергопотребление;
- интенсивность излучения системы;
- рабочая температура;
- устойчивость системы к внешним воздействиям (радиация, вибрация, воздействие кислот и т.п.).

Рассмотрим некоторые из них подробнее.

Производительность - количество выполняемых операций в единицу времени. Если, например, должен быть решен пакет задач $\{z_i\}_{i=1}^{I}$ вычислительной сложности Q_i, то производительность будет:

$$P = \frac{\sum_{i=1}^{I} Q_i}{T},$$

где T - время выполнения всего пакета задач.

Под ***вычислительной сложностью*** понимают количество операций, необходимое для решения задачи.

Для оценки текущей производительности в момент времени t можно использовать выражение

$$P(t) = \lim_{\Delta \to 0} \frac{Q(t, t+\Delta)}{\Delta},$$

где Q(t1,t2) - вычислительная сложность задач, выполненных системой на интервале времени [t1,t2).

Стоимость системы С включает не только затраты на создание или продажную стоимость, но также и стоимость поддержания системы в рабочем состоянии.

Стоимость единицы производительности, измеряемой в операциях в секунду, рассчитывается как C/P. Иногда используется инверсная величина P/C, характеризующая производительность получаемую на рубль затрат.

СРВ предназначены для решения определенных наборов задач, например, моделирование движения летательного аппарата, нахождение оптимальной траектории движения, выработка и передача управляющих воздействий исполнительным механизмам. Эти задачи должны быть решены в минимально возможное время или во время, не превышающее заданного. В соответствии с этим могут быть рассмотрены следующие критерии:

- минимизация общего времени выполнения;
- повышение количества задач, обрабатываемых в единицу времени - ***пропускной способности***.

Рассмотрим минимизацию общего времени выполнения T_o пакета из L задач. Если f_i есть момент времени завершения i-й задачи, то общее время выполнения $To=\max_{i=1,L} f_i$, и необходимо найти $\min \max_{i=1,L} f_i$. Этот критерий соответствует максимизации производительности системы при решении пакета задач, так как вычислительная сложность постоянна, а знаменатель в выражении для производительности минимизируется.

Каждая задача пакета может иметь ***директивный срок*** d_i - момент времени. к которому она должна быть завершена. При этом можно рассмотреть следующие критерии:

$$\min \sum_{i=1}^{L}(f_i - d_i),\ \min \sum_{i=1}^{L}| f_i - d_i |,\ \min \sum_{i=1}^{L}\max(0, f_i - d_i),$$

$$\min \max_{i=1,L}(f_i - d_i),\ \min \max_{i=1,L}| f_i - d_i |,\ \min \max_{i=1,L}(0, f_i - d_i).$$

Первые три из этих критериев используют среднее отклонение от директивных сроков, последние три используют максимальное отклонение. В первом критерии превышение директивных сроков для одних задач может быть скомпенсировано ранним решением других задач, во втором критерии штрафуются как раннее, так и позднее завершение задач, в третьем варианте штрафуется только задержка в решении задач. Для последних трех критериев большое значение имеет максимальное отклонение. Кроме того, могут вводиться веса, характеризующие важность задач и соответственно нарушений директивных сроков в их решении.

Для систем серверного типа, например, телефонных станций, узлов сообщений, штабов важен критерий среднего числа задач (пользователей), обслужен-

ных системой в единицу времени, т.е. пропускная способность системы. Здесь на первом месте стоит не вычислительная сложность выполненных задач, но количество обслуженных пользователей, стоящих за этими задачами, количество принятых решений. Пропускная способность максимизируется при минимизации среднего времени пребывания задач в системе. Пусть число задач - L, момент времени прибытия i-й задачи в систему - s_i, момент времени завершения обслуживания i-й задачи - f_i, i=1,..,L. Тогда среднее время пребывания задач в системе есть Tc = $\sum_{i=1}^{L}(f_i - s_i)/L$. Если время пребывания в системе i-й задачи $f_i - s_i, i = \overline{1, L}$, то можно сказать, что для i-й задачи каждую единицу времени $1/(f_i - s_i)$ часть задачи покидает систему и, в среднем, следующее количество задач - $L/\sum_{i=1}^{L}(f_i - s_i)$ - уходит из системы за единицу времени. Таким образом, для максимизации пропускной способности необходимо минимизировать среднее время пребывания задач в системе, а для максимизации производительности необходимо минимизировать максимальное время пребывания задач в системе.

Отметим, что минимизация T_o есть NP-трудная задача [2], имеющая экспоненциально растущее время решения с увеличением в системе количества процессоров и задач. Для получения близких к оптимальным решений таких задач обычно используются полиномиально-трудоемкие эвристические алгоритмы.

Так, для оптимизации T_o может использоваться алгоритм LPT (Largest Processing time task - first) с временной сложностью $O(L\log_2 L+Ln)$, где L - количество задач, n - количество идентичных процессоров вычислительной системы. Суть алгоритма в предварительном упорядочении задач по убыванию времени выполнения и в назначении их в таком порядке на освобождающиеся процессоры. При относительно небольшом количестве процессоров наиболее трудоемкой в данном алгоритме является сортировка. В [3] доказано, что при решении любого пакета независимых задач П (порядок их решения произволен) $T_o(П,LPT)/T_o(П,ОПТ)<4/3-1/(3n)$, где $T_o(П, ОПТ)$ - оптимальное время исполнения пакета П, т.е. ошибка алгоритма LPT не превышает 33% в худшем случае.

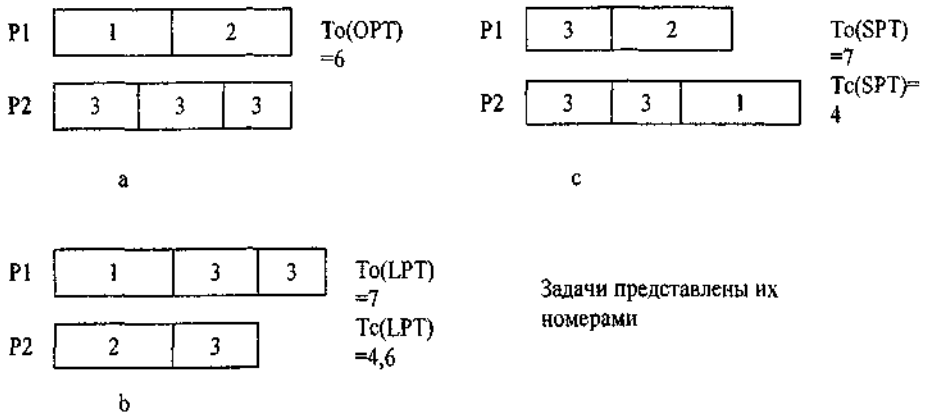


Рис. 1.6. Пример распределения задач по процессорам

Практически та же задача оптимизации T_c решается алгоритмом SPT (Shortest Processing time task - first) той же сложности, что и LPT. Отличие состоит в том, что в алгоритме SPT задача с меньшим временем выполнения назначается раньше.

Рассмотрим пример. Пусть задачи пакета П имеют следующие времена выполнения: (3,3,2,2,2). Тогда в случае 2-процессорной системы (n=2) T_o(П, OPT)=6 (рис.1.6,a), T_o(П, LPT), даваемое алгоритмом LPT, равно 7, T_c(П,LPT)=(3+3+5+5+7)/5=23/5=4.6 (рис. 1.6,b), T_o(П, SPT)=7, T_c(П, SPT)=(2+2+4+5+7)/5=20/5=4 (рис. 1.6,c).

При проектировании СРВ необходимо принимать во внимание различные критерии, зачастую противоречивые (производительность - стоимость, производительность - размер и т.д.). Очевидно, что оптимальное решение этой задачи чрезвычайно сложно, так как необходимо проводить оптимизацию по многим критериям. На практике, в таких случаях используются следующие упрощающие подходы:

- Использование одного общего критерия, получаемого как одно выражение. Обычно рассчитывается взвешенная сумма критериев с весами из интервала (0,1], характеризующими важность соответствующих критериев, причем сумма весов равна 1.
- Оптимизация по одному критерию с использованием значений других критериев в качестве ограничений.

Таким образом, проектирование СРВ - сложная многокритериальная задача, в результате решения которой должна быть получена ВС, имеющая необходимые с точки зрения успешного управления соответствующими объектами характери-

стики. Для достижения этих целей необходимо эффективное использование ресурсов ВС. Как видно из рассмотренного выше примера, изменение порядка выполнения задач без всякой модификации оборудования, увеличения цены, массы и т.п., может привести к существенному изменению системных характеристик (в нашем примере, порядка 10%, но можно привести более сильные примеры). Основные ресурсы системы - процессоры, процессорное время, оперативная память, внешняя память - используются выполняющимися программами. Следовательно, мы рассмотрим вопросы управления процессами и системными ресурсами.

Обычно СРВ должны управлять множеством объектов одновременно, поэтому даже в случае однопроцессорных систем необходимо поддерживать многозадачность, потому что невозможно сказать, что сначала мы завершим управление первым объектом, а потом приступим к управлению вторым, из-за неограниченности времени существования объектов управления (например, металлургические печи работают десятки лет). Поэтому для СРВ очень важно оптимальное управление параллельными процессами, ответственными за соответствующие объекты управления. Для получения более быстрой реакции -выдачи управляющих воздействий - в СРВ широко используется распараллеливание вычислений между несколькими процессорами, поэтому мы рассмотрим вопросы распараллеливания и связанные с ними проблемы.

СРВ предназначены для управления реальными объектами, и стоимость их сбоя может быть исключительно высокой (например, ошибка управления ядерным реактором Чернобыльской АЭС). Значит, должны быть использованы методы повышения надежности программ и оборудования СРВ. Мы рассмотрим эти вопросы применительно к методам проектирования, языкам и средствам разработки.

СРВ - сложные аппаратно-программные комплексы, в которых должна присутствовать модель управляемого объекта, средства взаимодействия с внешним миром с помощью устройств ввода-вывода. Должны быть обеспечены известные человеку-оператору средства управления: панель управления АЭС, самолета, автомобиля и т.п. Эти вопросы успешно решаются с помощью систем SCADA (Supervisor Control And Data Acquisition - контроль и получение данных), информация по которым будет приведена в пособии.

Очень важно для СРВ взаимодействие с внешними устройствами, которое обычно осуществляется с помощью драйверов. Далее будут освещены некоторые вопросы создания драйверов.

Одной из основных частей СРВ является операционная система, на базе которой работают программные компоненты системы. Для этих целей выделяют отдельный класс операционных систем - операционные системы реального времени. Такие операционные системы обладают рядом особенностей, которые мы в дальнейшем обсудим. Будут рассмотрены примеры существующих операционных систем реального времени и действующих СРВ.

Контрольные вопросы и задания

1. Дайте определение СРВ.
2. Какие основные типы СРВ вам известны?
3. Почему СРВ имеют иерархическую структуру?
4. Каковы особенности СРВ низшего уровня иерархии (контроллеры)?
5. Почему время дискретизации имеет важное значение? На что влияет этот параметр?
6. Каковы особенности СРВ верхнего уровня иерархии (системы управления)?
7. Перечислите критерии эффективности СРВ.
8. Что такое пропускная способность системы? Как ее можно увеличить?
9. Какие подходы используются для уменьшения общего времени исполнения задач?
10. Как можно учесть несколько критериев эффективности при проектировании СРВ?
11. Пусть однопроцессорная СРВ с приоритетами и прерыванием задач должна периодически управлять тремя объектами. Каждый объект управляется соответствующей задачей. Периодичность запросов на задачи, их сложность (количество операций) и приоритеты задач указаны ниже:

Номер задачи	1	2	3
Период, миллисекунды	5	50	100
Количество операций*10^5	5	50	100
Приоритет	1	2	3

Высший приоритет - 1 (приоритет наиболее важной задачи). Каждая задача может быть отложена, но ее исполнение должно завершиться до момента следующего вызова этой задачи (до истечения интервала времени, равного периоду задачи). Переключение контекста требует 100 операций и использует стек времени исполнения.

- Нарисуйте временную диаграмму, отражающую наихудший для обслуживания сценарий появления запросов на обслуживание (взрыв запросов).
- Рассчитайте минимальную производительность процессора СРВ, удовлетворяющую системным требованиям.
- Рассчитайте минимальный размер стека времени исполнения.

Глава 2. Методы и средства проектирования систем реального времени

Как уже было отмечено выше, СРВ являются вычислительными системами, предназначенными для работы в условиях временных ограничений. Они должны обеспечивать повышенный уровень надежности. Это приводит к тому, что СРВ используют все средства современных компьютерных технологий в программном и аппаратном обеспечении. Большинство прогрессивных методов из области вычислительной техники применяются при разработке СРВ.

В этой главе кратко рассмотрены основные этапы и стандарты разработки программного обеспечения (ПО), его жизненный цикл. Приведены некоторые наиболее распространенные методы описания и моделирования сложных систем, позволяющие создавать для них эффективное и надежное ПО.

2.1. Жизненный цикл программного обеспечения, стандарты

Мы в основном уделим внимание процессу проектирования ПО. СРВ представляют собой сложные системы, и проектирование программ для них следует вести, руководствуясь правилами разработки сложного ПО коллективом специалистов. Такое ПО должно быть хорошо документировано во избежание последующего непонимания программистами при стыковке его частей. Кроме того, должна иметься возможность тестирования системы в соответствии с начальными требованиями.

Обычно жизненный цикл ПО состоит из таких этапов, как спецификация, проектирование, внедрение, тестирование и поддержка. В соответствии со стандартом MIL-STD-2167A [4] процесс разработки ПО должен включать следующие основные виды работ, которые могут перекрываться и применяться итерационно:

- анализ/разработка системных требований;
- анализ требований к программному обеспечению,
- предварительная разработка;
- детальная разработка;
- кодирование и тестирование программных модулей (CSC - Computer Software Unit);
- интеграция и тестирование компонент ПО (CSC - Computer Software Component);
- тестирование конфигурационных частей ПО (CSCI - Computer Software Configuration Item);
- системное интегрирование и тестирование.

Компонент ПО - это элемент конфигурационной части программного обеспечения. Компоненты могут быть декомпозированы далее в другие компоненты и программные модули. В ноябре 1994 этот стандарт был заменен на MIL-STD-498.

Отметим, что данный стандарт не является единственным, и, например, стандарт ISO 9000 в своем составе имеет стандарт ISO 9000-3, посвященный разработке ПО.

Рассмотрим еще один подход к проектированию ПО - модель водопада [5]. Эта модель предполагает использование при проектировании программ последовательности следующих шагов:

1. Фаза концепции - определить цели проекта или исследовать возможность их достижения. Выходной документ - белая бумага.

2. Фаза требований - решить, что продукт должен делать. Формируется документ с требованиями, которые бывают функциональными и нефункциональными. К последним относятся, например, язык программирования, стиль программирования и т.п. Документ должен быть полным, корректным (соответствовать общим законам), непротиворечивым, каждое требование должно иметь возможность проверки.

3. Фаза разработки. Цель фазы - показать, каким образом реализовать проект в соответствии с требованиями. Формируется проектная документация, описывающая разбиение ПО на модули, тестовые примеры, детальная документация по ПО. Модули должны скрывать реализацию используемых в них решений от остальной системы и должны обладать полностью определенными интерфейсами.

4. Фаза программирования - построение системы. Создаются программные коды. Обычно каждый программист делает свою часть (модуль) ясно зная интерфейсы и цели своей части. На этом этапе полезно осуществлять контроль версий. Разные части системы должны быть разработаны, отлажены, пройти тесты и быть интегрированы в систему в целом.

5. Фаза тестирования. Цель - проверить, удовлетворяет ли система требованиям. На выходе фазы формируются отчеты о тестировании.

6. Фаза поддержки. Производится поддержка системы, ее сопровождение, создаются отчеты о поддержке, включающие информацию о размещении продукта, поддержке пользователей, коррекции программных ошибок.

Заметим, что обычно эти действия выполняются итерационно. Реализация каждой следующей фазы может приводить к обнаружению ошибок в предыдущих фазах, что может вести к невозможности реализации следующей фазы без модификации предыдущих, т.е. приводить к итерированию фаз. Другой подход заключается в использовании спиральной модели, в которой сначала делается грубый прототип, он анализируется, принимается, затем производится следующее приближение и так продолжается до достижения нужного результата.

2.2. Методы спецификации и проектирования систем реального времени

Спецификация выполняется заказчиком и указывает, что и при каких условиях должен делать продукт; разработка определяет, как достичь этих целей. Правильные, непротиворечивые и грамотные спецификации очень важны для создаваемой системы. Поэтому созданию спецификаций и описания системы необходимо уделить особое внимание.

Существует ряд методов создания спецификаций системы. Далее мы рассмотрим некоторые из них.

2.2.1. Естественные языки

Естественные языки хорошо понимаются людьми, но достаточно расплывчаты и не могут ясно выразить цели работы. Их нельзя автоматически преобразовать в выполняемые коды. Например, известная задача синхронизации "обедающие философы" может быть описана так: в комнате имеется k философов (мудрецов), чередующих размышления с приемом пищи. Для каждого мудреца выделена своя тарелка, причем философу требуется две вилки, чтобы поесть, более того, он может использовать только вилки, расположенные по бокам его тарелки. Требуется синхронизировать философов так, чтобы каждый из них мог за конечное время приступить к обеду. Считается, что периоды размышления и приема пищи конечны, но неизвестны заранее.

2.2.2. Математическая формулировка

Математическая формулировка точна и не имеет неопределенностей. Она дает возможность использования методов автоматической генерации кодов, оптимизации и верификации. Тем не менее эти методы могут быть тяжелы для восприятия разработчикам ПО.

Приведем формулировку задачи о философах с использованием математического метода. Имеется k параллельных процессов-философов Pi, i = 0, ..k-1. Имеется k булевых переменных (вилок) f_i, i = 0,..,k-1. Каждый процесс может быть в одном из трех состояний T(thinking - размышление), E(eating - питание), W (waiting - ожидание). Начальное состояние каждого процесса есть T и начальное состояние каждой f_i равно TRUE, i = 0,...k-1 (вилка доступна). Процесс находится в состоянии T в течение некоторого времени tt, которое может считаться случайной величиной. По истечении времени tt процесс Pi анализирует наличие необходимых вилок: f_i and $f_{(i+1)mod k}$ == TRUE. Если результат - TRUE, то он забирает эти вилки: $f_i = f_{(i+1)mod k}$ = FALSE и переходит в состояние E, в котором находится также некоторое случайное время te. После истечения времени te он возвращает вилки: $f_i = f_{(i+1)mod k}$ =TRUE и переходит в состояние T. В случае, когда результат анализа вилок равен FALSE, процесс Pi переходит в состояние W, в котором он периодически проверяет наличие необходимых вилок: f_i and $f_{(i+1)mod k}$ ==TRUE до их появле-

ния. При доступности вилок он берет их: $f_i = f_{(i+1) mod k}$ =FALSE и переходит в состояние Е.

2.2.3. Блок-схемы

Блок-схемы представляют собой старейший хорошо понимаемый метод проектирования ПО. Принципы их построения хорошо известны и не будут описываться подробно. Блок-схемы не годятся для больших проектов, содержащих более 10 000 инструкций и проектов с параллельными процессами. Такое ограничение обусловлено возрастающей сложностью схем, влекущей за собой трудности в понимании.

2.2.4. Структурные схемы

Структурные схемы показывают взаимосвязь между модулями систем посредством дерева вызовов. Дерево вызовов представляет собой декомпозицию системы на модули и показывает, как модули нижнего уровня вызываются модулями верхнего уровня. К недостаткам структурных схем можно отнести отсутствие условного ветвления.

2.2.5. Карты Джексона

Карты Джексона [6, 7] являются расширением структурных схем и используют для моделирования системы ***структурные диаграммы сущностей*** и ***сетевые диаграммы*** .

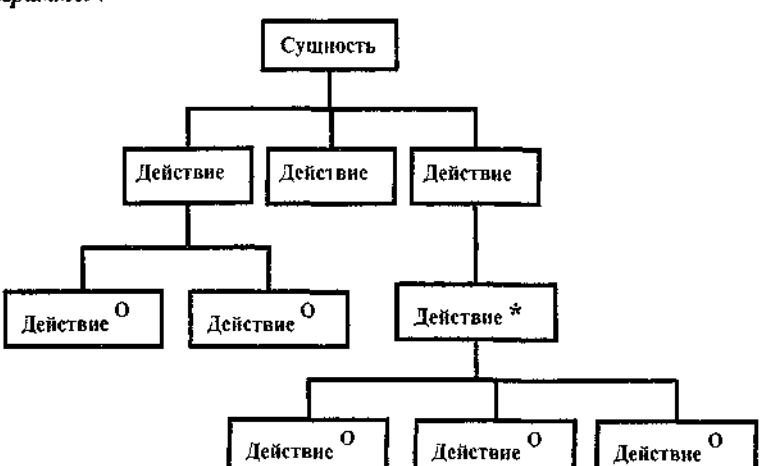


Рис. 2.1. Структурные диаграммы сущностей

Структурные диаграммы сущностей иллюстрируют временной порядок действий сущностей в системе (рис. 2.1).

Сущностью называется объект, который действует и подвергается воздействию системы.

Действия выполняются сущностями и воздействуют на другие сущности. Они соединены с корневой сущностью и друг с другом в соответствии с иерархическим отношением "родитель-ребенок".

Рис. 2.2 показывает конструкцию "Последовательность", которая используется для отображения действий выполняемых слева-направо.

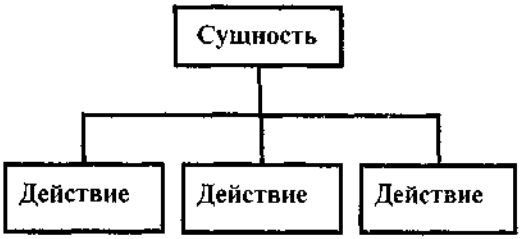


Рис. 2.2. Конструкция "Последовательность"

На рис. 2.3 показана конструкция "Выбор". Она используется для представления выбора между двумя или более взаимоисключающими действиями. Действия маркируются малой буквой "о" (опция) в правом верхнем углу.

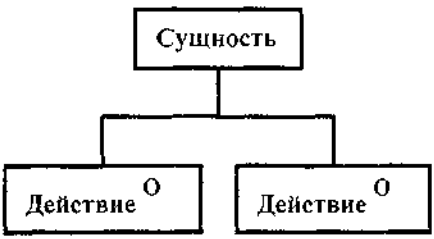


Рис. 2.3. Конструкция "Выбор"

Если некоторое действие должно повторяться, то используется конструкция "Итерация" (рис. 2.4).

Рис. 2.4. Конструкция "Итерация"

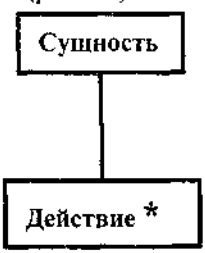


Рис. 2.5. Пустой компонент

Пустой компонент (рис. 2.5) используется тогда, когда ничего делать не надо. Например, в случае ветвления, пустой компонент может показывать альтернативу "ничего не делать".

Сетевые диаграммы (рис. 2.6) отображают взаимодействие между процессами. Они называются также диаграммами спецификации системы.

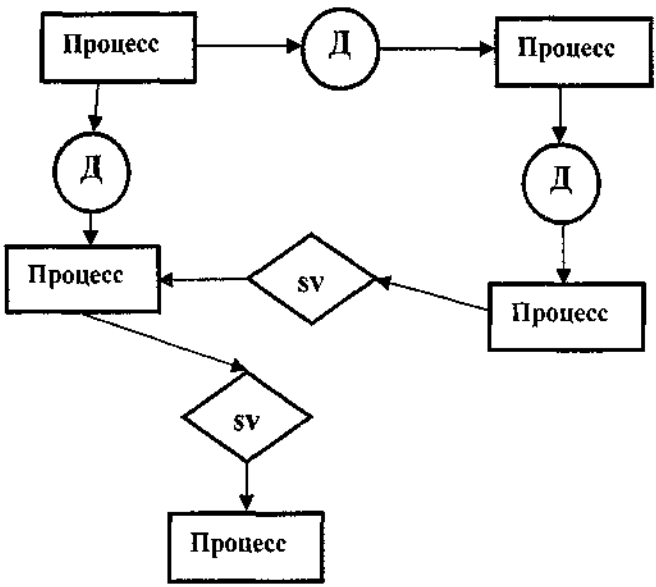


Рис. 2.6. Сетевые диаграммы Джексона

Каждый процесс моделирует некоторую базовую функцию системы. Процессы обычно связаны между собой потоками данных, которые определяют, какая информация проходит между ними (рис. 2.7).

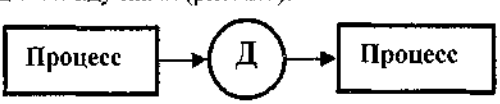


Рис. 2.7. Поток данных между процессами

Векторы состояний (рис. 2.8) представляют альтернативный вариант соединения процессов. Они определяют характеристики или состояния сущностей, изменяемых процессом.

Рис. 2.8. Вектор состояний

С помощью рассмотренных диаграмм можно, например, описать "обедающих философов", как показано на рис. 2.9, 2.10.

Рис. 2.9. Структурная диаграмма "Обедающие философы"

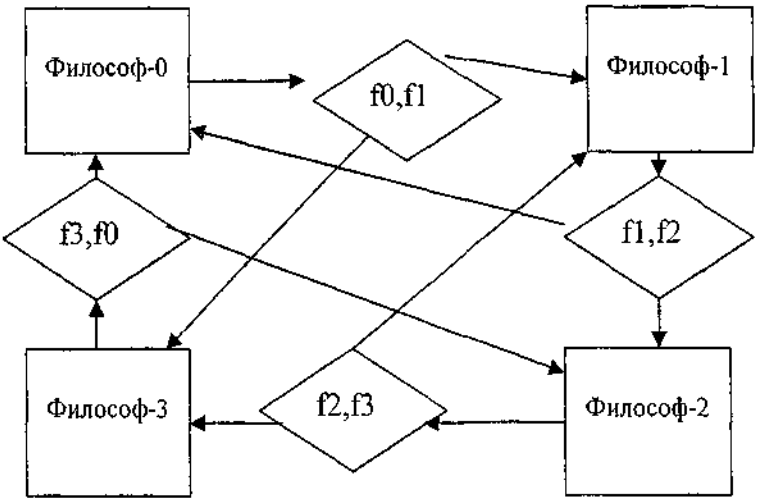


Рис. 2.10. Сетевая диаграмма "Обедающие философы"

2.2.6. Псевдокод и языки проектирования программ

Псевдокоды - языки высокого уровня, освобожденные от мелких деталей. Обычно в псевдокоде используются конструкции похожие на те, что имеются в C, Pascal и других алгоритмических языках. Псевдокоды также называют языками проектирования программ (ЯПП или PDL - programming design language). Для ЯПП обычно имеются трансляторы в какой-либо из языков программирования. Языки программирования Ada, Modula-2 могут быть использованы в качестве ЯПП. Следует отметить, что ЯПП - это тоже язык программирования, с которым пользователь должен быть знаком, причем стоимость разработки и поддержки инструментария для языков проектирования достаточно высока.

Приведем пример использования псевдокода для задачи "обедающие философы".

```
'Dining philosophers':
k - число процессов
fi=1, i=0,..,k-1
Pi(i=0,..,k-1){
 State from {T,E,W};
 State=T;
 While(1){
    Case state{
    T: сформировать случайное целое tt>0; while(tt>0)tt--;
        If (fi and f(i+1)modk){
          fi= f(i+1)modk =0;state =E;}
        Else state = W;
```

```
    E: сформировать случайное целое te>0; while(te>0)te--;
        f1= f(i+1)modk =1;state =T;
    W: If (f1 and f(i+1)modk){
         f1= f(i+1)modk =0;state =E;}
     }//end case
    }//end while
  }//end Pi
```

Запуск параллельных процессов-философов можно произвести, например, так.

```
  par i=0,..,k-1//запустить процессы параллельно
    Pi;
```

2.2.7. Конечные автоматы

Конечный автомат есть вектор (X,Y,S,S0,TF X*S->S,OF X*S->Y), где X и Y - множества входных и выходных сигналов, S - множество состояний автомата, TF и OF - функции переходов и выходов, определяющие следующее состояние и выход в зависимости от текущего состояния и входного сигнала

Есть три метода представления конечных автоматов теоретико-множественный, в виде диаграмм, матричный

Различают два класса автоматов автоматы Мили и автоматы Мура Математически они эквивалентны В автоматах Мура выход зависит только от текущего состояния, а в автоматах Мили - от состояния и входного сигнала

На рис 2 11 показан распознаватель четности, представленный в виде конечного автомата Мура Конечное состояние автомата показано двойной линией, начальное - стрелкой Ребра маркированы сигналами, вызывающими переходы между состояниями Предполагается, что в каждый дискретный момент времени может прийти сигнал

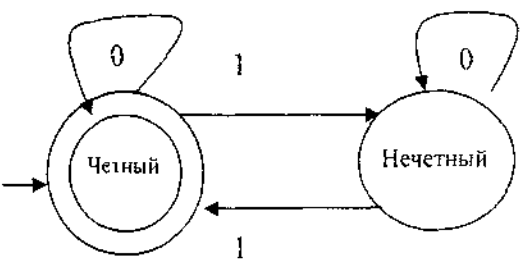


Рис 2 11 Автомат Мура, распознающий четность

Тот же распознаватель на рис 2 12 представлен как автомат Мили

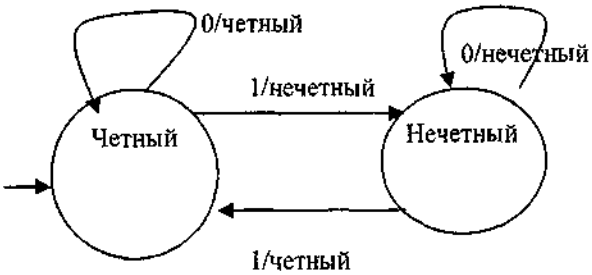


Рис. 2.12. Автомат Мили, распознающий четность

2.2.8. Диаграммы потоков данных

Диаграммы потоков данных [8] иллюстрируют в терминах входов/выходов, как данные обрабатываются системой.

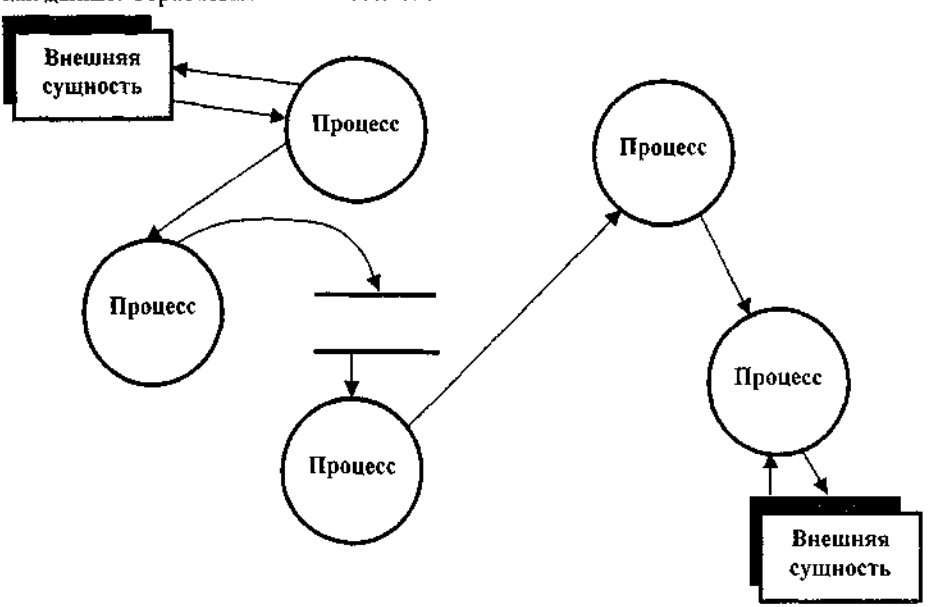


Рис. 2.13. Диаграмма потоков данных

Процесс преобразует входные потоки данных в выходные (рис. 2.13). Данные хранятся в системных хранилищах, называемых также файлами или репозиториями.

Потоки данных - конвейеры, через которые текут пакеты информации. Стрелки помечаются именами протекающих данных.

Внешние сущности - объекты вне системы, с которыми система взаимодействует. Они являются истоками и стоками для системных входов и выходов.

Вершина, представляющая один процесс на диаграмме верхнего уровня, может быть расширена в более детальную диаграмму потоков данных. Сначала показывается контекстная диаграмма, а затем приводятся детальные описания.

Контекстная диаграмма (рис. 2.14) - это диаграмма верхнего уровня 0. Она содержит только одну вершину процесса, представляющего систему в целом и ее связи с внешним миром.

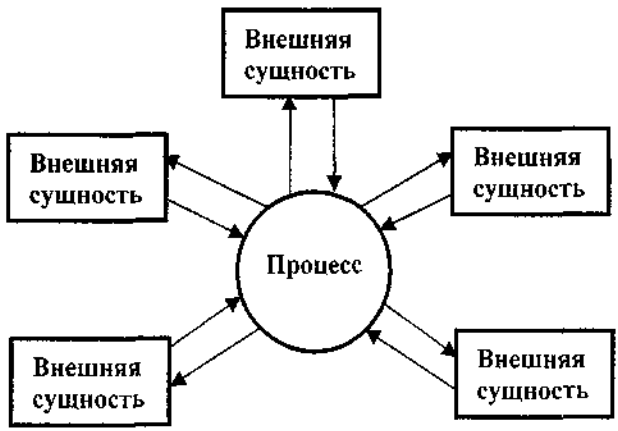


Рис. 2.14. Контекстная диаграмма

Диаграмма верхнего уровня показывает главные процессы системы. Каждый из этих процессов может быть разбит на подпроцессы до достижения уровня, хорошо представимого псевдокодом.

На рис. 2.15 приведен пример диаграммы потоков данных для системы заказа товаров посредством кредитной карты.

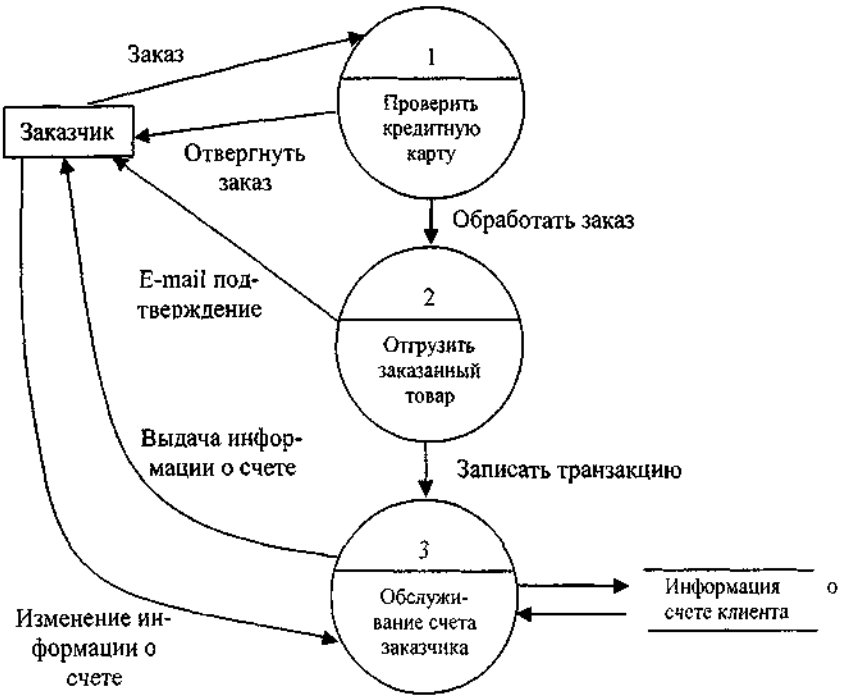


Рис. 2.15. Пример диаграммы потоков данных

2.2.9. Сети Петри

Сети Петри хорошо подходят для представления параллельных процессов. Это графы, имеющие вершины двух типов: ***места*** и ***переходы***. Только вершины разного типа могут быть соединены ребрами, т.е. места с переходами и наоборот.

В начальный момент времени сеть Петри маркирована метками, ассоциированными с местами. Каждое место может иметь несколько меток, представляющих единицы ресурса. Для каждого перехода должны быть определены: входные места, выходы которых являются входами для вершины-перехода; выходные места, с которыми связаны выходы этой вершины-перехода. Переход срабатывает, когда каждое из входных мест имеет, по крайней мере, одну метку. Срабатывание есть неделимое действие (транзакция), приводящее к уменьшению на единицу числа меток в каждом входном месте и к увеличению на единицу числа меток в каждом выходном месте. Если два перехода не имеют общих входных и выходных мест, они могут сработать независимо и параллельно. Если несколько переходов могут сработать, и они имеют общее входное место только с одной меткой, то только один из этих переходов может сработать, причем какой из них - не определяется. Переходы могут быть рассмотрены как процессы, ожидающие готовности вход-

ных сигналов. Когда все входы готовы, процесс выполняется и поставляет информацию в выходные места. В сетях Петри времена выполнения процессов не рассматриваются.

Рассмотрим представление пяти обедающих философов в виде сетей Петри (рис. 2.16). Для каждого философа показаны его состояния размышления (Mi) и питания (Ei). Вилки обозначены как Ci.

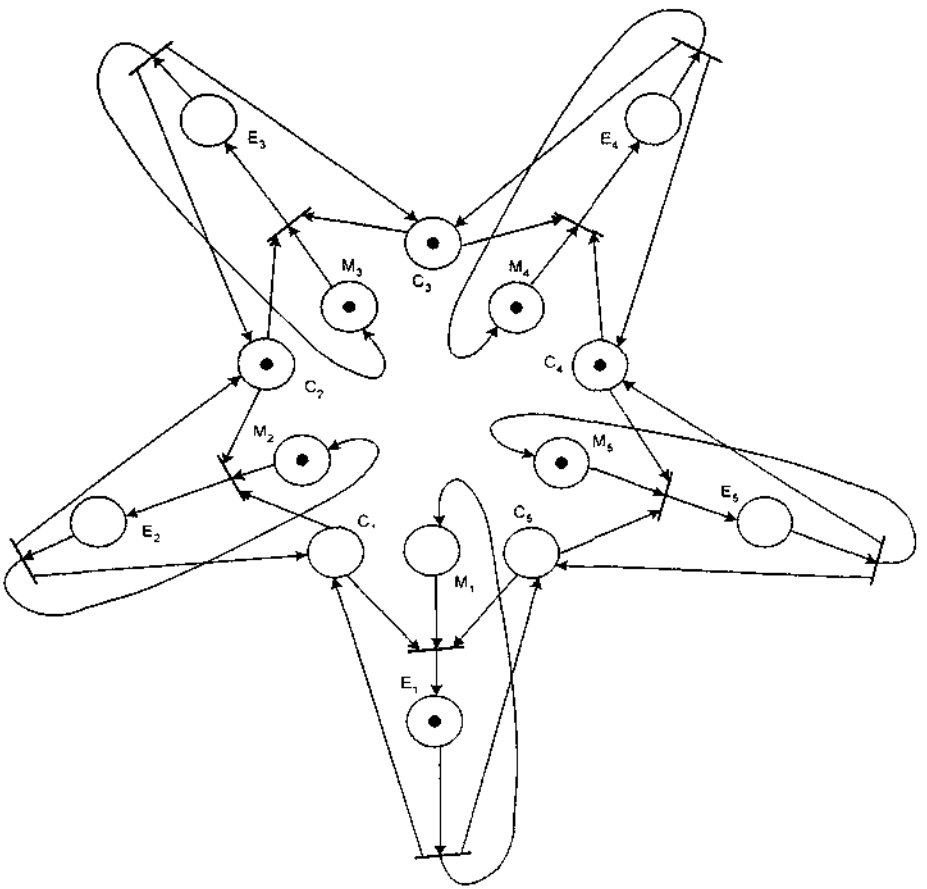
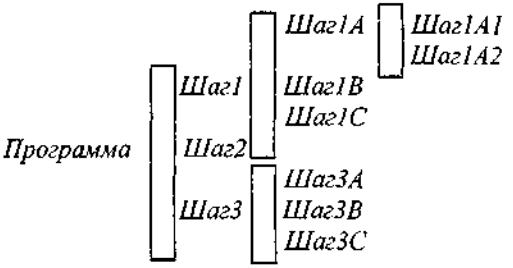


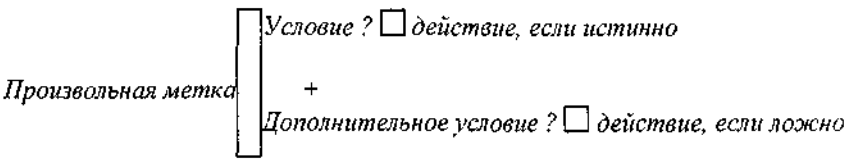
Рис. 2.16. Представление задачи "обедающие философы" в виде сети Петри

2.2.10. Представление Варнье-Орра

Нотация Варнье-Орра [9] подобна картам Джексона, но использует фигурные скобки для показа блоков. Отметим, что мы применяем ниже прямоугольники вместо скобок. Общая структура программы может быть записана примерно так:



Конструкция If-then-else может быть представлена как:



Цикл WHILE обозначается так:

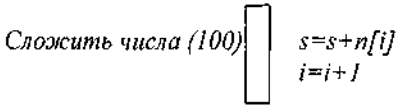
Произвольная метка (условие, W) ☐ *действие*

Цикл Repeat … until обозначается аналогично, но вместо "W" используется "U".

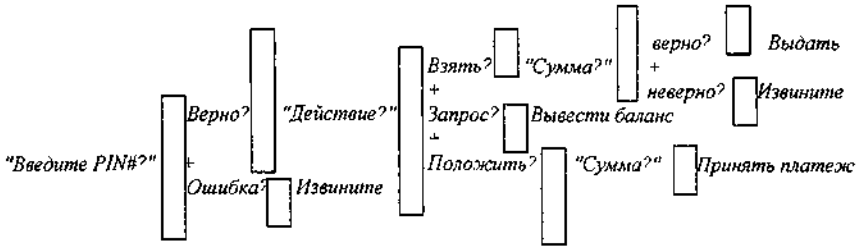
Цикл с параметром можно представить следующим образом:

Произвольная метка (n) ☐ *действие*

Например, сложение 100 чисел, полагая, что изначально i=1, s=0 выглядит так:



В качестве иллюстрации, изобразим схему работы банкомата:



2.2.11. Диаграммы состояний

Диаграммы состояний соединяют в себе конечные автоматы и диаграммы потоков данных. Они могут описывать асинхронные операции, называемые широковещательным взаимодействием, используют иерархическое представление путем вложения диаграмм в диаграммы более высокого уровня. С помощью диаграмм состояний можно описывать параллельные процессы, представляемые AND-состояниями - вложенными компонентами. Если состояние Y состоит из AND-компонент A и B, то Y называется ***ортогональным произведением*** A и B, и если в состояние Y входят извне, то одновременно входят в состояния A и B. если не оговорено противное. Ортогональность представляется штриховыми линиями. Взаимодействие между ортогональными состояниями осуществляется через глобальную память. Широковещательное взаимодействие представляется переходами между ортогональными состояниями на основе одного события. С помощью широковещательных взаимодействий могут быть представлены цепные реакции, т.е. одно событие может вызывать другие.

На рис. 2.17 если происходит событие e, то первый из трех параллельных процессов переходит из состояния A в состояние B, и происходит событие f. Это вызывает переход второго процесса из состояния C в состояние D. в результате чего происходит g. Событие g вызывает переход третьего процесса из состояния E в состояние F. Таким образом, представлена цепная реакция длины 2.

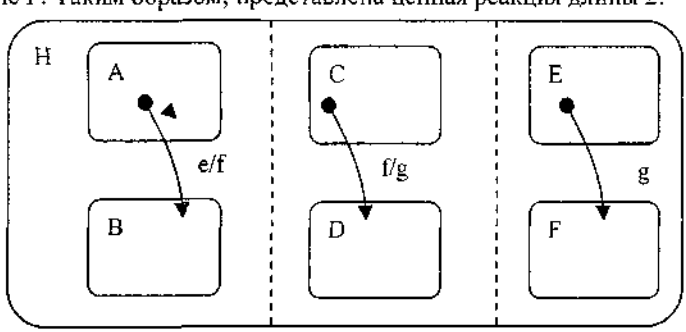


Рис. 2.17. Пример диаграммы состояний с цепной реакцией

Рассмотрим систему, представленную диаграммой с четырьмя состояниями (рис. 2.18).

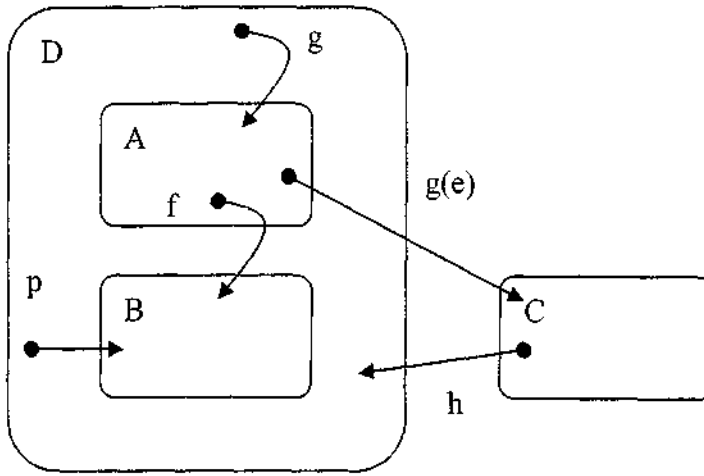


Рис. 2.18. Пример диаграммы состояний

Состояние D содержит в себе состояния A и B. Состояние D входит в состояние A, когда происходит событие g, или в состояние B, если происходит событие p. В D можно войти переходом из состояния C, когда происходит событие h. В состояние B входим из состояния A при наступлении события f. В состояние C система попадает из состояния A, если происходит событие g и при этом выполнено условие e. *При этом первое наступление события g приводит к* переходу в A, а следующее - к переходу в C.

В общем случае одно событие может вызывать множественные переходы в параллельных подпроцессах.

2.2.12. Унифицированный язык моделирования (UML)

Большая часть представленного ниже материала взята из электронного учебника [10]; использованные для иллюстраций примеры проектов взяты из испытательной версии SmartDraw6, доступной по адресу www.smartdraw.com. Полезная информация по унифицированному языку моделирования (Unified modeling language - UML) приведена в [11-15].

Объектно-ориентированная система описаний возникла из работ Гради Буча (Grady Booch), Джеймса Рамбо (James Rumbaugh), Ивара Якобсона (Ivar Jacobson) Эти исследователи объединили свои технологии в одну стандартную, унифицированную модель. Сегодня UML принят организацией Object Management Group (OMG) как стандарт для моделирования объектно-ориентированных программ и систем.

Типы диаграмм UML

UML определяет девять типов диаграмм:

1. ***Диаграммы классов/пакетов*** (Class/package Diagram). Классовые диаграммы являются основой каждого объектно-ориентированного метода, включая и UML. Они описывают статическую структуру системы. Пакетные диаграммы - это подмножество классовых диаграмм, но разработчики обычно рассматривают их как отдельный тип. Пакетные диаграммы организуют элементы системы в связанные группы для минимизации связей между пакетами.
2. ***Объектные диаграммы*** (Object Diagram) описывают статическую структуру системы в некоторый момент времени. Они могут использоваться для проверки правильности классовых диаграмм.
3. ***Диаграммы вариантов использования*** (Use case Diagram) моделируют функциональность системы с помощью участников и используемых ими сервисов.
4. ***Диаграммы последовательности*** (Sequence Diagram) описывают взаимодействия между классами в терминах последовательности обмена сообщениями во времени.
5. ***Диаграммы взаимодействия*** (Collaboration Diagram) представляют взаимодействие между объектами как серии последовательностей сообщений и описывают как статическую структуру системы, так и ее динамическое поведение.
6. ***Диаграммы состояний*** (Statechart Diagram) описывают динамическое поведение системы в ответ на внешние воздействия. Они особенно полезны в моделировании объектов, состояния которых определяются наступлением определенных событий.
7. ***Диаграммы активности*** (Activity Diagram) иллюстрируют динамическую природу системы моделированием потока управления от одного действия к другому. Действие представляет операцию над некоторым классом в системе, которое приводит к изменению состояния системы. Обычно диаграммы активности используются для моделирования потоков работ или бизнес-процессов и внутренних операций.
8. ***Диаграммы компонент*** (Component Diagram) описывают организацию физических программных компонент, включая исходные коды, библиотеки времени выполнения, исполняемые файлы.
9. ***Диаграммы размещения*** (Deployment Diagram) описывают физические ресурсы системы, включая узлы, компоненты, соединения.

Классовые диаграммы

Классы представляют сущности с общими свойствами (рис. 2.19). ***Связи*** описывают отношения между классами.

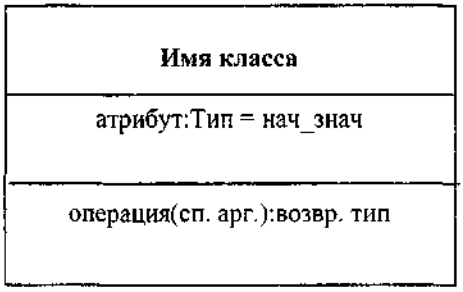


Рис. 2.19. Формат описания класса

Как видно из рис. 2.19 классы обозначаются прямоугольниками, разделенными горизонтально на части. В верхней части помещается имя класса, свойства - во второй части, методы (операции) - в третьей. Классы могут быть ***абстрактными*** - иметь только интерфейс без реализации, что полезно для платформенно-независимых приложений. Абстрактные классы показывают с помощью метки {abstract}

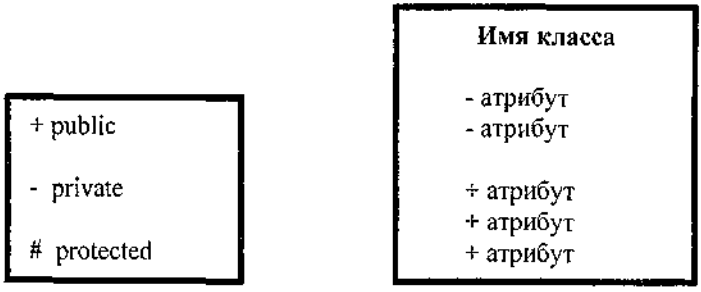


Рис. 2.20. Маркеры видимости полей и методов класса

Маркеры видимости (рис. 2.20) используются для указания того, кто может иметь доступ к внутренним объектам класса. Маркер видимости ***Private*** прячет информацию от доступа извне класса. Маркер видимости ***Public*** позволяет всем другим классам видеть помеченную им информацию. Маркер видимости ***Protected*** позволяет дочерним классам получать доступ к информации, унаследованной от родительского класса.

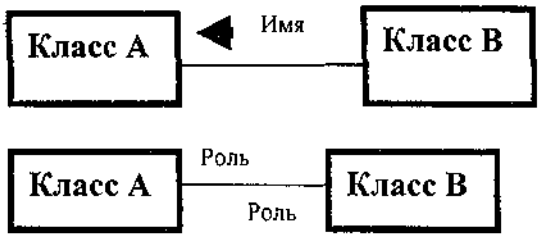


Рис. 2.21. Связи

Связи представляют статические отношения между классами (рис. 2.21). Имя связи помещают выше, над или под линией, соединяющей классы. Залитая стрелка используется для указания направления связи. Роли указываются у концов связи. Роли показывают, каким образом классы видят друг друга. Заметим, что обычно отмечают или имя связи, или роли.

Множественные отображения - кардинальные числа, помещаемые у концов связи. Эти символы показывают число экземпляров одного класса соединенных с одним экземпляром другого. Например, одна компания может иметь одного или многих работников, но каждый работник может работать только на одну компанию (рис. 2.22).

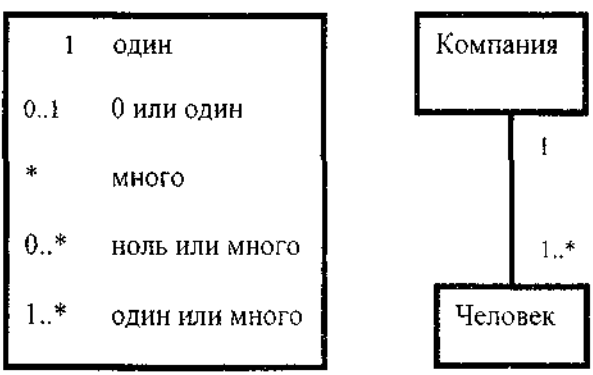


Рис. 2.22. Множественное отображение в связи компания-человек

Ограничения - это предположения, т.е. булевские предикаты, которые для системы всегда должны быть истинны. Например, сумма денежных активов филиалов банка должна быть не меньше суммы выданных кредитов, или аргумент операции извлечения квадратного корня должен быть больше или равен 0. Ограничения помещают внутри фигурных скобок "{}" (рис. 2.23).

Направление связи показывают стрелкой. Если Class1 показывает на Class2, это значит, что Class1 отвечает за выходную информацию связанных с ним объек-

тов класса Class2. Связи без стрелок трактуются как с неизвестным направлением или как двунаправленные. Если у конца связи нет имени, по умолчанию используется имя присоединенного класса.

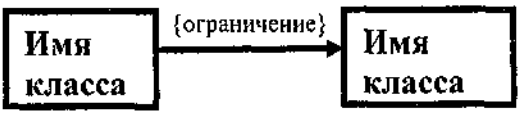


Рис. 2.23. Ограничение

Агрегация является отношением "часть-целое" (машина имеет двигатель, здание имеет этажи и т.д.).

Композиция есть специальный вид агрегации, который обозначает сильную связь между классом Class A (целым) и классом Class B (его частью). При композиции объект-часть может принадлежать только одному целому; предполагается, что части рождаются и умирают вместе с целым. Каждое удаление целого приводит к каскадному удалению частей.

Композицию показывают залитым ромбом (рис. 2.24). Пустой ромб используется для показа простого отношения агрегации, в котором класс "целое" играет более важную роль, чем класс "часть", но классы не зависят друг от друга. Конец с ромбом в случае как композиции, так и агрегации показывают на стороне класса "целое".

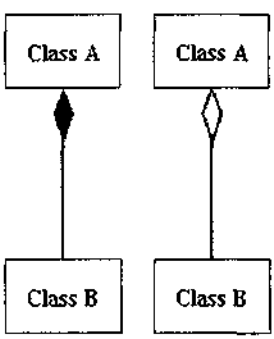


Рис. 2.24. Композиция и агрегация

Обобщение по другому называют наследованием или отношением "есть". Оно показывает отношение между двумя классами, где один класс есть специализация другого. Например, Хонда есть тип машины. Поэтому класс "Хонда" находится в отношении обобщения с классом "Машина".

Агрегация и наследование - разные по смыслу отношения. Если есть отношение агрегации, агрегат (целое) может получать доступ только к PUBLIC функциям класса "часть". С другой стороны, наследование позволяет наследующему классу получать доступ как к PUBLIC, так и к PROTECTED функциям суперклас-

са (класса-родителя). Обобщение показывают треугольником на стороне суперкласса, направленным к суперклассу (рис. 2.25).

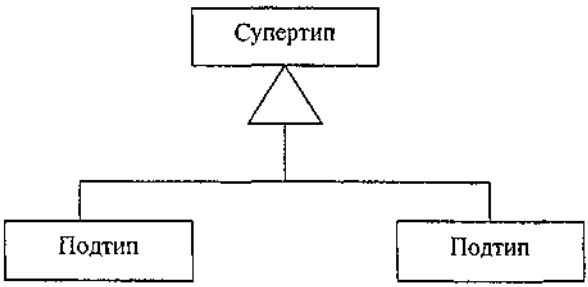


Рис. 2.25. Обобщение

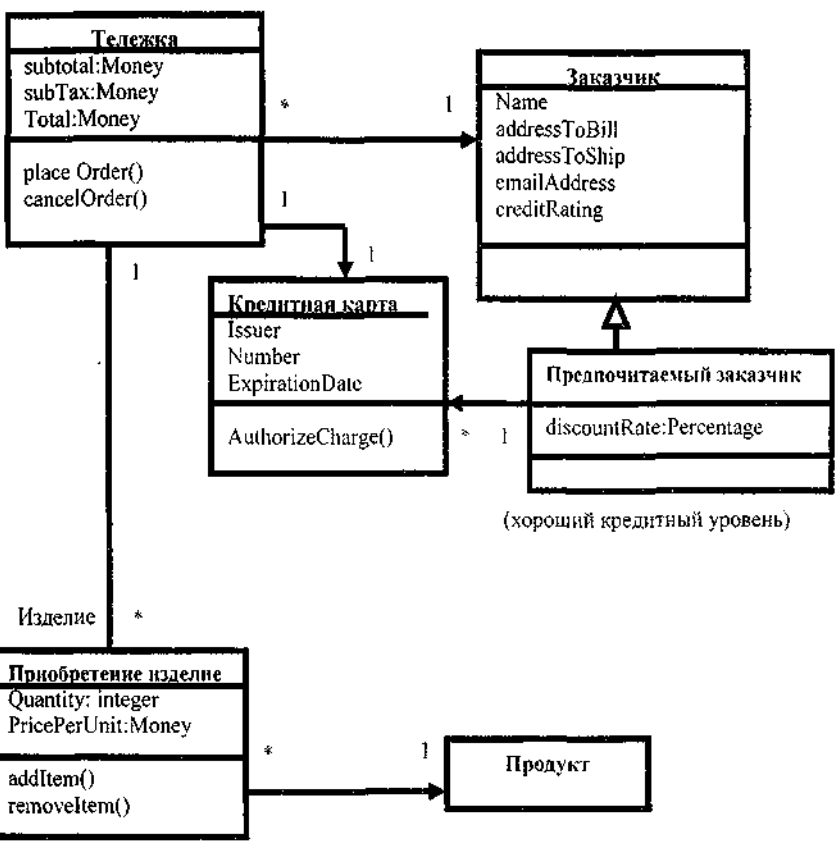


Рис. 2.26. Классовая диаграмма описания электронной тележки в магазине

На рис. 2.26 приведен пример классовой диаграммы для описания электронной тележки для товаров в магазине.

Пакетные диаграммы

Для представления пакетов используют прямоугольники с табличками. Имя пакета пишется внутри таблички или внутри прямоугольника. Аналогично классам, можно указывать свойства пакета (рис. 2.27).

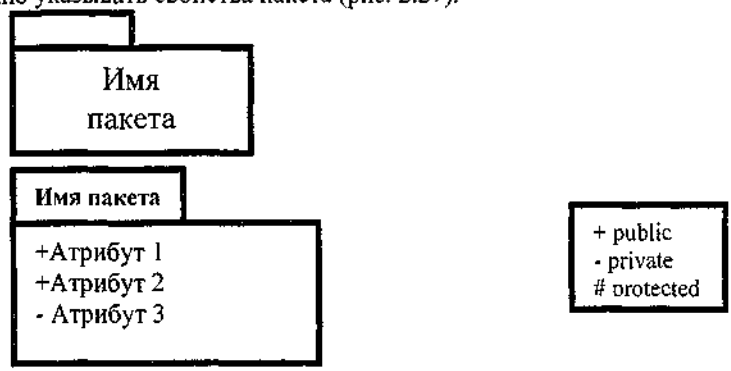


Рис. 2.27. Формат описания диаграмм пакетов

Маркеры видимости определяют, кто может получать доступ к информации внутри пакета. Действуют они так же, как в классовых диаграммах.

Зависимость определяет отношение, в котором изменения в одном пакете будут влиять на другой (рис. 2.28). Импорт есть тип зависимости, в котором один пакет получает доступ к содержимому второго. Зависимость между двумя пакетами существует, если существует зависимость между какими-либо двумя классами этих пакетов. Если P1->P2, это значит, что пакет P1 зависит от пакета P2.

Рис. 2.28. Зависимость в диаграмме пакетов

На рис. 2.29 показана диаграмма пакетов для обработки заказов.

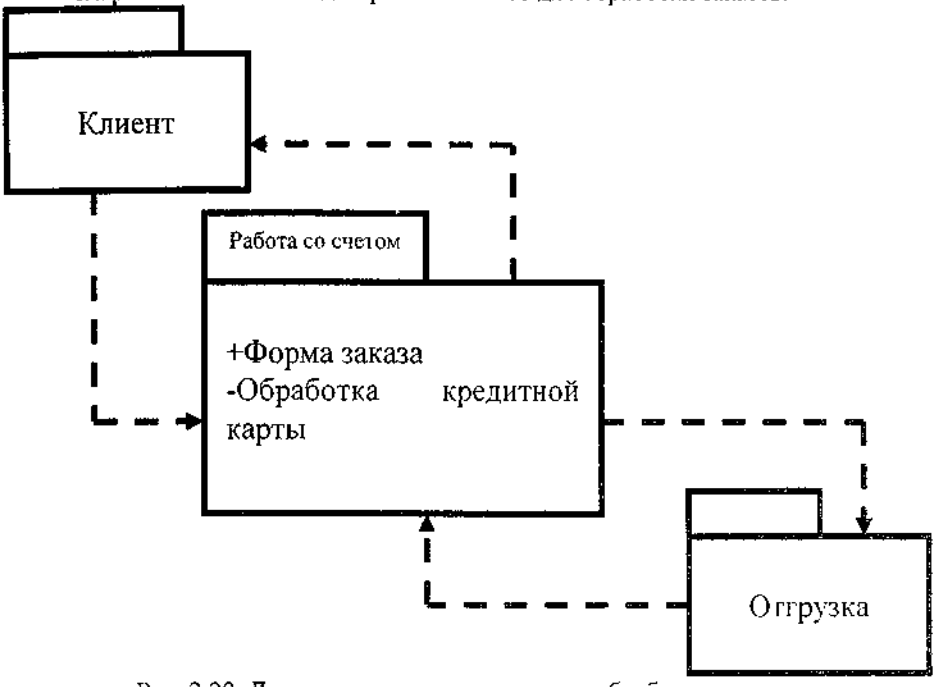


Рис. 2.29. Диаграмма пакетов для задачи обработки заказов

Объектные диаграммы UML

Объектные диаграммы также очень близки к классовым диаграммам. Как объект является частным случаем - экземпляром класса, так и объектная диаграмма может рассматриваться как частный случай классовой диаграммы. Объектные диаграммы описывают статическую структуру системы в конкретный момент времени и могут использоваться для проверки классовых диаграмм.

Рис. 2.30. Примеры описаний объектов

Каждый объект представляется прямоугольником, содержащим имя объекта и его класс, подчеркнутые и разделенные двоеточием (рис. 2.30).

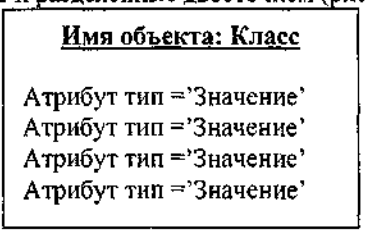


Рис. 2.31. Объект с атрибутами

Как и для классов, для объектов можно указывать список атрибутов в отдельном разделе (рис. 2.31). Однако в противоположность классам, атрибуты объектов должны иметь значения.

Объекты, которые управляют потоком действий, называются ***активными*** они показываются с более толстой границей (рис. 2.32).

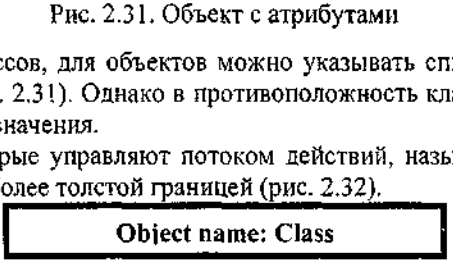


Рис. 2.32. Активный объект

Можно представить множество объектов одним символом, если атрибуты индивидуальных объектов не важны (рис. 2.33).

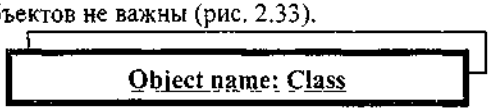


Рис. 2.33. Множество объектов

Связи являются экземплярами отношений (рис. 2.34). Можно рисовать связи с помощью линий, используемых в классовых диаграммах.

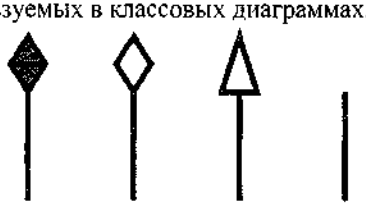


Рис. 2.34. Связи в объектных диаграммах

Объекты, играющие более одной роли, могут быть автосвязанными (рис 2.35). Например, если Марк, помощник администратора, также играет роль помощника по продажам, и эти две позиции связаны, то экземпляр этих двух классов будет автосвязан.

Рис. 2.35. Автосвязь

На рис. 2.36 приведена классовая диаграмма, а на рис. 2.37 соответствующая ей объектная диаграмма.

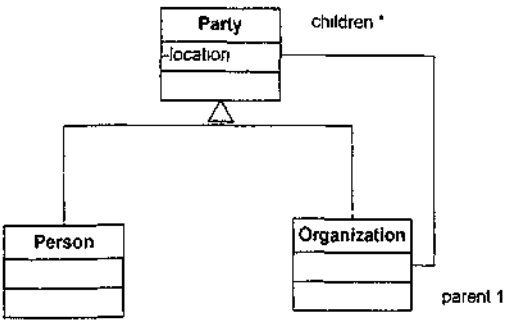


Рис. 2.36. Пример классовой диаграммы

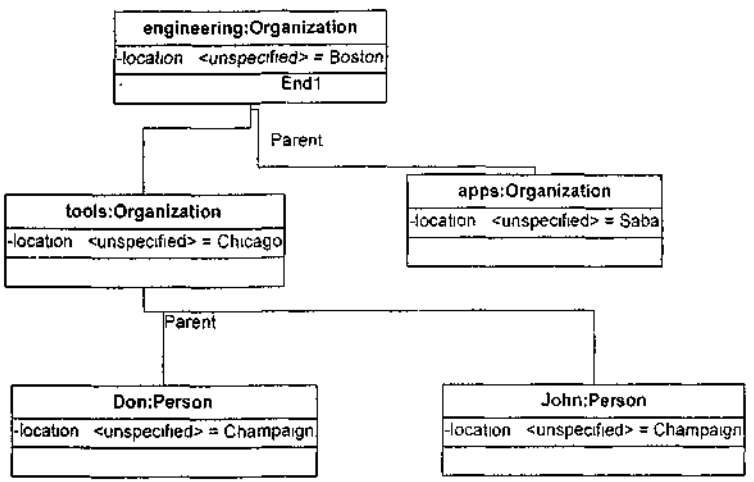


Рис. 2.37. Пример объектной диаграммы

Диаграммы использования

Диаграммы использования моделируют функциональность систем с помощью участников и сервисов.

Сервисы - это функции, предоставляемые системой пользователям.

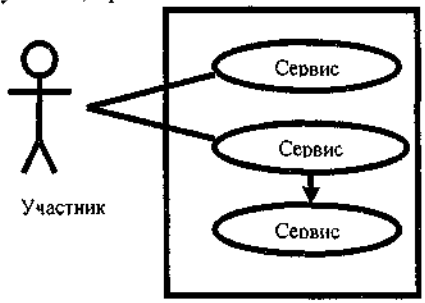


Рис. 2.38. Общий вид диаграммы использования

Система представляется прямоугольником, внутри которого указываются ее сервисы. Участники (actors) помещаются вне этого прямоугольника (рис. 2.38).

Каждый сервис обозначается овалом (рис. 2.38). Овал помечается глаголами, дающими представление о назначении системного сервиса.

Рис. 2.39. Система-участник другой системы

Участники - это пользователи системы (рис. 2.38). Когда система является участником-пользователем другой системы, она помечается стереотипом <<actor>> (рис. 2.39).

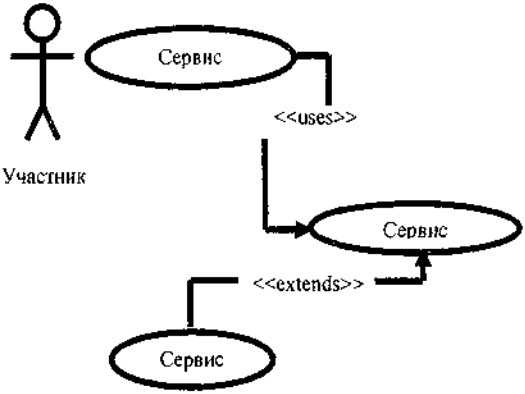


Рис. 2.40. Отношения

Отношения между участниками и сервисами показываются простой линией (рис. 2.40). Для отношений между сервисами используют стрелки, помеченные либо как "uses", либо как "extends". Отношение "использует" ("uses") указывает

что один сервис нуждается в услугах другого для выполнения своей задачи. Это необходимо для того, чтобы избежать повторов. Отношение "расширяет" ("extends") указывает альтернативную опцию при определенной ситуации и используется для описания отклонений от нормального поведения.

На рис. 2.41 приведен пример диаграммы использования для выполнения строительных работ.

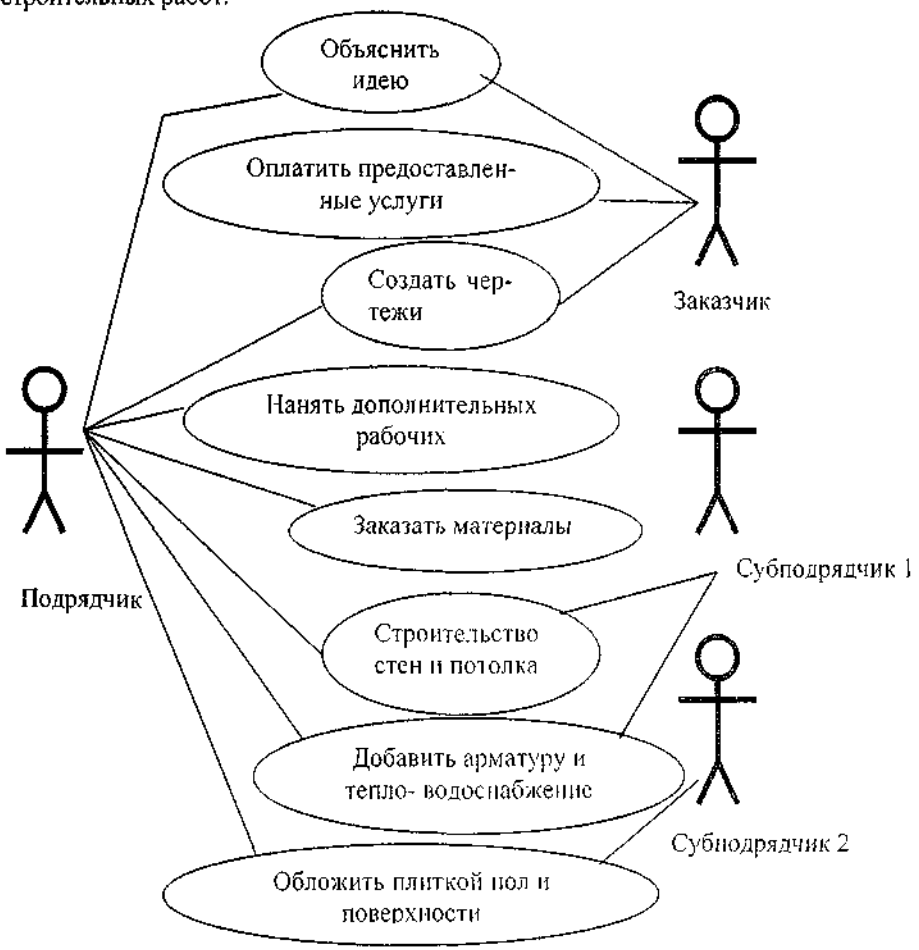


Рис. 2.41. Диаграмма использования для выполнения строительных работ

Диаграммы последовательностей

Диаграммы последовательностей описывают взаимодействия между классами в терминах обмена сообщениями во времени. Роли классов описывают пове-

дение объектов. Используются символы UML-объектов для показа ролей классов, но без перечисления списка атрибутов.

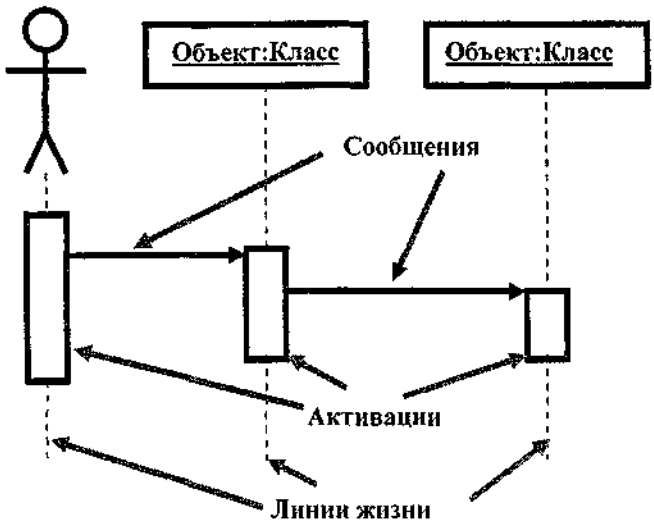
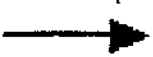


Рис. 2.42. Активации, сообщения и линии жизни

Прямоугольники - ***активации*** - (рис. 2.42) представляют время, необходимое объекту для выполнения задания.

Сообщения (рис. 2.42) обозначаются стрелками трех видов. Простая стрелка (flat)  - сообщение не ожидает ответа, управление переходит к получателю (отправитель завершается); синхронное (call)  - нормальный вызов процедуры - отправитель теряет управление до завершения обработки сообщения получателем, затем управление возвращается отправителю. асинхронное  - сообщение не требует ответа, но в отличие от случая flat отправитель остается активным и может посылать другие сообщения.

Линии жизни (рис. 2.42) - это вертикальные штриховые линии, которые показывают присутствие объекта во времени.

Объекты могут быть уничтожены с помощью стрелок, помеченных "< < destroy > >" , направленных в X (рис. 2.43).

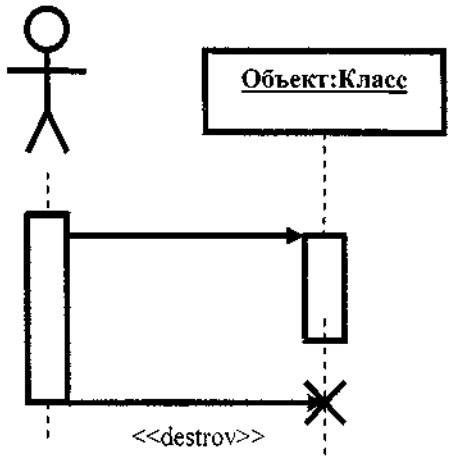


Рис. 2.43. Обозначение уничтожения объекта

Повторение, *или* ***цикл***, *в диаграмме последовательностей показывается* прямоугольником (рис. 2.44). Условие выхода из цикла помещается в его левом нижнем углу в квадратных скобках.

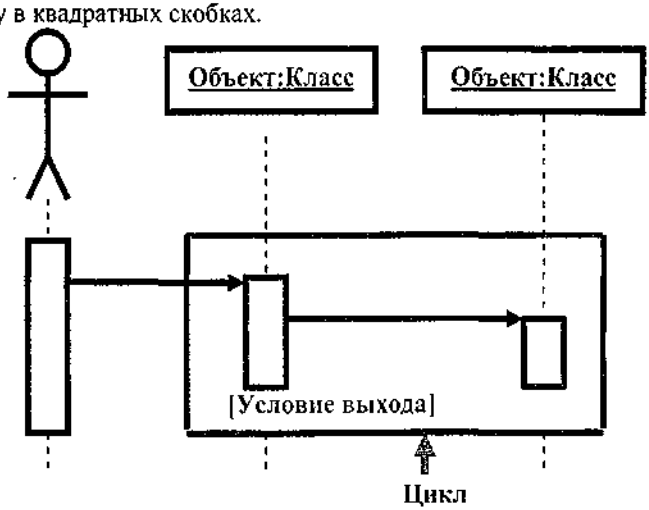


Рис. 2.44. Обозначение цикла

На рис. 2.45 приведен пример диаграммы последовательностей

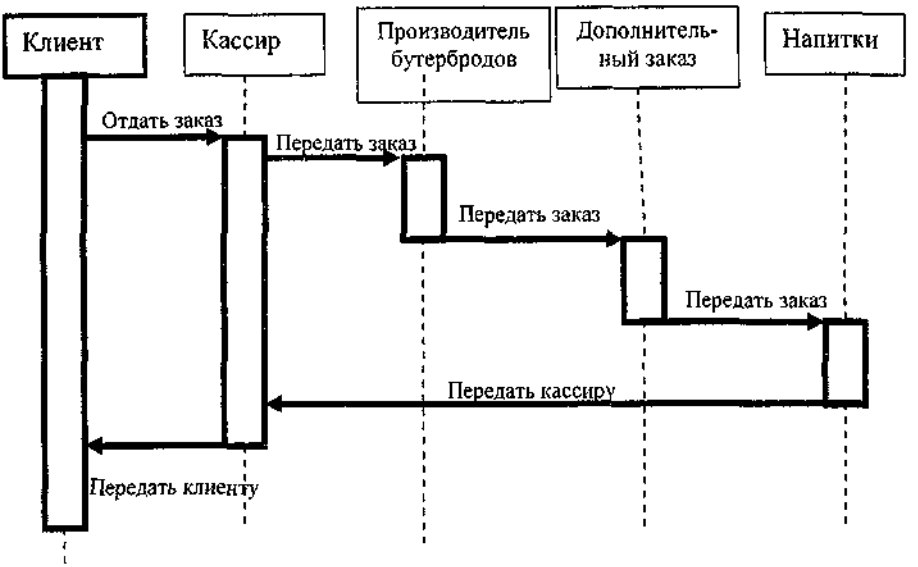


Рис. 2.45. Диаграмма последовательностей для процесса подготовки заказа

Диаграммы взаимодействия UML

Диаграммы взаимодействия показывают отношения между объектами в терминах последовательности сообщений. Диаграммы взаимодействия представляют комбинацию информации, взятой из диаграмм классов, последовательностей и использования и описывают как статические, так и динамические свойства системы.

Роли классов показывают поведение объектов. Для представления ролей используют символы объектов UML, но без перечисления атрибутов объектов

Роли связей описывают, как связь будет вести себя в конкретной ситуации. Для их представления используют простые линии, помеченные меткой: "<<стереотип>>";

В отличие от диаграмм последовательностей, диаграммы взаимодействия явным образом не показывают время, используя для этого нумерацию сообщений в процессе выполнения. Последовательность номеров может быть вложенной. Например, вложенные сообщения первого сообщения показываются как 1.1, 1.2, 1.3 и т.д. Условия для сообщения обычно помещают в скобках сразу за его номером. Для указания цикла используют символ * после номера сообщения.

На рис. 2.47 приведен пример диаграммы взаимодействия.

Рис. 2.47. Диаграмма взаимодействия сервера и браузера

Диаграммы состояний UML

Диаграммы состояний показывают поведение классов в ответ на внешние воздействия. Эти диаграммы моделируют динамику передачи потока управления от одного состояния к другому внутри системы.

Рис. 2.48. Состояние

Состояния показывают ситуации в течение жизни объекта. Изображаются как прямоугольники с закругленными углами (рис. 2.48).

Сплошная линия представляет ***переход*** между различными состояниями объекта (рис. 2.49). Она помечается событием, вызвавшим переход, и действием, которое должно быть выполнено в результате этого.

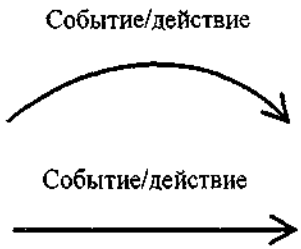


Рис. 2.49. Переходы

Залитый круг с выходящей из него стрелкой является начальным состоянием объекта (рис. 2.50). Стрелка, указывающая на залитый круг внутри другой окружности, - это конечное состояние объекта.

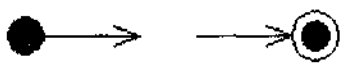


Рис. 2.50. Начальное и конечное состояния

Короткий толстый отрезок с двумя переходами в него показывает синхронизацию управления (рис. 2.51). Короткий толстый отрезок с двумя переходами из него представляет разветвление потока управления, что создает множественные состояния.

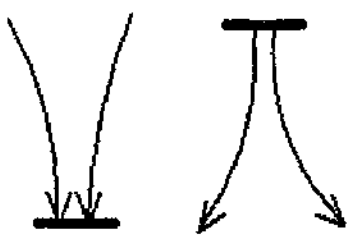


Рис. 2.51. Синхронизация и разветвление потока управления

На рис. 2.52 приведен пример диаграммы состояний для электросушилки

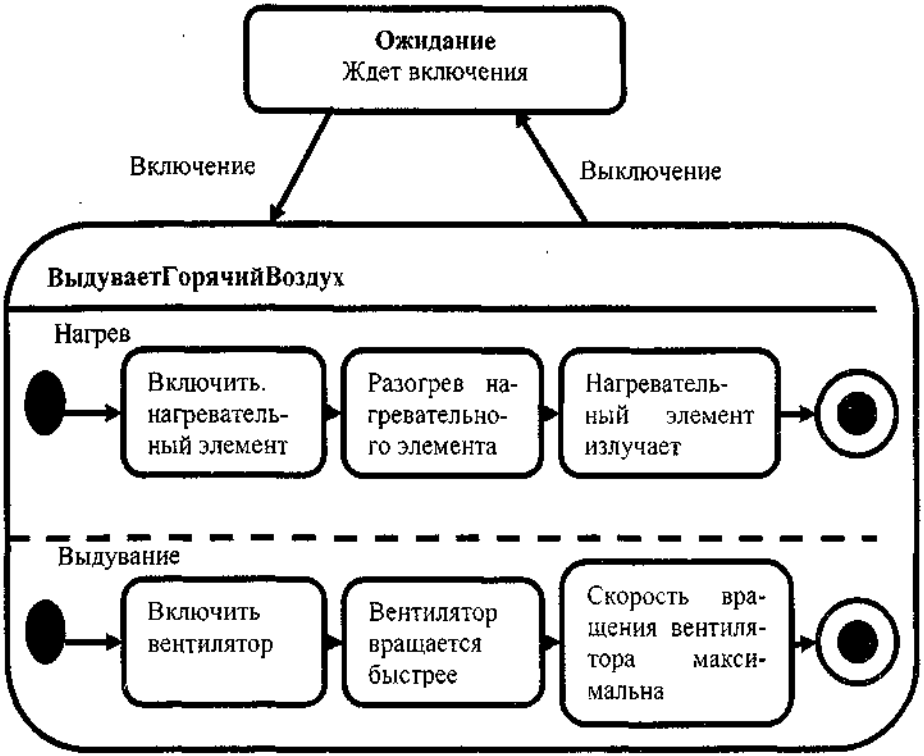


Рис. 2.52. Диаграмма состояний для электросушилки

Диаграммы активности UML

Диаграммы активности показывают динамическую природу системы моделированием потока управления от активности к активности.

Активность представляет операцию над некоторым классом в системе, приводящую к изменению состояния системы. Обычно диаграммы активности используются для моделирования потоков работ и бизнес-процессов. Это частный случай диаграмм состояний, поэтому здесь применяются те же соглашения.

Состояния активности отображают непрерываемые действия объектов (рис. 2.53).

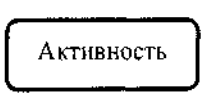


Рис. 2.53. Состояние активности

Поток действий показывается связями между состояниями активности (рис. 2.54).

Рис. 2.54. Поток действий

Поток объектов указывает на создание или модификацию объектов активностями (рис. 2.55). Стрелка от активности к объекту означает, что действие создает или модифицирует объект. Стрелка от объекта к активности означает, что активность использует этот объект.

Рис. 2.55. Поток объектов

Залитый кружок с выходящей стрелкой является ***начальным активным состоянием*** (рис. 2.56). Стрелка, указывающая на залитый кружок внутри другой окружности, представляет ***конечное активное состояние***.

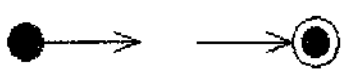


Рис. 2.56. Начальное и конечное состояния

Ромбом показывается решение с альтернативными выходами. Выходные ***альтернативы*** должны быть помечены условием срабатывания. Одна из альтернатив может помечаться с помощью "else" (рис. 2.57).

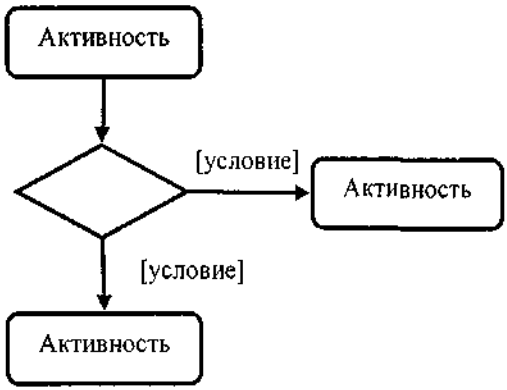


Рис. 2.57. Ветвление

Линия синхронизации (жирная) помогает описывать параллельные переходы. Синхронизация также называется разветвлением и соединением (forking and joining) (рис.2.58) - сначала управление разветвляется, затем объединяется.

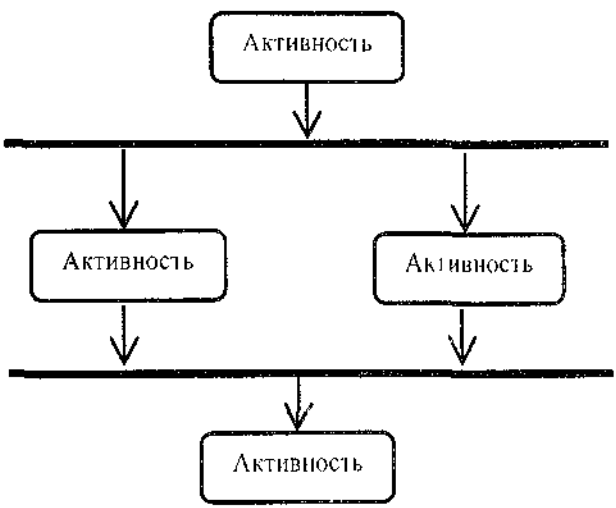


Рис. 2.58. Синхронизация

Дорожки объединяют связанные активности в одну колонку. Каждая дорожка помечается ответственным за нее классом (рис. 2.59).

На рис. 2.60 представлен пример обработки заказов.

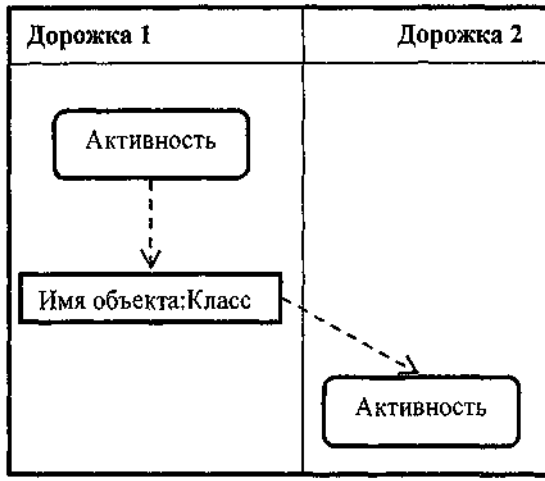


Рис. 2.59. Дорожки

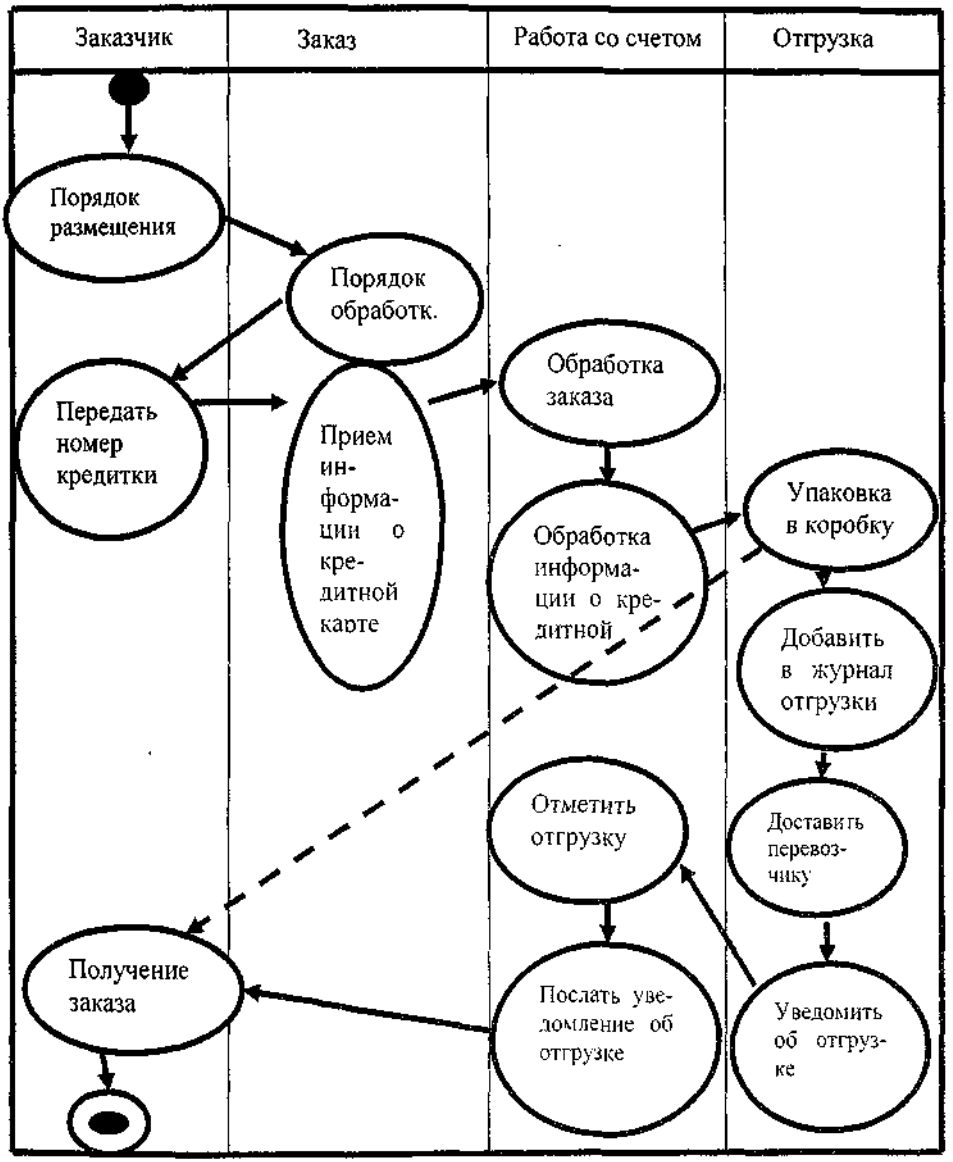


Рис. 2.60. Диаграмма активности обработки заказов

Диаграммы компонент UML

Диаграммы компонент описывают организацию физических компонент в системе.

Компонент - это физический строительный блок системы. Он представлен прямоугольником с табличками (рис. 2.61).

Интерфейс описывает группу операций используемых или создаваемых компонентами (рис. 2.61).

Рис. 2.61. Графическое представление компонента (слева) и интерфейса (справа)

Зависимости отмечаются штриховыми линиями (рис. 2.62). Они показывают, как изменения в одних компонентах могут повлиять на необходимость изменения других. Учитываются также зависимости по связям и компиляции.

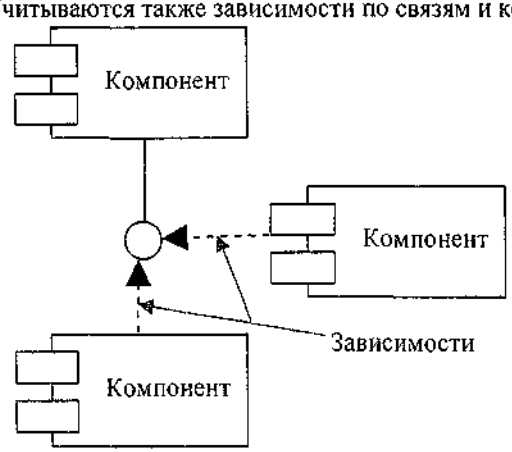


Рис. 2.62. Зависимости

Диаграммы размещения UML

Диаграммы размещения показывают физические ресурсы системы, включая узлы, компоненты, связи.

Узел (рис. 2.63) - это физический ресурс, который исполняет коды компонент.

Связь (рис. 2.63) показывает физическое соединение компонент, например, Ethernet, с их помощью представляют пути коммуникаций, по которым система будет передавать информацию.

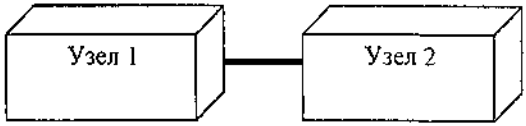


Рис. 2.63. Два узла и связи между ними

Компоненты помещаются внутри узлов там, где они должны находиться (рис. 2.64).

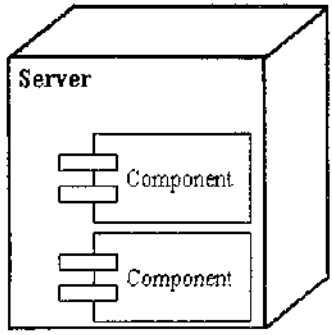


Рис. 2.64. Размещение компонентов Component в узле Server

На рис. 2.66 приведена диаграмма размещения для корпоративной сети.

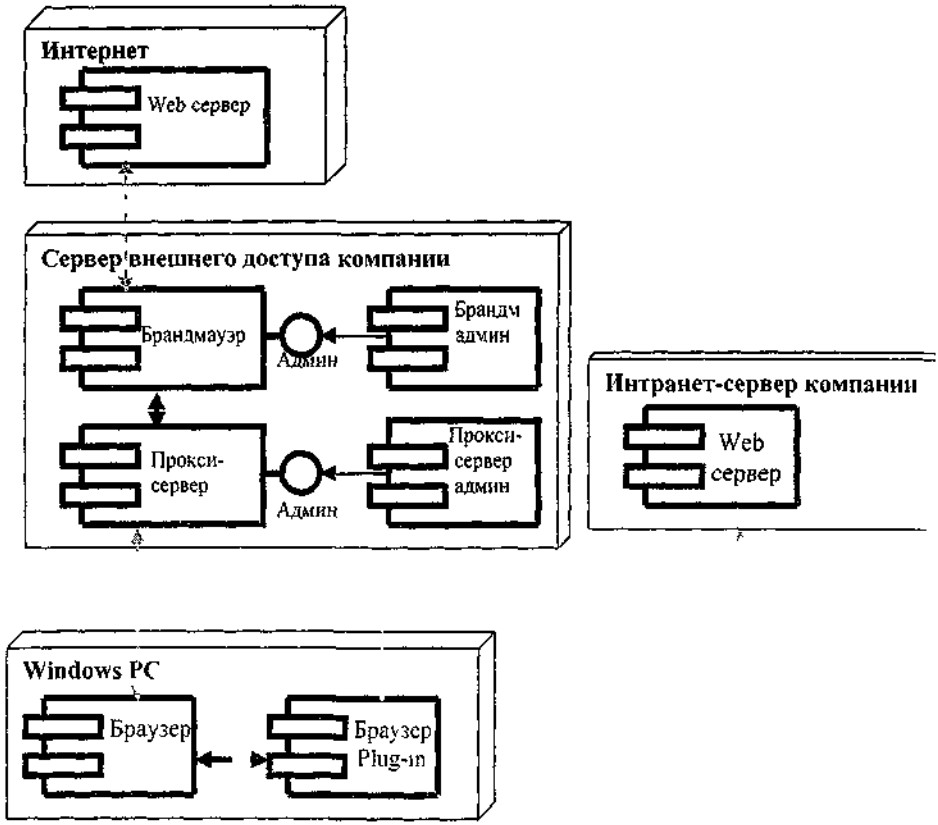


Рис 2 65 Диаграмма размещения для корпоративной сети

2.2.13. ROOM-диаграммы

ROOM [16-19] (real-time object-oriented modeling) - метод объектно ориентированного моделирования для СРВ Участник, или программная машина центральное понятие в методе ROOM ROOM-диаграммы показывают как структуру, так и поведение участника, и могут использоваться совместно с UML

С помощью структурных ROOM-диаграмм описывают участников и интерфейсы между ними (рис 2 66)

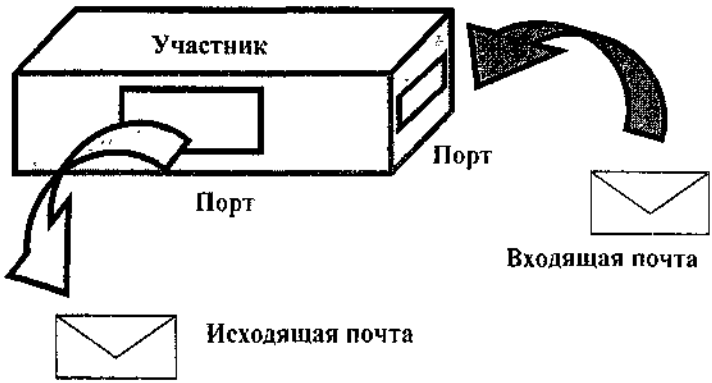


Рис. 2.66. Пример структурной ROOM-диаграммы

Участник - это активная компонента программной системы. Участники взаимодействуют с внешней средой через порты. Динамический участник порождается и уничтожается содержащим его участником.

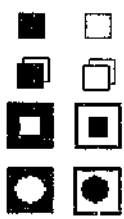


Рис. 2.67. Условные обозначения портов разных типов

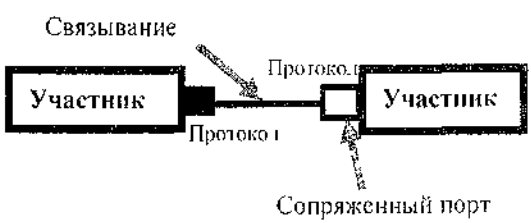


Рис. 2.68. Сопряженные порт, протокол, связывание

Порты (рис. 2.67) обеспечивают интерфейс между участниками с помощью протоколов, которые определяют информацию, которой они будут обмениваться. Порты бывают следующих типов:

- Реле-порт - разделяет интерфейс между внутренним классом и содержащим его классом.
- Сопряженный порт (рис. 2.68) - работает по тому же протоколу, что и напарник, но в инверсном режиме (входы являются выходами и наоборот). Обычно показывают в инверсных цветах.
- Внешний концевой порт (рис. 2.69) - взаимодействует с машиной состояний участника, или его поведением.
- Внутренний концевой порт (рис. 2.69) - соединяет компоненту участника с поведением охватывающего участника. Показывается так же, как и внешний концевой порт, но помещается внутри границы контейнера, а не на ней.

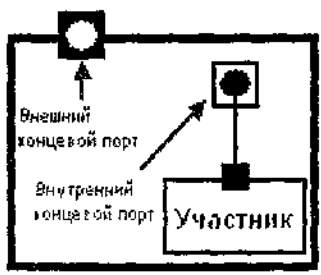


Рис. 2.69. Внешний и внутренний концевые порты

Протоколы определяют, какими сообщениями обмениваются участники (рис. 2.68).

Связывание существует между участниками, разделяющими протоколы (рис. 2.68).

ROOM-диаграммы поведения включают ROOM-карты и карты последовательностей сообщений.

ROOM-карты

ROOM-карты подобны диаграммам состояний. В ROOM событие происходит, когда сообщение послано через интерфейс участника. Компонента поведения участника может быть в различных состояниях. Она или ожидает новое сообщение, или занята обработкой некоторого произошедшего уже события. Переходы в ROOM-карте возбуждаются по прибытию сообщений через порты, соединенные с компонентой поведения. Начальный переход делается при создании нового поведения, т.е. при создании нового участника. Начальное состояние показано как "I" в кружке и является *псевдосостоянием*, так как участник не может в нем находиться.

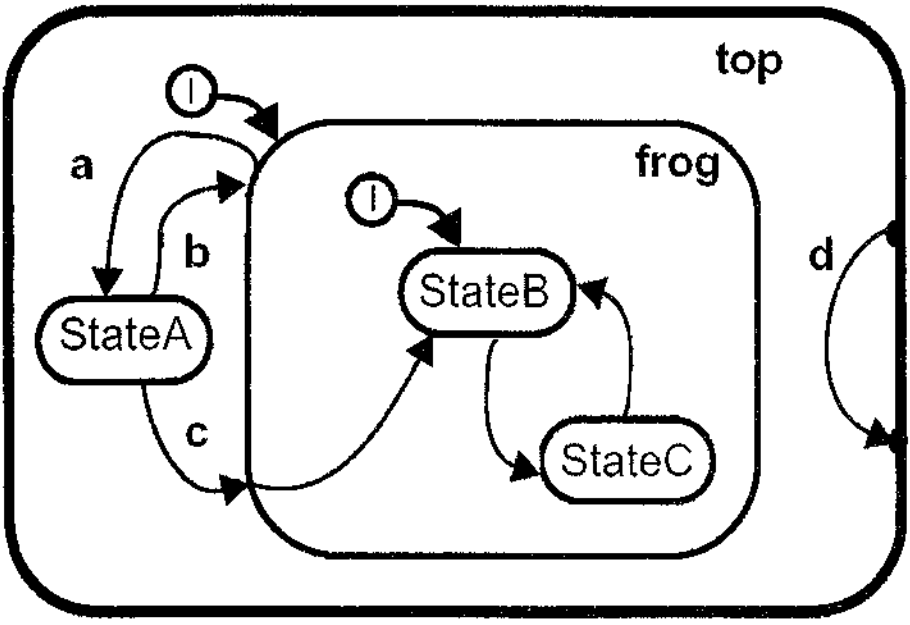


Рис. 2.70. Пример ROOM-карты

На рис. 2.70 переход 'a' делается, когда соответствующее событие происходит независимо от состояния, в котором находится участник frog. Переход 'b' возвращает в историческое состояние, т.е. в состояние, которое было при последней активности участника frog. Переход 'd' происходит из любого текущего состояния, это групповой переход верхнего уровня, который может быть удобен для представления обработки исключений. Этот переход заканчивается на границе состояния верхнего уровня. Когда возможные действия будут выполнены, поведение возвращается в историческое состояние, т.е. в то, что было перед наступлением события.

Карты последовательностей сообщений

В таблице ниже приведено описание протокола "Гость". На рис. 2.71 показана последовательность обмена сообщениями по данному протоколу между участниками.

Протокол Гость

Входные сигналы	Класс данных	Выходные сигналы	Класс данных
ТукТук	NULL	КтоТам	NULL
ЭтоЯ	NULL	Заходи	NULL

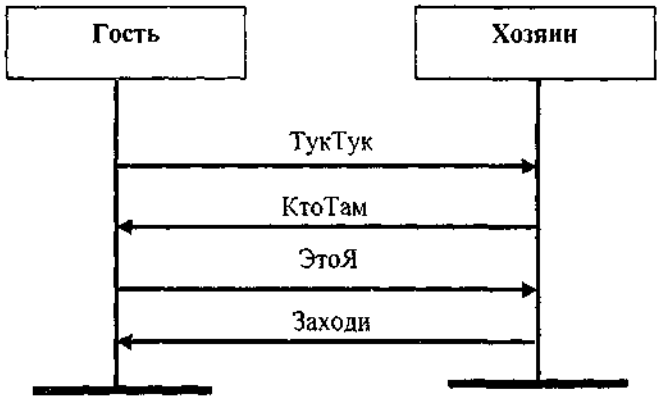


Рис. 2.71. Карта последовательности сообщений по протоколу "Гость"

Описание охраняемого помещения с помощью ROOM-диаграмм

Рассмотрим описание системы, приведенной на рис. 2.72 и служащей для разграничения доступа в охраняемую комнату.

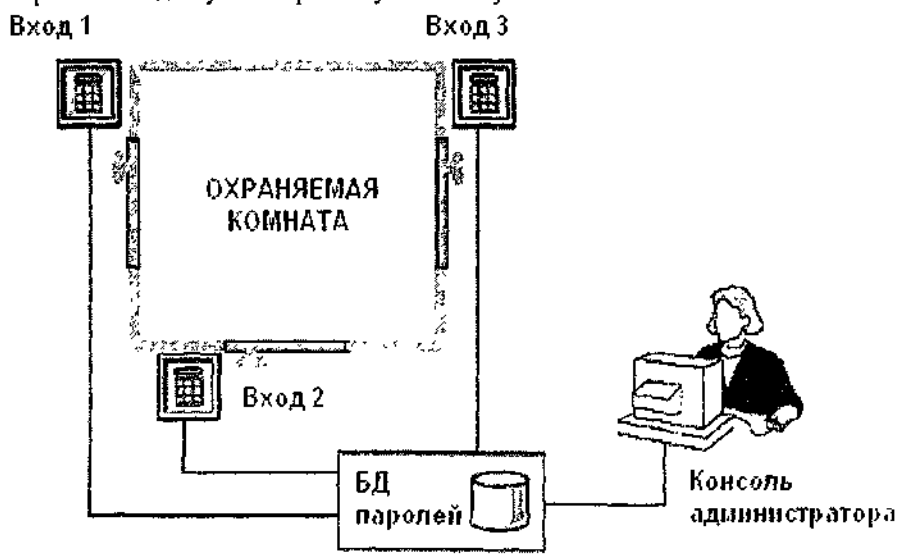


Рис. 2.72. Система ограничения доступа в охраняемую комнату

Необходимая аппаратура может быть представлена, как показано рис. 2.73.

Сценарий использования может быть таким:

1. Пользователь вводит идентификатор на цифровой клавиатуре.

2. Пользователь вводит персональный пароль, соответствующий идентификатору на цифровой клавиатуре.
3. Система проверяет корректность и соответствие друг другу идентификатора и пароля в базе данных.
4. Для правильных сочетаний идентификатора и пароля дверь открывается на 3 секунды, в течение которых пользователь может войти.

На рис. 2.74 приведена карта последовательности сообщений.

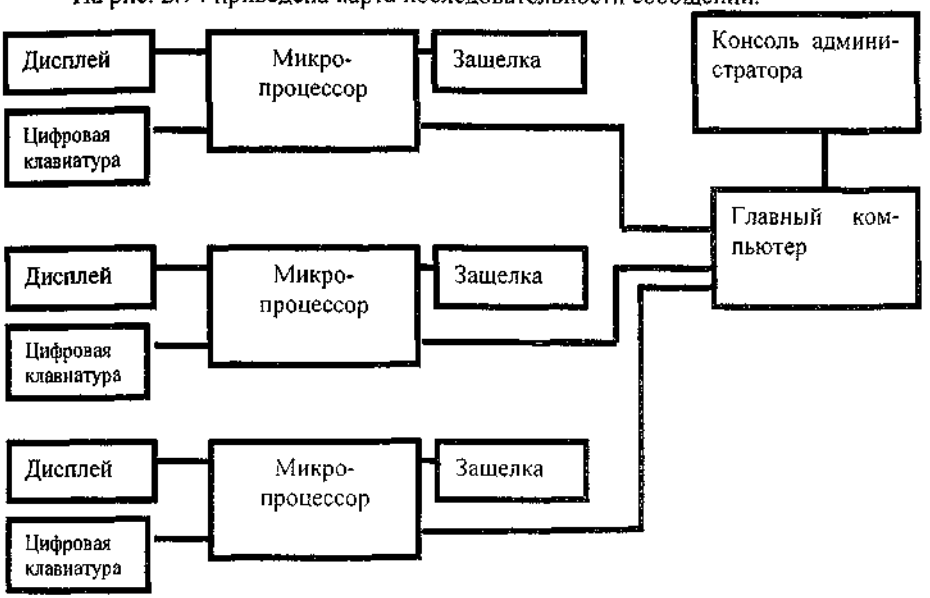


Рис. 2.73. Схема необходимой аппаратуры

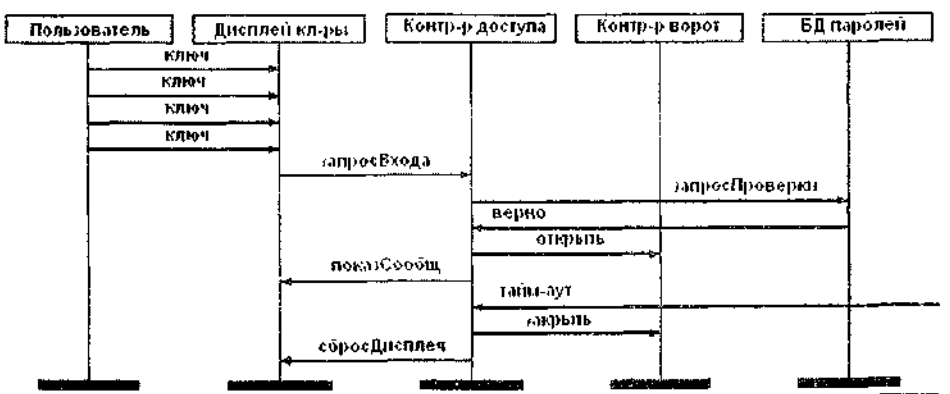


Рис. 2.74. Карта последовательности сообщений

Структуру системы можно получить из аппаратного представления, как показано на рис. 2.75.

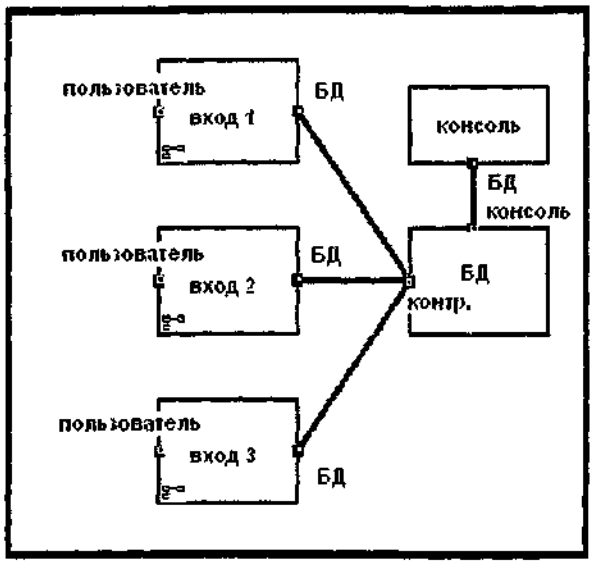


Рис. 2.75. Укрупненная структура системы

Каждый вход на рис. 2.75 может быть представлен, как показано на рис 2.76.

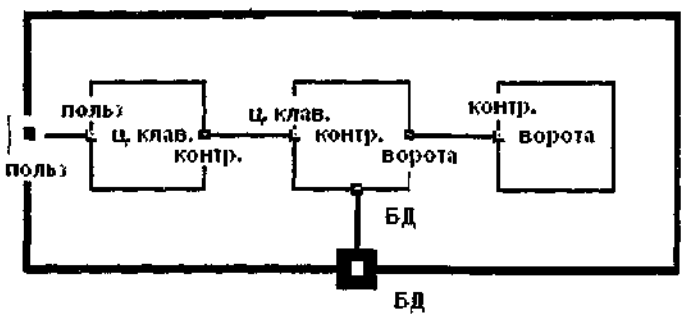


Рис. 2.76. Схема каждого входа

Из карты последовательности сообщений выводится поведение, приведенное на рис. 2.77.

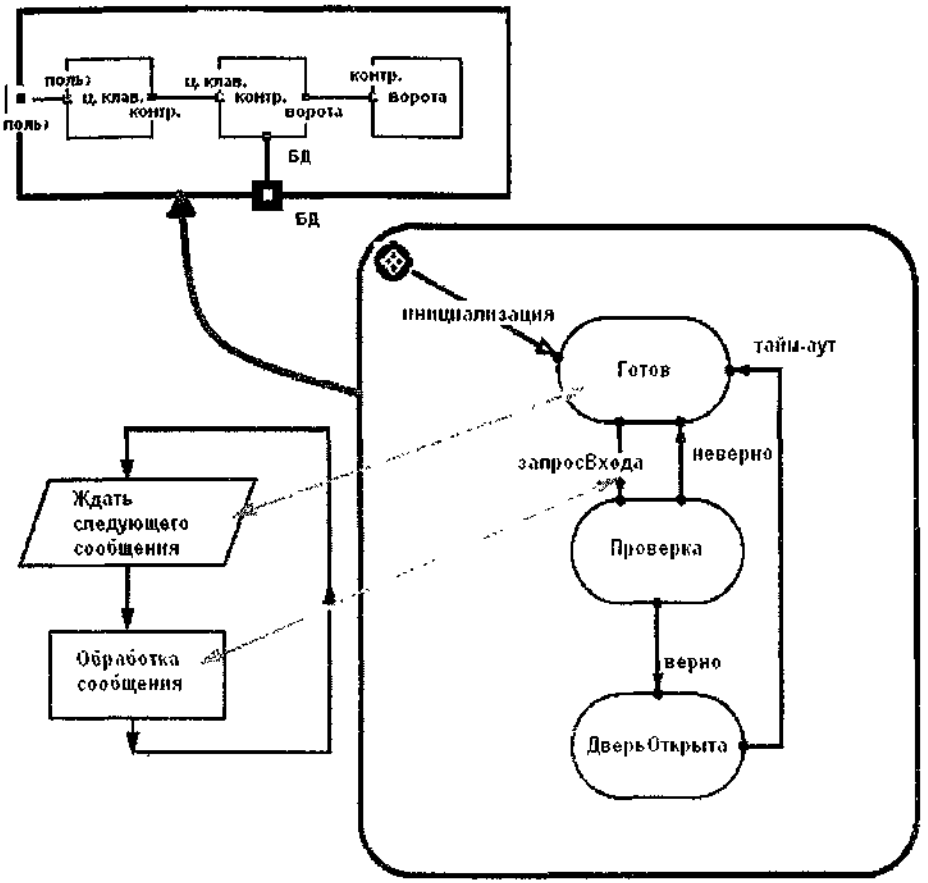


Рис. 2.77. Поведение системы

Итак, подход ROOM подобен UML, но вводит явным образом порты и протоколы для взаимодействия. Концевые порты получают сигналы из внешнего мира и должны иметь очереди, так как система обрабатывает каждый следующий сигнал до завершения без прерываний, что соответствует модели исполнения - исполнять-до-завершения, а в это время могут прийти новые сигналы, которые будут накапливаться в очереди. Состояния ROOM не имеют параллельных (AND) подсостояний, параллельные процессы моделируются различными участниками.

Контрольные вопросы и задания

1. В чем состоит идея модели водопада для проектирования сложного ПО?
2. Что такое спецификация на естественных языках? Перечислите ее достоинства и недостатки.

3. Каковы основные особенности математических спецификаций?
4. Что такое блок-схемы и структурные схемы?
5. Какие диаграммы могут применяться в картах Джексона?
6. Что такое псевдокод? Каковы его отличительные особенности?
7. Рассмотрим СРВ, которая должна обрабатывать бесконечные потоки целых чисел из 3 внешних источников, поступающие через порты, нумерованные. соответственно, 1, 2, 3. В системе имеется три кооперативно работающих процесса (кооперативная многозадачность). Каждый входной поток данных обрабатывается соответствующим процессом - вычисляются текущие среднее, минимальное и максимальное значения: $CurrAvg_n = \frac{\sum_{i=1}^{n} x_i}{n}, CurrMin_n = \min_{i=1,n} x_i, CurrMax_n = \max_{i=1,n} x_i$, где n - общее число чисел x_i, поступивших к текущему моменту времени.

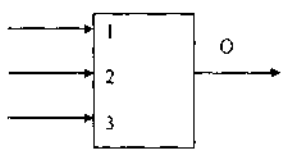


Рис. 2.78. Схема мультиплексора

Эти значения должны быть пересчитаны для каждого нового входного сигнала и выданы через порт O вместе с номером соответствующего входного порта как запись вида: CurrAvg, CurrMin, CurrMax, Port#. Например, если входной поток для порта 1 был (1,2,3,4,..), то отвечающий за этот порт процесс 1 выдаст следующую последовательность записей ((1,1,1,1),(1.5,1,2,1),(2,1,3,1),(2.5,1,4,1),..). Предполагаем, что процесс. данные для которого отсутствуют в текущий момент, передает управление следующему процессу. Напишите псевдокод (C- или Pascal-подобный), реализующий такую систему.

8. Что такое конечные автоматы? В чем отличие конечного автомата Мура от конечного автомата Мили?
9. Опишите однопроцессорную систему со стратегией Round-Robin (см. 3.2.3 использования процессорного времени для очереди из пяти задач как автомат Мили с входным алфавитом (ЗадачаПришла, ЗадачаКончилась), и выходным алфавитом (ЗадачаПринята, ЗадачаНеПринята). Значения сигналов: ЗадачаПришла - новая задача появилась в системе; ЗадачаКончилась - задача покидает систему по завершении решения, ЗадачаПринята - задача поставлена в оче-

редь задач на выполнение, ЗадачаНеПринята - задача не принята в очередь (нет свободного места).

10. Как можно описать систему с помощью диаграмм потоков данных?
11. Сети Петри: основная идея?
12. Что такое представление Варнье-Орра?
13. В чем преимущества диаграмм состояний?
14. Диаграммы каких типов включает в себя UML? Для чего каждый из типов диаграмм предназначен?
15. Для чего используются объектные, классовые и пакетные диаграммы?
16. Какие связи могут применяться в классовых диаграммах?
17. Что из себя представляют диаграммы использования?
18. Что такое диаграммы последовательностей?
19. Охарактеризуйте диаграммы взаимодействия UML.
20. Для чего нужны диаграммы состояний и диаграммы активности?
21. Что можно описать посредством диаграмм компонент и размещения?
22. С помощью классовых диаграмм UML опишите множество следующих классов и отношения между ними: система управления автомобилем, система управления тормозами, система управления потреблением топлива, система рулевого управления, автомобиль с системой управления автомобилем, Volkswagen Polo с системой управления автомобилем, Mercedes-300 с системой управления автомобилем, рулевое колесо, тормозная система, колесо, система газа (ускорения), система питания, двигатель. Покажите только имена классов (не показывайте атрибуты и методы).
23. Какие бывают ROOM-диаграммы?
24. Что такое порты и какими они могут быть?
25. Для чего нужны ROOM-карты, карты последовательности сообщений?
26. Рассмотрим систему управления автомобилем, состоящую из системы управления газом (ускорения), системы управления тормозами, системы управления потреблением топлива. Система управления ускорения имеет методы: больше_газа(), меньше_газа(); система управления тормозами имеет метод тормозить(); система управления потребления топлива имеет методы: увеличить_потребление(), уменьшить_потребление(). Названия методов отражают их смысл. Нарисуйте структурную ROOM-диаграмму. Нарисуйте ROOM-диаграмму последовательности сообщений для двух взаимоисключающих ситуаций:
    1) нажата педаль газа; 2) нажата педаль тормоза.

Глава 3. Управление процессами

Вкратце можно сказать, что процесс - это выполняющаяся программа. Управление выполнением программ является одной из наиболее важных задач любой вычислительной системы и в особенности СРВ. Поэтому в этой главе мы уделим внимание основным подходам к управлению процессами.

3.1. Основные сведения о процессах

Рассмотрим кратко исполнение программ применительно к процессорам семейства x86.

Программы обычно готовятся на языке программирования высокого уровня (исходные файлы .c, .pas и т.п.), компилируются в объектные коды (машинные коды с неразрешенными внешними ссылками, .obj файлы). Потом они компонуются с другими объектными и библиотечными файлами, в результате чего получаются исполняемые файлы (.exe, .com), которые содержат двоичный образ памяти процессора и практически без изменений загружаются в оперативную память, каждая ячейка которой - обычно, байт - имеет взаимнооднозначное связанное с ней число - ее адрес.

Для запуска программы ее ***точка входа*** - адрес ячейки, в которой находится первая команда программы, - загружается в ***программный счетчик*** (Program Counter - PC) - регистр, который представлен в процессорах Intel регистровой парой CS:IP (CS - Code Segment, IP - Instruction Pointer). Каждый из этих регистров 16-разрядный и содержит 2-байтное слово, поэтому с его помощью можно адресовать пространство 4Гб. Адреса являются 4-байтными структурами, имеющими 2 части: сегмент - 2 старших байта; смещение - 2 младших байта. Когда адрес загружается в PC, сегмент (смещение) загружается в CS (IP). Итак, адреса в процессорах Intel представляются двумя числами и для реального режима работы процессора вычисление исполняемого адреса требуемого байта (его номера) производится так:

Исполняемый адрес = 16**Сегмент* + *Смещение*.

В защищенном режиме соответствующие вычисления, отображающие адрес, представленный сегментом и смещением в одно число, производятся более сложным образом. Такое двухсловное представление адреса полезно с точки зрения возможности использования относительной адресации при компиляции программ, а когда программа загружается, она может быть настроена на соответствующий участок памяти установкой сегментных регистров практически без ее модификации.

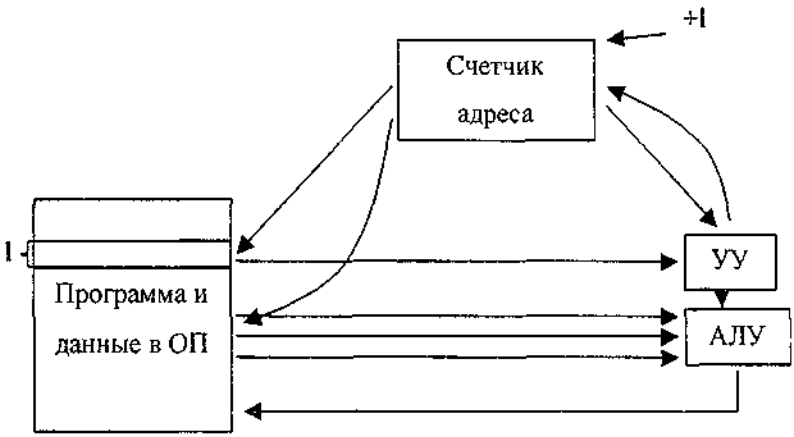


Рис. 3.1. Схема работы центрального процессора

Центральный процессор (ЦП, Central Processing Unit - CPU), имеющий устройство управления (УУ, Control Unit - CU) и арифметико-логическое устройство (АЛУ, Arithmetic-Logic Unit - ALU), работает достаточно просто, повторяя бесконечно одни и те же действия (рис. 3.1):

- чтение байта, адресуемого PC;
- декодирование прочитанного и определение того, что за инструкция должна выполняться (ее структура (формат), определение числа операндов и их местоположение (память, регистры, непосредственно в команде), способ адресации (прямая, косвенная)), модификация PC;
- чтение операндов во внутренние регистры, которые используются АЛУ как входы совместно с командой, определяющей операцию над операндами;
- получение результата во внутренних регистрах и их сохранение в месте, определенном исполняемой командой.

Этим завершается исполнение текущей команды, но программа является множеством команд, которые должны быть исполнены одна за другой. Поэтому после завершения текущей команды центральный процессор снова повторяет описанные выше действия, читая инструкцию с позиции, адресуемой PC, но сейчас он уже показывает на следующую команду, так как после декодирования предыдущей команды и определения ее длины, PC был наращен на длину исполняемой команды. Поэтому по завершении текущей команды в PC хранится адрес первого байта следующей команды.

Так может быть реализован алгоритм линейной структуры. Однако в большинстве программ имеются ветвления и циклы, нарушающие естественный ход выполнения программы. Ветвления и циклы выполняются с помощью команд пе-

реходов (JMP, GOTO). Они указывают, что следующая исполняемая команда находится по адресу, заданному ее операндом. Такие команды вызывают модификацию PC. Для запуска любой программы достаточно загрузить адрес ее точки входа в регистр PC, и тогда следующей исполняемой командой будет первая команда программы.

Таким образом, выполнение программ производится с помощью PC, который порождает один поток исполняемых команд: одна команда за другой последовательно, эти команды обрабатывают один поток данных, представленных их операндами. Поэтому по классификации Флинна (Flynn) такие компьютеры, работающие согласно модели программного управления фон Неймана (von Neumann) относятся к системам с одним потоком команд и одним потоком данных (ОКОД Single Instruction stream-Single Data stream - SISD). Поток команд обычно называют ***нитью управления***. Следовательно, в обычных компьютерах нить управления реализована с помощью регистра PC.

Процессы - это выполняющиеся программы, имеющие связанные с ними потоки команд или нити управления.

Переключение между множественными процессами может быть организовано загрузкой в PC адреса соответствующего кода. Естественно, что не может быть процесса без исполняемого кода, заданного программой; каждый процесс ассоциирован в каждый момент времени с некоторой программой. Но может быть много процессов, ассоциированных с одной и той же программой. Такие процессы исполняют один и тот же код, но находятся в разной фазе выполнения. Кроме того, может быть много программ, ассоциированных с одним процессом.

Так, можно запустить некоторую программу из текущего процесса с помощью, например, функции execl:

```
printf("\nEnter name of program");
gets(prog);
execl(prog, /*список аргументов*/,NULL);
```

Имя программного файла вместе с путем к нему вводится в переменную prog, и функция execl загружает программу из указанного файла вместо текущей используя в качестве параметров командной строки аргументы из списка аргументов, завершающегося символом NULL.

Таким образом, программа и процесс находятся в отношении многие-ко многим. Следовательно, нельзя идентифицировать процесс выполняемой им программой. Поэтому процессы представляются в операционной системе своими ***блоками управления процессом*** (БУП, Process Control Blocks - PCB).

3.2. Организация многозадачности на однопроцессорных системах

Рассмотрим основные подходы к организации выполнения множества процессов на одном процессоре.

3.2.1. Последовательное исполнение процессов (однозадачный режим)

На рис. 3.2 показана схема последовательного исполнения двух процессов, которое состоит из двух фаз: активное исполнение на центральном процессоре и работа с устройствами ввода-вывода (УВВ). Процессы один за другим подходят к процессору и обрабатываются от начала до конца. Таким образом, даже если процесс по какой-либо причине не занимает центральный процессор или устройства ввода-вывода, и соответствующий блок простаивает, другой процесс не может воспользоваться свободным оборудованием.

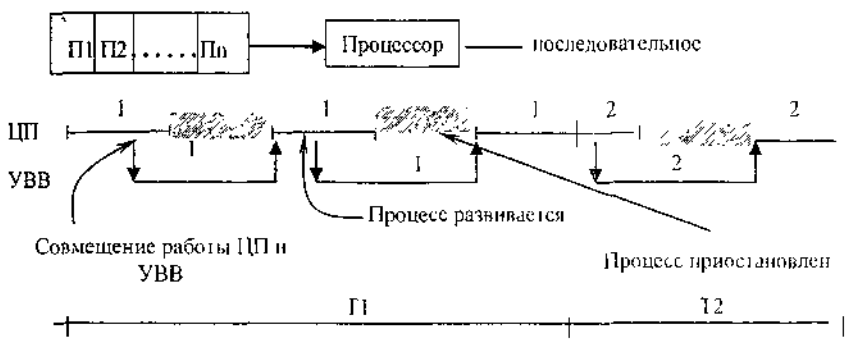


Рис. 3.2 Последовательное исполнение процессов

3.2.2. Кооперативная многозадачность

При кооперативной многозадачности простаивающий в ожидании внешнего события (например, завершения ввода/вывода) процессор может выполнять программу другого процесса (рис 3.3). Это возможно в случае, когда ввод/вывод управляется специальным процессором ввода/вывода, при этом общее время исполнения двух процессов Tpar может быть меньше времени последовательного исполнения процессов, что видно на соответствующих временных диаграммах.

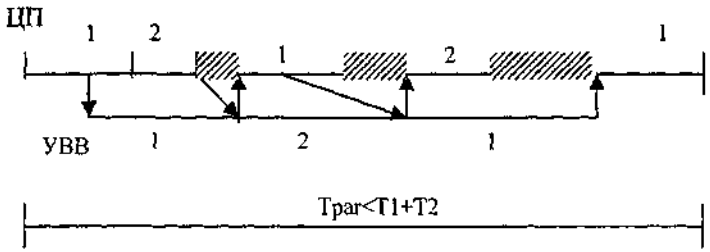


Рис. 3.3. Кооперативная многозадачность

3.2.3. Вытесняющая многозадачность

В случае вытесняющей многозадачности, которую также называют квантованием времени или круговым обслуживанием, каждый процесс может находиться в одном из следующих состояний (рис. 3.4):

- исполнение - процесс получил процессорный ресурс и развивается;
- блокировка - развитие процесса невозможно, так как он ожидает наступления некоторого внешнего события;
- готовность - процесс находится в очереди на получение очередного кванта процессорного времени - процессорного ресурса.

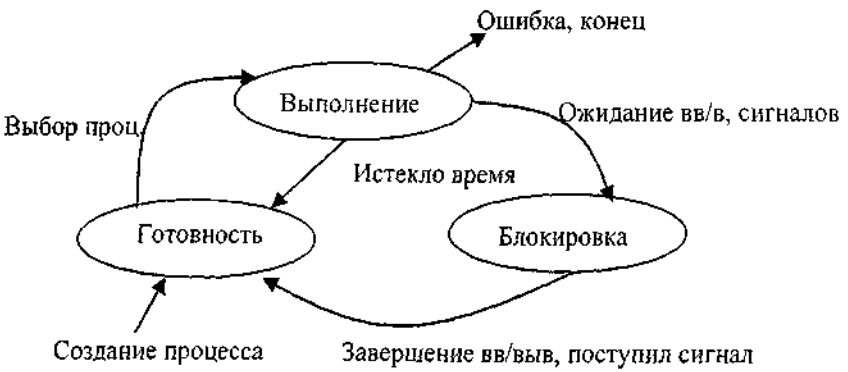


Рис. 3.4. Состояния процесса при вытесняющей многозадачности

Процессы обслуживаются согласно дисциплине FIFO (First-In-First-Out первый пришел - первый ушел, т.е. очередь). Не обслуженный полностью в выделенный квант времени процесс возвращается в конец очереди для ожидания следующей порции времени (рис. 3.5). Данный алгоритм получил название ***кругового циклического обслуживания*** (RR - round robin).

Рис. 3.5. Очередь FIFO к процессору при вытесняющей многозадачности

Для представления процессов в очереди используются блоки управления процессами. Рассмотрим более подробно организацию очереди (рис. 3.6).

Блок управления процессом имеет следующую примерную структуру: идентификатор процесса, состояние, другая информация о процессе:

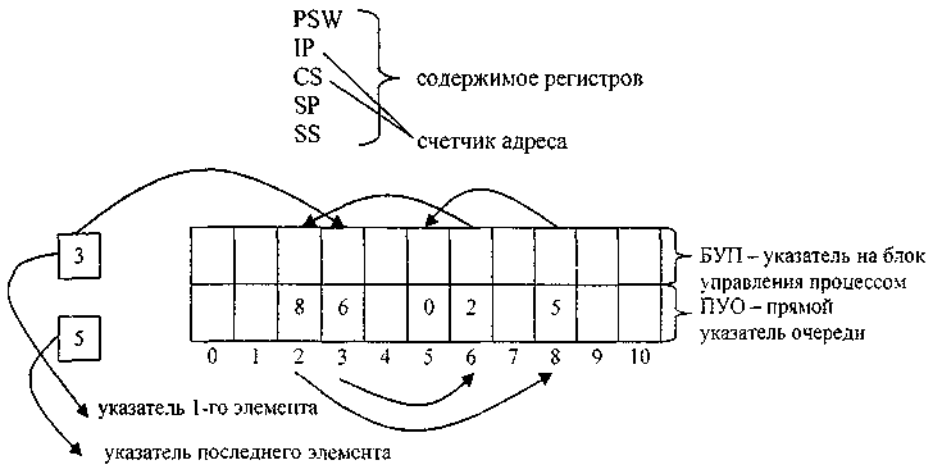


Рис. 3.6. Организация очереди равноприоритетных процессов (значение поля БУП не показано)

Когда центральный процессор переключается с процесса на процесс, нужно:

- Запомнить состояние исполняемого процесса в блоке управления процессом.
- Выбрать следующий процесс для исполнения с помощью указателя на первый элемент.
- Удалить из очереди переводимый в состояние выполнения процесс, т.е. установить указатель на первый элемент (УПЭ) равным полю ПУО удаленного из очереди процесса. Например, для ситуации на рис. 3.6 процесс №3 удаляется из очереди, так как на него указывает регистр УПЭ, УПЭ полагается равным 6, взятым из поля ПУО третьего процесса. После такой модификации очередь будет содержать 4 процесса.
- Если прерванный процесс должен вернуться в очередь готовых процессов, т.е. он не завершен еще, то его надо поместить в хвост очереди: ПУО старого последнего элемента, на который ссылается указатель на последний элемент (в нашем случае это процесс №5) положить номеру

прерванного процесса (например, №4), ПУО прерванного процесса (в нашем случае - № 4) положить равным 0, после чего загрузить в указатель на последний элемент (УПоЭ) номер прерванного процесса (в нашем случае - № 4).

- Модифицировать состояние только что выбранного для исполнения процесса на "Выполняется" в его блоке управления процессом, восстановить состояние процессора для его выполнения и передать ему управление.

В приведенной выше схеме процессы идентифицируются их положением в используемой для организации очереди таблице. Предполагается, что все они одинакового приоритета. Рассмотрим случай различных приоритетов.

Для СРВ разные процессы могут иметь различную важность, и это может быть отражено приоритетами: более важные процессы должны обслуживаться вначале, при отсутствии высокоприоритетных процессов могут выполняться низкоприоритетные процессы. Для каждого приоритета имеется своя очередь на выполнение (рис. 3.7).

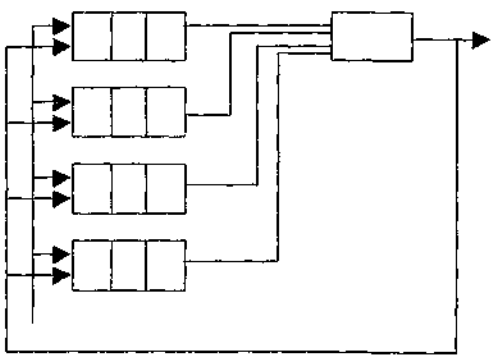


Рис. 3.7. Разноприоритетные очереди к процессору

Различают несколько дисциплин извлечения процессов из очередей и постановки обратно:

1. Процессы поступают в очередь соответствующего приоритета и после получения кванта процессорного времени возвращаются в хвост той же очереди.
2. Вновь прибывающие процессы помещаются в одну и ту же очередь, но получении кванта процессорного времени возвращаются в хвост очереди более низкого приоритета. Это влечет разделение задач по требуемому времени обслуживания - более короткие задачи будут обслужены в первую очередь. Такой подход соответствует максимизации пропускной способности при использовании алгоритма SPT, но при неизвест

ном заранее времени выполнения задач, и получил название ***алгоритм FB*** (foreground-background).

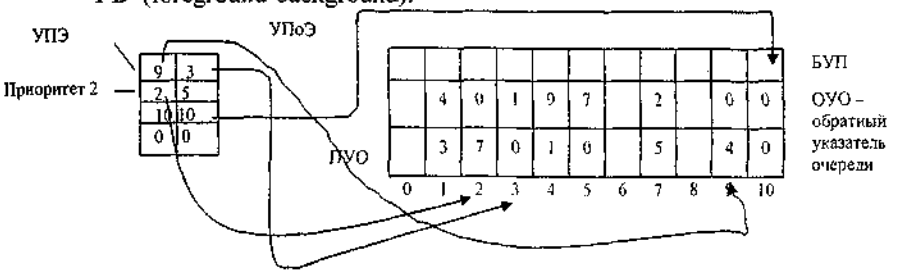


Рис. 3.8. Организация очереди разноприоритетных процессов

Для реализации очередей нескольких приоритетов используется тот же подход, что и для одной очереди, но уже с несколькими парами регистров указателя на последний элемент и указателя на первый элемент (рис. 3.8). Каждая такая пара определяет очередь соответствующего приоритета, все эти очереди разделяют одну и ту же таблицу процессов. На рис. 3.8 очередь 1-го приоритета состоит из процессов: 9, 4, 1, 3; очередь 2-го приоритета - из процессов: 2, 7, 5 и очередь 3-го приоритета включает только один 10-й процесс. Для более эффективной работы с очередями в этой таблице процессы связаны в оба направления с помощью полей ПУО и ОУО (обратный указатель очереди). Это может быть полезно при выполнении операций модификации приоритетов. Так, например, если мы хотим изменить приоритет i-го процесса и сделать его равным x, то процесс должен быть удален из своего списка и подсоединен к концу x-го списка. Удаление может быть произведено так (рис. 3.9):

```
ПУО(ОУО(i))=ПУО(i)
ОУО(ПУО(i))=ОУО(i)
```

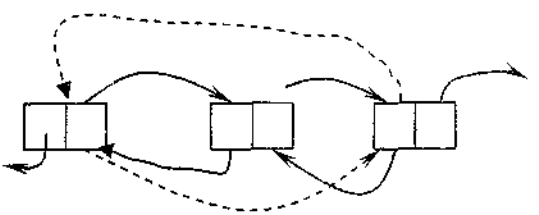


Рис. 3.9. Удаление процесса из очереди

3.2.4. Особенности смены состояний процессов

На рис. 3.10 приведена детальная схема перехода процесса из одного состояния в другое.

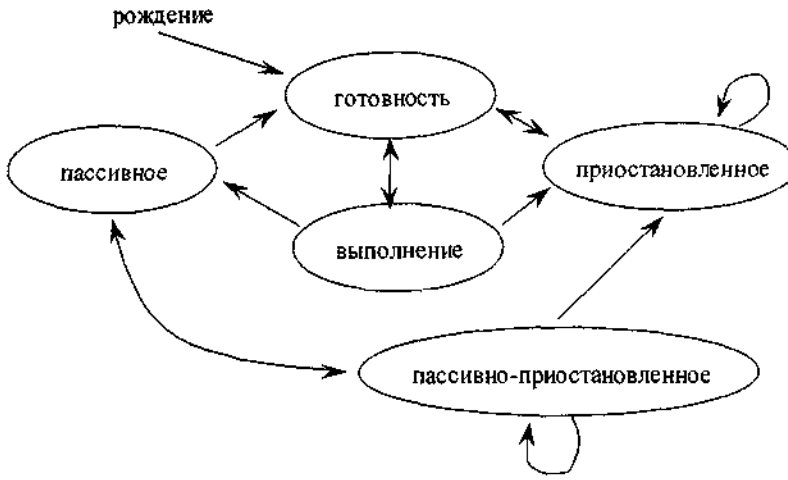


Рис. 3.10. Подробное описание состояний процесса

Процесс переходит из состояния выполнения в пассивное, когда он делае системный вызов SLEEP или когда ожидает завершение запроса на ввод/вывод Запрос SLEEP имеет параметр тайм-аута. Процесс переходит из пассивного со стояния в активное - состояние готовности, или из пассивно-приостановленно состояния в приостановленное состояние, когда истекает тайм-аут или завершает ся операция ввода/вывода. Переход из состояния выполнения в приостановлено состояние происходит, когда процесс подвешивает (приостанавливает) себя сис темной командой SUSPEND TASK, при этом глубина подвешивания полагается равной 1. Процесс переводится из состояния активности (готовности) в подвешен ное состояние, если некоторый другой процесс делает системный вызов SUSPEN TASK с ID первого процесса в качестве параметра вызова; глубина подвешивани также полагается равной 1. Если такой системный вызов сделан для процесса, уж подвешенного или пассивно-подвешенного, глубина подвешивания последнег наращивается на 1. Глубина подвешивания уменьшается на 1, если некоторы процесс делает системный вызов RESUME TASK для приостановленного или пас сивно-приостановленного процесса. Приостановленный (пассивно приостановленный) процесс переходит в состояние готовности (пассивное, соот ветственно), если глубина подвешивания становится равной 0.

3.3. Прерывания и их обработка

Как отмечалось ранее, для организации кругового циклического обслужи вания с квантованием времени необходимо прерывать выполнение текущего про цесса по истечении выделенного ему кванта времени. Вычислительная систем имеет специальное устройство - таймер, сигналы которого используются для веде

ия системного времени, работы с дисками, а также при определении тайм-аутов. дея обработки сигналов таймера та же, что и для обработки сигналов других ешних устройств. Вопрос обработки прерываний важен не только для организа- ии вытесняющей многозадачности, но и для общения вычислительной системы с ешним миром, что имеет особое значение для СРВ, которые должны интенсивно ботать с внешними устройствами. Поэтому рассмотрим вкратце понятие преры- ния и его обработку применительно к процессорам x86.

Прерывания - это сигналы, приходящие от внешних устройств в заранее известные моменты времени. Они могут появиться в любой момент времени и истема должна быть готова отреагировать на них адекватно. Эти сигналы посту- ают на специальные входы процессора. Устройства могут быть разных типов, и ждое из них может требовать специфичной обработки. Таким образом, сигнал рерывания приходит вместе с его номером, зависящим от типа устройства, вы- вшего сигнал прерывания. Например, сигнал от таймера имеет номер 08h.

В процессорах Intel предполагается, что может быть 256 различных типов рерываний со своими уникальными номерами, лежащими в диапазоне от 0 до 5. Каждый тип прерывания должен быть обработан соответствующей ***процеду- й обработки прерывания***. Таким образом, для каждого из 256 возможных пре- ваний должна вызываться специфическая программа, и эти программы могут сать не только разработчики операционной системы, но и прикладные програм- исты. Более того, такие программы могут находиться в различных местах опера- вной памяти. Возникает вопрос: как процессор узнает местоположение процеду- обработки прерывания?

Адрес такого обработчика, называемый ***вектором прерывания***, должен ранiться в предопределенном месте, а именно, в так называемой таблице преры- ний, занимающей 1Кб, начиная с байта с адресом 0. Эта таблица содержит век- ры прерываний, являющиеся адресами соответствующих процедур обработки рерываний. Каждый такой вектор прерывания занимает 4 байта: 2 младших байта смещение адреса обработчика прерывания, 2 старших байта - сегмент этого адре- . Адрес вектора прерывания номер N может быть рассчитан по формуле 4 N: так, ктор прерывания для таймера хранится в байтах 32,33,34,35, причем байты 32,33 держат смещение адреса обработчика прерывания от таймера, а байты 34,35 ют сегментную часть этого адреса.

Таким образом, чтобы заставить процессор реагировать на сигналы внеш- их устройств, достаточно в соответствующих 4-х байтах вектора прерывания ука- ть адрес нужного обработчика. Но обычно обработка прерываний несколько ожнее. Связано это с тем, что, возможно, прежний обработчик делал какую-то жную для системы работу, и мы не должны нанести вред системе, просто заме- в его собственным обработчиком. Следовательно, необходимо сделать новую работку и сохранить старую. Также обычно по завершению программы требует- почистить память за собой и восстановить старый обработчик прерываний. Та-

ким образом, как правило, сохраняется старый вектор прерывания в памяти нашей программы, прерывание перехватывается заменой старого вектора прерывания на новый - адрес нашего обработчика прерывания. Первой операцией нового обработчика прерывания является вызов старого обработчика, который сделает некоторую важную для системы работу, о которой мы можем ничего и не знать. По завершении старого обработчика управление возвращается новому обработчику, и он выполняет свою работу.

Следует отметить, что когда приходит сигнал прерывания, после определения адреса обработчика прерывания, но до передачи управления ему, процессор кладет в стек 3 двухбайтных слова: регистр флагов (Flags register, Processor Status Word - PSW) и регистры CS, IP в указанной последовательности. Это делается для обеспечения возможности возврата управления из обработчика прерывания в прерванную программу, так как следующая команда, которая должна была в ней исполняться, адресуется программным счетчиком - регистровой парой CS:IP. Регистр флагов используется для анализа условий и необходим при организации ветвлений в программах. Таким образом, когда обработчик прерывания получает управление, он сохраняет в своей памяти (обычно, в стеке) все регистры (по крайней мере, используемые), выполняет необходимые действия, восстанавливает все ранее сохраненные регистры и передает управление прерванному приложению командой iret (Interruption RETurn). Эта команда берет из стека в обратном порядке значения IP, CS, PSW и загружает их в соответствующие регистры. После загрузки этих регистров прерванный процесс продолжает работу, как будто он и не прекращался.

Следует обратить внимание на то, что обработка прерывания начинается только после полного завершения команды процессора (не оператора языка высокого уровня!), выполняемой на момент поступления прерывания.

Кроме того, необходимо иметь в виду, что когда вызывается старый обработчик прерывания обычным программным вызовом (call), то в стеке сохранятся только CS:IP, но не PSW. При возврате же из старого обработчика прерывания будет использована команда iret, которая возьмет из стека 3 слова, вместо положенных туда двух, что приведет к порче стека и неправильной работе системы. Во избежание этого, перед вызовом старого обработчика надо положить в стек PSW. Часто в компиляторах имеются специальные директивы для описания процедур обработчиков прерываний, тогда и при обычном их вызове в стек автоматически будут положены необходимые 3 слова. Кроме того, подобные процедуры, как правило, самостоятельно сохраняют при входе и восстанавливают при выходе основные регистры процессора.

Выше были рассмотрены аппаратные прерывания, но те же самые обработчики могут быть запущены специальной командой процессора - int, что соответствует так называемому программному прерыванию. Эта возможность широко используется в программировании, в частности, для управления экраном использует-

ся прерывание 10h, для работы с дисками - 13h, для работы с клавиатурой - 16h и т.д. Все рассмотренное выше справедливо и для программных прерываний, но в этом случае сигнал прерывания исходит от программы, а не от аппаратуры.

При возникновении прерывания процессор запрещает прерывания установкой специального флага PSW, что делает невозможным остановку процедуры обработки прерывания другой аналогичной подпрограммой. В случае аппаратных прерываний, запрет происходит не только на уровне процессора, но и на уровне контроллера прерываний. Поэтому, чтобы разрешить программные прерывания требуется установить флаг регистра PSW, а для аппаратных прерываний необходимо еще и модифицировать настройки контроллера прерываний. Программные прерывания обычно разрешаются автоматически при возврате из процедур их обработки путем автоматического восстановления регистра флагов, который на момент вызова прерывания содержал установленный бит разрешения прерываний. В случае контроллера прерываний такого не происходит и аппаратные прерывания необходимо разрешать в процедуре обработки прерывания.

3.4. Семафорные операции

Пусть необходимо разрешить только одному из двух процессов исполнять критическую секцию кода, т.е. часть программы, в которой он работает с критическим ресурсом. Такой ресурс должен использоваться одновременно не более чем одним процессом. Для этого можно использовать код, приведенный на рис. 3.11 и базирующийся на применении семафорной переменной.

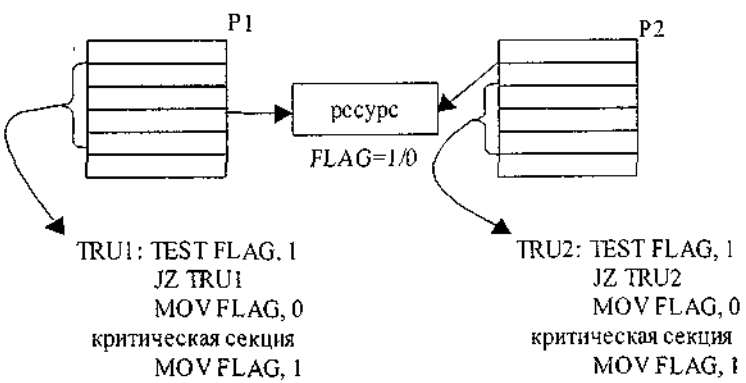


Рис. 3.11. Организация доступа к критической секции с помощью одного флага

Одновременный вход в критическую секцию возможен при следующем сценарии: команда TEST исполняется первым процессом и застает FLAG=1, а перед исполнением команды MOV FLAG, 0 управление передается второму процессу. Команда TEST исполняется вторым процессом, который также видит FLAG=1,

т.е. он тоже получает доступ к ресурсу. Для предотвращения такой ситуации можно использовать два флажка, каждый из которых нулевым значением показывает желание соответствующего процесса войти в критическую секцию:

```
P1:                              P2:
        MOV FLAG1, 0                     MOV FLAG2, 0
CHECK1: TEST FLAG2, 1            CHECK2: TEST FLAG1, 1
        JZ CHECK1                        JZ CHECK2
        Критическая секция               Критическая секция
        MOV FLAG1,1                      MOV FLAG2,1
```

Такое решение гарантирует взаимное исключение, но порождает другую проблему - тупика или взаимной блокировки процессов. Если переключение с первого процесса на второй произойдет между командами MOV и TEST, то второй процесс может тоже сбросить свой флаг и оба флажка будут в нулевом состоянии. Для обеспечения взаимного исключения и предотвращения тупиков необходимо ввести третий флажок, который показывал бы на более приоритетный процесс в случае возникновения конфликта. Эта идея воплощена в алгоритме Дейкстры, который мы рассмотрим далее для случая N процессов. Но заранее можно сказать, что этот алгоритм весьма непрост. Более простое решение может быть представлено следующим кодом:

```
Pi:
      MOV AL, 0
TRY:  XCHG AL, FLAG
      TEST AL, AL
      JZ TRY
            Критическая секция
      MOV FLAG,1
```

В этом коде используется специальная команда XCHG, присущая процессорам x86 и меняющая в одном машинном цикле содержимое своих операндов. В данном случае обменялись содержимым регистр AL и переменная FLAG.

Но и это решение не является наилучшим, так как ждущие процессы находятся в состоянии активного ожидания (в цикле проверки регистра AL) и тратят впустую процессорное время. Такого недостатка лишен следующий код, использующий специальный системный компонент - монитор или диспетчер, управляющий выполнением процессов:

```
Pi:
AGAIN:      MOV AL, 0
            XCHG AL, SEMA
            TEST AL, AL
            JNZ NEXT
```

```
        Вызов монитора (процесс замораживается,
        пассивное ожидание)
        JMP AGAIN
NEXT:
        Критическая секция
```

Такая операция проверки семафорной переменной, в результате которой процесс может быть подвешен, называется P-операцией над переменной SEMA. Обратная операция, называемая V, используется для пробуждения спящего процесса. Она может быть реализована так:

```
MOV SEMA, 1
Вызов монитора (ожидающий процесс активируется)
```

Диспетчер анализирует очередь процессов, ожидающих у переменной SEMA, и пробуждает один из них. Семафоры могут использоваться не только для взаимного исключения, но и для других задач синхронизации, например:

Процесс A	Процесс B
PA1: активизировать B	PB1: ждать сигнала от A
...	...
PA2: ждать сигнала от B	PB2: активизировать A

Такая схема взаимодействия может быть реализована с помощью двух семафоров:

WAKEA, WAKEB=0

Процесс P1	Процесс P2
PA1: V(WAKEB)	PB1: P(WAKEB)
...	...
PA2: P(WAKEA)	PB2: V(WAKEA)

3.5. Типовые задачи синхронизации

Для начала дадим еще одно определение процесса. ***Процесс*** - вычисление, применение конечного множества операций к конечному набору данных. Два процесса считаются параллельными, если первая операция одного процесса начинает выполняться до завершения другого. Ресурсы, используемые несколькими процессами, называются ***разделяемыми***. ***Критический ресурс*** - это разделяемый ресурс, который одновременно может использоваться не более чем одним процессом. ***Критический интервал*** или ***критическая секция*** - участок кода, где процесс работает с критическим ресурсом. Взаимосвязанные процессы - это процессы, использующие общий ресурс или обменивающиеся информацией. ***Синхронизация***

процессов - ограничения, накладываемые на порядок выполнения процессов. Эти ограничения задаются с помощью ***правил синхронизации***, которые описываются с помощью ***механизмов синхронизации*** (примитивов).

К типовым задачам синхронизации можно отнести:

- взаимное исключение;
- обедающие философы;
- поставщики - потребители;
- читатели - писатели.

Задача о взаимном исключении формулируется так: есть несколько процессов, программы которых содержат участки, где процессы обращаются к разделяемому ресурсу. Требуется исключить одновременные обращения процессов к критическому интервалу. При этом необходимо, чтобы задержка любого процесса вне его критического интервала не влияла на развитие других процессов. Решение должно быть симметричным для всех процессов, т.е. все процессы равноправны. Решение не должно допускать общих и локальных тупиков. ***Общий тупик*** - это взаимная блокировка всех процессов; ***локальный тупик*** - взаимная блокировка одного или нескольких процессов.

Задача "обедающие философы": на круглом столе находятся k тарелок с едой, между которыми лежит столько же вилок, k>=2. В комнате имеется k философов, чередующих философские размышления с принятием пищи. За каждым философом закреплена своя тарелка; для еды философу нужны две вилки, причем он может использовать только вилки, примыкающие к его тарелке. Требуется так синхронизировать философов, чтобы каждый из них мог получить за ограниченное время доступ к своей тарелке. Предполагается, что длительность еды и размышлений философа конечна, но заранее недетерминирована.

Задача "поставщики-потребители": имеется ограниченный буфер на m мест (m порций информации). Он является критическим ресурсом для процессов двух типов: процессы-поставщики, получая доступ к ресурсу, помещают на свободное место порцию информации; процессы-потребители, получая доступ к ресурсу, считывают из него порцию информации. Требуется исключить одновременный доступ процессов к ресурсу. При полном опустошении буфера задерживаются процессы-потребители, при полном заполнении буфера задерживаются процессы-поставщики.

Задача "читатели-писатели": имеется разделяемый ресурс - область памяти, к которой требуется обеспечить доступ процессам двух типов: процессы-читатели могут получать доступ к ресурсу одновременно, они считывают информацию (неразрушающее считывание); процессы-писатели взаимно исключают друг друга и читателей. Известны два варианта этой задачи:

1. читатели, изъявившие желание получить доступ к ресурсу, должны получить его как можно быстрее;

2. читатели, изъявившие желание получить доступ к ресурсу, должны получить его как можно быстрее, если отсутствуют запросы от писателей. Писатель, требующий доступ к ресурсу, должен получить его как можно быстрее, но после обслуживания читателей, подошедших к ресурсу до первого писателя.

Во многих приложениях требуется барьерная синхронизация N параллельных процессов - они все должны прийти к некоторой точке и только после этого все процессы, запросившие барьерную синхронизацию, могут продолжиться.

3.6. Механизм семафоров

Пусть S - ***семафор*** - переменная специального типа с целочисленными значениями, над которой определены две операции: P (закрытие) и V (открытие). Определим эти операции.

P(S): если S≥1, то процесс продолжает выполняться, а S уменьшается на единицу; если S = 0, то процесс задерживается, а имя его передается в очередь процессов, ожидающих доступа к данному ресурсу (обычно семафоры связывают с некоторыми ресурсами).

V(S): если в очереди к семафору S есть процессы, то один из них выбирается и активизируется (переводится в состояние готовности: помещается в очередь процессов, претендующих на процессорное время); если в очереди нет процессов, то выполняется операция S = S+1 (при условии непревышения результатом максимально допустимого значения семафора; для двоичного семафора $S \in \{0,1\}$).

Операции P и V неделимы, т.е. над одним семафором одновременно может работать только один процесс. Так, процесс, начавший работу с семафором, должен ее закончить до переключения процессора на другой процесс в случае квазипараллельных процессов.

Все упомянутые в разд. 3.5 задачи синхронизации можно решить с помощью семафоров. Рассмотрим задачу взаимного исключения:

```
S=1;
Процесс_i:
P(S);
Критическая секция
V(S);
```

При неаккуратном использовании семафорных переменных могут возникать тупиковые состояния. Например, тупик возможен в случае представленных ниже двух процессов при их одновременном выходе на второй вызов P:

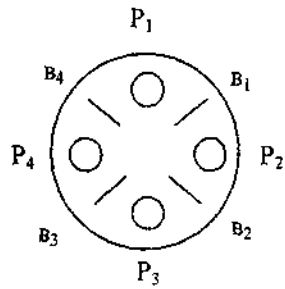
```
S1=1          S2=1
P(S1)         P(S2)
  P(S2)         P(S1)
  ...           ..
```

```
   V(S2)        V(S1)
V(S1)        V(S2)
```

Следует, однако, отметить, что приведенные в примере процессы могут работать и неограниченно долгое время, не попадая в тупик.

3.7. Семафорное решение задачи о философах

Обозначим через P_1, P_2, P_3, P_4 - процессы-философы; b_1, b_2, b_3, b_4 - вилки. Для еды философу необходимо использовать две соседние вилки одновременно (рис. 3.12).



P_1, P_2, P_3, P_4 – философы

B_1, B_2, B_3, B_4 – вилки

Для еды необходимо иметь 2 вилки одновременно

Рис. 3.12. Расположение 4-х философов за круглым столом

Алгоритм работы каждого философа [20] предполагает использование пяти семафоров (S_i, $i = \overline{1,4}$; S). Семафор S предназначен для взаимного исключения процессов при получении доступа к массиву b, отвечающему за вилки. Каждый семафор S_i предназначен для подвешивания i-го философа в том случае, если в результате проверки вилок он обнаружит, что нужных ему вилок в наличии нет. Приведем алгоритм работы первого философа, остальные работают аналогично.

```
P1:   P(S[1]);
       P(S);
        if (b1>0)&(b2>0) then begin
         b1:=0; b2:=0;
         V(S);
                 {питание}
         P(S);
          b1:=1; b2:=1;
         V(S);
         for i:=1 to 4 do V(S[i]);
          {философствование}
        end
        else
```

```
        V(S);
    goto P1;
```

3.8. Разделение общих процедур

Общие процедуры, допускающие мультиплексирование по времени, называются ***реентерабельными***. Одновременно возможно исполнение нескольких экземпляров реентерабельной процедуры. При разделении общих процедур требуется, чтобы при каждом запуске локальные переменные повторно инициализировались. Эти процедуры должны содержать чистый код без побочных эффектов, т.е. реентерабельные процедуры не должны модифицировать глобальные данные, они работают только со своими локальными переменными, расположенными в их собственном стеке.

На рис. 3.13 приведен пример организации доступа к памяти при одновременном вызове реентерабельной процедуры двумя процессами. Учитывая тот факт, что каждый экземпляр процедуры работает со своим собственным сегментом данных и стеком, они не мешают друг другу и могут нормально существовать в параллельном варианте.

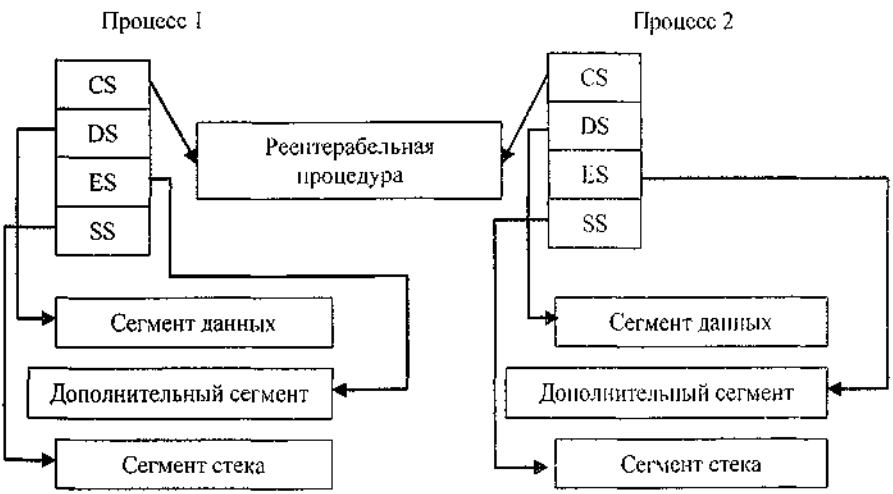


Рис. 3.13. Пример организации реентерабельных процессов

Рассмотрим два примера описания одной и той же функции на языке Си.

Пример 1:

```
void func(int n)
{
      for(int j=0;j<=n;j++);
}
```

Пример 2:

```
int j;
void func(int n)
{
      for(j=0;j<=n;j++);
}
```

Первый пример функции является реентерабельным, так как все необходимые функции данные сохраняются в ее локальном стеке и доступны только ей. Следовательно, новый вызов этой функции создаст новые данные в другом месте стека, и не будет возникать конфликтных ситуаций.

Вторая функция реентерабельной не является, т.к. использует глобальную переменную j, что может привести к конфликту между несколькими одновременно исполняющимися экземплярами этой функции.

3.9. Механизмы синхронизации параллельных процессов

3.9.1. Классификация. Механизмы с активным ожиданием

Механизмы синхронизации параллельных процессов можно разделить на две большие группы:

- централизованные (используются в сосредоточенных системах);
- децентрализованные (используются в распределенных системах).

Простейший механизм централизованной синхронизации - активное ожидание: циклический анализ разделяемой процессами переменной.

Рассмотрим решение задачи взаимного исключения для N процессов - алгоритм Дейкстры.

Массив C(N) содержит N элементов; если C(I) = 1, то i-й процесс не хочет войти в критическую секцию, если C(I) = 0 - хочет войти (исходно все элементы массива C - единицы). Условие вхождения в критический интервал:

```
L:    C(I)=0
      FOR J=1 TO N DO BEGIN
            IF (I<>J) AND (C(J)=0) GOTO L
      END;
```

Это решение может привести к тупику. Чтобы его не возникло, нужен еще один массив: B(N+1), который используется для фиксации факта активности i-го процесса (B(0) соответствует фиктивному процессу) и переменная *очередь*. Исходно переменная *очередь*, характеризующая приоритет процесса, равна 0, а все элементы массива B равны единице (неактивны). Приведем текст i-го процесса:

```
Pi:   B(I):=0
L1:   IF (ОЧЕРЕДЬ=I) GOTO M
L:    C(I):=1
```

```
      IF (B(ОЧЕРЕДЬ)=1) ' ОЧЕРЕДЬ:=I
      GOTO L1
M:    C(I):=0
      FOR J=1 TO N DO
            IF (I<>J) AND (C(J)=0) GOTO L
      КРИТИЧЕСКИЙ ИНТЕРВАЛ
      C(I):=1
      B(I):=1
      ОЧЕРЕДЬ:=0
```

Если в месте программы, помеченном звездочкой, прервался первый процесс, затем второй, то потом они приходят на M, при этом один из них может проскочить в критический интервал, остальные возвращаются на метку L. Если же ни один не проскочил, то возвращаются все процессы, после метки L и оператора IF уходя на L1 и проверяя условие *очередь* = i. Так как *очередь* равна номеру последнего изменившего ее процесса, то все процессы, кроме одного, крутятся в цикле L1 - GOTO L1, а процесс, последним изменивший переменную, пройдет вторично на метку M и далее - в критический интервал.

Более простое решение дано в [21,22]. Оно использует круговой буфер *procphase[N],* i-й элемент которого соответствует i-му процессу, i = 0,..,N-1, изначально все элементы имеют значение out_cr (процесс - вне критической секции - *out of critical section*), а переменная *turn* указывает на процесс, очередной для исполнения (инициализирована 0). Рассмотрим протокол входа в критическую секцию: перед входом каждый процесс полагает procphase[i]=want_cr, показывая желание войти в критическую секцию. Если нет других процессов с номерами между turn и i (в круговом порядке), тоже изъявивших желание, то i-й процесс устанавливает свое право входа в критическую секцию: procphase[i]=claim_cr. Если нет другого процесса j с procphase[j]=claim_cr, то i-й процесс входит в критическую секцию, полагая turn=i. В случае существования такого процесса j, i-й процесс повторяет протокол входа. По завершении критической секции i-й процесс исполняет протокол выхода: проверяет другие процессы в круговом порядке i+1, i+2,.., N-1,0,..,i-1, и если находит некоторый процесс j в состоянии want_cr (или claim_cr), то полагает turn=j, в противном случае turn не меняется. Код этого решения:

```
//глобальные переменные
int procphase[N]={out_cr,..,out_cr}, turn=0;
Pi(){ //i-й процесс, i=0,..,N-1
 //протокол входа
 do{
   procphase[i]=want_cr;
   j=turn;
   //1-й цикл while
   while(j!=i){ //не проверяем для j=i
```

```
      if(procphase[j]==out_cr) j=(j+1)%N;
      else j=turn;       // рестартуем сканирование
   }
   procphase [i]=claim_cr;
   j=(i+1)%N; //нет других желающих процессов
   //2-й цикл while
   // есть ли другие соревнующиеся:
   while(procphase[j]!=claim_cr)j=(j+1)%N;
 }while((j!=i)||(turn!=i&&procphase[turn]!=out_cr));
//конец цикла do
 turn=i;
//конец протокола входа
// критическая секция
 //выходной протокол для i-го процесса
 j=(turn+1)%N;
 while(procphase[j]==out_cr)j=(j+1)%N;
 turn=j;//если имеется другой процесс, то turn будет моди-
фицирован,
//иначе останется равным i
procphase[i]=out_cr;
}
```

В случае повторения входного протокола (например, если turn = 0, процесс i2 (i2<i1) проверяет условие входа в 1-м цикле while, а процесс i1 входит во 2-й цикл while перед тем, как процесс i2 модифицирует свой элемент procphase, но они встретятся во 2-м цикле while - i2 модифицирует procphase[i2] до проверки этого значения процессом i1), только один из процессов-соперников - ближайший к turn - найдет значения out_cr в массиве procphase между turn и собственным номером, так как этот массив уже модифицирован, и поэтому, тупик не возникнет и ближайший к turn процесс пройдет дальше.

Рассмотрим также алгоритм Л.Лампорта "булочной лавки" [22, 23]. Основная идея алгоритма "булочной": каждый вновь прибывающий процесс получает талончик на обслуживание с номером; процесс с наименьшим номером на талончике обслуживается следующим. Из-за неатомарности операции присваивания у некоторых процессов могут оказаться одинаковые талоны - в этом случае первым обслуживается процесс с меньшим идентификатором. Разделяемые данные для алгоритма:

```
//фиксирует, что процесс получает
shared enum {false, true} choosing[n];
//талончик; инициируется значениями false (никто талончики
не получает)
shared int ticket[n];
```

```
//массив талончиков; инициируется нулями
//n - число процессов.
```

Введем отношение < такое, что: (a,b) < (c,d), если a < c или если a == c и b < d.

Структура процесса с идентификатором i, i=0,..,n-1, для алгоритма "булочной" приведена ниже

```
choosing[i] = true;//собирается взять талончик
ticket[i] = max(ticket[0], ..., ticket[n-1]) + 1;
choosing[i] = false;//взял талончик
for(j = 0; j < n; j++){
  while(choosing[j]); //ждать, пока j-й берет талончик
    while((ticket[j]!=0&&
          (ticket[j],j) < (ticket[i],i));
          //ждать, когда
          //обслужится j-й предшествующий
} //конец for
//критическая секция
ticket[i] = 0;
//остальной код
```

3.9.2. Механизм событий

С событиями связываются операции: ждать событие и объявить событие. Событие представляется специальной переменной, которая может находиться в *установленном или сброшенном состоянии*. *В отличие от семафорной операции* V при объявлении события отпускаются все процессы, стоящие в очереди к этому событию. Как правило, механизм событий используется для синхронизации главного и подчиненного процессов.

Пример на PL/1:

```
PROG: PROC MAIN (MAIN, TASK)
      CALL A EVENT (V1)
      CALL B EVENT (V2)
      WAIT (V1, V2)
```

Здесь программа PROG вызывает две процедуры - A и B - как сопрограммы. Они исполняются параллельно основной программе. С каждой из этих сопрограмм связывается событие, которое устанавливается при их завершении. Таким образом, главная программа ожидает завершения обеих сопрограмм.

Отметим, что механизм событий не позволяет решать задачу взаимного исключения.

3.9.3. Механизмы критических областей

Как уже говорилось ранее, критическая секция - часть кода, где осуществляется работа с разделяемым ресурсом, доступ к которому ограничен.

В языке Модула-2 критические области реализованы в виде:

```
VAR
      V: SHARED NUM
BEGIN
      REGION V DO S1, …, SN; END;
```

Может быть введено условие входа в критическую секцию:

```
REGION V WHEN B DO S1, …, SN; END;
```

Работает это так: если условие не выполняется - происходит блокировка процесса до выполнения условия, но вход в критическую секцию возможен только по одному процессу, взаимно исключая друг друга. Например, в случае задачи "поставщики-потребители" доступ производителя к буферу возможен только при условии наличия свободного места в нем и отсутствии процессов, работающих с ним:

```
REGION Буфер WHEN Неполон(Буфер) DO ПоместитьВ(Буфер, Порция); END;
```

Кандидаты на вход в условную критическую секцию - процессы, для которых условие истинно.

3.9.4. Механизм мониторов

Монитор - программный компонент, в котором определены разделяемые переменные и заданы операции над этими переменными. Процедуры монитора взаимно исключают друг друга, т.е. два процесса не могут одновременно выполнять код одной и той же или разных процедур монитора. Если при обращении процесса к монитору требуемый ресурс занят, то процесс должен быть задержан. С каждой причиной задержки процесса связывается переменная типа condition.

К переменным типа condition применяются две операции: wait - ставит процесс в очередь к данной переменной, signal - позволяет активизировать ожидающий процесс. Механизм мониторов используется в языках Модула-2, Java. Ниже приведена часть программы на языке Модула-2, реализующая двоичный семафор

```
RESOURCE: MONITOR;
VAR
      BUSY: BOOLEAN;
      NONBUSY: CONDITION;{переменные монитора}
AQUIRE: PROC; {операция монитора}
      BEGIN
```

```
            IF BUSY THEN NONBUSY.WAIT
            BUSY:=TRUE
END;
RELEASE: PROC; {операция монитора}
      BEGIN
            BUSY:=FALSE;
            NONBUSY.SIGNAL;
      END;
BEGIN{инициализация переменных монитора}
      BUSY:=FALSE
END.
```

3.9.5. Управляющие выражения

Управляющие выражения позволяют с помощью специальных операций указывать последовательность действий. К числу таких операций относятся:

- ";" - последовательное выполнение;
- "," - возможность параллельного выполнения;
- "*" - многократное недетерминированное исполнение.

Рассмотрим задачу, представленную графом на рис. 3.14. Здесь действия исполняются в следующем порядке: сначала A, потом B и C параллельно некоторое количество раз, затем D.

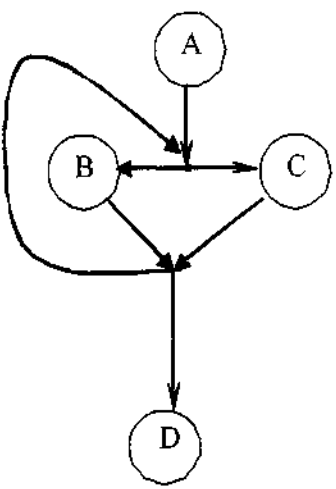


Рис. 3.14. Управляющий граф задачи

Управляющее выражение для графа на рис. 3.14 будет:

```
PATH
      A;*(B, C);D
END;
```

3.9.6. Приоритетное обслуживание и критические ресурсы

Приоритетное обслуживание задач может использоваться системой при разграничении доступа к критическим ресурсам.

Приоритеты бывают двух основных типов: статические и динамические Статические приоритеты назначаются задачам во время их старта и уже не меняются в процессе выполнения. В случае динамических приоритетов начальный приоритет задачи может меняться в зависимости от нужд системы в процессе исполнения.

При использовании только статических приоритетов и наличии критических ресурсов, может наблюдаться так называемая ***инверсия приоритетов***, когда высокоприоритетный процесс вынужден пропускать вперед низкоприоритетные Для начала рассмотрим пример распределения ресурсов на основе схемы возрастания директивного срока DM (Deadline Monotonic), когда больший приоритет получают процессы с меньшим директивным сроком. Эта схема может приводить к инверсии приоритетов.

Таблица 3 1

Основная информация о процессах

Процесс	Приоритет	Последовательность исполнения	Время старта
A	1	EQQQQE	0
B	2	EE	2
C	3	EVVE	2
D	4	EEQVE	4

Задачи для примера приведены в табл. 3.1, из которой видно, что запросы на выполнение задач появляются в моменты времени 0, 2, 2, 4, и процесс D имеет самый высокий приоритет, что в нашем случае означает минимальный директивный срок. Процессы в течение предоставленного кванта времени либо просто исполняются, не требуя критических ресурсов - E, либо требуют доступа к критическим ресурсам (Q или V). Характер их поведения во времени представлен в столбце "Последовательность исполнения", а временная диаграмма приведена на рис 3.15. Здесь каждый квадратик представляет единицу времени. Ряд квадратиков соответствует оси времени для процессов A, B, C, D. "P" внутри квадратика символизирует состояние "прерван" соответствующего процесса, "b" - блокировку из-за запроса на ресурс. Видим, что высокоприоритетная задача D должна ожидать

момента завершения низкоприоритетных задач B, C и почти что момента завершения самой низкоприоритетной задачи A (время пребывания D в системе - 12).

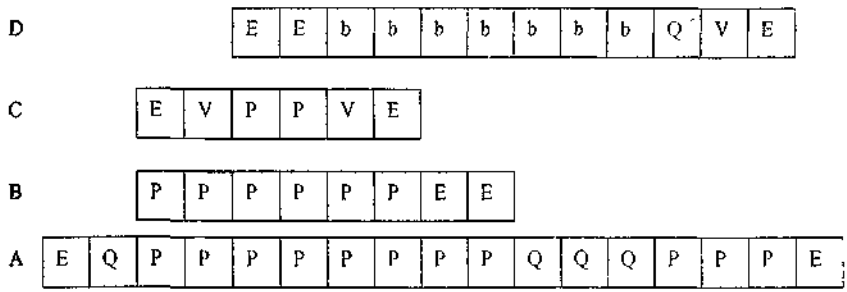


Рис. 3.15. Пример инверсии приоритетов при использовании схемы DM возрастания директивного срока

При управлении блокировкой разделяемых ресурсов часто используются два основных подхода: Original Ceiling Priority Protocol (OCPP) и Immediate Ceiling Priority Protocol (ICPP) [24]. Когда любой из этих протоколов применяется в однопроцессорной системе, то высокоприоритетный процесс может быть блокирован низкоприоритетным процессом не более одного раза за время своего выполнения. Кроме того, тупики и транзитивные блокировки (процесс с блокирован процессом b, заблокированным процессом a, и т.д.) невозможны, взаимное исключение процессов при доступе к критическим ресурсам гарантировано. Это достигается путем так называемого наследования приоритетов. Например, задача, заблокировавшая высокоприоритетный процесс, на время блокировки может работать на уровне приоритета заблокированного процесса, т.е. наряду со статическими приоритетами могут использоваться и динамические.

Рассмотрим протокол OCPP:

1. Каждый процесс имеет статически назначенный приоритет, например, по схеме DM возрастания директивного срока или по схеме RM (Rate Monotonic) возрастания периодичности запуска (меньше период - больше приоритет, т.е. приоритет отдается более высокочастотным задачам).
2. Каждый критический ресурс имеет статически определенную величину "потолка" - максимальный из приоритетов процессов, которые его используют. Предполагаем, что более высокий приоритет представлен большим числовым значением.
3. Процесс имеет динамический приоритет, являющийся максимумом из собственного статического приоритета и приоритета, который он наследует благодаря блокировке более высокоприоритетного процесса.

Т.е. когда процесс блокирует высокоприоритетную задачу, его приоритет временно становится равным этому более высокому приоритету.

4. Процесс может занять ресурс, только если его динамический приоритет выше, чем потолок каждого в данный момент занятого ресурса, исключая те, что он уже занял.

Протокол гарантирует, что если ресурс занят процессом a, и это может привести к блокировке высокоприоритетного процесса b, то ни один другой ресурс, который может занять процесс b, не будет занят ни одним процессом отличным от a. Процесс может быть задержан не только при попытке занять ресурс, который уже занят, но и при попытке захватить ресурс, если это может привести к множественной блокировке высокоприоритетных процессов.

Рассмотрим теперь применение к той же системе задач (табл. 3.1) дисциплины OCPP (рис. 3.16).

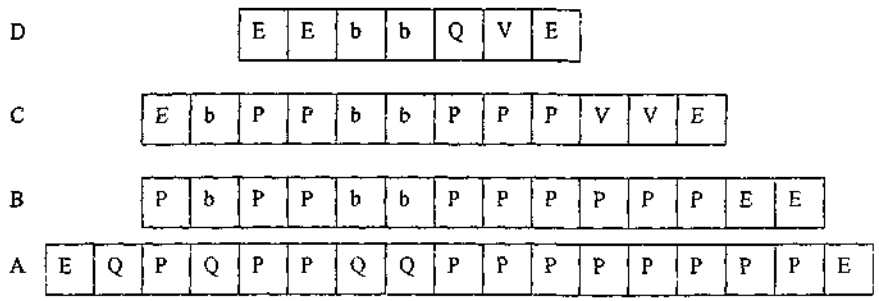


Рис. 3.16. Пример наследования приоритетов, протокол OCPP

Процесс A снова занимает первую критическую секцию Q, так как ни один ресурс еще не занят. Он снова прерывается процессом C, но теперь попытка C занять вторую секцию (V) неудачна, так как его приоритет (3) не выше, чем потолок каждого занятого в этот момент ресурса (он равен 4, так как Q должен быть использован процессом D). В момент 3, A блокирует C, и, следовательно, выполняется на приоритете 3. Высокоприоритетный процесс D прерывает A в момент 4, но соответственно блокируется, когда пытается получить доступ к Q. Следовательно, A возобновится (на приоритете 4) до освобождения им Q - при этом его приоритет падает обратно до 1. Теперь D возобновляется до завершения (его время пребывания в системе - 7). Как уже отмечалось, преимуществом протокола является то, что высокоприоритетный процесс может быть блокирован только раз (на одну свою активацию) любым более низкоприоритетным процессом. Однако из рисунка видно, что процессы B и C блокированы дважды. В действительности это одна блокировка, разбитая на две из-за прерывания процессом D.

Рассмотрим протокол ICPP:

1. Каждый процесс имеет назначенный ему статически приоритет.
2. Каждый критический ресурс имеет статическую величину потолка - максимум из приоритетов процессов, которые должны его использовать.
3. Процесс имеет динамический приоритет - максимум из своего статического приоритета и величин потолка занятых им ресурсов.

Рассмотрим снова тот же пример, но применим теперь протокол ICPP (рис. 3.17).

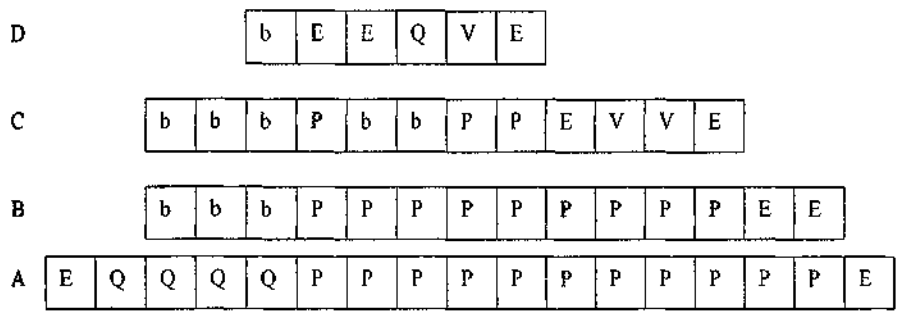


Рис. 3.17. Пример наследования приоритетов, протокол ICPP

Процесс A, занявший Q в момент 1, следующие четыре единицы времени выполняется на приоритете 4. Следовательно, другие процессы блокируются. Когда A освобождает Q, и его приоритет понижается, другие процессы исполняются в порядке своих приоритетов. В этом примере процессы блокируются до начала их исполнения, а время отклика для процесса D теперь - 6.

Таким образом, протоколы OCPP и ICPP позволяют избежать инверсии приоритетов, при этом последний протокол проще и в рассмотренном примере оказался эффективнее.

3.9.7. Механизм рандеву языка Ада

Механизм рандеву используется в языке Ада для синхронизации параллельно выполняемых задач. Для его описания требуется кратко познакомиться с основными чертами этого языка, что имеет смысл еще и потому, что в настоящее время Ада - один из основных языков, используемых в мире для создания СРВ.

Краткое описание языка Ада

При изложении материала мы следуем в основном [24-29].

Язык ADA был разработан в начале 1980-х (ADA-83) в ответ на требование Министерства обороны США создать стандартный язык разработки критичных к надежности приложений, в первую очередь, СРВ. Дальнейшее развитие язык получил в стандарте ADA-95.

Структура программы

Программа языка ADA имеет блочную структуру:

```
declare
  <описания>
begin
  <последовательность операторов>
exception
  <обработчики исключений>
end;
```

Например:

```
simple:
declare
  Temp:Integer:=A;
begin
 A:=B;
 B:=Temp;
End simple;
```

Комментарии начинаются знаками "- -". Блок может иметь имя. При именовании блока, имя блока должно указываться перед блоком и после *end*, завершающего блок. Так, в примере выше, имя блока - *simple*.

Основные типы данных языка Ада

Определение переменных нужного типа осуществляется так:

```
<Имя переменной>:<имя типа>
<Имя переменной>:<имя типа>=<начальное значение>
<Имя переменной>:<имя типа> range <диапазон значений>
```

Например:

```
i:Integer;
i:Integer=10;
i:Integer range 0..16383;
```

Тип *Integer* задает целые числа в диапазоне от -2^{31} до $+(2^{31} - 1)$. Язык Ада, как и Паскаль, не различает регистры.

Вещественные типы задаются с помощью ключевого слова *Float*. Также имеется возможность определять новые вещественные типы:

```
type New_Float is digits 10 range -1.0e18..1.0e18;
subtype Crude_Float is New_Float digits 2;
subtype Pos_New_Float is New_Float range 0.0..1000.0;
type Scaled_Int is delta 0.05 range -100.00..100.00;
```

Перечисляемые типы:

```
type Computer_Language is (Assembler, Cobol, Lisp, Pascal,
Ada);
```

Логический тип описан в пакете *Standard*, подключаемом к любой программе Ада:

```
type Boolean is (False, True);
```

Символьные типы бывают двух разновидностей: *Character* (256 символов) и *Wide_Character* (64K символов).

Язык допускает определение подтипов. Подтипы совместимы по присваиванию со своими родительскими типами и определяются следующим образом:

```
subtype <Имя подтипа> is <Базовый тип>;
subtype Data is Integer;
subtype Byte is Integer range 0..255;
```

Также возможно определение совершенно новых типов, несовместимых по присваиванию с родительскими типами, но допустимые значения которых такие же, как у базового типа:

```
type <Имя типа> is new <Базовый тип>;
```

Например, тип *Person* для описания человека и тип *Bank* для описания банка может быть физически одинаков - строка из 20 символов, но они должны будут трактоваться, как данные разных типов.

Массивы определяются так:

```
type <имя_массива> is array (<спецификация_индекса>,..) of
<тип_элементов_массива>;
```

Примеры:

```
Max: Const Integer:=10; -- определение константы
type Reading_T is array (0..Max-1) of  Float;
Size: Const Integer:=Max-1;
type Switches_T is array(0..Size, 0..Size) of Boolean;
Reading: Reading_T;
Switches: Switches_T;
```

Массив может быть объявлен непосредственно:

```
Reading1 : array(0..Max-1) of Float;
```

В этом случае массив называется анонимным, поскольку он не имеет явного типа, и будет несовместим по присваиванию с другими массивами, даже такими, которые описаны так же. Кроме того, такие массивы не могут быть использованы

как параметры подпрограмм. В общем случае рекомендуется избегать использования анонимных массивов.

При обращении к элементам массива индексы указываются в круглых скобках. Возможно присваивание значений сразу нескольким элементам массива:

```
type Stock_Level is Integer range 0..20_000;
type Pet is (Dog, Budgie, Rabbit);
type Pet_Stock is array(Pet) of Stock_Level;
Store_1_Stock : Pet_Stock;
. . .
Store_1_Stock := (5, 4, 300);
Store_1_Stock := (Dog..Rabbit => 0);
```

Допускается использование отрезков массива для одномерных массивов:

```
type Stack is array (1..50) of Integer;
Calculator_Workspace : Stack;
. . .
Calculator_Workspace (5 .. 10) := (5, 6, 7, 8, 9, 10);
Calculator_Workspace (25 .. 30) := Calculator_Workspace (1
.. 10);
```

В языке существует понятие массива-константы:

```
type Months is (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
Sep, Oct, Nov, Dec);
subtype Month_Days is Integer range 1..31;
type Month_Length is array (Jan..Dec) of Month_Days;
Days_In_Month : constant Month_Length := (31, 28, 31, 30,
31, 30, 31, 31, 30, 31, 30, 31);
```

Предопределенный тип массива String задает символьные строки:

```
My_Name    : String (1..20);
My_Surname : String (21..50);
```

Массиву может присваиваться значение другого массива того же типа.

Проверки на равенство и на неравенство доступны почти для всех типов Ада. Два массива считаются равными, если каждый элемент первого массива равен соответствующему элементу второго массива.

Записи описываются следующим образом:

```
type <имя_записи> is
    record
        <имя_поля_1> : <тип_поля_1>;
        <имя_поля_2> : <тип_поля_2>;
```

```
        . . .
        <имя_поля_N> : <тип_поля_N>;
    end record;
```

Например:

```
type Bicycle is
    record
        Frame       : Construction;
        Maker       : Manufacturer;
        Front_Brake : Brake_Type;
        Rear_Brake  : Brake_Type;
    end record;
My_Bicycle  : Bicycle;
```

Можно задавать значения полей записи по умолчанию. Доступ к полям записи осуществляется через точку.

```
type Date_T is record
  Day: Day_T:=1;
  Month: Month_T:=1;
  Year: Year_T;
end;
D: Date_T;
begin
  D.Year:=2003; -- доступ к полю записи
  D:=(26,4,2003); --полное присваивание
  -- полное присваивание по имени:
  D:=(Year=>2003, Day=>26, Month=>4)
end
```

В Ада существуют ссылочные типы данных. При этом имеется ряд характерных особенностей:

- Ссылочные типы не могут быть использованы для доступа к произвольному адресу памяти.
- Значения ссылочного типа не рассматриваются как целочисленные, а значит, они не поддерживают адресную арифметику.
- Значения ссылочного типа могут ссылаться только на значения того типа, который был указан при описании ссылочного типа.

Пример использования ссылочного типа:

```
type Person_Name is new String (1 .. 4);
type Person_Age is Integer range 1 .. 150;
```

```
type Person is
    record
        Name : Person_Name;
        Age  : Person_Age;
    end record;
-- описание ссылочного типа:
type Person_Ptr is access Person;
-- переменные (объекты) ссылочного типа:
Someone, SomeOne1 : Person_Ptr;
```

Пустая ссылка записывается, как *null*.

Создание объекта в памяти:

```
Someone  := new Person; --без инициализации
Someone1 := new Person'( Name => "John", --с инициализацией
                         Age  => 43);
```

Разыменование, т.е. получение значения переменной по ее адресу, осуществляется неявным образом:

```
Someone.Name := "Fred";
Someone.Age  := 33;
```

С помощью зарезервированного слова all можно обратиться ко всему содержимому ссылки целиком:

```
--копирование значений полей структуры:
Someone.all := SomeOne1.all;
```

Списковые структуры могут определяться так:

```
type Element;                    -- неполное описание типа

type Element_Ptr is access Element;

type Element is                  -- полное описание типа
    record
        Value : Integer;
        Next  : Element_Ptr;
    end record;
-- переменная, которая будет началом списка:
Head_Element : Element_Ptr;
```

Существует два способа освобождения пространства, которое было распределено в области динамической памяти под данные ссылочного типа, для его последующего повторного использования:

- библиотека времени выполнения выполняет неявное освобождение распределенного пространства, когда объект выходит из области видимости, т.е. производится сборка мусора, как в языках C# или Java;
- выполнение явного освобождения пространства динамической памяти в программе.

Для первого случая примечательно то, что если тип объекта описан на уровне библиотеки, то освобождение памяти не произойдет вплоть до завершения работы программы.

Если необходимо освободить память явным образом подобно тому, как это делает системный вызов free в UNIX, то можно конкретизировать настраиваемую процедуру *Ada.Unchecked_Deallocation*, для которой определено:

```
generic
   type OBJECT is limited private;
   type NAME   is access OBJECT;
procedure UNCHECKED_DEALLOCATION(X : in out NAME);
```

Например:

```
procedure Free is new Ada.Unchecked_Deallocation(Object =>
Element, Name => Element_Ptr);
```

Такая процедура FREE имеет следующий эффект:

- по завершении FREE(X) значение X есть null;
- FREE(X), когда X уже null, не имеет эффекта;
- FREE(X), когда X не равен null, указывает на то, что объект, обозначенный X, больше не нужен, и что отведенная ему память может быть повторно использована.

Если X и Y означают один и тот же объект, то доступ к нему с помощью Y будет ошибочным, если предпринимается после вызова FREE(X); эффект такого доступа в языке не определен.

В Ада используется явное преобразование типов. Кроме того, при преобразовании типа Float в тип Integer производится округление полученного значения по правилам математики:

```
Cnt: Integer;
FCnt: Float=10.5;
. . .
Cnt=Integer(FCnt); -- Cnt будет содержать 11
```

Управляющие структуры

Пустая инструкция - *null* - используется в тех случаях, когда не требуется выполнение каких-либо действий, но согласно синтаксису языка должна быть записана хотя бы одна инструкция.

Условные инструкции:

```
if <логическое_выражение> then
    <последовательность инструкций 1>
{elsif <логическое_выражение> then
    <последовательность инструкций 2>}
[else
    <последовательность инструкций>]
end if;
```

В приведенном выше описании и в последующих описаниях {x} - метасимвол повторения x ноль или более раз, [x] - метасимвол повторения x ноль или один раз.

Пример условного оператора:

```
if MONTH = DECEMBER and DAY = 31 then
       MONTH := JANUARY;
       DAY   := 1;
       YEAR  := YEAR + 1;
end if;
```

Инструкция выбора:

```
case  <выражение>  is
   when <значение_выбора> =>
        <последовательность инструкций>
  {when <значение_выбора> =>
        <последовательность инструкций>}
  [when others => <последовательность инструкций>]
end case;
```

Пример:

```
case Letter is
    when 'a'..'z'| 'A'..'Z' => Put ("letter");
    when '0'..'9'           => Put ("digit! Value is");
                               Put (letter);
    when ''' | '"' | '`'    => Put ("quote mark");
    when '&'                => Put ("ampersand");
    when others             => Put ("something else");
end case;
```

В языке имеются три циклические конструкции: простой цикл, цикл пока и цикл с параметром.

Простой (бесконечный) цикл описывается так:

```
loop
    <инструкции тела цикла>
end loop;
```

Цикл пока (работает по истинности условия):

```
while <логическое выражение> loop
    <инструкции тела цикла>
end loop;
```

Цикл с параметром:

```
for <счетчик> in [reverse] <диапазон значений счетчика>
loop
    <инструкции тела цикла>
end loop;
```

Например, вывод на экран целых значений от 1 до 20, а также от 20 до 1 может выглядеть так:

```
for Count in 1..20 loop --переменная цикла декларируется по
                        --умолчанию
    Put (Count);
end loop;

for Count in reverse 20..1 loop
    Put (Count);
end loop;
```

Любой дискретный тип может использоваться для указания диапазона значений переменной-счетчика:

```
declare
    subtype List is Integer range 1..10;
begin
    for Count in List loop
        Put (Count);
    end loop;
end;
```

Для выхода из цикла могут использоваться конструкции безусловного и условного прерывания цикла соответственно:

```
exit;
exit when <логическое_выражение>;
```

Инструкция перехода *goto* предусмотрена для использования в языке Ада, в исключительных ситуациях, и имеет следующий вид:

```
goto Label;
```

Здесь *Label* - идентификатор, помечающий инструкцию программы, к которой необходимо выполнить переход. Например:

```
<<AFTER>> X := 1;
..
goto after;
```

Отметим, что нельзя выполнить переход внутрь условной инструкции *if*, внутрь цикла (*loop*) и за пределы подпрограммы.

Подпрограммы

Подпрограммы в Ада, как правило, состоят из двух частей: спецификации и тела. ***Спецификация*** описывает интерфейс обращения к подпрограмме, другими словами - "что" обеспечивает подпрограмма (заголовок: название и список формальных параметров). ***Тело подпрограммы*** описывает детали реализации алгоритма работы подпрограммы, т.е. "как" подпрограмма устроена. Заголовок в спецификации и в описании тела подпрограммы должны совпадать.

Каждый формальный параметр подпрограммы имеет имя, тип и режим передачи.

В Ада существует два типа подпрограмм: процедуры и функции.

Формат описания тела процедуры приведен в табл. 3.2.

Таблица 3.2

Синтаксис описания тела процедуры.

`procedure <имя процедуры> [ (<формальные параметры>)] is`	Заголовок процедуры, определяющий имя процедуры и ее формальные параметры (если они есть).
`. . .`	Описательная (или декларативная) часть, которая может содержать локальные описания типов, переменных, констант, подпрограмм...
`begin`	
`. . .`	Исполнительная часть процедуры, которая описывает алгоритм работы процедуры; обязана содержать хотя бы одну инст

	рукцию.
end [<имя процедуры>];	Здесь указание имени процедуры опционально.

Примечательно требование языка, чтобы исполнительная часть процедуры содержала хотя бы одну инструкцию. Поэтому, как правило, на этапе проектирования, при написании процедур-заглушек используется пустая инструкция *null*.

При вызове процедуры указывается имя процедуры и список ее параметров в круглых скобках.

Функции во многом подобны процедурам (табл. 3.3), за исключением того, что они возвращают значение в вызвавшую их программу.

Таблица 3.3

Синтаксис тела описания функции

function <имя функции> [(<формальные параметры>)] return <тип возвращаемого значения> is	Заголовок функции, определяющий имя функции, ее формальные параметры (если они есть) и тип возвращаемого значения.
. . .	Описательная (или декларативная) часть, которая может содержать локальные описания типов, переменных, констант, подпрограмм...
begin	
. . .	Исполнительная часть функции, которая описывает алгоритм работы функции; обязана содержать хотя бы одну инструкцию возврата значения - `return`.
end [<имя функции>];	Здесь указание имени функции опционально.

Для формальных параметров подпрограмм используются следующие режимы: *in, in out, out*. По умолчанию используется режим *in*.

При режиме *in*, формальный параметр инициализируется значением фактического параметра, при этом внутри подпрограммы он является константой и разрешает только чтение значения ассоциированного фактического параметра.

Режим *in out* непосредственно соответствует параметрам, передаваемым по ссылке (подобно var-параметрам языка Паскаль).

При *out* не происходит начальной инициализации формального параметра значением фактического. Такой параметр служит только для передачи некоторого значения в точку вызова подпрограммы.

Для передачи параметров в функцию разрешается использовать только режим *in*. Поэтому функция через свои параметры может только импортировать данные из среды, вызвавшей ее. При этом параметры функции не могут быть использованы для изменения значений переменных в точке вызова функции. Таким образом, в отличие от традиций языка Си, функции Ада не обладают побочными эффектами.

Также как и Паскаль, и в отличие от Си, Ада позволяет встраивать одни подпрограммы в другие, конструируя один общий компилируемый модуль.

Возможно использование параметров по умолчанию:

```
procedure Print_Lines(No_Of_Lines: in Integer := 1);
```

Допускается перегрузка (совмещение) процедур и функций, а также перегрузка операторов, т.е. создание нескольких подпрограмм с одинаковыми именами, но разными списками параметров.

Пример описания тела процедуры вычисления корней квадратного уравнения:

```
Procedure Quadratic (a,b,c: in float; R1, r2: out float;
Ok:out Boolean) is
 Z: float;
 Begin
   Z:=b*b-4.0*a*c;
   if z<0.0 or a=0.0 then
    Ok:=false;R1:=0;R2:=0;
    Return;
   end if;
   ok:=true;
   r1:=(-b+sqrt(z))/(2.0*a); r2:=(-b-sqrt(z))/(2.0*a);
  end quadratic;
```

Функция поиска минимума двух чисел:

```
Function minimum(x,y: in integer) return integer is
Begin
  If x>y then
    Return y;
  Else
     Return x;
  End if;
End minimum;
```

Отметим, что в Ада возможно использование указателей на процедуры и их передача в качестве параметров. Рассмотрим пример с выдачей сообщений об ошибках.

Следующее определение типа языка Ада (Error_Report) дает возможность ввести указатель на процедуру, которая в качестве параметра принимает строку сообщения об ошибке. Далее определяется процедура с параметром такого типа (указатель на процедуру), вызов ее делается с фактическим параметром - указателем на конкретную процедуру соответствующего типа (в примере - на процедуру Operator_Warning).

```
Type ErrorReport is access procedure (Reason: in string);
Procedure OperatorWarning (Message: in String) is
Begin
  --информировать оператора об ошибке
end;
procedure ComplexCalculation(ErrorTo: ErrorReport) is
begin
--если зафиксирована ошибка в сложном вычислении
 ErrorTo("Giving Up");
End ComplexCalculation;
. . .
  ComplexCalculation(OperatorWarning'Access);-- передача
--указателя на процедуру
. . .
```

Процедуры могут быть использованы как расширения кода (inline expansion). Директива (прагма) Inline требует использовать расширения кода для указанных подпрограмм где это возможно

```
pragma INLINE (name {, name});
```

Исключения

Исключения предоставляют программе возможность обработки ошибок времени исполнения.

Существует пять исключений, которые стандартно предопределены в языке программирования Ада. Они приведены в табл 3 4.

Таблица 3.4

Стандартные исключения языка Ада

Исключение	Описание
Constraint_Error	Ошибка ограничения
Numeric_Error	Ошибка числа
Program_Error	Ошибка программы
Storage_Error	Ошибка памяти
Tasking_Error	Ошибка задачи

Возможно определение пользовательских исключений. Например

```
My_Very_Own_Exception : exception;
```

```
Another_Exception      : exception;
```

Исключение генерируются так:

```
raise <исключение>;
```

Обработка исключений идет в разделе *exception* блока Ада-программы. Например:

```
exception
    when Constraint_Error =>
        Put("that number should be between 1 and 20");
    when others =>
        Put("some other error occurred");
end;
```

Параллельные задачи в языке Ада. Рандеву

Задачи (tasks) Ада представляют параллельные процессы. Входы (entries) задач аналогичны процедурам, но используются для межзадачных коммуникаций. Они могут обладать входными и выходными параметрами.

Рассмотрим пример полной программы на языке Ada с задачами и использованием типа SEMAPHORE для их синхронизации (одна из демонстрационных программ к пакету SmallADA - свободно распространяемой среды программирования Ада, реализованной кафедрой Department of Electrical Engineering and Computer Science The George Washington University Washington, DC 20052 USA в 1991)

```
--включить и использовать пакет SMALL_SP:
with SMALL_SP;
use package SMALL_SP
procedure FOURTASK is
- в оригинале рассматриваются 4 задачи
  SCREEN: SEMAPHORE := 1; -- глобальный семафор перерисовк
  task CYCLIC_1 is
    entry START;-- задание входа задачи
  end CYCLIC_1; -- интерфейс задачи 1

  task CYCLIC_2 is
    entry START; -- задание входа задачи
  end CYCLIC_2; -- интерфейс задачи 2
-- мы вводим только 2 задачи
  task body CYCLIC_1 is -- тело задачи 1
    T : FLOAT;
  begin
    accept START;-- ожидание вызова входа
```

```
    WAIT(SCREEN); -- ожидание доступа к экрану
    CURSORAT(1,1); PUT("CYCLIC_1 RUNS EVERY 3.0 SECONDS");
    SIGNAL(SCREEN); -- освобождаем семафор
    T := CLOCK + 3.0;
    for HOW_MANY in 1..20
    loop
      WAIT(SCREEN);
      CURSORAT(2,1); PUT("        ");
      SIGNAL(SCREEN);
      delay T - CLOCK;
      T := CLOCK + 3.0;
      WAIT(SCREEN);
      CURSORAT(2,1); PUT("RUNNING"); PUT(HOW_MANY);
      SIGNAL(SCREEN);
    end loop;
  end CYCLIC_1;

  task body CYCLIC_2 is -- тело задачи 2
    T : FLOAT;
  begin
    accept START; -- ожидание вызова входа
    WAIT(SCREEN);
    CURSORAT(3,1); PUT("CYCLIC_2 RUNS EVERY 5.0 SECONDS");
    SIGNAL(SCREEN);
    -- отличается от задачи 1 только задержкой:
    T := CLOCK + 5.0;
    for HOW_MANY in 1..10 -- и количеством итераций
    loop
      WAIT(SCREEN);
      CURSORAT(4,1); PUT("        ");
      SIGNAL(SCREEN);
      delay T - CLOCK;
      T := CLOCK + 5.0;
      WAIT(SCREEN);
      CURSORAT(4,1); PUT("RUNNING"); PUT(HOW_MANY);
      SIGNAL(SCREEN);
    end loop;
  end CYCLIC_2;
-- задачи 3 и 4 абсолютно подобны этим двум задачам,
-- поэтому не рассматриваются
begin
  CYCLIC_1.START; --запуск задачи
```

```
  CYCLIC_2.START; --запуск задачи
end FOURTASK;
```

В этом примере, также как и в следующей реализации задачи, моделирующей тип семафора, входы задачи используются только для синхронизации без передачи информации.

Ниже приведена реализация бинарного семафора с помощью задачи *SEMAPHOR*.

```
TASK SEMAPHOR IS
  ENTRY P; -- объявление входа P
  ENTRY V; -- объявление входа V
END SEMAPHOR;
--ТЕЛО ЗАДАЧИ:
TASK BODY SEMAPHOR IS
BEGIN
  LOOP
    ACCEPT P; -- оператор приема входа P
    ACCEPT V; -- оператор приема входа V
  END LOOP;
END SEMAPHOR;
```

При обращении к входу вызывающая задача может быть обслужена немедленно, только если никакой другой процесс не обслуживается в данный момент Если процесс при вызове задерживается, то он помещается в очередь к соответствующему входу. Если задача выходит на оператор ACCEPT приема входа, а вызова этого входа нет, то задача задерживается до появления вызова. При появлении вызова, если задача выполняет оператор приема, то осуществляется обмен информацией или же только синхронизация процессов. До завершения выполнения операторов критической секции, указанных в операторе приема *accept*, вызывающая задача пассивна (оператор критической секции может отсутствовать, как в приведенных примерах). По завершении критической секции обе задачи выполняются параллельно. Именно такое взаимодействие называется ***рандеву***.

Обращения к семафору со стороны других задач, например, для взаимного исключения, будут выглядеть так:

```
. . .
SEMAPHOR.P;
--Критическая секция
SEMAPHOR.V;
. . .
```

Отметим, что предложенная реализация семафора не совсем соответствует классическому варианту, так как вызов входа V какой-либо задачей приведет к

блокировке в случае, если с ее стороны не было вызова P. Это неверно, так как по определению операция V не приводит к блокировке, а лишь наращивает счетчик семафора или оставляет его без изменения при достижении максимального значения.

Для более гибкой работы с входами язык Ада поддерживает специальную конструкцию *SELECT*:

```
Select
  When <Булевское выражение> =>
    Accept <вход> do
       <Инструкции обмена информацией> --операторы
                                       --критической секции
    end <вход>
  -- любая последовательность инструкций, исполняемых после
  -- окончания рандеву
  Or when . . .
    -- и т.д.
    . . .
end select
```

При выполнении инструкции *select* задача-сервер циклически опрашивает свои входы на наличие вызова рандеву от задач-клиентов, причем происходит это без блокировки в состоянии ожидания вызова рандеву на каком-либо входе. Опрос продолжается до тех пор, пока не будет обнаружен вызов рандеву на каком-либо входе, который соответствует одной из перечисленных инструкций принятия рандеву. После обнаружения вызова выполняется соответствующая альтернатива. Завершение обработки альтернативы приводит к завершению инструкции отбора.

Отметим, что пока операторы, стоящие после *accept* между *do* и *end* выполняются принявшей вызов входа задачей, задача, вызвавшая вход, блокируется, т.е. это и есть критическая секция.

Кроме того, в операторе *select* в качестве альтернатив могут быть использованы конструкции *terminate*, *else* или *delay*. Например, альтернатива *Terminate* выбирается, когда больше нет задач, которые могут вызвать один из входов перечисленных в операторе *select*, при ее выборе задача завершается.

Рассмотрим решение задачи "поставщики - потребители" средствами рандеву языка Ада. Здесь для представления буфера используется задача MAILBOX, у которой определены две операции: SEND - послать сообщение (поместить в буфер), RECEIVE - получить сообщение (извлечь из буфера).

```
SUBTYPE MESSAGE IS STRING[100]; -- это тип сообщения

TASK MAILBOX IS -- описание задачи
  ENTRY SEND(INMAIL: IN MESSAGE);--тип связи относительно
```

```
                                        -- задачи MAILBOX
  ENTRY RECEIVE(OUTMAIL: OUT MESSAGE);
END TASK MAILBOX;

TASK BODY MAILBOX IS -- тело задачи
  SIZE: CONST INTEGER:=20; -- размер буфера
  BUFFER: ARRAY (1..SIZE) OF MESSAGE; -- буфер сообщений
  COUNT: INTEGER:=0; -- текущее кол-во сообщений
  NEXTIN, NEXTOUT: INTEGER:=1; -- указатели буфера
BEGIN
  LOOP
    SELECT
      WHEN COUNT<SIZE=> -- если буфер неполон
        ACCEPT SEND (INMAIL: IN MESSAGE) DO
          BUFFER(NEXTIN):=INMAIL;--информационный обмен
        END;
        NEXTIN:=NEXTIN MOD SIZE+1;
        COUNT:=COUNT+1;
      OR WHEN COUNT>0=> -- если буфер не пуст
        ACCEPT RECEIVE (OUTMAIL: OUT MESSAGE) DO
          OUTMAIL:=BUFFER(NEXTOUT); --информационный обме-
        END;
        NEXTOUT:=NEXTOUT MOD SIZE+1;
        COUNT:=COUNT-1;
    END SELECT;
  END LOOP;
END MAILBOX;
```

Почтовый ящик может принимать вызовы двух входов в заранее неизвестной последовательности. Поэтому, если принять какую-либо последовательность обслуживания вызовов входов, можно получить ситуацию тупика, когда процессы не смогут развиваться. В этом примере использован оператор *SELECT*, в котором определены две альтернативы. Оператор анализирует готовность соответствующей альтернативы, наличие вызова входа и истинность логического выражения в конструкции *when*. Если, по крайней мере, одна альтернатива готова к обслуживанию, то выполняется соответствующий оператор *ACCEPT*, сообщения от одной задачи передаются другой в рамках критической секции. Если нет готовых входов задача подвешивается до появления таковых. Аналогичную конструкцию альтернативного ввода далее мы рассмотрим и для языка ОККАМ (3.9.8).

Описание некоторых директив компилятора Ада-95

По сравнению с Ада-83 стандарт Ада-95 был расширен новыми директивами компилятора, задаваемыми ключевым словом *pragma*.

Так, директива компилятора *Locking_Policy* управляет блокировкой разделяемых ресурсов. В языке Ада протокол ICPP задается так:

```
pragma Locking_Policy(Ceiling_Locking);
```

Это единственная стратегия, предопределенная в любой реализации компилятора [28]. Другие стратегии также могут быть реализованы, их имена должны заканчиваться "_Locking", но стандарт их не определяет.

Директивы *Priority* и *Interrupt_Priority* используются для задач и обработчиков прерываний. Например:

```
task Controller is
  pragma Priority(10);
end Controller;
```

Директива *Task_Dispatching_Policy* позволяет выбирать стратегию диспетчирования. Так, *FIFO_Within_Priority* означает, что задачи назначаются в порядке очереди. Следовательно, когда задача становится готова к решению, она помещается в хвост очереди соответствующего приоритета. Имеется одно исключение из этого правила: когда задача прервана, то по активизации она помещается в голову очереди соответствующего приоритета.

3.9.8. Синхронизация в языке Оккам

Оккам - это язык параллельного программирования, разработанный для транспьютеров, предназначенных для создания распределенных мультипроцессорных систем. Поэтому, как в Ада, для обмена данными и синхронизации используется передача сообщений. В Оккаме все взаимодействия между процессами проводятся на основе однонаправленных синхронных небуферизуемых каналов. Пакеты данных, передаваемые по каналам, имеют структуру, задаваемую с помощью протоколов. Протоколы, используемые отправителем и приемником сообщения, должны совпадать.

Каналы определяются как переменные специального типа *CHAN*, и операция посылки значения выражения *a* в канал *ch* обозначается как *ch!a*, а получение сообщения из канала *ch* в переменную *b* обозначается *ch?b*.

Для иллюстрации работы каналов и синхронизации процессов рассмотрим процесс-мультиплексор, получающий данные с трех направлений и отправляющий их все в один выходной канал до тех пор, пока не придет сигнал по управляющему каналу Stop (рис. 3.18).

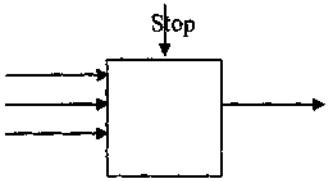


Рис. 3.18. Процесс-мультиплексор

```
BOOL Going:
SEQ
  Going:= TRUE
  WHILE Going
    BYTE char: -- спецификация процесса-блока
    INT any:
    ALT
      ALT index = 0 FOR 3
        InChan[index] ? char
          OutChan ! char
      Stop ? any
        Going:=FALSE
```

Следует иметь в виду, что стиль написания программы на языке Оккам важен. Одинаковый отступ означает, что соответствующие процессы (операторы) имеют один уровень вложенности. Вложенность процессов показывается отступом на 2 позиции вправо относительно охватывающего их процесса.

В приведенном выше примере сначала объявляется булевская переменная *Going*. Область видимости и существования переменной *Going* - внутри следующего за объявлением процесса *SEQ*, состоящего из двух процессов: присваивания значения *TRUE* переменной Going и составного процесса *WHILE*. Последний имеет только один процесс *ALT* с двумя локально определенными в нем (процессе ALT) переменными *char* типа *BYTE* и *any* типа *INT*. Использование здесь ключевых слов *char* и *any* для обозначения переменных возможно, так как это на самом деле не ключевые слова, потому что использованы малые буквы (ОККАМ, как и С, чувствителен к регистру).

В программе также использована конструкция репликатора или повторителя *ALT index = 0 FOR 3*, обеспечивающая в данном случае размножение частей процесса *ALT*: конструкция *FOR* порождает три части этого процесса одинаковой структуры, но отличающихся значением переменной index, которая принимает значения - 0,1,2. В развернутом виде это дает:

```
ALT
  InChan[0] ? char
    OutChan ! char
```

```
InChan[1] ? char
  OutChan ! char
InChan[2] ? char
  OutChan ! char
```

Такая репликация делается препроцессором, что приводит к модификации исходного кода перед компиляцией. Репликатор *FOR* похож на соответствующий цикл по форме, но это не цикл времени исполнения, а средство препроцессора, позволяющее кратко записывать части программы, незначительно отличающиеся друг от друга.

Процесс *ALT* срабатывает, когда, по крайней мере, одна из альтернатив готова, т.е. источник данных готов к передаче, причем в этом случае информация берется только из одного канала, из какого - зависит от реализации. Если ни один источник не готов, процесс *ALT* подвешивается до наступления этой готовности. Здесь предполагается, что канальные переменные InChan, Stop, OutChan описаны ранее, например, так

```
--указывается число элементов массива входных
--каналов и выходного с протоколом BYTE:
CHAN OF BYTE InChan[3] , OutChan:
-- определение управляющего канала с протоколом INT:
CHAN OF INT Stop:
```

Заметим, что определения в Оккаме всегда завершаются двоеточием.

Одной из разновидностей каналов в языке Оккам являются таймеры. Они также могут быть использованы для синхронизации и поддерживают простой протокол INT. Число таймеров не ограничено. Таймеры описываются так:

```
TIMER <имя таймера>:
```

Таймеры могут использоваться только для чтения и всегда готовы к вводу из них. Они выдают текущее значение счетчика, наращиваемого на единицу с каждым тиком физического таймера транспьютера (обычно с периодом в 1 мкс). Интервальный таймер реализован в Оккам конструкцией *AFTER*:

```
<имя таймера>?AFTER <выражение>
```

Здесь выражение определяет значение таймера, до достижения которого процесс должен быть подвешен.

Счетчик таймера по достижении максимального значения сбрасывается в 0 и затем снова наращивается. Как правило, в выражении интервального таймера используется модульная арифметика для принятия во внимание этого обстоятельства.

Следующий фрагмент кода может быть использован для подвешивания процесса на 1 минуту:

```
VAL TicksPerSecond IS 1000000:
VAL OneMinute IS 60*TicksPerSecond:
TIMER Clock:
SEQ
  Clock?Now
  Clock? AFTER Now PLUS OneMinute --сложение по модулю
```

В языке Оккам определены порты. Подобно таймерам, порты всегда считаются готовыми, но они могут использоваться для ввода и вывода информации определенного типа:

PORT OF type port:

Обычно порты служат для связи с внешними устройствами и имеют фиксированные адреса в оперативной памяти. Назначение адреса порту производится так:
PLACE port AT address:

В заключении следует сказать, что, несмотря на прекращение в настоящее время развития транспьютеров, они находят применение при реализации СРВ, а заложенные в них идеи получили развитие в современных процессорах.

3.9.9. Синхронизация в PVM

PVM (Parallel virtual machine) - это библиотека, обеспечивающая взаимодействие процессов в распределенных компьютерных системах - кластерах. Она ориентирована на обмен сообщениями между процессами, физически расположенными на разных машинах. Компьютеры, включенные в такую систему, образуют пул системных ресурсов, которые могут быть назначены созданным процессам или задачам.

Взаимодействие между задачами осуществляется с использованием их идентификационных номеров (task identifiers - TIDs). Заголовочный файл pvm3.h включается в программу для использования PVM. Запуск задач производится вызовом:

```
int  numt = pvm_spawn (
    char *task,  /* исполняемый файл                    */
    char **argv, /* аргументы командной строки          */
    int  flag,   /* опция запуска     0 - PVMTaskDefault -
                    PVM решает сама,
                    где запустить задачу*/
    char *where, /* место для запуска                   */
    int  ntask,  /* число запускаемых копий             */
    int  *tids   /* массив tids запущенных задач        */
   );
```

Для получения сообщений используется функция:

```
int  bufid = pvm_recv ( int tid, int msgtag );
```

Эта функция осуществляет блокирующее получение сообщения, т.е. ожидает прихода сообщения с тегом *msgtag* от задачи с идентификатором *tid*, затем создает новый буфер получения *bufid*, кладет туда полученное сообщение и завершается. Если *tid = -1*, то любое сообщение с тегом *msgtag*, посланное этой задаче, принимается ею; если *msgtag = -1*, то принимается любое сообщение, посланное данной задаче от задачи с идентификатором *tid*. Если *tid = msgtag = -1*, то принимается любое сообщение от любой задачи. Для отправки используется функция pvm_send() с таким же интерфейсом. Однако отправка сообщений не блокирует отправителя, он остается активным.

Функция

```
int info = pvm_bufinfo ( int bufid, int nbytes, int msgtag,
int tid )
```

возвращает значения *tid, msgtag* и длину в байтах *nbytes* для сообщения в буфере *bufid*. Эту функцию разумно использовать при неизвестном идентификаторе задачи и/или теге сообщения (*tid = -1*, *msgtag = -1*).

Для распаковки содержимого принятого буфера используется функция:

```
int  info = pvm_upkint( int *ip, int nitem, int stride )
```

Эта функция обратна функции упаковки pvm_pkint() с таким же интерфейсом.

Функция *pvm_initsend(PvmDataDefault)* должна быть использована перед отправкой каждого сообщения. Значение *PvmDataDefault* означает, что при кодировании данных в буфере различия в представлении данных на отправляющей и принимающей сторонах учитываются автоматически. Функции упаковки, распаковки и инициализации отправки данных используются для того, чтобы предоставить возможность работы PVM на гетерогенных кластерах, состоящих из вычислительных узлов разной архитектуры.

Одним из вариантов синхронизации в PVM является возможность блокирующего приема сообщения от другого процесса. В этом случае процесс, желающий принять информацию, будет находиться в состоянии ожидания до тех пор, пока такая информация не поступит, что и обеспечивает синхронизацию.

Отметим, что в отличие от Оккама и Ада в PVM посылка сообщений не приводит к блокировке посылающего процесса.

Идеология обмена сообщениями является одной из старейших и наиболее эффективных в случае распределенных систем. Аналогичные PVM идеи заложены и в широко распространенной библиотеке MPI

3.10. Тупики

Как уже отмечалось ранее, ***тупиком*** называется ситуация в системе, при которой ряд процессов не может развиваться. Напомним, что тупики бывают гло-

бальные, когда блокированы все процессы, и локальные, когда блокирована некоторая группа процессов.

Наличие тупика может привести к краху СРВ в связи с нарушением временных ограничений.

Существуют следующие необходимые условия возникновения тупиков:

- в системе имеются параллельные процессы;
- процессы используют общие критические ресурсы;
- процессы требуют во время своей работы дополнительные ресурсы;
- если ресурсов, нужных процессу, нет, то он ждет их, не освобождая ранее захваченные ресурсы.

Можно выделить несколько основных методов борьбы с тупиками:

- Обнаружение и устранение. Система отслеживает состояния процессов и в случае обнаружения тупика разрешает его, завершая некоторый процесс.
- Исключение тупиков. Например, перед началом работы процесс получает все необходимые ему ресурсы и ничего дополнительно не требует за все время своей работы.
- Обход тупиков. Это - компромиссный вариант, в нем дополнительные ресурсы предоставляются, только если новое состояние ресурсов будет гарантировать отсутствие тупиков.

Рассмотрим перечисленные методы подробнее.

3.10.1. Обнаружение тупиков

Для обнаружения тупиков используются ресурсные графы (рис. 3.19), т.е. графы с вершинами двух типов: процессы - ▭ и ресурсы - ○. Количество единиц ресурса указывается в кружке числом или точками по числу единиц ресурса.

Вершины разного типа соединяются дугами. Дуга идет от процесса к ресурсу, если процесс требует некоторого ресурса и наоборот, если ресурс ему предоставлен.

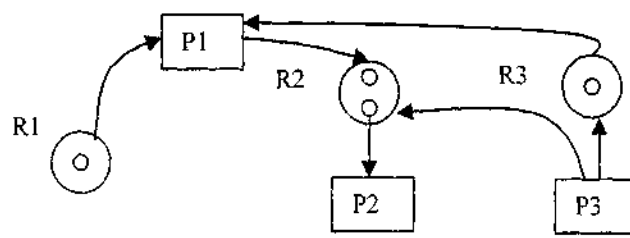


Рис. 3.19. Пример ресурсного графа

Тупик представляется циклом на ресурсном графе. Для обнаружения тупика используется процедура редукции ресурсного графа. Если после применения про-

цедуры редукции все ребра графа будут удалены, то тупика нет. В противном случае оставшиеся ребра образуют цикл, представляющий тупик. На рис. 3.19 процесс P1 имеет единицу ресурсов R1, R3 и требует единицу ресурса R2, процесс P2 имеет единицу ресурса R2 и ничего не требует, процесс P3 требует по единице ресурсов R2, R3 и ничего не имеет. Всего в системе 1 единица ресурса R1, 2 единицы ресурса R2, 1 единица ресурса R3.

Процедура редукции ресурсного графа такова: если ресурсы процессу предоставлены или могут быть предоставлены (они имеются в наличии в данный момент), или никакие ресурсы процессу не предоставлены, то все ребра, инцидентные этому процессу, могут быть удалены. Удаляются или все ребра, или ни одного! Редукция производится последовательно по всем процессам.

На рис. 3.19 можно убрать ребра, инцидентные P3, так как этот процесс не имеет ресурсов; ребра, инцидентные P1, так как имеется единица свободного ресурса R2; ребро, инцидентное P2, так как он не имеет требований. Все ребра при этом пропадают, т.е. тупика в данном случае нет. Рассмотрим теперь следующую ситуацию, представленную на рис. 3.20.

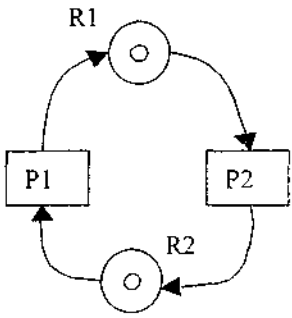


Рис. 3.20. Ресурсный граф с тупиком

P1 требует единицу R1 и уже получил R2, P2 требует единицу R2 и уже получил R1. Граф не может быть редуцирован, так как ни для одного из процессов его требование не может быть выполнено, т.е. мы имеем тупик.

Следует отметить, что из графа ресурсов может быть получен граф ожидания задач (ГОЗ), в котором присутствуют только вершины одного типа - процессы, а ребро направлено от процесса P1 к процессу P2 в случае, если P1 ожидает освобождения ресурсов, занятых P2.

Методы обнаружения тупиков могут быть классифицированы как ***централизованные*** и ***децентрализованные***.

Для централизованных методов характерно хранение ресурсного графа на некотором процессоре (вычислительном узле) и обработка его этим же процессо-

ром. Преимущество подхода - простота, недостаток - малая надежность и производительность, так как единственный процессор становится узким местом.

При децентрализованном подходе в обнаружении тупика принимают участие несколько процессоров системы. К недостаткам можно отнести значительную сложность реализации, к достоинствам - высокую надежность и производительность.

Рассмотрим ряд алгоритмов обнаружения тупиков.

Алгоритм Голдмана (Goldman)

Процессы обмениваются информацией в форме упорядоченных списков блокированных процессов (УСБП), в которых каждый процесс блокирован своим последователем. Считается, что процесс А блокирован другим процессом В, если последний использует критические ресурсы, требуемые процессу А.

На рис. 3.21 показан ГОЗ: Р1 ждет Р2, который ждет Р3 и т.д. Последний процесс Р5 исполняется. Когда он освободит ресурс, нужный Р4, тот возобновится. Показаны также компьютеры, на которых исполняются процессы: Р1, Р2 - на одном, Р3, Р4 - на другом, Р5 - на третьем. На этом графе нет цикла, следовательно нет тупика.

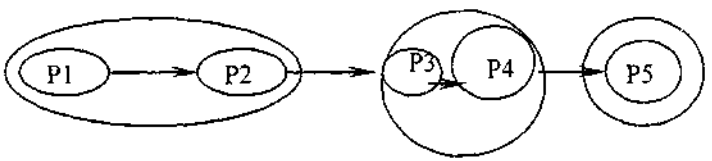


Рис. 3.21. Пример графа ожидающих задач для алгоритма Голдмана

Если Р1 инициирует процедуру обнаружения тупика, он пошлет Р2 список (Р1, Р2); Р2 добавит Р3 к нему и пошлет список процессу Р3, и т.д. Каждый процесс после получения УСБП анализирует его, пытаясь найти себя в нем дважды. Процесс, обнаруживший себя дважды в УСБП, фиксирует тупиковую ситуацию. В случае, если бы процесс Р5 был блокирован процессом Р1, то Р1 обнаружил бы тупик после получения УСБП от Р5.

Алгоритм Чэнди-Мисра-Хааса (Chandy-Misra-Haas)

Процессы обмениваются специальными пакетами - триплетами (i, j, k), где

i - номер процесса-отправителя;

j - номер компьютера, где находится процесс-отправитель;

k - номер узла (компьютера), где располагается блокирующий процесс-получатель.

Один из процессов инициирует проверку наличия тупика, посылая первый триплет блокирующему его процессу. Обнаружение тупика фиксируется в случае

получения триплета процессом-инициатором обратно, что иллюстрируется на рис. 3.22, на котором изображены три узловых компьютера и 10 процессов.

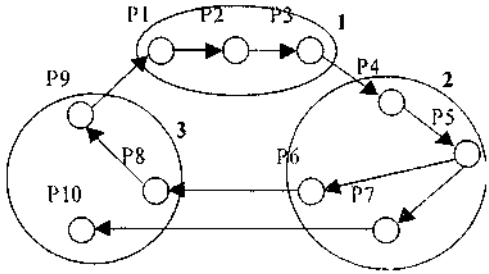


Рис. 3.22. Пример графа ожидающих задач для алгоритма Чэнди-Мисра-Хааса

Процесс P1 инициирует обнаружение тупика посылкой триплета (1,1,1) блокирующему его процессу P2 (покажем номера процессов приемников и получателей, разделяя их стрелкой, а в скобках - посылаемый триплет):
1→2(1,1,1), 2→3(2,1,1), 3→4(3,1,2), 4→5(4,2,2). 5→6(5,2,2), 5→7(5,2,2), 6→8(6,2,3), 8→9(8,3,3), 9→1(9,3,1), 7→10(7,2,3)

Процесс-инициатор P1. получив триплет (9,3,1), фиксирует тупик. Однако процессы, вовлеченные в тупик, в отличие от предыдущего алгоритма, подлежат определению путем обратного прохода по цепочке блокированных процессов.

Алгоритм Митчела-Меррита (Mitchell-Merritt)

Каждая вершина (процесс) ГОЗ получает две метки: локальную и глобальную. Локальные метки уникальны. Изначально глобальные метки равны локальным. Алгоритм обнаруживает тупик обратным распространением глобальных меток (в сторону, противоположную направлению ребер графа). Процесс, получив глобальную метку, сравнивает ее со своей глобальной меткой и максимальное из этих значений становится его новой глобальной меткой.

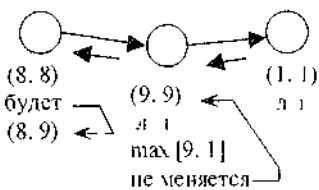


Рис. 3.23. Иллюстрация работы алгоритма Мичела-Меррита

Тупик обнаруживается, когда процесс получает собственную глобальную метку; эта метка имеет максимальное значение среди процессов, вовлеченных в тупик. Процесс, обнаруживший тупик, разрешает его, завершая себя. На рис 3.23

показана часть графа с рис. 3.22: только 3 процесса P8, P9, P1. Локальные метки помечены буквой "л", глобальные - "г". Если процесс P1 опять инициирует обнаружение тупика, он посылает свою глобальную метку 1 предшествующему процессу P9, у которого при этом глобальная метка не меняется. Далее, P9 посылает свою глобальную метку 9 предшествующему процессу P8, у которого глобальная метка меняется с 8 на большее значение 9. Затем P8 шлет свою новую глобальную метку процессу P7 и т.д. Когда максимальная в цикле метка 9 вернется процессу P9, он зафиксирует наличие тупика.

Алгоритм Брача-Тоег (Bracha-Toueg)

Алгоритм имеет две фазы:

1) сообщения-запросы на ресурсы распространяются вниз по ГОЗ (по направлению ребер);

2) сообщения о предоставлении ресурсов возвращаются обратно от активных процессов, активизируя процессы-получатели таких сообщений.

Процессы, не ставшие активными к концу 2-й фазы, считаются находящимися в тупике.

Иерархический подход - алгоритм Хо-Рамамурти (Ho-Rumamoorthy)

Компьютеры группируются в непересекающиеся кластеры. Периодически один из компьютеров назначается основным, и он выделяет главные компьютеры в кластерах. Основной компьютер запрашивает от главных компьютеров кластеров информацию о межкластерных транзакциях и внутрикластерных блокировках. Таким образом, главные компьютеры кластеров пытаются обнаружить тупики внутри кластеров, а главный компьютер системы пытается обнаружить межкластерные тупики.

Проблемы, связанные с обнаружением тупиков

С задачей обнаружения тупиков связан ряд проблем:

1. Доказательство корректности алгоритмов сложно из-за очень большого числа возможных комбинаций взаимодействий процессов и чувствительности тупиков к временным параметрам.
2. Проблема оценки производительности. Неясно, что использовать в качестве меры производительности: количество сообщений, которыми обмениваются вершины для обнаружения циклов; объем передаваемых сообщений; время существования тупиков; затраты памяти на хранение передаваемых сообщений.
3. Проблема разрешения тупиков. Тупик может быть разрешен завершением любого процесса, вовлеченного в тупик. Процесс, обнаруживший тупик, может не знать, что другие процессы также обнаружили этот же тупик. Это может вести к завершению нескольких процессов, что не

эффективно. Следовательно, после обнаружения тупика нужен дополнительный обмен информацией. Для разрешения тупика необходимо:

1) Выбрать жертву (какой процесс прервать).
2) Прервать выбранный процесс, освободить все удерживаемые им ресурсы, восстановив при этом их прежние значения.
3) Удалить информацию о жертве из специальных информационных структур, используемых для определения тупика (например, из ГОЗ).

4. Ложные тупики. Рассмотрим ГОЗ на рис. 3.24. Предположим, что верхний цикл обнаружен процессом Т3, разорвавшим его удалением ребра (Т4, Т5); одновременно Т8 мог также обнаружить тупик, и могло случиться так, что процесс Т3 убрал (Т4,Т5) после того, как сообщение об обнаружении тупика, инициированное Т8, прошло через него. Это может привести к разрешению нижнего тупика (цикла), которого уже к этому времени реально нет.

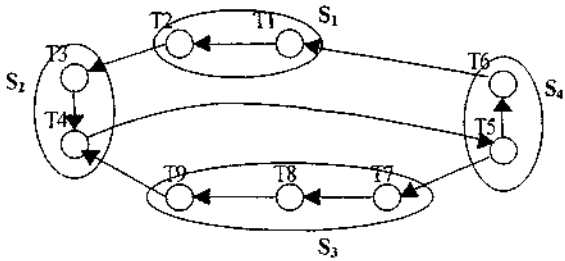


Рис. 3.24. Пример графа ожидающих задач с возможными ложными тупиками

3.10.2. Методы исключения тупиков

Можно выделить два основных метода, направленных на исключение возникновения тупиков:

1. Процесс не запускается, пока ему не предоставлены все требуемые ресурсы, причем он в дальнейшем не требует дополнительных ресурсов вплоть до своего завершения.
2. Процесс, не получивший требуемый ресурс, прерывается с освобождением предоставленных ему ресурсов.

3.10.3. Обход тупиков

Одним из методов обхода тупиков является алгоритм Хабермана (Habermann). Рассмотрим его подробнее.

Предполагается, что имеется N процессов $P_1,\ldots P_N$, которые требуют ресурсы. Также есть *a* единиц ресурсов. Для каждого процесса указан вектор $b=(b_1, \ldots$

b_N), каждый компонент которого содержит максимальный запрос к ресурсам соответствующего процесса, и вектор $c=(c_1, \ldots, c_N)$, который содержит текущее количество ресурсов у процессов. Здесь мы рассматриваем случай ресурсов одного типа, запросы представляются векторами, доступные ресурсы - скаляром, но алгоритм исходно сформулирован для ресурсов многих типов, когда запросы представляются матрицами, доступные ресурсы - вектором. В обоих случаях формально алгоритм выглядит одинаково, но первый случай проще для рассмотрения, поэтому мы на нем и остановимся.

Ресурсное состояние называется ***реализуемым***, если выполняются следующие соотношения:

$$\begin{cases} (\forall k)(bk \le a) & (3.1) \\ c \le b & (3.2) \\ \sum_{i=1}^{N} c_i \le a & (3.3) \end{cases}$$

Количество свободных ресурсов $r(t) = a - \sum_{i=1}^{N} c_i$ (t - текущее время) для реализуемых состояний удовлетворяет:

$$r(t) \ge 0 \qquad (3.4)$$

Если реализуемое состояние гарантирует отсутствие тупиков, оно называется ***безопасным***, иначе - ***опасным***. Реализуемое состояние (a,b,c) безопасно, если существует такая последовательность $S = (P_{i_1}, P_{i_2}, \ldots, P_{i_N})$ процессов, что:

$$\left(\forall P_{i_j} \in S\right)\left(b_{i_j} \le \left[r(t) + \sum_{k<j} c_{i_k}(t)\right]\right) \quad j = \overline{1, N} \qquad (3.5)$$

Такая последовательность S называется безопасной. Условие (3.5) говорит о том, что требование процесса P_{i_j} на ресурсы не должно превосходить суммы свободных ресурсов и суммарного количества тех ресурсов, которые освободит предшествующие ему в последовательности процессы.

Важность концепции безопасности в том, что, отправляясь из безопасного состояния, можно без тупиков выделить ресурс всем процессам даже тогда, когда каждый процесс захватывает все затребованные им ресурсы и не освобождает их до своего завершения, если удовлетворять их запросы на дополнительные ресурсы в порядке следования процессов в безопасной последовательности S.

Большое значение имеет также утверждение: если состояние ресурсов безопасное и частичная последовательность S процессов удовлетворяет (3.5), она может быть расширена. Это значит, что если последовательность не расширяется, то состояние не безопасное, и не надо проводить дополнительные переборы. Это

важно, так как количество разных последовательностей N процессов - N!, и дает возможность сформулировать нетрудоемкий алгоритм определения безопасной последовательности.

Алгоритм определения безопасной последовательности

1. $S=\varnothing$. Пустая последовательность удовлетворяет (3.5).
2. Пытаемся расширить S так, чтобы она удовлетворяла (3.5) после присоединения к ней некоторого процесса P_k. Эта попытка может оказаться неуспешной по причине: a) все процессы уже в S, что означает безопасность состояния: b) ни одно расширение не удовлетворяет (3.5), что означает не безопасность состояния.

Рассмотрим этот алгоритм более детально: пусть S - частично сформированная последовательность; S* - дополнение S (т.е. все процессы, не попавшие в S); T - множество кандидатов на включение в S (T и S* связаны); R - количество свободных ресурсов.

```
{
  S=∅;
  T=S*;
  while (T<>∅) {
    //1-й элемент берется как кандидат на включение:
    Candidate=T[1];
    //исключаем его из множества не рассмотренных
    //кандидатов:
    T=T-{T[1]};
    //проверяем (3.5):
    if (B[Candidate]-C[Candidate]≤R-∑c[i]){
      S=S∪{ Candidate };//расширяем, если (3.5) выполнено
      T=S*;//пересчитываем множество не рассмотренных
           //кандидатов
    }
  }
  SAFE=(S*==∅);//безопасно, если дополнение пусто
}
```

Порядок появления запросов может не соответствовать порядку процессов в S. В таком случае определение безопасности может проводиться для каждого запроса. Приведенный выше алгоритм может быть упрощен благодаря следующему утверждению: если безопасное состояние модифицируется назначением дополнительного ресурса процессу P_k, и если некоторая последовательность S (быть мо-

жет, частичная) содержит P_k и удовлетворяет (3.5), то новое состояние также безопасно.

Пример использования алгоритма Хабермана

Рассмотрим пример. Пусть число процессов N=5, a=10 - количество системных ресурсов, b - вектор максимальных запросов на ресурсы, c - вектор предоставленных процессам ресурсов:

$$b=\begin{pmatrix}5\\8\\7\\6\\9\end{pmatrix},\quad c=\begin{pmatrix}2\\3\\1\\1\\3\end{pmatrix}.$$

Безопасно ли это состояние (a, b, c)? Реализуемо ли оно?
2+3+1+1+3=10 ➔ условия (3.1)-(3.3) выполнены ➔ состояние реализуемо.
Проверим это состояние на безопасность.
S=∅, R=10-(2+3+1+1+3)=0
T=S*: T={1,2,3,4,5}
Candidate=1
T={2,3,4,5}
B1-C1=5-2=3≤0; Нет
T={2,3,4,5}
Candidate =2
T={3,4,5}
B2-C2=8-3=5≤0; Нет
T={3,4,5}
Candidate =3
T={4,5}
B3-C3=7-1=6≤0; Нет
T={4,5}
Candidate =4
T={5}
B4-C4=6-1=5≤0; Нет
T={5}
Candidate =5
T=∅ -- пусто
B5-C5=9-3=6≤0; Нет
T=∅
S*≠∅ - не пусто.

Следовательно, это состояние небезопасно.

Рассмотрим предыдущую систему с видоизмененными начальными состояниями:

$$b = \begin{pmatrix} 5 \\ 8 \\ 7 \\ 6 \\ 9 \end{pmatrix}, \quad c = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 2 \end{pmatrix}.$$

Состояние по-прежнему реализуемо, что нетрудно проверить. Удостоверимся, что оно безопасно.

B1-C1=5-1=4≤4+0 - Да
S=∅ ∪ {1}={1}
T={2,3,4,5}
Candidate=2
T={3,4,5}
B2-C2=8-1=7≤4+1; Нет
T={3,4,5}
Candidate =3
T={4,5}
B3-C3=7-1=6≤4+1; Нет
T={4,5}
Candidate =4
T={5}
B4-C4=6-1=5≤4+1; Да
S={1,4}
T={2,3,5} - Дополнение S
Candidate =2
T={3,5}
B2-C2=8-1=7≤4+1+1; Нет
T={3,5}
Candidate =3
T={5}
B3-C3=7-1=6≤4+1+1; Да
S={1,4,3}
T={2,5} - Дополнение S
Candidate =2
T={5}
B2-C2=8-1=7≤4+1+1+1; Да
S={1,4,3,2}

Candidate =5

T=∅ - Пустое

B5-C5=9-2=7≤4+1+1+1+1; Да

S={1,4,3,2,5}

T=0

S*=∅ - пусто

Следовательно, состояние безопасно. Рассмотрим развитие процессов в динамике.

Таблица 3.5

Временные параметры выделения ресурсов процессам

P1	$\frac{(0,3)}{1}$	$\frac{(3,7)}{3}$	$\frac{(7,11)}{5}$	$\frac{(11,\infty)}{0}$	
P2	$\frac{(0,5)}{1}$	$\frac{(5,11)}{4}$	$\frac{(11,14)}{5}$	$\frac{(14,17)}{8}$	$\frac{(17,\infty)}{0}$
P3	$\frac{(0,7)}{1}$	$\frac{(7,10)}{6}$	$\frac{(10,13)}{7}$	$\frac{(13,\infty)}{0}$	
P4	$\frac{(0,2)}{1}$	$\frac{(2,9)}{3}$	$\frac{(9,12)}{6}$	$\frac{(12,\infty)}{0}$	
P5	$\frac{(0,6)}{2}$	$\frac{(6,12)}{4}$	$\frac{(12,15)}{7}$	$\frac{(15,20)}{9}$	$\frac{(20,\infty)}{0}$

В таблице 3.5, например, $\frac{(0,3)}{1}$ в первой строке, означает, что P1 требует единицу ресурса от 0 до момента 3 (в отсутствие других процессов). Так как реально процессы должны работать совместно, могут быть задержки в назначении ресурсов в связи с их отсутствием в данный момент, но длины интервалов времени, в течение которых процесс работает на одних и тех же ресурсах, инвариантны. Безопасная последовательность в соответствии с начальными запросами процессов к ресурсам (1,4,3,2,5), что было получено ранее. График выделения ресурсов процессам приведен на рис. 3.25.

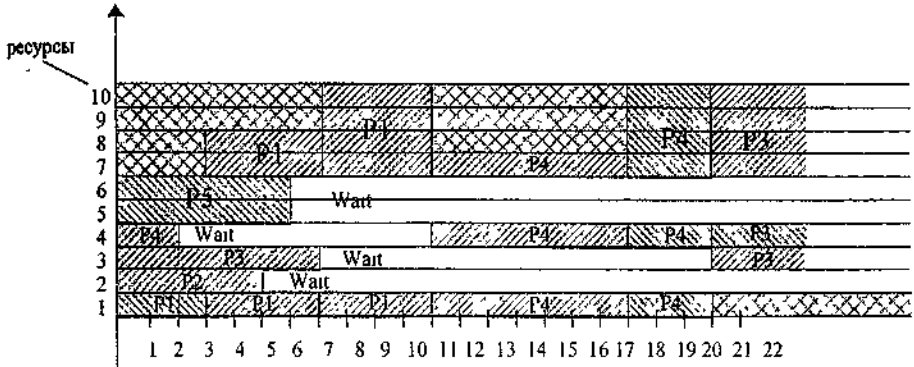


Рис. 3.25. График выделения ресурсов процессам (неполный)

Дополнительные ресурсы назначаются в соответствии с безопасной последовательностью (1, 4, 3, 2, 5). В момент 3 P1 запросит дополнительно 2 единицы ресурса и получит их, так как он первый в последовательности. А процесс P4, запрашивающий дополнительные ресурсы в момент 2, блокируется, так как он не первый в последовательности, хотя возможность удовлетворения его запроса имеется (4 свободных единицы ресурса, а требуются - 2) На рис. 3.25 для обозначения состояний процессов и ресурсов использованы обозначения:

 – выполнение P

Wait – ожидание

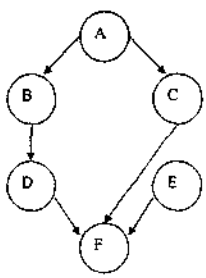
 – свободен

Заметим, что подобная идея реализуется в алгоритме банкира - предоставлять ресурсы, если результирующее состояние исключает возникновение тупиков.

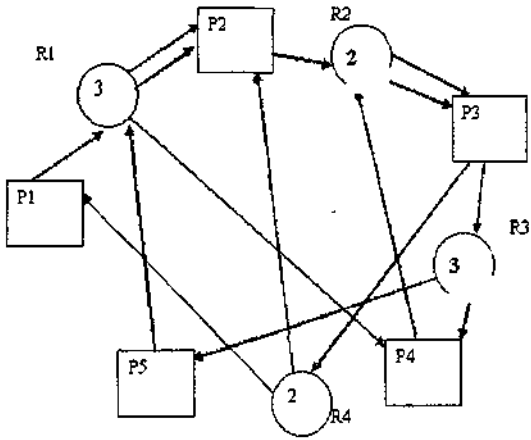
Контрольные вопросы и задания

1. Что такое процесс, нить управления, программный счетчик, блок управления процессом?
2. Как организуется последовательное выполнение процессов?
3. Что такое кооперативная и вытесняющая многозадачность?
4. В каких основных состояниях может находиться процесс?
5. Как организуется обслуживание процессов без приоритетов и с приоритетами?
6. Что такое прерывания? Для чего они нужны? Как обрабатываются прерывания на процессорах семейства x86?
7. Напишите выражение для вычисления исполнительного (физического) адреса обработчика прерывания 08h, используя его вектор прерывания

8. Что такое семафорные операции? Какова их область применения?
9. Какие типовые задачи синхронизации Вы знаете?
10. Охарактеризуйте механизм семафоров.
11. Как посредством семафоров можно решить задачу "обедающие философы"?
12. Что такое реентерабельность?
13. Дайте классификацию механизмов синхронизации процессов. Как работает алгоритм Дейкстры синхронизации N процессов?
14. Опишите механизмы событий, критических областей, мониторов и управляющих выражений.
15. Напишите управляющее выражение, описывающее следующую схему вычислений:



16. Для чего вводится наследование приоритетов? Что такое инверсия приоритетов? Вследствие чего она может проявиться?
17. Опишите основные управляющие структуры языка Ада.
18. В чем особенность использования ссылочных типов данных в языке Ада?
19. Как в языке Ада описываются подпрограммы?
20. Как обрабатываются ошибки в языке Ада.
21. Каким образом организуется многозадачность в языке Ада? Что такое рандеву?
22. Как производится синхронизация процессов в языке Оккам?
23. Что такое тупики? Какими они бывают? Сформулируйте основные условия возникновения тупиков. Какие методы борьбы с тупиками Вы знаете?
24. Что такое ресурсный граф? Как он может помочь при обнаружении тупика?
25. Пусть текущее ресурсное состояние (процессы P1..P5, ресурсы R1..R4) представлено следующим ресурсным графом:



Определите, используя процедуру редукции, является ли текущее состояние тупиком. Если тупик существует, какие процессы вовлечены в него?

26. Постройте граф ожидания задач на основе графа из задания 25. Виден ли на нем тупик?
27. Опишите алгоритмы Голдмана и Чэнди-Мисра-Хааса.
28. Как работают алгоритмы Митчела-Меррита, Брача-Тоег, Хо-Рамамурти?
29. Какие есть проблемы при обнаружении тупиков?
30. Какие методы исключения тупиков Вы знаете?
31. В чем суть алгоритма Хабермана обхода тупиков? Пусть количество доступных единиц ресурса a=10, число параллельных процессов n=4, их текущие запросы c=(5,2,1,1), их максимальные запросы b=(7,10,8,2). Найдите безопасную последовательность процессов, покажите все шаги ее нахождения.

Глава 4. Управление памятью

Память принято делить на два основных класса: оперативная и внешняя.

Оперативная память отличается высоким быстродействием, как правило, энергозависима и имеет относительно небольшие объемы. В ней хранятся данные и исполняемый код, именно с ней напрямую работает процессор и периферийные устройства.

Внешняя память, как правило, используется для долговременного хранения больших массивов информации, может располагаться как на сменных (дискеты, оптические диски, ленты и т.п.), так и на встроенных носителях (жесткие диски). Внешняя память отличается значительными объемами и низким быстродействием по сравнению с оперативной памятью.

Эффективное управление оперативной и внешней памятью является важной задачей при построении СРВ.

4.1. Управление памятью в однозадачном режиме

Для однозадачных систем можно выделить такие части оперативной памяти, как:

- сегмент кода;
- сегмент данных;
- куча (динамически выделяемая память);
- стек.

Эти области памяти располагаются последовательно от младших адресов к старшим. Сегменты кода и данных распределяются статически, их размеры не меняются во время исполнения программы. Сегмент кода используется для хранения исполняемого машинного кода программы. Сегмент данных предназначен для хранения глобальных структур данных программы.

Сегменты кучи и стека используются для динамического выделения памяти. Изначально они пустые, но во время своего выполнения программа может выдавать запросы на дополнительную память в стеке и/или в куче.

Запросы на стек появляются при вызове процедур или функций: фактические параметры и локальные переменные вызываемых функций размещаются в стеке. По завершении подпрограммы соответствующее место в стеке освобождается. Стек управляется по дисциплине LIFO (Last-In-First-Out). Он растет от старших адресов к младшим и управляется с помощью указателя стека, адресующего последнее помещенное в стек слово. В процессорах Intel указателем стека является регистровая пара SS:SP. Когда данные помещаются в стек, указатель стека уменьшается на размер этих данных; если он становится меньше минимально допустимого адреса - верхнего адреса кучи, возникает прерывание по переполнению стека. Это может случиться, например, в случае использования рекурсивных вызовов

подпрограмм большого уровня вложенности. Когда данные считываются из стека, указатель стека просто увеличивается на размер считанных данных, сами данные при этом физически не уничтожаются. Таким образом, свободное пространство стека представляет собой непрерывный участок памяти от первого байта выше кучи до первого байта, ниже адресуемого указателем стека, так как порядок освобождения памяти противоположен ее выделению.

Куча нужна для хранения данных, размеры которых неизвестны во время компиляции. Для управления этой памятью используются специальные функции или операторы, например, malloc(), calloc() в языке C, процедура new в Pascal, оператор new в Ada, C++. Данные, размещенные в куче, могут быть удалены в произвольном порядке, не обязательно противоположном их размещению. Это приводит к фрагментации кучи, и, несмотря на наличие достаточно большого суммарного свободного объема кучи, возможность удовлетворения очередного запроса будет определяться размером наибольшего непрерывного участка свободной памяти в куче. Например, запрос, требующий 70К памяти не сможет быть удовлетворен, если в куче свободные участки имеют размеры 30К, 40К, 50К, 60К. Всего свободной памяти 180К, но непрерывный участок памяти максимального размера - 60К, что меньше требуемых 70К.

Информацию о состоянии кучи можно получить стандартными функциями. Например, в Borland C++ 3.1 функция coreleft() возвращает размер свободного участка кучи выше последнего размещенного в ней блока, т.е. размер непрерывного участка непосредственно под стеком.

Если объем доступной памяти достаточно велик, а нужно минимизировать время выполнения программы, можно использовать встроенные вызовы функций (inline). В этом случае вместо передачи управления коду функции происходит его подстановка непосредственно в точку вызова, и исходный текст фактически расширяется кодом функции. Это ведет к увеличению размера программы, но может способствовать уменьшению времени ее работы, что существенно для приложений реального времени. Ниже дан пример объявления функции для расширения:

```
inline int max( int a , int b )
{
if( a > b ) return a;
return b;
}
```

В случае ограниченной памяти и больших программ памяти может оказаться недостаточно. В таких ситуациях могут использоваться оверлеи. Применение оверлеев позволяет части программы, не нужные в данный момент, заменять другими. Корневой оверлейный сегмент, вызывающий остальные сегменты, должен существовать в памяти постоянно. Для описания оверлейной структуры используются представления программ в виде деревьев вызовов, в чем-то напоминающих

диаграммы Джексона. Размер необходимой памяти при использовании оверлеев определяется максимальным путем на дереве вызовов, в то время как при обычном подходе размер памяти определяется суммой длин всех путей. В качестве меры длины используется размер памяти, необходимой для соответствующей процедуры в дереве вызовов.

Многозадачность в простейшем случае может быть организована свопированием: текущая программа вытесняется из памяти на диск с запоминанием ее состояния; очередная программа загружается с диска и работает определенный квант времени, затем все повторяется. Такой подход прост, но малоэффективен из-за больших накладных расходов на работу с внешней памятью.

4.2. Управление памятью в многозадачном режиме

Существует несколько основных подходов при распределении памяти в многозадачном режиме:

1. С ***фиксированными разделами*** и ***абсолютной адресацией*** - память разделена на заданное число разделов одинаковой длины; задача транслируется в абсолютных адресах с указанием номера раздела. Такой подход, во-первых, фиксирует максимальное количество задач, во-вторых, обладает низкой гибкостью из-за абсолютной адресации, что может привести к тому, что, несмотря на наличие достаточного количества памяти, некоторая задача не сможет выполняться, так как ее раздел занят другой задачей.
2. С ***фиксированными разделами***, но ***с относительной адресацией*** - модули перемещаемы, но число одновременно находящихся в памяти задач фиксировано, что является несомненным недостатком. Кроме того, небольшая задача будет расходовать столько же памяти, сколько и объемная, т.е. память используется не оптимально.
3. С ***переменными разделами*** - модули перемещаемы, число разделов не фиксировано, размер раздела равен размеру задачи. Данный подход устраняет недостатки, присущие предыдущим, но и у него есть проблемы. Так, свободные участки памяти могут появляться из-за завершения некоторых задач в середине занятого участка, что ведет к фрагментации памяти, т.е., как и в случае с кучей однозадачных систем, суммарный объем свободной памяти может быть большим, а разместить очередную задачу нельзя, так как нет непрерывного участка памяти достаточного размера. С этой проблемой можно бороться периодической сборкой мусора, называемой дефрагментацией, что требует перемещения задач в памяти, и на это обычно достаточно большое время система будет в нерабочем состоянии, что может быть неприемлемо для приложений реального времени.

4. ***Виртуальная память*** - подход к использованию физической оперативной памяти, имеющей ограниченный объем, как к неограниченной, с помощью внешней памяти. Так, если на адресацию отводится 32 разряда, то имеется возможность подключения 4 Гб виртуальной памяти. Задачи работают в виртуальном адресном пространстве, не заботясь, действительно ли части их кода и данных находятся в физической памяти. За перемещение памяти задачи между оперативной и внешней памятью отвечает операционная система.

Чаще всего в современных системах применяется виртуальная память. Рассмотрим различные варианты ее организации.

4.2.1. Страничная организация виртуальной памяти

Страничная организация виртуальной памяти предполагает разбиение всего адресного пространства на блоки одинаковой длины - ***страницы***.

Страницы оперативной (физической) памяти подобны страницам виртуальной памяти, но их меньше. Адрес в виртуальной памяти представляется парой (p,d) - номер страницы и смещение в ней, соответственно. Для адреса в оперативной памяти принято обозначение - (p',d). Страницы виртуальной памяти подгружаются в страницы оперативной памяти. Так, на рис. 4.1 виртуальный адрес (2,10) преобразуется в физический (3,10).

Рис. 4.1. Страничная организация виртуальной памяти (показаны номера страниц)

Для каждого процесса имеется таблица страниц доступной ему виртуальной памяти. Регистр базового адреса указывает на начальный адрес этой таблицы (рис. 4.2). Каждая строка таблицы содержит: бит присутствия страницы в оперативной памяти; номер физической страницы, соответствующей виртуальной странице. Если бит присутствия равен нулю, то p' содержит расположение виртуальной страницы p на внешнем носителе. Строка i таблицы соответствует i-й виртуальной странице.

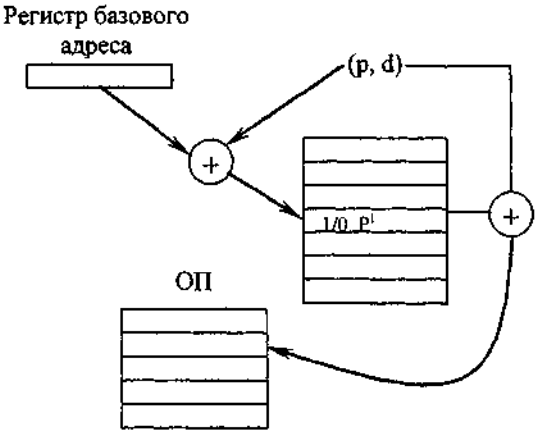


Рис. 4.2. Формирование адреса при страничной организации памяти

Если страница отсутствует в оперативной памяти, то возникает прерывание по отсутствию страницы, после чего следует поиск страницы во внешней памяти и ее загрузка в память оперативную. Заметим, что загрузка страницы в оперативную память может быть связана с вытеснением страниц, уже присутствующих в ней. После загрузки нужной страницы выполнение процесса возобновляется.

Страничная организация памяти позволяет нескольким процессам совместно использовать одни и те же страницы, что минимизирует необходимую память при исполнении процессами одной и той же программы.

При использовании страничной адресации возникают проблемы:

- Необходимо выбрать, какими страницами виртуальной памяти загружать оперативную память. Если выбор неверен, то это может привести к дополнительным операциям подкачки страниц, которые выполняются достаточно медленно.
- Следует решить, каков должен быть размер страниц. Как небольшой, так и большой размер страницы может привести к высоким накладным расходам, требуется выбрать наилучший размер.
- Необходимо выполнять преобразование адресов, что ведет к дополнительным накладным расходам процессорного времени при обращении к странице.
- Требуется реализовать эффективное вытеснение страниц из оперативной памяти с тем, чтобы потом загрузить на это место данные из виртуальной памяти.

4.2.2. Сегментная организация

При сегментной организации память может быть поделена на фрагменты произвольной заранее не фиксированной длины - ***сегменты***.

Для каждого процесса заводится таблица сегментов. Структура записи таблицы сегментов: бит-признак присутствия сегмента, длина сегмента, биты-признаки защиты, адрес сегмента в оперативной или внешней памяти.

Признак защиты используется для ограничения доступа процесса к сегментам и обычно состоит из четырех битов: бит R (защита по чтению), W (защита от записи), E (защита по исполнению), A (защита по добавлению). Для всех них значение "1" разрешает доступ по операции, "0" - запрещает. Отметим, что наличие признаков защиты характерно и для страничной организации памяти.

Преобразование виртуальных адресов в реальные производится аналогично страничной организации, однако есть отличия, что связано, например, с необходимостью проверки невыхода обращения за границы сегментов, при этом проверка для сегментов стека производится с учетом его роста вниз, в то время, как остальные растут вверх.

Сегментной организации присущи следующие проблемы:

- для каждого сегмента нужно знать и учитывать его длину;
- виртуальные сегменты различаются по размеру и в оперативной памяти существуют свободные непрерывные участки разного размера, из чего следует, что нужно выбирать место назначения для сегмента;
- фрагментация памяти.

4.2.3. Сегментно-страничная организация

Сегментно-страничная организация объединяет первые два подхода. Виртуальный адрес выглядит следующим образом: (s, p, d). В нем указаны номер сегмента, номер страницы и смещение соответственно. Память представлена в виде множества сегментов разной длины, каждый сегмент является совокупностью страниц одинакового размера.

Для каждого процесса заводится таблица сегментов, начальный адрес которой хранится в специальном регистре. По номеру сегмента s производится обращение к s-й строке таблицы сегментов, которая содержит начальный адрес таблицы страниц данного сегмента. Если сегмент присутствует, то по адресу S производится обращение к таблице страниц этого сегмента, строка p которой содержит информацию об интересующей нас виртуальной странице. Там хранится номер физической страницы, отведенной под реальную страницу. Следовательно, для реализации одного обращения к памяти необходимо сделать три обращения: два - к таблицам сегментов и страниц, третье - непосредственно в ячейку.

Таким образом, недостатком является сложность получения реального адреса в памяти.

4.2.4. Использование ассоциативных таблиц

Использование виртуальной памяти ведет к падению производительности при доступе к памяти в 2-3 раза, так как на одно обращение к памяти надо сделать 2-3 обращения из-за преобразования адресов. Поэтому для поддержки виртуальной памяти используется специальная аппаратура, например, ассоциативная память, очень быстрая, но небольшого размера. Структура ассоциативной таблицы имеет вид, представленный на рис. 4.3.

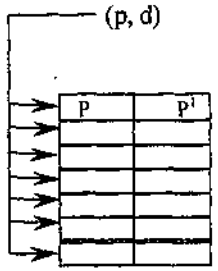


Рис. 4.3. Структура ассоциативной памяти

Значение p используется как ключ поиска, результат выдается из строки, для которой ключ совпал с этим номером. Обычно в ассоциативной памяти хранятся несколько десятков дескрипторов страниц - элементов таблицы страниц, к которым были сделаны обращения в последнее время. По номеру виртуальной страницы p отыскивается нужный дескриптор, из которого берется адрес p^1 оперативной памяти, где эта страница расположена. Время обращения к ассоциативной памяти порядка времени обращения к регистрам, что приводит к сохранению уровня производительности системы при использовании виртуальной памяти.

4.2.5. Проблемы при использовании виртуальной памяти

Использование виртуальной памяти ведет к возникновению ряда проблем: загрузки, размещения, выталкивания. Вкратце обсудим их.

Проблема загрузки состоит в том, что необходимо определить момент времени копирования страницы (сегмента) из внешней памяти во внутреннюю. Возможно два варианта: по запросу (пока нет обращения, страница не загружается), с упреждением (запроса нет, блок уже перемещается в оперативную память). Достоинством первого варианта является экономия памяти, недостатком - при возникновении запроса теряется время в ожидании подкачки. Подкачка с упреждением, как правило, позволяет осуществить более быстрое обращение к странице, однако загруженная заранее страница занимает место, более того, может находиться в памяти безо всякой пользы.

Проблема размещения имеется при использовании сегментной организации и отсутствует при страничной или сегментно-страничной организации. Как уже

говорилось выше, она сводится к необходимости выбора подходящего свободного пространства и проблеме фрагментации памяти.

Проблема выталкивания заключается в том, что необходимо определить наилучшую страницу или сегмент для удаления из оперативной памяти с целью освобождения места при нехватке памяти.

Основные подходы к выталкиванию:

1. Принцип оптимальности. Следует заменить ту страницу, к которой в дальнейшем не будет обращений в течение самого длительного времени. Этот принцип физически нереализуем.
2. Выталкивание случайной страницы.
3. Принцип FIFO. Выталкивается первая пришедшая, т.е. дольше всего находящаяся в памяти страница.
4. LRU (Least Recently Used). Выталкивается страница, дольше всего не использовавшаяся.
5. MRU (Most Recently Used). Выталкивается страница, наиболее недавно (только что) использовавшаяся.
6. NUR (Not Used Recently). Выталкивается страница, не использовавшаяся недавно.

В варианте NUR для отслеживания времени необходимы всего 2 бита признака: обращения (нуль, если не было обращений). модификации (нуль, если страница не изменялась). Выталкивается страница с минимальным значением этой пары бит. Всего может быть четыре возможных значения (табл. 4.1.).

Таблица 4.1

Возможные значения пары бит

Обращение	Модификация
0	0
0	1
1	0
1	1

Вариант сочетания битов 01 соответствует физически абсурдной ситуации, в которой была модификация без обращения к странице, что объясняется периодическим сбросом бита признака обращения, но не модификации! Модифицированная ("грязная") страница должна сохраняться на диске при вытеснении, немодифицированную можно просто заместить новой.

7. Используется рабочее множество страниц. Этот подход опирается на ***свойство локальности***: если программа начала работать с некоторой областью памяти, то в течение некоторого времени она будет продолжать работать с этой же областью памяти. Главное в таком подходе - удачно определить множество страниц, с которыми процесс работает наиболее интенсивно. Для этой цели, как правило, используются статистические данные, накапливаемые во время работы процесса.

4.3. Управление внешней памятью

При рассмотрении структуры диска мы следуем [30].

Магнитные диски в настоящее время широко используются как основная внешняя память. Рассмотрим кратко организацию файлов на дисках и проблему оптимизации времени доступа к данным. Для простоты будут рассмотрены гибкие диски и файловая система FAT.

Магнитные поверхности диска разбиты на концентрические дорожки, поделенные в свою очередь на секторы . Первый сектор (поверхность 0, дорожка 0, сектор 0) всегда используется для хранения загрузочной записи. Далее несколько секторов хранят FAT (file allocation table - таблица размещения файлов), которая записывается в двух копиях. Число секторов под FAT зависит от емкости диска. За FAT в последовательных секторах располагается корневая директория, число секторов под которую тоже зависит от емкости диска. При назначении секторов под таблицы FAT и директорию дисковая операционная система использует сначала все секторы дорожки 0, поверхности 0, затем переходит на поверхность 1 и использует дорожку 0, затем переходит на поверхность 0 дорожку 1, затем - на поверхность 1 дорожку 1 и т.д. Иными словами, происходит проход последовательно по цилиндрам - дорожкам с одним номером с разных поверхностей, что минимизирует перемещения магнитной головки.

Когда диск форматируется, секторы и дорожки организуются так, чтобы контроллер диска мог получить доступ к информации на диске. Некоторые секторы используется при этом для хранения системной информации - загрузочный сектор, таблицы FAT, директория; другие могут использоваться для хранения файлов пользователя. Первый сектор используется для хранения загрузочной записи. Из загрузочной записи компьютер узнает тип диска, размер секторов, количество секторов и другую важную информацию, необходимую для базовой системы ввода/вывода (BIOS) и операционной системы.

Единица распределения места под файлы - кластер - некоторое количество последовательных секторов на диске. Размер кластера хранится в загрузочной записи в одном байте со смещением 0x0D от начала записи. Число секторов под одну копию FAT хранится в двух байтах со смещением 0x16, число элементов корневой директории записано в двух байтах со смещением 0x11. Каждый элемент директории имеет размер 32 байта, структура которых представлена в табл. 4.2.

Таблица 4.2

Структура элемента директории

Байты (H)	Возможное содержание	Интерпретация
0-7	45 44 20 20 20 20 20 20	"ED " - имя файла
8-A	45 48 45	"EXE" - расширение
B	20	Архивный файл - байт атрибутов.

Байты (Н)	Возможное содержание	Интерпретация
		Смысл битов: 0 - только для чтения; 1 -скрытый; 2 - системный; 3 - метка тома; 4 - поддиректорий; 5 - архивный файл (если 1)
C-15		Зарезервировано
16-17	E0 96 (использовать как 96 E0)	1001 0110 1110 0000 00000 =0 с 2-секундным инкрементом: биты 4-0 представляют число секунд; 110111 = 55 мин: биты 10-5 представляют минуты; 10010= 18 часов: биты 15-11 представляют часы
18-19	98 0C (использовать как 0C 98)	0000 1100 1001 1000 11000= день 24: биты 4-0 представляют день; 0100 =месяц 4: биты 8-5 представляют месяц; 0000110=год 6 1980+6=1986: биты 15-9 представляют год
1A-1B	02 00	Первый кластер файла 0002
1C-1F	9B 10 00 00	Размер файла в байтах 00 00 10 9B=4251

Элементы директории указывают на первый кластер файла. Остальные кластеры файла, образующие цепочку, могут быть найдены из таблицы FAT, которая критична для доступа к файлам и для повышения надежности поддерживается в двух копиях.

Таблица FAT состоит из множества элементов. Размер элементов может быть разным для дисков разного типа, при большем размере элемента потенциально доступное дисковое пространство тоже увеличивается (FAT12 - использует 12-битные элементы, FAT16 - 16-битные элементы и т.п.). Первые два элемента (входа) FAT содержат байт описания среды (например, F9 - двухсторонняя дискета 5 ¼", 15 секторов/дорожка), за которым идут 0xF для заполнения свободного места в этих элементах. Остальные входы находятся во взаимнооднозначном соответствии с кластерами диска: каждый вход FAT отображает состояние (свободен, зарезервирован, испорченный, часть файла) соответствующего кластера.

Диск имеет следующую логическую структуру: загрузочный сектор, зарезервированные секторы, FAT#1, FAT#2, директория, данные. Когда номер класте-

ра преобразуется в номер сектора, следует принять во внимание, что первый кластер данных имеет номер 2 (sector=data_start+(cluster_number-2)*sectors_per_cluster). Входы FAT12 могут иметь значения, указанные в табл. 4.3.

Таблица 4.3

Входы FAT12

Код	Значение
0x000	Неиспользуемый кластер: кластер никогда не использовался
0x001	Свободный кластер: использовался, но сейчас свободен
0x002-0xFEF	Кластер используется файлом
0xFF0-0xFF6	Зарезервирован
0xFF7	Плохой кластер, не может быть использован
0xFF8-0xFFF	Последний кластер файла

Пример содержания таблицы FAT12:

Байт: 0 1 2 3 4 5 6 7 8 9 A B C D E ..

Значение:F9 FF FF 03 60 00 00 10 00 FF 7F FF 00 00 00 ..

Первые два элемента расположены в байтах 0-2. Третий элемент (соответствует кластеру номер 2) имеет свои части в байтах 3-4 (смещение в FAT=floor(cluster_number*3/2)): 03 60, после перестановки мест получаем 60 03, и 3 младшие (для кластеров с четными номерами) цифры 003 составляют содержимое входа 2, ссылающегося на вход 3. Вход 3 находится в байтах 4-5: 60 00, после перестановки - 00 60, 3 старшие цифры (для нечетных номеров кластеров) есть 006, т.е. он ссылается на кластер 6. Вход 6 находится в байтах 9-A: FF 7F, после перестановки - 7F FF, младшие 3 цифры (для кластеров с четными номерами) - FFF, так что это последний кластер в цепи кластеров, выделенных файлу.

Кластеры одного файла могут быть расположены в разных местах диска, что приводит к проблеме фрагментации дисковой памяти, замедляющей доступ к ней. Эта проблема решается с помощью трудоемкой процедуры дефрагментации, упорядочивающей информацию на диске так, чтобы кластеры одного файла располагались один за другим.

Кроме того, кластеры разных файлов располагаются в различных местах диска, что приводит к проблеме оптимизации доступа к диску для системы, в которой параллельно исполняются несколько процессов. Например, порядок обслуживания запросов на доступ к диску может зависеть от местоположения в данный момент магнитной головки и от размещения целевых частей файла на диске так, чтобы минимизировать перемещения магнитной головки.

Обычно используются следующие стратегии доступа к диску, реализуемые как на программном, так и на аппаратном уровне:

1. Пришедший первый запрос первым и обслуживается (FIFO).

2. Сначала обслуживается запрос с минимальным временем перемещения магнитной головки.
3. Scan (сканирование) - магнитная головка циклически движется от внешних дорожек к внутренним и обратно, обслуживая встречающиеся по пути запросы - алгоритм лифта. Направление движения меняется, если нет больше запросов в текущем направлении.
4. Cyclic scan (циклическое сканирование) - запросы обслуживаются только по пути к внутренним дорожкам, затем головка прыжком возвращается назад, и процесс повторяется.
5. N-step scan (N-шаговое сканирование) - головка движется вперед и назад, обслуживая запросы, появившиеся перед началом движения в текущем направлении. Новые запросы переупорядочиваются для более эффективного обслуживания при обратном движении.

Контрольные вопросы и задания

1. Какие виды памяти Вы знаете?
2. В чем особенности управления памятью в однозадачном режиме?
3. Охарактеризуйте режимы с фиксированными и переменными разделами.
4. Что такое виртуальная память? Какие варианты организации виртуальной памяти Вы знаете?
5. Сколько числовых значений требуется для организации адресации в сегментной модели виртуальной памяти?
6. Для чего используются ассоциативные таблицы при доступе к памяти?
7. Какие проблемы, возникающие при использовании виртуальной памяти, Вы можете выделить?
8. Какова структура диска для файловых систем семейства FAT?
9. Что содержит в себе FAT: кластеры, номера кластеров, файлы, имена файлов или ничего из перечисленного?
10. Что содержат в себе элементы директории: размер файла, имя файла, ссылку на первый элемент файла, все перечисленное или ничего из перечисленного?
11. Каков размер элемента FAT12 в битах?
12. Какие стратегии оптимизации доступа к диску Вы знаете?

Глава 5. Управление процессорами

Процессор является системным ресурсом, также как внешняя и оперативная память. Поэтому эффективная вычислительная система должна умело распоряжаться процессорным временем. Под управлением процессором мы будем подразумевать стратегии распределения процессорного времени между задачами, обрабатываемыми системой. Управление процессором тесно перекликается с задачей управления процессами.

Однопроцессорные системы, как правило, используют круговое циклическое обслуживание процессов (RR), о котором говорилось ранее в п. 3.2.3.

Кроме того, обычно применяется дисциплина переднего-заднего плана (FB) - каждая задача поступает в очередь своего приоритета (см. п. 3.2.3).

Две часто используемые стратегии назначения приоритетов в СРВ - это RM (по возрастанию периода) и DM (по возрастанию директивного срока решения задач). В этих стратегиях больший приоритет получают задачи с меньшим периодом и директивным сроком соответственно. Задачи меньшего приоритета обслуживаются только при отсутствии задач более высокого приоритета. Новые поступающие высокоприоритетные задачи могут либо прерывать обслуживание низкоприоритетной задачи, либо нет.

Рассмотрим некоторые из подходов при планировании задач в однопроцессорных и многопроцессорных системах.

5.1. Планирование периодических задач

При изложении данного материала мы используем [1].

Как правило, любая вычислительная система исполняет ряд задач, носящих повторяющийся или периодический характер. Такие задачи возникают в системе в определенные моменты времени. Примером может быть процесс опроса клавиатуры или датчиков, который повторяется через заданные промежутки времени. Множество периодических задач имелось в системе, рассмотренной в качестве примера в главе 1.

Периодические задачи T(p,e,d,φ) характеризуются периодом исполнения p, временем исполнения e (длительностью исполнения), относительным директивным сроком (определяет временной интервал, в течение которого задача после появления в системе должна быть закончена), фазой φ, показывающей момент времени первого запуска задачи (момент $i-$го запуска подсчитываются как $\varphi+(i-1)p$). По умолчанию, директивный срок равен периоду, и фаза равна 0, если эти величины не определены явно. ***Загруженность*** задачи определяется как u=e/p. Необходимое условие реализуемости системы из n периодических задач на однопроцессорной системе:

$$\sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{e_i}{p_i} \le 1$$

Расписание выполнения системы периодических задач строится для ***гиперпериода*** - наименьшего общего кратного периодов задач. Гиперпериод - временной отрезок, на котором может быть построено расписание выполнения системы периодических задач. Затем это расписание можно будет просто скопировать на следующих временных промежутках. Например, для задач T1(4,1), T2(5,1.8), T3(20,1), T4(20,2) загруженности: 0,25; 0,36; 0,05; 0.1; суммарная загруженность - 0,76; гиперпериод - 20, поэтому достаточно построить расписание выполнения задач на интервале длины 20, каждый следующий интервал той же длины будет повторять предыдущий.

Расписание представляется как таблица, элементы которой соответствуют работам и показывают время начала каждой работы. В нашем примере для интервала длины 20 мы имеем 5 работ от задачи T1, 4 работы от T2, 1 работу от T3 и одну работу от T4. Каждая из них должна быть назначена на решение внутри гиперпериода так, чтобы она завершилась до своего директивного срока. Такое расписание может быть построено заранее, не в процессе вычислений, и поэтому для этого могут использоваться достаточно трудоемкие алгоритмы. Например, для рассмотренного набора задач может быть построено расписание, приведенное на рис. 5.1.

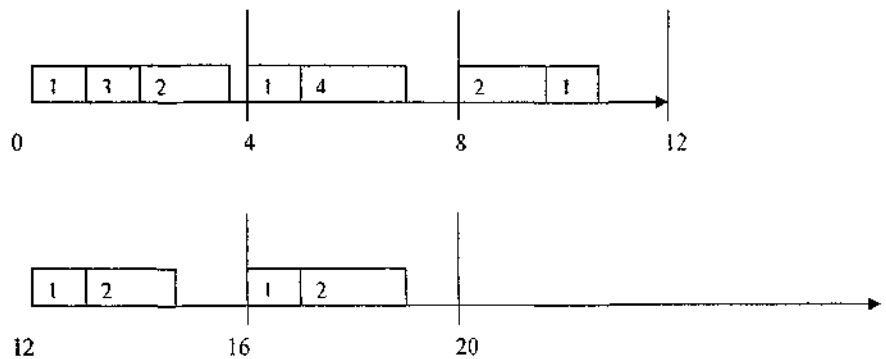


Рис. 5.1. Пример расписания для четырех задач

На рис. 5.1 номера задач показаны в прямоугольниках, моменты времени показаны ниже оси абсцисс. Такие расписания называются ***циклическими*** или ***временными расписаниями***, так как принятия решений о назначении задач делаются в определенные моменты времени, независимо от событий в системе, например, таких, как появление или завершение работ и пр.

Следует подчеркнуть, что при загруженности U, большей 1, расписание, удовлетворяющее директивным срокам, построить нельзя, а при загруженности, не большей 1, существование таких расписаний возможно, но не гарантируется, и выяснение возможности их существования является сложной задачей. Например, имеем 2 периодические задачи T1(2,1), T2(3,1.5). Загруженности задач равны: u1=1/2=0,2, u2=1,5/3=0,5, суммарная загруженность U= u1+u2=0,5+0,5=1, поэтому эта система задач не является нереализуемой. Но если строить расписание для них алгоритмом, назначающим больший приоритет задаче с меньшим периодом (RM - rate monotonic), то на гиперпериоде 6 получается расписание, приведенное на рис. 5.2.

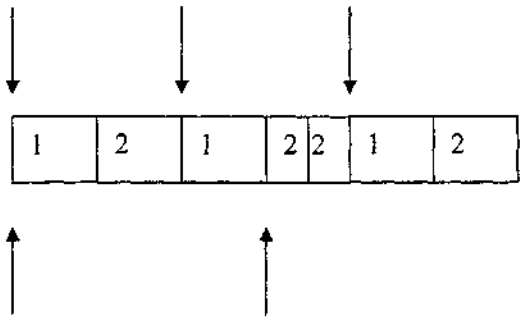


Рис. 5.2. Расписание для гиперпериода 6 и алгоритма RM

На рис. 5.2 три верхние стрелки показывают моменты появления в системе задачи №1 (три задачи за 6 единиц времени), а нижние две стрелки помечают моменты появления в системе задачи №2 с периодом 3 (появляется 2 раза за гиперпериод). Видно, что в момент 2 исполнение задачи №2 прерывается более высокоприоритетной задачей №1. Задача №2 возобновляется в момент времени 3 и завершается в момент времени 3.5 после своего директивного срока, равного 3 (по умолчанию, директивный срок равен периоду). Итак, алгоритм RM дает нарушение директивного срока для первого запуска задачи №2.

В то же время, если рассмотреть алгоритм EDF (Earliest Deadline First, ранние директивные сроки первые), который назначает задачам приоритеты динамически - чем меньше времени остается до истечения директивного срока, тем приоритет выше, то для той же системы из двух задач можно получить расписание, показанное на рис. 5.3.

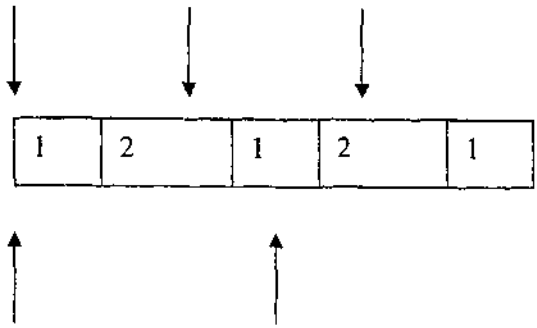


Рис. 5.3. Расписание для гиперпериода 6 и алгоритма EDF

В этом случае, когда второй раз появляется задача №1 в момент 2, для нее директивный срок равен 4 (2+2). Директивный срок для исполняющейся задачи №2 есть 3 (0+3), что меньше, чем 4, поэтому приоритет задачи №2 выше, чем у задачи №1, и задача №2 не прерывается в момент времени 2, а завершается в момент времени 3 без нарушения директивного срока. В момент 3 запускается вторая копия задачи №2, но ее директивный срок 3+3=6 больше директивного срока задачи №1, поэтому второй выпуск задачи №1 продолжается до завершения в момент времени 3.5, после чего назначается задача №2, а по ее завершении в момент 5 - 3-й выпуск задачи №1. Видно, что в этом расписании для обеих задач соблюдены директивные сроки.

Для циклических расписаний желательно иметь некоторую предопределенную структуру, которая определяет моменты времени назначения задач регулярным образом. Моменты времени принятия решений о назначении задач делят ось времени на равные интервалы, называемые фреймами. Каждый фрейм имеет длину f - размер фрейма. Фаза каждой периодической задачи является неотрицательным целым кратным размера фрейма. Другими словами, первая работа каждой задачи появляется в момент начала некоторого фрейма. В интервалах между фреймами планировщик может запускать непериодические задачи, если они есть. Размер фрейма должен удовлетворять ряду ограничений. Ограничение (5.1) показывает, что фрейм должен быть достаточно большим, чтобы любая задача могла быть решена внутри него целиком.

$$f \geq \max_{1 \leq i \leq n}(e_i) \tag{5.1}$$

Размер фрейма должен быть делителем гиперпериода, следовательно, f должно быть делителем периода, по крайней мере, одной задачи:

$$(\exists i \in \{1,\ldots n\})(p_i \bmod f \equiv 0) \tag{5.2}$$

Внутри каждого фрейма может исполняться несколько задач, но фреймы должны идти один за другим. Если некоторая задача появляется в момент t' внут-

ри фрейма, начавшегося в момент t, она может быть назначена в любой следующий фрейм, но должна закончиться до своего относительного директивного срока D. Могут быть две ситуации: $t=t'$, или $t<t'$. Так как задача должна завершиться внутри фрейма, накладываем условие: $f \le D$ для первого случая, а для 2-го: $t+2f \le t'+D$, так как задача может стартовать только во втором фрейме после своего появления (рис. 5.4).

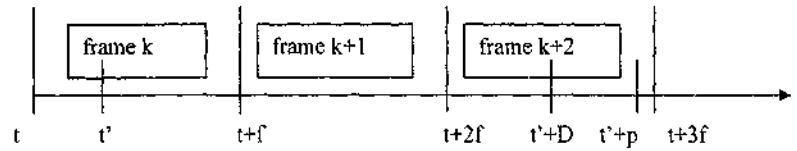


Рис. 5.4. Соотношение между размером фрейма и директивным сроком

Следовательно, $2f-(t'-t) \le D$. Так как разность (t'-t) не меньше наибольшего общего делителя gcd(p_i, f), можем записать:

$$2f - \gcd(p_i, f) \le D_i, i = \overline{1,n}. \qquad (5.3)$$

Условия (5.1)-(5.3) являются ограничениями на размер фрейма. Например для рассмотренных ранее четырех задач с гиперпериодом H=20, в соответствии с (5.2), получаем следующие возможные размеры фрейма: 2, 4, 5, 10, 20. Все они удовлетворяют (5.1). Однако только значение 2 удовлетворяет (5.3):

2*2-gcd(p1,2)=4-gcd(4,2)=4-2=2<=D1=p1=4,
2*2-gcd(p2,2)=4-gcd(5,2)=4-1=3<=D2=5,
2*2-gcd(p3,2)=4-gcd(20,2)=4-2=2<=D3=20,
2*2-gcd(p4,2)=4-gcd(20,2)=4-2=2<=D4=20,
но
2*5-gcd(p1,5)=10-gcd(4,5)=10-1=9>D1=4,
2*10-gcd(p1,10)=20-gcd(4,10)=20-2=18>D1=4,
2*20-gcd(p1,20)=20-gcd(4,20)=20-4=16>D1=4.

Таким образом, только фрейм размера 2 удовлетворяет всем условиям. Циклическое расписание с размером фрейма 2 приведено на рис. 5.5.

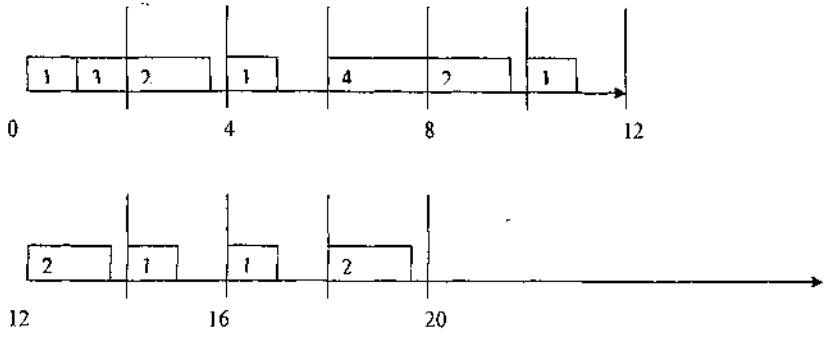


Рис. 5.5. Циклическое расписание для размера фрейма, равного 2

Расписания на рис. 5.1 и 5.5 эквивалентны в смысле удовлетворения директивных сроков периодических задач. Но расписание на рис. 5.5 дает больше возможностей для более раннего назначения апериодических задач, если они появляются.

Иногда параметры задач таковы, что невозможно найти размер фрейма, удовлетворяющий всем условиям, например, в случае системы задач T={(4,1), (5,2,D=7), (20,5)}. Чтобы удовлетворить (5.1), нужно, чтобы f>=5. Чтобы удовлетворить (5.3), нужно, чтобы f<=4:

2*4-gcd(p1,4)=2*4-gcd(4,4)=8-4=4<=D1=4,

2*4-gcd(p2,4)=2*4-gcd(5,4)=8-1=7<=D2=7,

2*4-gcd(p3,4)=2*4-gcd(20,4)=8-4=4<=D3=20, но

2*5-gcd(p1,5)=2*5-gcd(4,5)=10-1=9>D1=4,

2*10-gcd(p1,10)=2*10-gcd(4,10)=20-2=18>D1=4,

2*20-gcd(p1,20)=2*20-gcd(4,20)=40-4=36>D1=4.

Таким образом, имеем противоречивые требования к размеру фрейма. В такой ситуации некоторые задачи принудительно разбиваются на подзадачи меньшего размера. В расписании на рис. 5.6 задача 3 разбита на три подзадачи 3,1; 3,2; 3,3 с временами исполнения соответственно 1, 3, 1.

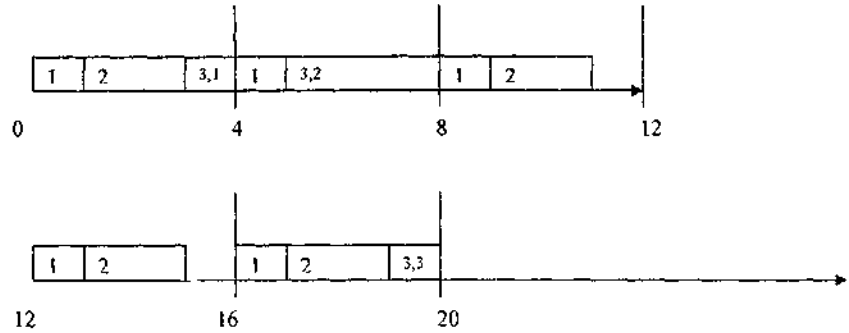


Рис. 5.6. Пример расписания с разбиением задачи на подзадачи

Как уже отмечалось, интервалы между фреймами могут использоваться для выполнения апериодических задач. Рассмотрим расписание для некоторых периодических задач и предположим, что три апериодические задачи A1, A2, A3 появились как раз перед моментами времени 4; 9,5; 10.5 и требуют 1,5; 0,5 и 2 единицы времени для исполнения соответственно.

Такое расписание может быть получено в случае запуска этих задач в конце каждого фрейма после периодических задач (рис. 5.7). На рисунке периодические задачи показаны серым цветом.

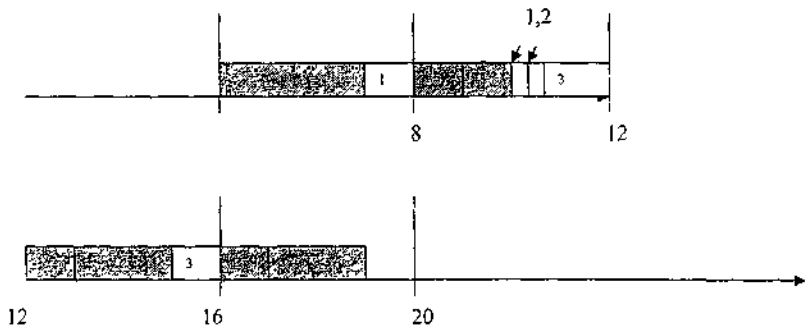


Рис. 5.7. Циклическое расписание с планированием апериодических задач

В расписании на рис. 5.7 времена отклика (пребывания в системе) для трех апериодических задач: 6,5; 1,5; 5,5 со средним значением равным 4,5.

Более эффективный подход - кража резервов - предполагает предоставление процессора апериодическим задачам в начале каждого фрейма (рис. 5.8). Времена отклика: 4,5; 0,5; 2,5 соответственно, среднее время - 2,5.

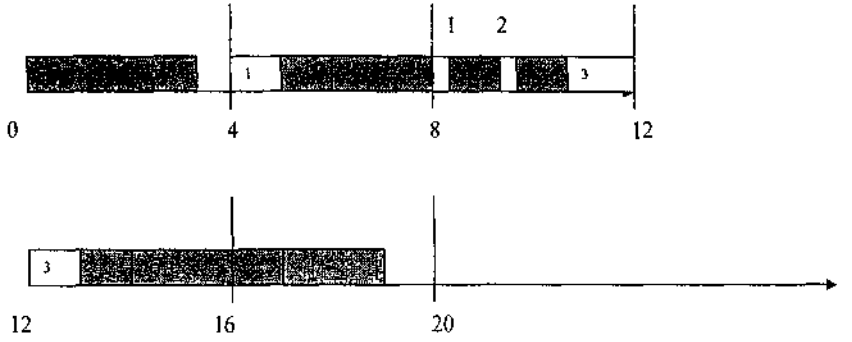


Рис. 5.8. Планирование апериодических задач в начале каждого фрейма

Планировщик пытается назначить апериодические задачи, если они есть, в начале каждого фрейма или внутри фрейма по окончании периодической задачи, если можно удовлетворить директивный срок апериодической задачи и не нарушить директивный срок ранее назначенных апериодических, а также периодических задач. В случае отсутствия такой возможности планировщик должен уведомить высший уровень управления о возникшей ситуации с тем, чтобы предпринять соответствующие меры для улучшения ситуации. Так, например, если система контроля качества обнаружит неисправную деталь на конвейерной ленте, она запустит апериодическую задачу передвижения руки робота к этой детали, чтобы снять ее с конвейера. Если же будет обнаружено, что эта задача не может быть решена в директивный срок (пока деталь еще в зоне досягаемости руки робота), то система должна выдать предупреждение как можно скорее для выполнения коррекции - замедления хода конвейера, остановки конвейера, сообщения оператору о необходимости ручного удаления бракованной детали. Предупреждение не должно прийти слишком поздно, иначе деталь может быть уже упакована.

5.2. Многопроцессорные системы

В случае многопроцессорных систем физически параллельные процессоры используются для исполнения задач, что может существенно повысить пропускную способность и производительность системы. Обсудим несколько алгоритмов распределения процессорных ресурсов для многопроцессорных систем.

5.2.1. Алгоритм Макнотона

Рассмотрим систему с n идентичными процессорами, на которой необходимо решить L независимых задач, каждая задача решается одним процессором на

протяжении времени t_i, $i = \overline{1, L}$. Необходимо построить расписание решения системы задач за минимальное время.

Рассматриваемый ниже алгоритм был предложен Макнотоном в 1959 г. и направлен на построение оптимального по длине расписания с не более чем n-1 прерываниями выполнения задач.

Суть алгоритма:

1. Подсчитывается величина $\Theta = \max\left(\frac{1}{n}\sum_{i=1}^{L} t_i, \max_{i=\overline{1,L}} t_i\right) \le T_0$. Здесь $\frac{1}{n}\sum_{i=1}^{L} t_i$ - время решения пакета задач на n процессорах; T_0 - оптимальное время решения пакета задач.
2. Задачи упорядочиваются по убыванию времени решения и назначаются в таком порядке справа налево от момента Θ, начиная с n-го процессора. В худшем случае имеется (n-1) прерывание одной или нескольких задач.

Рассмотрим пример. Пусть n=2, L=4, времена решения задач: (5,4,3,2). Тогда $\Theta = \max(7,5) = 7$. Получаем расписание, приведенное на рис. 5.9.

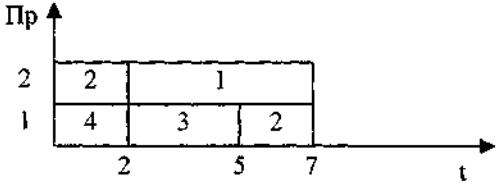


Рис. 5.9. Расписание решения пакета задач для алгоритма Макнотона

5.2.2. Алгоритм Коффмана-Грэхема

Для этого алгоритма, как и для алгоритма Макнотона, полагается, что имеется n идентичных процессоров, на которых необходимо решить L задач. Отличие состоит в том, что задачи являются взаимосвязанными.

На множестве задач определен частичный порядок <, который задает отношение предшествования между задачами. Так, запись $z_i < z_j$ означает, что задача z_i является предшественником задачи z_j и должна быть завершена до начала z_j. Если не существует такой задачи z_k, что справедливо $z_i < z_k < z_j$, то z_j называется непосредственным потомком z_i, а z_i является непосредственным предшественником z_j.

Задача называется конечной, если не имеет потомков, и начальной, если не имеет предшественников.

Множество задач может быть представлено в виде графа, вершины которого - задачи. На графе ориентированная дуга идет от z_i к z_j, если z_j есть непосредственный потомок z_i (рис. 5.10).

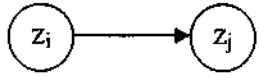


Рис. 5.10. Связь предшественник-потомок

Идея этого алгоритма состоит в том, чтобы назначать в первую очередь задачи, наиболее удаленные от конечной вершины. В связи с этим вводится понятие ***уровня задачи*** - длины максимального пути на графе задач, связывающего данную задачу с конечной.

Введем ряд обозначений: $s(z_0)$ - это множество непосредственных потомков для z_0; $\alpha(z_0)$ - метка z_0; $N(z)$ - убывающая последовательность целых чисел, полученных упорядочиванием множества $\{\alpha(z_j)/\, z_j \in s(z)\}$.

Алгоритм:

1. Выбирается задача z_0 из набора задач, для которой $s(z_0)=0$ (конечная задача), и полагается $\alpha(z_0)=1$. Остальным конечным задачам присваиваются значения α, нарастающие с шагом 1.
2. Пусть для некоторого $i \le L$ метки $1,\ldots,i-1$ уже расставлены. Пусть R - множество задач, не имеющих непомеченных потомков, т. е. для всех $z \in R$ определено $N(z)$. Пусть $z^* \in R$ - такая задача, что $N(z^*) \le N(z)$ (лексикографически) для всех $z \in R$. Если таких задач несколько, то в качестве z^* можно брать любую. Тогда $\alpha(z^*)=i$.
3. Когда все задачи помечены, строится список задач С в соответствии с убыванием меток.
4. Задачи, не имеющие предшественников или предшественники которых уже решены, назначаются на освобождающиеся процессоры в порядке, определяемом построенным списком С (слева направо).

В алгоритме предполагается, что все задачи набора имеют время решения 1 (некоторая единица времени). Если это не так, то задачи работают с прерыванием и длительность одного кванта времени для каждой разбитой задачи составляет 1.

Рассмотрим пример. Пусть граф набора задач имеет вид, показанный на рис. 5.11.

Для графа на рис. 5.11 одна из возможных разметок совпадает с нумерацией задач, т.е. $\alpha(z_j)=j$, $c=(z_{19}, z_{18},\ldots, z_1)$. Расписание для трех процессоров приведено в табл. 5.1.

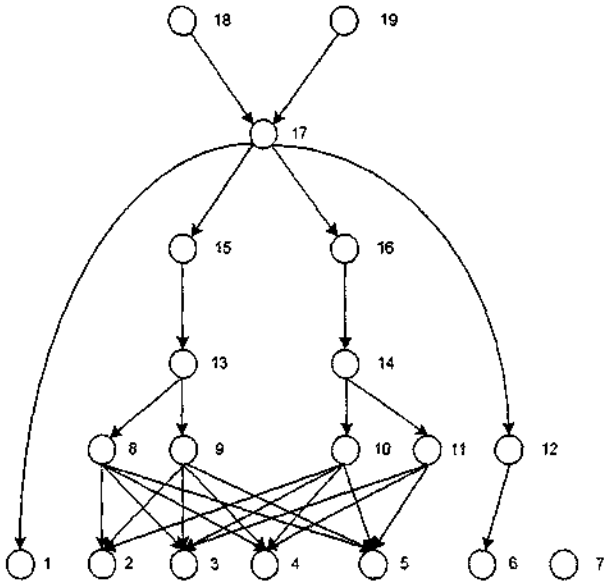


Рис. 5.11. Граф набора задач

Таблица 5.1

Расписание для трех процессоров

P3	7		12	6	9		3				
P2	18		15	13	10	1	4				
P1	19	17	16	14	11	8	5	2			
t	1	2	3	4	5	6	7	8	9	10	11

Отметим, что алгоритм не дает оптимального расписания. Его оценка:

$$T \le \left(2 - \frac{2}{n}\right) T_0, \text{ при } n \ge 2.$$

Иными словами, в худшем случае будет превышение T_0 в два раза.

5.2.3. Многопроцессорные системы с периодическим планированием задач

Для систем с периодическими задачами может использоваться планирование, основанное на заполнении урн (bin-packing). Один из таких алгоритмов RMFF (Rate Monotonic First Fit, RMFF - первый подходящий по убыванию частоты), который назначает на подходящий процессор очередную задачу из списка задач, упорядоченных по неубыванию периода. Предполагается, что очередная

задача может быть назначена на процессор i, если суммарная загруженность i-го процессора не превзойдет URM(n(i)), где URM(n)= $n(2^{1/n}-1)$ и n(i) - суммарное количество задач, назначенных на i-й процессор. Когда n=2, эта функция дает приблизительно 0,82 и убывает монотонно сверху до ln 2 (0,693) при n, стремящемся к бесконечности (табл. 5.2).

Таблица 5.2

Значения функции URM(n) для нескольких значений n

URM(n)	1	0.828	0.78	0.75	0.745
N	1	2	3	4	5

Рассмотрим пример. Пусть задачи и их загруженности равны значениям, показанным в табл. 5.3.

Таблица 5.3

Характеристики системы периодических процессов

Ti	ui	Ti	ui	Ti	ui
(2,1)	0.5	(4.5,0.1)	0.022	(8,1)	0.125
(2.5,0.1)	0.04	(5,1)	0.2	(8.5,0.1)	0.012
(3,1)	0.333	(6,1)	0.167	(9,1)	0.111
(4,1)	0.25	(7,1)	0.143		

Если число процессоров равно 3, то в соответствии с алгоритмом RMFF мы получим распределение задач по процессорам, показанное в табл. 5.4.

Таблица 5.4

Распределение задач по трем процессорам

Процессор P1	Процессор P2	Процессор P3
(2,1), (2.5,0.1), (4.5,0.1), (6,1), (8.5,0.1) с суммарной загруженностью 0,741	(3,1), (4,1), (7,1) с суммарной загруженностью 0,726	(5,1), (8,1), (9,1) с суммарной загруженностью 0,436

Суммарная загруженность процессоров принимается во внимание, чтобы гарантировать существование соответствующего однопроцессорного RM-расписания, удовлетворяющего директивным срокам.

Контрольные вопросы и задания

1. Какими параметрами характеризуются периодические задачи?
2. Что такое циклические расписания, гиперпериод?
3. Каковы основные условия для размера фрейма?
4. Каким образом осуществляется планирование апериодических задач?
5. Определите общую загрузку системы из четырех периодических задач: P1(период=3, исполнение=1), P2(3,1), P3(4,1), P4(3,2). Постройте расписание для этих задач, если это возможно.

6. Охарактеризуйте алгоритмы Макнотона, Коффмана-Грэхема.
7. Как производится распределение процессорного времени для множества процессоров и периодических задач?

Глава 6. Драйверы устройств

Эта глава базируется на материалах из [31-34].

Драйверы устройств - это специальные программы, используемые для организации доступа к внешним устройствам.

По существу драйверы являются абстракцией устройств, предоставляя операционной системе унифицированный интерфейс для доступа к ним.

В однозадачных операционных системах, например, MS DOS, они имеют простую структуру на базе COM-файлов. Многозадачные операционные системы, снабженные механизмами защиты памяти и разграничения уровней защиты ядра и пользовательских программ, предъявляют к драйверам устройств значительно более высокие требования.

6.1. Драйверы устройств операционной системы Windows

Применительно к операционным системам семейства Windows будут рассмотрены драйверы устройств формата WDM (windows driver model), который поддерживается Windows 98, ME, NT, 2000, XP, 2003.

6.1.1. Общая структура операционной системы Windows NT/2000

Структура операционной системы Windows 2000 представлена на рис. 6.1.

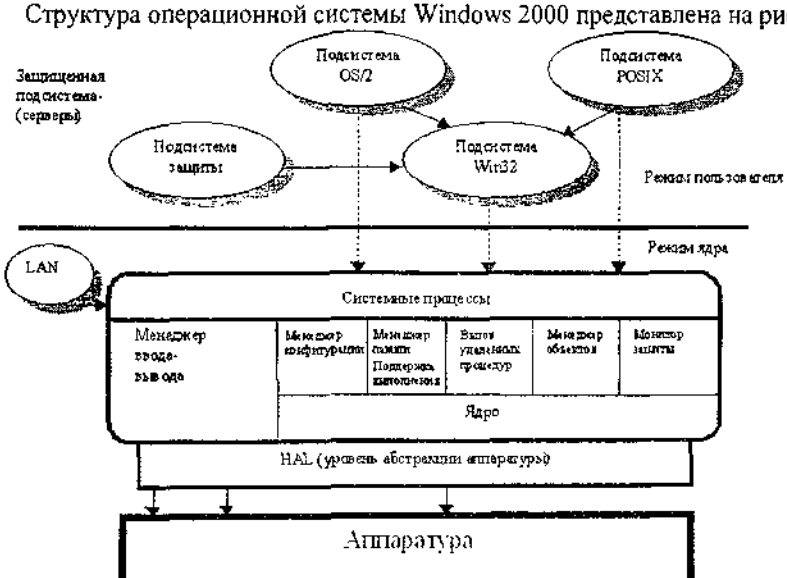


Рис. 6.1. Структура операционной системы Windows NT/2000

Будем рассматривать операционную систему с учетом специфики процессоров x86, которые в защищенном режиме, начиная с i386, предоставляют четыре уровня безопасности, привилегий, исполнения программы. Таким образом, имеются четыре кольца привилегий с номерами 0, 1, 2, 3, причем кольцо 0 имеет высший уровень привилегий безопасности. Операционная система Windows NT/2000 использует 0-е и 3-е кольца привилегий. Нулевое кольцо применяется для частей операционной системы режима ядра, пользовательские приложения и значительная часть операционной системы выполняются в третьем кольце привилегий. Защищенный режим предоставляет средства многозадачности, сегментную и страничную организации виртуальной памяти.

Каждая задача пользователя получает 4Гб виртуального адресного пространства. Максимальный размер сегмента - 4Гб, размер страницы - 4Кб. Состояния задач сохраняются в специальных сегментах состояния задачи (TSS - Task State Segments). Такая структура данных уже встречалась ранее и называлась блоком управления процессом, в ней хранятся состояния регистров и прочая информация, необходимая для нормального восстановления задачи после прерывания. Сегмент TSS имеет также карту ввода/вывода (Input-Output Map), показывающую для каждого из возможных 64K портов ввода/вывода возможность доступа к нему.

Операционная система Windows NT/2000 характеризуется следующими отличительными особенностями:

- модель модифицированного микроядра;
- эмуляция нескольких операционных систем;
- независимость от архитектуры процессора;
- объектная модель;
- поддержка многопоточности;
- вытесняющая многозадачность;
- виртуальная память с подкачкой страниц по требованию;
- мультипроцессорная обработка;
- интегрированная поддержка сети.

Идея микроядра состоит в исполнении большинства компонент операционной системы как процессов пользователей. Исключение составляет небольшая часть операционной системы - собственно микроядро, его компоненты исполняются в привилегированном режиме. Windows NT/2000 использует подход, известный как модифицированное микроядро, что является промежуточным случаем между чистым микроядром и монолитной операционной системой. В этом подходе подсистемы окружения (Environmental Subsystem) и неотъемлемые подсистемы (Integral Subsystems) работают в пользовательском режиме. Подсистемы окружения обеспечивают прикладным программам интерфейс программирования, специфичный для некоторых операционных систем (Win32, Windows 3.x, POSIX, OS/2, DOS).

Подсистема Win32 является основной и используется как сервер для реализации других программных интерфейсов (например, psxdll.dll для программ POSIX). Подсистема окружения Win32 работает с помощью kernel32.dll, user32.dll, gdi32.dll, некоторых других клиентских dll и серверного процесса csrss.exe (Client/Server Runtime SubSystem). Управление окнами и сообщениями реализуется user32.dll; графикой - gdi32.dll; основные службы, такие, как ввод/вывод, управление процессами и потоками, управление памятью, синхронизацией - kernel32.dll. Начиная с Windows NT 4.0 функции USER и GDI перемещены в режим ядра и исполняются в контексте процесса. Код режима ядра, реализующий функции USER и GDI, находится в win32k.sys (пользовательская часть - в csrss.exe).

Неотъемлемыми подсистемами являются:

- подсистема безопасности (Security Subsystem);
- диспетчер управления службами (Service Control Manager), отвечающий за процессы-демоны и драйверы устройств;
- компоненты службы локализации RPC (RPC Locator Service), дающие возможность процессам работать в сети, используя логические имена устройств (служба отображает логические имена в сетевые перед RPC-запросом);
- исполнительные компоненты (Executive Components), включающие: диспетчер объектов (Object manager) - управляет ресурсами и реализует глобальное пространство имен; монитор безопасности (Security Reference Monitor) - реализует модель безопасности NT на основе идентификаторов безопасности (Security Identifiers, SID) и списков разграничительного контроля доступа (Discretionary Access Control List, DACL); диспетчер виртуальной памяти (Virtual Memory Manager) - определяет адресное пространство процесса и распределяет физическую память; диспетчер ввода/вывода (I/O Manager) - обеспечивает интерфейс между прикладными программами и драйверами устройств; диспетчер кэша (Cache Manager) - реализует глобальный файловый кэш: средство вызова локальных процедур (Local Procedure Call Facility) - обеспечивает эффективные межпроцессные коммуникации: диспетчер конфигурации (Configuration Manager) - управляет реестром; диспетчер процессов (Process Manager) - отвечает за создание и уничтожение процессов и потоков; поддержка среды Win32 (Win32 Support) - реализует управление окнами (сообщения и графика): диспетчер Plug-and-Play (Plug-and-Play manager) - уведомляет драйверы о включении/выключении устройств; диспетчер электропитания (Power Manager) - контролирует состояние электропитания; исполнительный модуль (Executive Support) - реализует управление очередями, системной областью памяти, обеспечивает системные рабочие потоки.

Ядро (Kernel - находится в ntoskrnl.exe для однопроцессорных систем, в ntkrnlmp.exe - для мультипроцессоров) отвечает за обработку прерываний, планирование потоков, межпроцессную синхронизацию. Драйверы устройств и исполнительные компоненты вызывают процедуры ядра, идентификаторы которых имеют префикс Ke(rnel). Ядро экспортирует объекты-диспетчеры (dispatcher objects) и управляющие объекты (control objects).

Объекты-диспетчеры используются для планирования и синхронизации и имеют атрибут, определяющий их состояние - занят или свободен. К этому классу объектов относятся: события, мьютексы, семафоры, таймеры и т.п.

Управляющие объекты используются для контроля над системными операциями. Управляющими объектами являются:

- APC-объект (Asynchronous Procedure Call) - содержит адрес процедуры асинхронного вызова и указатель на поток, который будет исполнять этот вызов;
- DPC-объект (Deferred Procedure Call) - содержит адрес процедуры отложенного вызова;
- объект-прерывание (Interruption object) - устанавливает соответствие между вектором прерывания и процедурой обработки прерывания (Interrupt Service Routine, ISR) драйвера устройства.

Функции ядра сильно зависят от аппаратуры. Слой абстрагирования от оборудования (Hardware Abstraction Layer, HAL) - тонкий слой кода между ядром и аппаратурой, взаимодействующий напрямую с процессором, шинами и другим оборудованием. Он обеспечивает функциям ядра стандартный интерфейс к оборудованию, драйверам, диспетчеру ввода/вывода и располагается в hal.dll. Процедуры HAL имеют префикс Hal. HAL обеспечивает независимость операционной системы от оборудования.

6.1.2. Использование памяти

Память в NT/2000 использует плоскую модель, каждый сегмент имеет размер 4Гб. Основные типы сегментов приведены в табл. 6.1.

Таблица 6.1

Основные типы сегментов Windows

Селектор Hex	Назначение	База	Предел	Уровень дескриптора привилегий (DPL)	Тип
08	Code32	0000 0000	ffff ffff	0	RE
10	Data32	0000 0000	ffff ffff	0	RW
1b	Code32	0000 0000	ffff ffff	3	RE

23	Data32	0000 0000	ffff ffff	3	RW

Тип селектора определяет, какие операции можно производить с памятью, и может быть равен: R - чтение, W - запись, E - выполнение.

Системное адресное пространство занимает верхние 2Гб, нижние 2Гб отведено под адресное пространство пользователя. Системное адресное пространство одно и то же для всех процессов. В виртуальном адресе 16-битный селектор имеет структуру, приведенную в табл. 6.2.

Таблица 6.2

Структура селектора

Номер бита	15 3	2	1, 0
Значение	index	TI	RPL

Index - номер селектора.

TI (Table indicator) - указатель таблицы:

0 - селектор указывает на глобальную таблицу дескрипторов GDT;

1 - указывает на локальную таблицу дескрипторов LDT.

RPL (Requested Privelege level) - требуемый уровень привилегий (0 или 3).

Например, для сегмента кода ядра TI=0, index=1, RPL=0, поэтому селектор = 0x08 (GDT[1]).

Системное адресное пространство имеет структуру, показанную в табл. 6.3.

Таблица 6.3

Структура системного адресного пространства

Содержимое	Адрес
HAL	0xffff ffff
Информация дампа сбоя	
Невыгружаемый пул	
Выгружаемый пул	
Отображение менеджера кэша	
Гиперпространство менеджера кэша	
Директория страниц	
Таблица страниц	
Файлы, отображаемые в память	
Образ ОС	0x8000 0000

6.1.3. Система приоритетов

Windows NT/2000 имеет двухуровневую модель приоритетов:

- приоритеты высшего уровня (уровни запроса прерываний - Interrupt ReQuest Level - IRQL), управляются аппаратными и программными прерываниями;

- приоритеты низшего уровня (приоритеты планирования) - управляются планировщиком.

В табл. 6.4 приведена информация по используемым уровням запроса прерываний, их назначению и происхождению.

Таблица 6.4

Уровни запроса прерываний в Windows

Генерируются	Имя IRQL	Назначение
Аппаратура	HIHGEST_LEVEL	Проверка оборудования, ошибки шины
	POWER_LEVEL	Прерывания по отключению питания
	IPI_LEVEL	Межпроцессорные сигналы для мультипроцессорных систем
	CLOCK2_LEVEL	Интервальный таймер 2
	CLOCK1_LEVEL	Интервальный таймер 1
	PROFILE_LEVEL	Таймер профилирования (измерение производительности)
	DIRQLs	Платформеннозависимое число уровней для прерываний устройств ввода/вывода
Программы	DISPATCH_LEVEL	Планирование потоков и выполнение отложенных вызовов процедур
	APC_LEVEL	Выполнение асинхронных процедур
	PASSIVE_LEVEL	Уровень нормального исполнения потоков

Каждый нормально исполняющийся поток получает приоритет из диапазона 0-31 (минимальный приоритет 0 используется фоновым процессом обнуления страниц). Номера 16-31 соответствуют приоритетам реального времени. Номера 0-15 закрепляются за динамическими приоритетами.

Потоки реального времени прерываются только при появлении процессов более высокого приоритета. Такие процессы должны передавать управление другим процессам сами. Для процессов с динамическими приоритетами планировщик может временно повышать приоритет долго ожидающих процессов.

Все потоки с IRQL>=DISPATCH_LEVEL не планируются диспетчером, не могут ждать событий диспетчера (события, мьютексы и т.д.), и должны использо-

вать только невытесняемую память (non-paged pool). Драйверы также используют невытесняемую память. Коды пользовательского режима и режима ядра используют для доступа к памяти разные селекторы, поэтому разные виртуальные адреса могут соответствовать одному участку физической памяти. Диспетчер виртуальной памяти (Virtual Memory Manager, VMM) использует специальный список дескрипторов памяти (Memory Descriptor List, MDL) для описания физической памяти, применяемой в качестве буфера для конкретного процесса.

Когда драйвер работает в режиме ядра, отображение пользовательской памяти на невытесняемую память для временного хранения передаваемых данных (при записи - из пользовательского адресного пространства в системное, при чтении - из системного адресного пространства в пользовательское) производится VMM с помощью MDL.

6.1.4. Инструментарий создания драйверов для Windows NT/2000

Драйверы могут разрабатываться с помощью инструментария Driver Development Kit (DDK95, 98, 2000, XP), обычно работающего с Microsoft Visual C++ версии не ниже, чем 4.1. Отладка драйверов в режиме ядра обеспечивается отладчиками WinDbg (разработчик - Microsoft), SoftICE (разработчик - NuMega). Утилита BUILD из DDK собирает исполняемый файл драйвера. Инструментарий DDK имеет множество примеров.

6.1.5. Модель драйвера устройства

Драйвер - автономная программная компонента, которая может загружаться и выгружаться. Драйверы имеют унифицированный интерфейс, что позволяет диспетчеру ввода/вывода (I/O Manager) работать с ними, не зная деталей их внутренней реализации. Драйверы могут иметь слоистую структуру, что дает возможность их повторного использования. Их можно рассматривать как специальные динамически подключаемые библиотеки, хранящиеся в файлах с расширением ".sys" и находящиеся в директории \WINDOWS\System32\Drivers.

Драйвер не является процессом и не имеет потока исполнения, а исполняется в контексте того процесса, который к нему обратился.

Система драйверов Windows NT/2000 похожа на слоеный пирог, в котором на каждом уровне находятся определенные драйвера, обменивающиеся пакетами запроса ввода/вывода (IORP) между собой, аппаратурой и пользовательским приложением.

С точки зрения расположения драйверов в системе их можно классифицировать как:

- драйверы высшего уровня;
- драйверы промежуточного уровня;
- драйверы низшего уровня.

Драйверы высшего уровня осуществляют непосредственный контакт с пользовательской программой. Драйверы промежуточного уровня производят преобразование приходящих им данных от драйверов более высокого уровня и передачу пакета запроса далее по цепочке драйверов. Наконец, драйверы низшего уровня непосредственно контактируют с аппаратурой компьютера.

В случае, когда один и тот же драйвер выполняет все три функции, его называют монолитным, но в Windows 2000 это встречается редко.

С точки зрения сферы использования драйверы можно условно разделить на следующие типы:

- Драйверы режима ядра (Kernel mode drivers). Это основной тип драйвера. Такие драйверы используются для решения общих задач: управление памятью, шинами, прерываниями, файловыми системами, устройствами хранения данных и т.п.
- Графические драйверы (Graphics drivers). Как правило, создаются одновременно с самой видеокартой. Очень сложны в написании, так как должны учитывать множество противоречивых требований и поддерживать большое количество стандартов.
- Мультимедиа-драйверы (Multimedia drivers). Драйверы для аудиоустройств (считывание, воспроизведение и компрессия аудиоданных), устройств работы с видео (захват и компрессия видеоданных), позиционных устройств (джойстики, световые перья, планшеты и пр.)
- Сетевые драйверы (Network drivers) Эти драйверы работают с сетью и сетевыми протоколами на всех уровнях.
- Драйверы для виртуальных машин MS-DOS (Virtual DOS Drivers). Имитируют устаревшую операционную систему с целью обратной совместимости. Постепенно переходят в раздел рудиментарных.

Система ввода-вывода Windows 2000 имеет следующие особенности:

- Менеджер ввода-вывода NT предоставляет интерфейс для всех драйверов режима ядра, включая драйверы физических устройств, драйверы логических устройств и драйверы файловых систем.
- Операции ввода-вывода послойные. Это значит, что вызов, сделанный пользователем, проходит через несколько драйверов, генерируя несколько пакетов запросов на ввод-вывод и "по пути" обращаясь к необходимым драйверам. К примеру, когда приложение пытается открыть файл, подсистема ввода-вывода Windows делает запрос к драйверу файловой системы; драйвер файловой системы обращается к промежуточному драйверу; и лишь промежуточный драйвер обращается непосредственно к винчестеру. Такая архитектура построения системы существенно повышает ее гибкость и снижает общую стоимость разработки.
- Разработчик драйвера обязан реализовать несколько стандартных функций, к которым будет обращаться диспетчер ввода-вывода.

Архитектура драйвера Windows NT использует модель точек входа, в которой диспетчер ввода/вывода вызывает специфическую подпрограмму в драйвере, когда требуется, чтобы драйвер выполнил определенное действие. В каждую точку входа передается необходимый набор параметров для драйвера, чтобы дать ему возможность выполнить требуемую функцию.

Имеются следующие основные точки входа (классы точек входа):

1. **DriverEntry.** Диспетчер ввода/вывода вызывает эту функцию при первоначальной загрузке драйвера. Внутри этой функции драйверы выполняют инициализацию как себя, так и любых устройств, которыми они управляют. Эта точка входа должна быть у всех NT-драйверов.
2. **Диспетчерские (Dispatch) точки входа.** Диспетчерские точки входа драйвера вызываются диспетчером ввода/вывода, чтобы инициировать через драйвер некоторую операцию ввода/вывода.
3. **Unload.** Диспетчер ввода/вывода вызывает эту точку входа, чтобы запросить драйвер подготовиться к немедленному удалению из системы.
4. **Reinitialize.** Диспетчер ввода/вывода вызывает эту точку входа, если она была зарегистрирована, чтобы позволить драйверу выполнить вторичную инициализацию.
5. **Точка входа процедуры обработки прерывания (ISR- Interrupt Service Routine).** Эта точка входа присутствует только, если драйвер поддерживает обработку прерывания: его ISR будет вызываться всякий раз, когда одно из его устройств запрашивает аппаратное прерывание.
6. **Точки входа вызовов отложенных процедур (DPC - Deferred Procedure Call).** Драйвер использует эти точки входа, чтобы завершить работу, которая должна быть сделана в результате появления прерывания или другого специального условия, но была временно отложена.
7. **IoTimer.** Для драйверов, которые инициализировали и запустили поддержку IoTimer, диспетчер ввода/вывода вызывает эту точку входа каждую единицу времени.

Рассмотрим некоторые аспекты функционирования драйверов на примере драйвера устройства genport.sys, входящего в состав DDK. Файл, содержащий его описание, называется genport.inf и может быть использован при установке драйвера из мастера установки оборудования, входящего в состав Windows.

После установки драйвера информация о нем хранится в реестре операционной системы. Так, точка входа в драйвер - адрес функции с именем DriverEntry - определяется значением ключа Start.

Драйверы должны работать с внешними устройствами. Каждый раз, когда драйвер открывается, создается соответствующий файловый объект посредством функции API Win32 CreateFile. Причем реально это может быть не файл, тем не менее, одинаковое представление используется для описания всех внешних взаимодействий. Например, для работы с портом используем:

```
//для использования CreateFile, определенного в
//Winbase.h
#include <windows.h>
HANDLE  hndFile;              // Хэндл устройства
hndFile = CreateFile(
       "\\\\.\\GpdDev",     // Открыть "файл" устройства
          GENERIC_READ,     //вид доступа
          FILE_SHARE_READ, //последовательные операции
                           //открытия будут успешны только
                           //в случае доступа по чтению
          NULL, //указатель на атрибуты безопасности
          OPEN_EXISTING, //неудача, если файл не существует
          0, // определяет атрибуты файла
          NULL //определяет хэндлер с доступом GENERIC_READ
               //к файлу шаблона
);
```

Имя файла, используемое для вызова драйвера, определяется в Windows NT/2000 как

```
// NT имя устройства
#define GPD_DEVICE_NAME L"\\Device\\Gpd0"
```

Также для совместимости с NT 3 51 необходимо такое определение

```
#define DOS_DEVICE_NAME L"\\DosDevices\\GpdDev"
```

Между этими двумя именами должна быть установлена символическая связь

```
NTSTATUS GpdAddDevice(
    IN PDRIVER_OBJECT DriverObject,
    IN PDEVICE_OBJECT PhysicalDeviceObject
    )
/*++

Описание подпрограммы:

    Подсистема Plug&Play запрашивает драйвер для некоторого
устройства.
    Необходимо определить, нужно ли нам находиться в стеке
драйверов для данного устройства.
    Создать объект-устройство для помещения его в стек.
    Инициализировать этот объект-устройство
    Вернуть статусный код с сообщением об успехе.
```

```
Аргументы:

    DeviceObject - указатель на объект-устройство.

    PhysicalDeviceObject -  указатель на объект-устройство,
        созданный нижестоящим драйвером шины.
Возвращаемое значение:
    Статусный код NT.
--*/
{
    NTSTATUS                status = STATUS_SUCCESS;
    PDEVICE_OBJECT          deviceObject = NULL;
    PLOCAL_DEVICE_INFO      deviceInfo;
    UNICODE_STRING          ntDeviceName;
    UNICODE_STRING          win32DeviceName;

    PAGED_CODE();

    RtlInitUnicodeString(&ntDeviceName, GPD_DEVICE_NAME);

    //
    // создаем объект-устройство.
    //

    status = IoCreateDevice (DriverObject,
                             sizeof (LOCAL_DEVICE_INFO),
                             &ntDeviceName,
                             GPD_TYPE,
                             0,
                             FALSE,
                             &deviceObject);


    if (!NT_SUCCESS (status)) {
        // Нет памяти для создания объекта-устройства или
        // другой объект с таким же именем уже существует
        return status;
    }

    RtlInitUnicodeString(&win32DeviceName,
DOS_DEVICE_NAME);
```

```
    status = IoCreateSymbolicLink( &win32DeviceName,
&ntDeviceName );

    if (!NT_SUCCESS(status))    // если нельзя создать связь
    {                           // то прерывать инсталляцию.
        IoDeleteDevice(deviceObject);
        return status;
    }
```

Таким образом, драйверы используют многочисленные объекты: объект-драйвер (DRIVER_OBJECT), объект-устройство (DEVICE_OBJECT), объект-контроллер (CONTROLLER_OBJECT), объект-адаптер (ADAPTER_OBJECT). объект-прерывание (INTERRUPT_OBJECT).

Вызов драйвера (из CreateFile, например) обрабатывается сначала диспетчером ввода/вывода, который передает запросы драйверу как пакеты запроса ввода/вывода (I/O Request Packet, IRP), рассматриваемые соответствующей диспетчерской функцией (Dispatch Routine) драйвера. Диспетчерская функция драйвера проверяет параметры запроса, и если они корректны, передает IRP функции Start I/O Routine драйвера, которая использует содержимое IRP для запуска операции на устройстве. Когда операция завершится, функция драйвера DpcForISR фиксирует код финального состояния в IRP и возвращает его обратно диспетчеру ввода/вывода. Диспетчер ввода/вывода использует информацию из IRP для завершения обработки запроса и посылает пользователю код финального состояния.

В случае уровневых драйверов процедура значительно сложнее. Один IRP может передаваться между несколькими уровнями драйверов прежде, чем запрос будет завершен. Кроме того, драйверы верхнего уровня могут создавать новые IRP и передавать их другим драйверам.

Приведем общую схему взаимодействия драйвера с другими компонентами

Диспетчер ввода/вывода находит функцию DriverEntry по ее обязательному имени. Эта функция вызывается при загрузке драйвера. Для каждого драйвера создается объект-драйвер (Driver Object) и функция DriverEntry заполняет его поля (рис. 6.2). Она определяет устройства, которыми драйвер должен управлять, выделяет или подтверждает использование ресурсов (порты, прерывания, устройства прямого доступа к памяти - DMA), и делает имя драйвера видимым для всей системы.

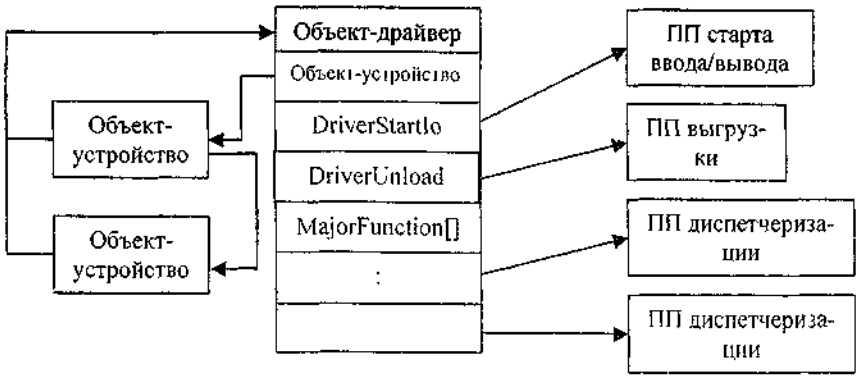


Рис. 6.2. Схема взаимодействия драйвера с другими компонентами

Функция Unload вызывается, когда драйвер должен быть динамически выгружен. Она должна произвести операции, противоположные тем, что сделаны DriverEntry.

Когда диспетчер ввода/вывода получает запрос от приложения, он преобразует тип запроса (чтение, запись и т.д.) в код функции. Диспетчер ввода/вывода идентифицирует драйвер и вызывает соответствующую функцию драйвера. Диспетчерская функция драйвера проверяет запрос и, если необходимо, делает запрос диспетчеру ввода/вывода на постановку этого запроса в очередь к устройству для выполнения реальной работы. Затем она возвращает управление диспетчеру ввода/вывода, помечая запрос как ожидающий обработки.

Каждый драйвер должен предоставлять такие диспетчерские функции, как CreateDispatch (для операции CreateFile) и CloseDispatch (для операции CloseHandle). В зависимости от устройства драйвер может иметь специальные диспетчерские функции. Если драйвер не поддерживает соответствующую операцию, приложение получает сообщение об ошибке.

Функция "Начать ввод/вывод" (Start I/O Routine) вызывается диспетчером ввода/вывода каждый раз, когда устройство должно начать передачу данных. Такой запрос генерируется всегда, когда процесс ввода/вывода завершается и есть очередной запрос в очереди. Функция "Начать ввод/вывод", предоставляемая драйвером, выделяет необходимые для работы устройства ресурсы и запускает устройство.

Диспетчер прерываний ядра вызывает функцию обработки прерываний драйвера (Interrupt Service Routine, ISR) всякий раз, когда устройство генерирует прерывание. Эта ISR доступна с помощью объекта-устройства (Device Object) и объекта-расширения устройства (Device Extension Object), показанных на рис. 6.3.

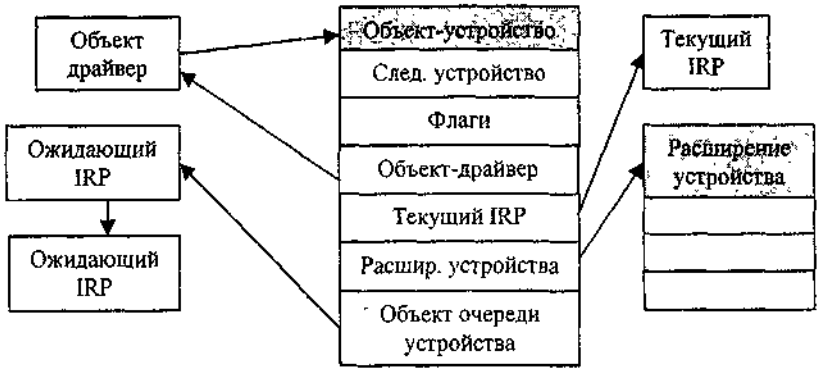


Рис. 6.3. Объект-устройство и связанные с ним объекты

Объект-расширение устройства имеет указатель на объект-прерывание (Interrupt Object), связанный с устройством, а у последнего есть указатель на ISR драйвера. Только минимальная обработка должна выполняться функцией ISR. Если требуется значительное время для выполнения обработки, она откладывается путем вызова функции DPC (Deferred Procedure Call) из ISR. Эта функция выполнится позднее, когда система будет работать на IRQL ниже, чем DIRQL (IRQL для устройств). Драйвер может иметь несколько (включая ноль) функций DPC для выполнения или завершения операций устройства.

Все коды ядра должны быть реентерабельными, и диспетчер ввода/вывода обеспечивает необходимые сервисы для синхронизации использующих разделяемые ресурсы процессов. Когда драйверу нужно обратиться к общему ресурсу, он с помощью диспетчера ввода/вывода ставит свой запрос в очередь к этому ресурсу, указывая функцию, которую необходимо активизировать при доступности ресурса. Когда ресурс освобождается, диспетчер ввода/вывода вызывает эту функцию.

Драйвер может также иметь таймерные функции (Timer Routines) для отслеживания прошедшего времени, функции завершения ввода/вывода (I/O Completion Routines) для уведомления драйвера верхнего уровня о завершении операции драйвером нижнего уровня, функции прерывания ввода/вывода (Cancel I/O Routines) для обеспечения возможности прерывания запроса инициатором.

Взаимное исключение может реализовываться блокированием прерываний (*interrupt blocking)* при помощи повышения своего IRQL вызовом KeRaiseIrql. Затем IRQL понижается с помощью KeLowerIrql или использованием DPC. DPC выполняются на одном относительно низком IRQL последовательно. Синхронизация в многопроцессорном режиме может проводиться с помощью объектов *spin lock* Объекты *spin lock* - это объекты взаимного исключения, связанные с группой структур данных. Когда код режима ядра хочет работать с любой защищенной структурой данных, он сначала должен запросить соответствующий объект *spin*

lock (KeAcquireSpinLock запрашивает, KeReleaseSpinLock - освобождает после использования). Так как только один процессор в каждый момент времени может владеть объектом *spin lock*, структура данных защищается от порчи.

6.2. Драйвера устройств операционных систем семейства Unix

Семейство Unix включает множество операционных систем, подчас значительно отличающихся друг от друга. Тем не менее, многие основные принципы реализованы во всех Unix-системах. Это касается и драйверов устройств.

На рис. 6.4 показано место драйверов в системе Unix.

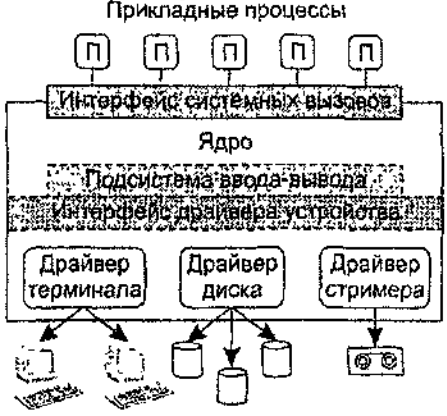


Рис. 6.4. Место драйверов в операционной системе Unix

В Unix драйверы, как и в случае Windows, относятся к подсистеме ввода-вывода. Подсистема ввода-вывода управляет всей независимой от аппаратуры частью операций ввода-вывода. Ей требуется обеспечить высокоуровневое процедурное представление устройств. С точки зрения подсистемы ввода-вывода устройство представляет собой "черный ящик", поддерживающий стандартный набор операций. Внутри каждого устройства эти операции реализованы по-разному, однако подсистема ввода-вывода не знает об этом

6.2.1. Классификация устройств и их драйверов

Единый интерфейс взаимодействия не подходит для всех существующих устройств, так как они существенно различаются между собой по функциональности и методам доступа. В системах UNIX все устройства разделены на два типа - *блочные* и *символьные*. Для каждого типа определен собственный интерфейс взаимодействия.

Блочное устройство хранит данные и производит ввод-вывод блоками фиксированного размера, доступными в произвольном порядке. Обычно размер блока

равняется 512 байтам, умноженным на 2 в степени, где степень больше либо равна нулю. В качестве примеров блочных устройств можно привести жесткие диски, флоппи-дисководы и приводы компакт-дисков. Файловая система UNIX способна храниться только на устройствах блочного типа. Ядро взаимодействует с такими устройствами через драйверы, используя специальные структуры *buf*, в которых содержится вся необходимая информация об операциях ввода-вывода.

Символьные устройства могут использоваться для хранения и передачи данных произвольного объема. Некоторые устройства этого типа умеют передавать информацию побайтно, вырабатывая каждый раз прерывание. Другие устройства способны поддерживать внутреннюю буферизацию. Ядро интерпретирует данные от символьных устройств как непрерывный поток байтов, доступный в последовательном режиме. Символьные устройства не в состоянии использовать произвольную адресацию и не поддерживают операцию поиска. Примерами устройств такого типа являются терминалы, принтеры, манипуляторы, звуковые карты и т.п.

Не все устройства можно отнести к одной из вышеописанных категорий. В системах UNIX любое устройство, не обладающее свойствами блочного типа, классифицируется как символьное. Некоторые устройства вообще не производят никакого ввода-вывода. Например, аппаратный таймер является устройством, обеспечивающим через определенный интервал времени прерывание работы процессора. Отображаемые в память дисплеи являются доступными в произвольном порядке, но они считаются символьными устройствами. Некоторые блочные устройства, такие как диски, обеспечивают символьный интерфейс взаимодействия, так как он является более эффективным для определенных операций.

Драйвер не всегда управляет физическим устройством. Он может использоваться в качестве интерфейса доступа, поддерживающего дополнительные функции. Например, драйвер mem позволяет считывать из адресов физической памяти или записывать по ним. Устройство null разрешает только запись в себя, производя удаление всех получаемых данных. Устройство zero является источником памяти, заполняемой нулями. Перечисленные выше устройства получили название *псевдоустройств*.

Важным преимуществом драйверов псевдоустройств является тот факт, что именно они позволяют сторонним производителям добавлять в ядро UNIX новые средства. Драйверы UNIX поддерживают общецелевую точку входа под названием *ioctl*. Эта процедура может быть вызвана с большим количеством специфичных для определенного драйвера команд в качестве входного параметра, что позволяет драйверам псевдоустройств обеспечивать на прикладном уровне богатый выбор способов вызова функций ядра, причем, нет нужды изменять коды ядра.

Современные системы UNIX поддерживают еще один класс устройств, называемый *STREAMS*. Устройства этого типа обычно используются для управления сетевыми интерфейсами и терминалами, а также заменяют собой устройства сим-

вольного типа для более ранних реализаций. С точки зрения сохранения обратной совместимости интерфейс драйверов STREAMS унаследован от символьных устройств

6.2.2. Запуск драйвера

Ядро запускает драйвер устройства в нескольких случаях.

- Настройка. Ядро обращается к драйверу во время загрузки системы с целью проверки и инициализации устройства.
- Ввод-вывод. Подсистема ввода-вывода вызывает драйвер для чтения или записи данных.
- Управление. Пользователь может создать управляющий запрос к устройству, например, открытие или закрытие устройства или перемотка ленты в накопителе.
- Прерывания. По завершении операции или при изменении состояния устройства генерируется прерывание.

Функции настройки устройства отрабатывают только один раз при загрузке системы. Операции ввода-вывода и управления являются синхронными. Они запускаются в ответ на определенные запросы процесса и выполняются в его контексте. Процедура *strategy* блочного драйвера является исключением из правила. Прерывания - это асинхронные события в системе, так как ядро не может заранее знать о времени их возникновения. Их обработка не производится в контексте какого-либо определенного процесса.

Драйвер делится на две части, которые обычно называются верхней и нижней половинами (top half и bottom half). Верхняя половина драйвера содержит синхронные процедуры, в нижней располагаются процедуры асинхронного типа Синхронные процедуры могут обращаться к адресному пространству и области вызывающего процесса и даже умеют переводить процесс в режим сна, если это необходимо. Процедуры нижней половины выполняются в системном контексте и, как правило, не имеют отношения к текущему процессу. В любом случае такие процедуры не обладают правом доступа к адресному пространству текущего процесса. Они не должны переводить процесс в режим сна, поскольку это может привести к блокировке другого процесса.

Обе составляющие драйвера должны обладать средствами синхронизации своих действий друг с другом. Если объект доступен обеим "половинам", то процедуры верхнего уровня должны произвести блокировку прерываний при манипуляции таким объектом. В ином случае, устройство может создать прерывание в тот момент, когда объект находится во внутренне противоречивом состоянии, что приведет к неожиданным результатам.

Кроме обращения ядра к драйверу, последний умеет сам загружать функции ядра для выполнения таких задач, как управление буфером, доступом или диспетчеризации событий.

6.2.3. Точки входа драйвера

Как и в Windows драйвера Unix построены на модели точек входа. Для каждого драйвера имеется специальная структура-переключатель, содержащая информацию о его точках входа. Определены следующие основные точки входа:

1. **d_open()**. Вызывается каждый раз при открытии устройства, может сделать устройство активным или инициализировать структуры данных. Устройства, требующие эксклюзивного доступа (такие как принтеры или накопители на магнитных лентах) могут устанавливать флаг при открытии и сбрасывать его при закрытии. Если флаг оказывается уже установленным, d_open() блокируется или завершается с ошибкой. Процедура является общей для символьных и блочных устройств.
2. **d_close()**. Вызывается в том случае, если освобождается последняя ссылка на устройство, то есть не остается ни одного процесса, для которого это устройство открыто. Процедура вправе завершить работу устройства или перевести его в автономный режим. Процедура является общей для символьных и блочных устройств.
3. **d_strategy()**. Общая точка входа для запросов на чтение и запись с устройства. Процедура получила такое название из-за возможности драйвера определять некоторую стратегию перераспределения ожидающих запросов с целью повышения производительности. Она работает асинхронно, т.е. если устройство занято, процедура просто помещает запрос в очередь и завершает работу. После завершения ввода-вывода обработчик прерываний удаляет из очереди следующий запрос и начинает новую операцию ввода-вывода.
4. **d_size()**. Используется дисковыми устройствами для определения размеров дисковых разделов.
5. **d_read()**. Считывает данные с символьного устройства.
6. **d_write()**. Записывает данные в символьное устройство.
7. **d_ioctl()**. Общая точка входа управляющих операций символьных устройств. Каждый драйвер может определить набор команд, загружаемых посредством интерфейса ioctl. Одним из параметров таких функций является cmd, целое число, соответствующее выполняемой команде. Параметр arg указывает на специфический для команды набор аргументов. Это удобная точка входа, поддерживающая богатый набор операций с устройством.
8. **d_segmap()**. Отображает память устройства в адресном пространстве процесса. Используется символьными устройствами, отображаемыми в память при применении системного вызова mmap.
9. **d_mmap()**. Применяется только в случае поддержки процедуры d_segmap(). Если d_segmap равняется NULL, системный вызов mmap по отношению к символьному устройству вызывает операцию

spec_segmap(), которая, в свою очередь, вызывает d_mmap(). Проверяет корректность указанного смещения устройства и завершает работу с возвратом соответствующего виртуального адреса.

10. **d_xpoll()**. Опрашивает устройство с целью проверки наступления определенного события. Может использоваться для проверки готовности устройства для чтения или записи без блокировки, если произошла ошибка и т. д.
11. **d_xhalt()**. Завершает работу устройства, управляемого драйвером. Вызывается при завершении работы системы либо при выгрузке драйвера из ядра.

Структуры переключателя немного отличаются друг от друга в различных реализациях UNIX. Например, в некоторых вариантах переключатель блочного устройства содержит дополнительные функции, такие как d_ioctl(), d_read() и d_write(). Другие версии могут поддерживать функции инициализации или реакции на сброс шины.

Все перечисленные процедуры (кроме d_xhalt() и d_strategy()) принадлежат к верхнему уровню. Процедура d_xhalt() вызывается при завершении работы, вследствие чего не может воспользоваться контекстом прикладного процесса и даже прерываниями. В любом случае эта процедура не переводится в режим сна.

Операция d_strategy() является специализированной. Она часто запускается с целью чтения или записи буферов, не относящихся к вызывающему ее процессу. Например, процесс пытается получить свободный буфер. Для этой цели он находит в списке свободных буферов первый по счету "грязный" буфер и вызывает процедуру strategy для выгрузки его содержимого на диск. После вызова операции записи процесс запрашивает следующий буфер (считая, что он свободен) и начинает его использовать. В дальнейшем процесс не интересуется состоянием записываемого буфера и не ожидает окончания операции записи. Дисковые операции ввода-вывода часто являются несинхронными (как в только что приведенном примере) и драйвер не должен блокировать вызывающий процесс.

Из вышесказанного можно заключить, что операция d_strategy() интерпретируется как процедура нижнего уровня. Она инициализирует операцию ввода-вывода и завершает работу, не ожидая окончания ввода-вывода. Если при получении запроса устройство окажется занято, d_strategy() просто добавит этот запрос во внутреннюю очередь устройства и завершит выполнение. Через некоторое время из кода прерывания будут вызваны другие процедуры нижнего уровня, которые удалят запрос из очереди и выполнят его. Если вызывающему процессу необходимо ждать завершения ввода-вывода, он производит это вне рамок процедуры d_strategy().

Точки входа драйвера для обработки прерываний и инициализации, как правило, недоступны посредством таблицы переключателя. Вместо этого они указываются в главном файле конфигурации, используемом при сборке ядра системы.

Такой файл содержит вхождения для каждого контроллера и драйвера. Во вхождении отражается также такая информация, как уровни приоритета прерывания, номер вектора прерываний и базовый адрес драйвера. Спецификации содержимого и формата файла зависят от реализации UNIX.

6.2.5. Файлы устройств

Внутри операционной системы драйверы идентифицируются парой чисел - старшим и младшим номером, что не очень удобно при работе с драйверами из программ. Поэтому система UNIX поддерживает единый интерфейс доступа к файлам и устройствам при помощи такого понятия, как *файл устройства* (device file), ассоциируемый с определенным устройством. По соглашению все файлы устройств находятся в каталоге /dev или его подкаталогах.

С точки зрения пользователя файл устройства ничем не отличается от обычного файла. Пользователь может открывать или закрывать такой файл, читать или писать данные и даже производить позиционирование по заданному смещению (однако последнее поддерживается лишь небольшим количеством устройств). Командный интерпретатор умеет перенаправлять потоки stdin, stdout или stderr в файл устройства. Их содержимое преобразуется в определенные действия устройства, представленного собственным файлом устройства. Например, запись данных в файл /dev/lpr приводит к печати этих данных на линейный принтер.

"Изнутри" файл устройства существенно отличается от обычного файла. Он не имеет блоков данных, хранимых на диске, но обладает постоянным индексным дескриптором в той файловой системе, где он расположен (обычно в корневой файловой системе). Ядро осуществляет преобразование прикладного номера устройства (пути) в его внутренний номер (пару: старший номер, младший номер).

Унификация пространств имен файлов и устройств дала массу преимуществ. Для ввода-вывода устройств используется тот же набор системных вызовов, что и для ввода-ввода файлов. Программисты могут создавать приложения, не вникая в подробности, является ли ввод или вывод программы обычным файлом или устройством. Пользователи видят систему единообразно, и для обращения к устройствам вправе использовать понятные символьные обозначения.

Еще одним важным преимуществом единого пространства имен является управление доступом и защита. Некоторые операционные системы, такие как DOS, предоставляют неограниченный доступ к устройствам всем пользователям, в то время как операционные системы мэйнфреймов не разрешают прямой доступ к устройствам. Ни одна из этих схем не является приемлемой. Унификация пространств имен файловой системы и устройств в UNIX позволила расширить механизм защиты файлов на устройства. В каждом файле устройства установлены права доступа на чтение/запись/выполнение для владельца, группы и остальных. Эти права инициализируются и изменяются стандартными методами как для обычных файлов. Традиционно некоторые устройства, в том числе диски, могут быть дос-

тупны напрямую только привилегированному пользователю, в то время как другие устройства, накопители на магнитных лентах, доступны всем пользователям.

Контрольные вопросы и задания

1. Для чего нужны драйверы устройств?
2. Каковы особенности операционных систем Windows?
3. Для чего нужен HAL?
4. Какая модель драйверов используется в Windows?
5. Приведите классификацию драйверов Windows с точки зрения пользователя.
6. Что такое точки входа драйверов? Какие они бывают для Windows?
7. Как можно получить доступ к драйверу в Windows?
8. Приведите классификацию драйверов в UNIX.
9. Что такое переключатели устройств?
10. Какие точки входа могут быть у драйверов в UNIX?
11. Как используются файлы устройств при доступе к драйверам в UNIX?

Глава 7. Системное и прикладное программное обеспечение систем реального времени

Программные комплексы СРВ содержат в себе как прикладную, так и системную составляющую. К системной составляющей можно отнести специальные операционные системы реального времени, к прикладной - SCADA-системы. Рассмотрим оба класса программного обеспечения подробно.

7.1. Операционные системы реального времени

Материал ниже базируется на [1].

Операционные системы реального времени (ОС РВ - real time operating systems, RTOS) значительно отличаются от обычных операционных систем. Так, например, на них накладываются жесткие временные ограничения на время отклика. Ниже мы рассмотрим особенности ОС РВ.

7.1.1. Основные характеристики операционных систем реального времени

Как правило, ОС РВ должны быть модульными и расширяемыми. Любую операционную систему можно условно разделить на уровни:

1. пользовательский интерфейс - оболочка - исполнительные модули операционной системы;
2. поддержка файлов и дисков - расширенные и основные модули ввода/вывода;
3. межзадачные синхронизация и взаимодействие - ядро;
4. планирование задач - микроядро;
5. управление потоками/задачами, управление блоками памяти - наноядро.

Заметим, что все три нижних уровня могут быть объединены в одно ядро.

Когда создается поток, с ним ассоциируется блок управления потоком. В дополнение к обычно используемой здесь информации, такой как идентификатор потока, начальный адрес, контекст, информация для синхронизации, для ОС РВ здесь может быть также тип задачи (периодическая, апериодическая), фаза, период, относительный директивный срок, количество запусков, список событий. Например, периодическая задача с конечным числом запусков завершается и может быть уничтожена ядром после исполнения указанное количество раз.

Часы РВ очень важны для ОС РВ, они дают сигнал прерывания через определенный промежуток времени, называемый тиком. Для большинства систем величина тика - 10 миллисекунд. По каждому прерыванию таймера ядро выполняет следующие действия:

1. Обрабатывает события таймера (таймер имеет очередь, и ядро определяет, какие события из очереди произошли) и выполняет соответствующие действия (например, активизацию некоторых задач).
2. Обновляет бюджет времени исполнения: уменьшает бюджет исполняющегося потока для выяснения того, продолжать его исполнение или прервать.
3. Обновляет очередь готовых потоков и возвращает управление. Если текущий поток должен быть прерван, производится перепланирование и управление передается потоку в голове очереди высшего приоритета.

ОС РВ обычно позволяют приложениям создавать собственные программные таймеры: периодические и одноразовые.

Основными временными характеристиками ОС РВ являются время отклика системы или время реакции системы, т.е. максимальное время, за которое система может отреагировать на внешнее событие (прерывание); время переключения контекста - время, за которое происходит переход от одной параллельной задачи к другой (от одной нити к другой).

В случае внешних прерываний, например, по приходу сообщения по сети, обработчик прерывания должен определить, какому потоку оно адресовано, доставить его в адресное пространство потока, выполнить протокол подтверждения для источника сообщения и т.д. Такие действия могут требовать десятки миллисекунд. Поэтому большинство ОС РВ разделяют обработку прерывания на две части: служба немедленной обработки прерывания, которая исполняется на высоком уровне запроса обработки прерываний, и служба отложенной обработки прерываний, которая исполняется на подходящем уровне приоритета. Первый шаг подготавливает всю необходимую информацию для последующей обработки нормальным образом в виде обычной задачи.

Как правило, ОС РВ поддерживают до 256 уровней приоритета. Приоритеты чаще всего назначаются по возрастанию периода (rate monotonic) и по возрастанию относительного директивного срока (deadline monotonic). Эти стратегии обеспечивают исполнение в первую очередь задач с меньшим периодом или ранним директивным сроком и были разобраны ранее (см. 3.9.6, 5.1, 5.2.3).

Потоки могут запрашивать режим работы без прерываний при использовании критических ресурсов, что также должно поддерживаться ОС РВ. Обычно поддерживаются протоколы наследования приоритетов - когда приложение использует высокоприоритетный ресурс, оно выполняется на этом высоком уровне приоритета. Эти протоколы также рассматривались ранее (см. 3.9.6).

В случае многопроцессорных систем ОС РВ должны иметь маску процессоров для каждой задачи, показывающую, на каких процессорах можно запускать эту задачу.

При использовании виртуальной памяти ОС РВ должна предоставлять приложению возможность фиксировать страницу в памяти, чтобы запретить ее выгрузку на диск.

Для повышения эффективности ОС РВ может иметь несколько планировщиков, свой для каждой группы задач: не РВ, РВ периодических, РВ апериодических, задач связи. Также должен быть общий планировщик, который решает, какую задачу из числа выбранных частными планировщиками запустить

7.1.2. Классификация операционных систем реального времени

Условно существующие ОС РВ можно разделить на три класса в зависимости от их архитектурных особенностей: монолитные, модульные и объектно-ориентированные.

ОС РВ с ***монолитной архитектурой*** можно представить, как показано на рис. 7.1. Она включает в себя:

- прикладной уровень: состоит из работающих прикладных процессов;
- системный уровень: состоит из монолитного ядра операционной системы, в котором можно выделить следующие части: интерфейс между приложениями и ядром (API), собственно ядро системы, интерфейс между ядром и оборудованием (драйверы устройств).

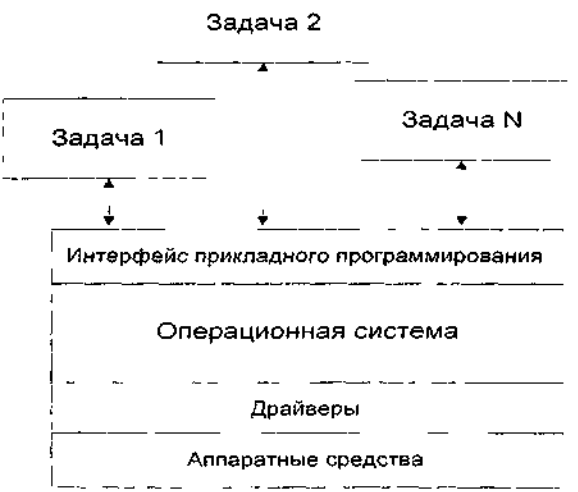


Рис 7.1. ОС РВ с монолитной архитектурой

Интерфейс в таких системах играет двойную роль С одной стороны, он нужен для управления взаимодействием прикладных процессов и системы С другой - для обеспечения непрерывности выполнения системных компонентов, т.е. отсутствия переключения задач во время исполнения системного кода

Основным преимуществом монолитной архитектуры по сравнению с другими архитектурами является ее относительно высокая производительность. Однако достигается это, как правило, за счет написания значительных частей системы на ассемблере.

Недостатки монолитной архитектуры:

1. Системные вызовы, требующие переключения уровней привилегий (от пользовательской задачи к ядру), должны быть реализованы как прерывания или специальный тип исключений. Это сильно увеличивает время их работы.
2. Ядро не может быть прервано пользовательской задачей. Это может приводить к тому, что высокоприоритетная задача не получит управления из-за работы низкоприоритетной.
3. Сложность переноса на новые аппаратные архитектуры из-за значительных ассемблерных вставок.
4. Негибкость и сложность развития: изменение части ядра системы требует его полной перекомпиляции.

Модульная архитектура появилась как попытка убрать интерфейс между приложениями и ядром, облегчить модернизацию системы и перенос ее на новые процессоры.

Здесь микроядро играет двойную роль (рис 7.2):

1. управление взаимодействием частей системы (например, менеджеров процессов и файлов);
2. обеспечение непрерывности выполнения кода системы, т.е. отсутствие переключения задач во время исполнения микроядра.

Рис. 7.2. ОС РВ на основе микроядра

Недостатки модульной архитектуры фактически те же, что и у монолитной, но теперь они сместились с уровня интерфейса на уровень микроядра. Системный интерфейс по-прежнему не допускает переключения задач во время работы микроядра, только сократилось время пребывания в этом состоянии, проблемы с переносимостью микроядра уменьшились в связи с сокращением его размера, но не исчезли вовсе.

В ***объектной архитектуре*** интерфейс между приложениями и ядром отсутствует вообще (рис. 7.3). Взаимодействие между компонентами системы - микроядрами и пользовательскими процессами осуществляется посредством обычного вызова функций, поскольку и система, и приложения написаны на одном языке (обычно C++). Это обеспечивает максимальную скорость системных вызовов.

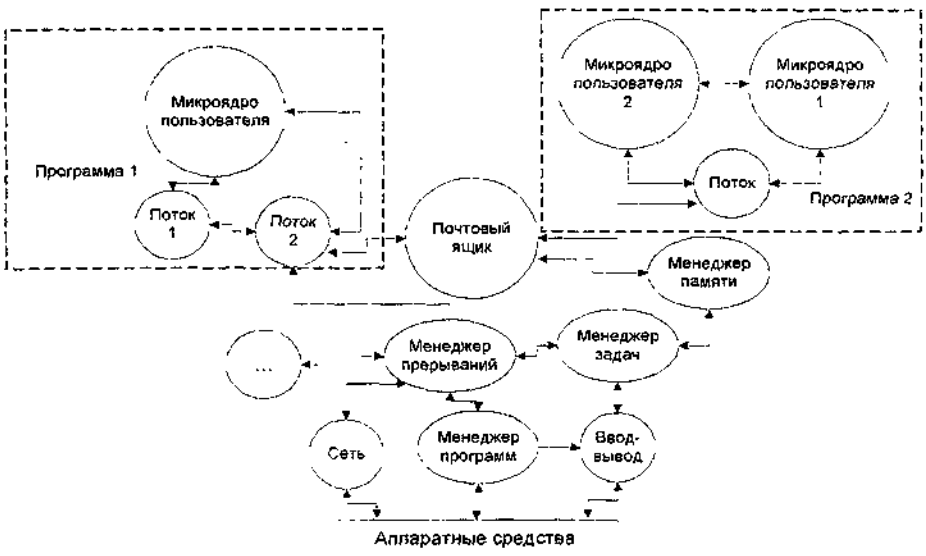


Рис. 7.3. Пример объектно-ориентированной ОС РВ

Фактическое равноправие всех компонент системы дает возможность переключения задач в любое время. Объектно-ориентированный подход обеспечивает модульность, безопасность, легкость модернизации и повторного использования кода.

В отличие от предыдущих системных архитектур не все компоненты самой операционной системы должны быть загружены в оперативную память. Если микроядро уже загружено для другого приложения, то повторно оно не загружается, а используется код и данные уже имеющегося микроядра. Все эти приемы позволяют сократить объем требуемой памяти.

Недостатками данной архитектуры является сложность реализации ОС РВ, построенных на ее основе, а также относительно невысокое быстродействие.

7.1.3. Наиболее известные операционные системы реального времени

Коммерческие ОС РВ имеют характеристики, в основном, удовлетворяющие стандарту - программному интерфейсу переносимых ОС РВ (Real-Time POSIX (Portable OS Interface) API).

К ОС РВ можно отнести LynxOs, pSOS, QNX, VRTX, vxWorks. Ниже приведены характеристики некоторых из них.

Операционная система QNX

ОС РВ QNX, производимая компанией QNX Software Systems (США), используется преимущественно в приложениях, где основными требованиями является функционирование в реальном времени, компактность и гибкая работа в сети. Учитывая свойства QNX как эффективной многозадачной, многопользовательской и многоконсольной ОС РВ с малым гарантированным временем отвега на событие, основные области ее применения - сложные автоматизированные системы управления процессами в промышленности, энергетике, на транспорте, в области коммуникаций, а также для автоматизации банковских систем, систем управления полетами и т.п. Вследствие своей компактности QNX может эффективно применяться при разработке надежных и недорогих интеллектуальных микроконтроллеров и бездисковых микрокомпьютеров со встроенным программным обеспечением, работающих как автономно, так и в составе сложных сетевых систем для управления роботами, медицинским, авиационным оборудованием и т.п.

Кроме того, QNX является высокопроизводительной сетевой ОС РВ для персональных компьютеров типа IBM PC, специально разработанной для систем управления процессами.

Основу QNX составляет приоритетно управляемое микроядро. Оно дает возможность работы в реальном времени в защищенном режиме, устраняет ограничения по памяти и не допускает нарушения целостности данных за счет устойчивой к отключениям питания файловой системы. Технология с микроядром является функционально наиболее подходящей для приложений с высокими требованиями на время реакции системы (порядка нескольких микросекунд).

Микроядро QNX имеет несколько преимуществ. Во-первых, как и любое ядро ОС, предназначенной для работы во встраиваемом варианте, ядро QNX очень мало в размерах. Так, для QNX 4.x оно занимает 12 Кб памяти. Во-вторых, микроядро содержит только две основные функции:

- управление прохождением сообщений - микроядро управляет прохождением всех сообщений между процессами;

- управление выполнением запущенных процессов (диспетчеризация процессов); диспетчер - это часть самого микроядра, и он запускается всякий раз, когда какой-либо процесс изменил свое состояние в результате получения сообщения или в результате выполнения аппаратного прерывания.

При обычных условиях ядро недоступно [35]. Оно становится доступно только по прямому вызову из какого-либо системного процесса или аппаратного прерывания.

Все службы QNX, исключая внутренние службы самого ядра, управляются системными процессами. Типичная конфигурация QNX включает в себя следующие системные процессы:

- менеджер процессов (Proc);
- менеджер файловой системы (Fsys);
- менеджер устройств (Dev);
- менеджер сети (Net).

Системные процессы практически ничем не отличаются от процессов пользователя, они не имеют ни скрытых, ни так называемых "частных" интерфейсов, в то же время системные процессы недоступны для прямого вызова из пользовательских процессов или прикладных программ.

Такая особенность QNX предоставляет широкие возможности расширения функциональности системы. При различных вариантах запуска исполняемой программы в QNX, она может быть представлена как системный процесс или как процесс пользователя.

При одновременной работе нескольких процессов, как в типичной многозадачной СРВ, ОС должна обеспечить устойчивый механизм для обмена информацией между запущенными процессами. QNX предоставляет простой, но при этом довольно мощный механизм межпроцессного обмена информацией - сообщения. Сообщения в QNX представляют собой пакет байтов, посланный от одного процесса к другому.

Сообщения доставляются в соответствии с приоритетами, и сервисный процесс исполняется на приоритете клиента (на уровне приоритета процесса клиента высшего приоритета из очереди). Сообщения в QNX передаются с помощью каналов. Функции QNX посылки и приема сообщений являются блокирующими (в POSIX отправитель не блокируется). Таким образом, обмен сообщениями обеспечивает не только передачу информации, но является способом синхронизации параллельной работы нескольких процессов.

Любой процесс на любом компьютере в сети может быть использован с другого компьютера. С точки зрения исполняемых в QNX программ нет различий между локальными и удаленными ресурсами, т.е. нет необходимости встраивать в программу дополнительные средства для доступа к удаленным ресурсам. Фактически программа должна иметь только средства для определения того, где нахо-

дится ресурс типа файла или устройства - на локальном компьютере или каком-либо другом узле сети.

Пользователи могут получать доступ к файлам в любом узле сети, использовать любое периферийное устройство и запускать любые программы на компьютерах, находящихся в сети, если, конечно, имеются соответствующие права доступа. QNX предоставляет широкие возможности по созданию приложений, состоящих из нескольких процессов и выполняющихся на нескольких распределенных компьютерах.

Отметим также, что QNX является многопроцессорной операционной системой. Она использует модель симметричного мультипроцессора с несколькими гигабайтами физической памяти.

Версия QNX - Neutrino, обеспечивает атомарные операции увеличения и уменьшения переменной, установки и очистки битов, битового дополнения. Она предоставляет 32 уровня приоритетов, использует дисциплины планирования FIFO, кругового циклического обслуживания, прерывания. Для синхронизации применяются семафоры, мьютексы, условные переменные.

Время переключения контекста для QNX составляет порядка 13 мкс, время реакции - 4,3 мкс.

Операционная система VxWorks

ОС РВ VxWorks создана компанией WindRiver System, США. Она относится к ОС жесткого РВ. Характерной чертой этой ОС является то, что благодаря ее развитым сетевым возможностям, вся разработка программного обеспечения ведется на инструментальном компьютере (хост-системе) с использованием кросс-средств, а последующее исполнение происходит на целевой машине под управлением VxWorks.

Отличительная черта системы - возможность управлять работой сложных комплексов реального времени и бортовых устройств, использующих процессорные элементы различных поставщиков. Три основных компонента данной ОС РВ образуют единую интегрированную среду: собственно ядро системы, управляющее процессором; набор средств межпроцессорного взаимодействия; комплект коммуникационных программ для работы с Ethernet или последовательными каналами связи.

Основные характеристики:

1. монолитная архитектура;
2. поддержка многозадачности и многопроцессорности;
3. 256 уровней приоритетов;
4. время реакции - 4 мкс, время переключения контекста - 15 мкс;
5. приоритетное планирование, прерываемое ядро;
6. минимальный размер: 22K;
7. средства синхронизации и взаимодействия: семафоры POSIX, очереди, сигналы.

Эта система была использована на марсоходе Mars Pathfinder, который NASA посылала на Марс в 1997 г. После спуска на Марс у марсохода были проблемы с постоянным перезапуском компьютера. Причиной стала инверсия приоритетов, приводившая к недопустимо большой задержке выполнения важных задач, что провоцировало перезапуск системы. Эта же операционная система использовалась в 2004 г. на марсоходах Spirit и Opportunity, у которых также возникали проблемы с перезагрузкой. Например, Spirit пытался выделить больше файлов, чем могла принять файловая система, основанная на RAM. Это вызывало исключение, приводящее к подвешиванию задачи, что вело, в свою очередь, к перезапуску, во время которого система пыталась перемонтировать флэш-файловую систему. Но программная утилита была не в состоянии выделить достаточно памяти для директории в RAM, вызывая ее прекращение и т.д [36].

Операционная система OS-9

OS-9 фирмы Microware относится к классу UNIX-подобных ОС РВ и подходит для приложений жесткого РВ. По своей сути OS-9 - это многозадачная ОС с вытесняющей приоритетной многозадачностью, допускающая возможность многопользовательской работы. Объектно-ориентированный модульный дизайн системы позволяет конфигурировать систему в очень широком диапазоне от встраиваемых систем до больших сетевых приложений. Согласно этой концепции все функциональные компоненты OS-9, включая ядро, иерархические файловые менеджеры, драйвера устройств и т.д., реализованы в виде независимых модулей. Все модули операционной системы позиционно независимы и могут быть размещены в ПЗУ, а также могут удаляться из системы в процессе ее функционирования без какой-либо повторной инсталляции или перекомпоновки.

К основным характеристикам данной ОС можно отнести:

1. многозадачность и многопроцессорность;
2. 65535 уровней приоритетов;
3. время реакции системы - 3 мкс;
4. планирование задач: приоритетное, FIFO, RR;
5. ядро, допускающее прерывание;
6. для синхронизации и взаимодействия применяются разделяемая память, сигналы, семафоры, события;
7. минимальный размер ОС - 16 К;
8. доступность средств кроссплатформенной разработки на базе Unix/Windows.

7.1.4. Операционные системы семейства Linux и задачи реального времени

Для задач РВ применяют расширения РВ для Linux: RTLLinux [37], KURT [38] и др. Эти продукты позволяют получить устойчивую среду РВ в рамках данной операционной системы. Приведем их краткое описание.

KURT представляет собой систему мягкого РВ Этот проект основан на минимальных изменениях ядра Linux и предоставляет разработчику два режима работы нормальный (normal mode) и РВ (real-time mode) В любой момент времени процесс, использующий библиотеку интерфейсов API KURT, может переключаться между этими двумя режимами, которые позволяют процессу работать как в режиме РВ, так и в нормальном режиме ядра Linux

Программный пакет KURT выполнен в виде отдельного системного модуля Linux RTMod, который становится дополнительным планировщиком РВ Планировщик РВ доступен в нескольких вариантах и может тактироваться от любого таймера в системе или от прерываний стандартного параллельного порта

Так как все процессы работают в общем пространстве процессов Linux, программист использует в своих программах стандартные API-интерфейсы Linux и может переключаться из одного режима в другой по событиям или в нужном месте программы с применением API-интерфейсов KURT При переключении в режим РВ все процессы в системе засыпают до момента освобождения ветви процесса РВ Это довольно удобно при реализации задач с большим количеством вычислений, требующих по своей сути механизмов РВ Примером может служить подмножество задач обработки аудио-видео информации

Стандартно планировщик RTMod тактируется от системного таймера и время переключения контекста задач РВ равно 10 мс Используя KURT совместно с системой UTIME можно довести время контекстного переключения задач до 1 мс Прерывания обрабатываются стандартным для Linux образом, т е используется механизм драйверов

API-интерфейс KURT делится на две части прикладную и системную Прикладная часть позволяет программисту управлять поведением своих процессов, а системный API-интерфейс KURT предназначен для манипулирования пользовательскими процессами и написания собственных планировщиков

Совершенно другой подход применен при реализации в Linux жесткого РВ К системам такого рода можно отнести RTLinux - дополнение к ядру Linux, которое позволяет управлять различными чувствительными ко времени реакции системы процессами

RTLinux - операционная система в которой маленькое ядро РВ сосуществует с POSIX-ядром Linux

Разработчики RTLinux пошли по пути который предусматривает запуск из ядра РВ ядра Linux как задачи с наименьшим приоритетом В RTLinux все прерывания обрабатываются ядром РВ и в случае отсутствия обработчика РВ передаются ядру Linux Фактически ядро Linux является простаивающей задачей ОС РВ, запускаемой только в том случае если никакая задача РВ не исполняется При этом на Linux-задачу накладываются некоторые ограничения которые, впрочем, прозрачны для программиста Linux не может выполнять следующие операции блокировать аппаратные прерывания предохранять себя от вытеснения другой

задачей. Ключ к реализации данной системы - эмулирующий систему управления прерываниями драйвер, к которому обращается Linux при попытке блокировать прерывания. В этом случае драйвер перехватывает запрос, сохраняет его и возвращает управление ядру Linux.

Ядро RTLinux спроектировано таким образом, что оно никогда не нуждается в ожидании освобождения ресурса, занятого Linux -процессом.

Ключевой принцип построения RTLinux - как можно больше использовать Linux и как можно меньше RTLinux. Действительно, Linux заботится об инициализации системы и устройств, а также о динамическом выделении ресурсов. На RTLinux ложится только планирование задач РВ и обработка прерываний. Процессы РВ реализованы в виде загружаемых модулей Linux для простоты запуска в контексте ядра, сохранения модульности и расширяемости системы.

7.1.5. Операционная система Windows NT и задачи реального времени

Windows NT проектировалась без учета требований, предъявляемых к ОС РВ - она разработана как операционная система общего назначения (точнее, как сетевая ОС). Тем не менее, благодаря тому, что Windows NT создавали разработчики ОС РВ, в нее заложены некоторые механизмы, присущие этим системам. Например, введены классы процессов РВ, которые планируются так же, как и в ОС РВ. Кроме того, при обслуживании прерываний используется очень эффективный алгоритм. Рассмотрим, достаточно ли этого, чтобы квалифицировать Windows NT как ОС РВ.

В СРВ не все задания имеют одинаковые приоритеты. Критичные по времени задания имеют высокие приоритеты. Другие, такие, как отображение состояния системы, запись сообщений о событиях в файле протокола или конфигурирование системы, имеют низкие приоритеты. Поэтому ОС должна уметь назначать заданиям различные приоритеты.

При оптимальном проектировании системы различным нитям присваиваются различные приоритеты. Но в силу особенностей ОС в Windows NT внутри одного процесса доступно только 5 (7, если учитывать два экстремальных) уровней приоритета. При таком ограничении большая часть нитей будет работать на одном и том же уровне и, если вы должны управлять одновременно несколькими критическими событиями, предсказуемость системы будет невысокой. Конечно, может быть много процессов. Но даже и тогда общее количество приоритетов всего лишь 16. Более того, время переключения контекста между нитями из различных процессов значительно больше, чем между нитями внутри одного процесса.

Классической для СРВ проблемой является проблема инверсии приоритетов. Системные вызовы синхронизации, такие, как мьютексы, семафоры и критические секции, должны уметь управлять наследованием приоритетов. В Windows NT это невозможно.

Еще одна проблема, относящаяся к инверсии приоритетов в Windows NT, связана со способом реализации некоторых вызовов графического интерфейса пользователя. Эти вызовы управляются синхронным образом (вызывающая нить приостанавливается до завершения обработки вызова) процессом, работающим в классе динамических приоритетов (не реального времени), что создает все предпосылки для инверсии приоритетов (разделяемый ресурс здесь - системный вызов интерфейса пользователя). Как видно, из-за относительно небольшого количества возможных приоритетов и проблемы их инверсии Windows NT пригодна только для простых (небольшое количество различных видов событий) СРВ.

Любая СРВ взаимодействует с внешним миром через аппаратуру компьютера. Внешние события преобразуются в прерывания и обрабатываются драйвером устройства.

Как уже отмечалось ранее, доступ к аппаратуре имеют только драйверы. Поскольку приложения РВ часто работают со специфическими внешними устройствами, требующими и специфического управления, разработчик системы РВ под Windows NT должен уметь разрабатывать драйверы устройств.

Драйвер отвечает за обработку прерываний, генерируемых устройством. Напомним, что в Windows NT для увеличения реактивности системы был разработан оригинальный механизм, при котором прерывания обрабатываются в два этапа. Сначала прерывание обрабатывается в очень короткой процедуре ISR (Interrupt Servicing Routine) внутри драйвера. Эта процедура выполняет только критические действия, блокируя при этом другие прерывания. Остальная часть обработки прерывания откладывается и ставится в очередь при помощи вызова процедуры отложенных прерываний DPC (Deferred Procedure Call).

Таким образом, обработка прерываний в Windows NT достаточно сложна. Более того, процедура обслуживания прерывания ISR должна быть написана очень грамотно и должна выполняться очень короткое время. Поэтому большинство драйверов выполняют основную часть обработки прерывания в DPC (которая может вытесняться только любой ISR). Различные DPC работают на одном и том же уровне приоритета и не могут вытеснять друг друга. Таким образом, время реакции разработанного драйвера сильно зависит от других драйверов.

Как видно из вышеизложенного, Windows NT, созданная для работы главным образом с классическими приложениями, не может служить хорошей платформой для приложений РВ по следующим причинам:

- количество доступных приоритетов в классе РВ слишком мало;
- проблема инверсии приоритетов не решена;
- занимаемая память слишком велика для встраиваемых приложений, а лицензия слишком дорога;
- драйверы устройств могут тратить слишком много времени на уровне DPC и их невозможно вытеснить другим DPC .

Однако Windows NT может использоваться при создании систем мягкого РВ со следующими особенностями:

- системы мягкого РВ, которые иногда допускают пропуск временных сроков;
- простые системы, где количество событий различного рода не слишком велико (предсказуемость DPC в этом случае значительно выше);
- нагрузка на центральный процессор всегда остается низкой (у системных процессов есть время для исполнения);
- ответственные участки заданий (или даже всё задание целиком) выполняются на уровне DPC или, лучше, в самой ISR.

7.2. Системы SCADA

Системы SCADA (Supervisor Control and Data Acquisition - управление и обработка данных оператором) широко используются для разработки, моделирования и создания рабочих мест автоматизированных систем управления технологическим процессом (АСУ ТП).

В функции систем SCADA входит управление технологическим процессом с пульта оператора, поэтому они предлагают широкий набор элементов управления, отображаемых на экране: кнопки, рубильники, ползунковые и поворотные регуляторы.

Для отображения информации нужны экранные формы, показывающие параметры производства: стрелочные, ползунковые или цифровые индикаторы, а также сигнализирующие табло различной формы и содержания.

Контроль правильности решений, принимаемых оператором, а также накопление статистики по работе системы требуют возможности создания архивов событий и поведений переменных процессов во времени. Эти функции также реализуются системами SCADA.

Для обеспечения надежного проектирования СРВ SCADA-системы предусматривают средства документирования как самого алгоритма управления, так и технологического процесса. Кроме того, SCADA-системы обычно включают в себя многооконный графический интерфейс, позволяют импортировать элементы отображения, дают возможность создания собственных библиотек алгоритмов, динамических объектов, элементов мнемосхем.

Исполнение основных компонент СРВ происходит с использованием ядра РВ, обеспечивающего предсказуемое время отклика на внешнее событие.

Работа с внешними устройствами осуществляется через драйверы оборудования нижнего уровня АСУ ТП.

Системы SCADA, как правило, предусматривают возможность децентрализованного контроля и управления технологическим процессом. При этом используются сетевые функции вычислительных систем. Поэтому системы SCADA предоставляют средства защиты от несанкционированного доступа.

Известно много коммерческих систем SCADA, предлагаемых разными производителями Например, система Genie компании Advantech

На рис 7 4 представлен скриншот этой системы в работе

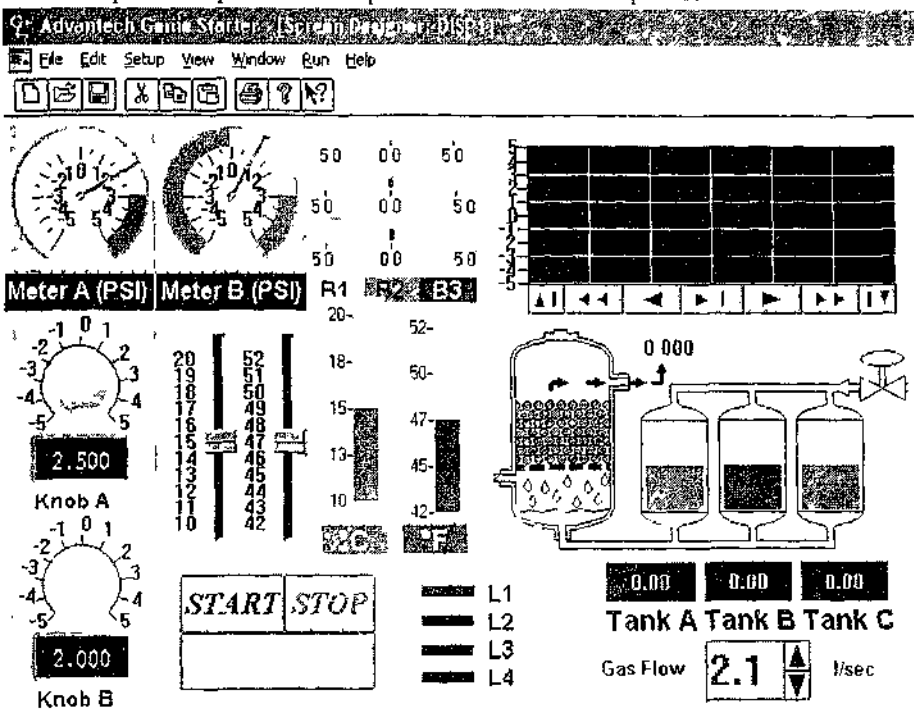


Рис 7 4 Скриншот работающей системы Advantech Genie

На рис 7 5 представлен проект системы соответствующей рис 7 4 Здесь набор инструментов для использования в проектах показан слева Справа отображаются основные блоки CPB и взаимосвязи между ними

SCADA-система Genie позволяет задавать алгоритмы обработки поступающих сигналов для любых блоков системы На рис 7 5 имеется два таких программных блока PRG1 и PRG2 Содержимое программного модуля PRG1 приведено на рис 7 6

Для каждого модуля в том числе программного задается список входов и выходов посредством которых они взаимодействуют с системой в целом

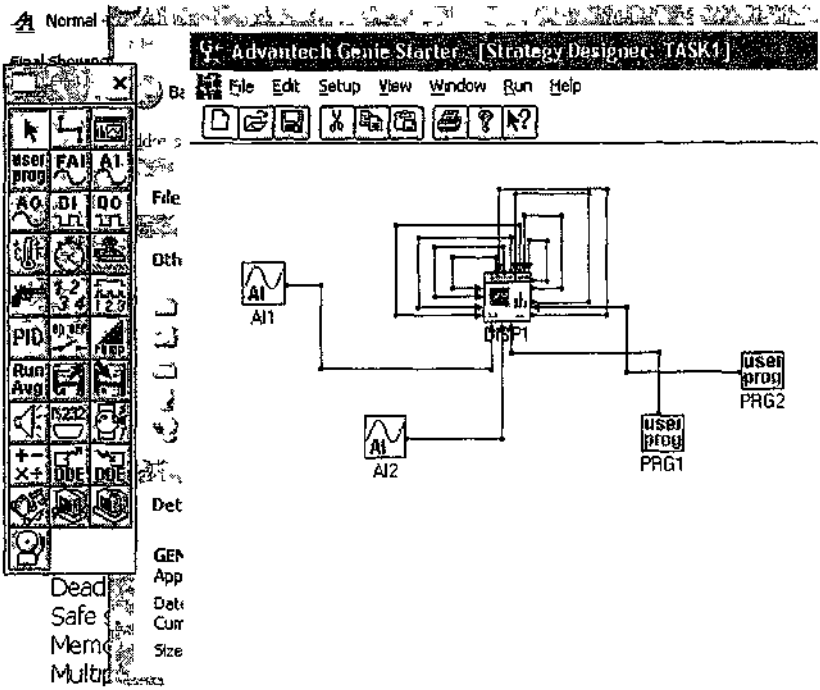


Рис 7 5 Пример проекта

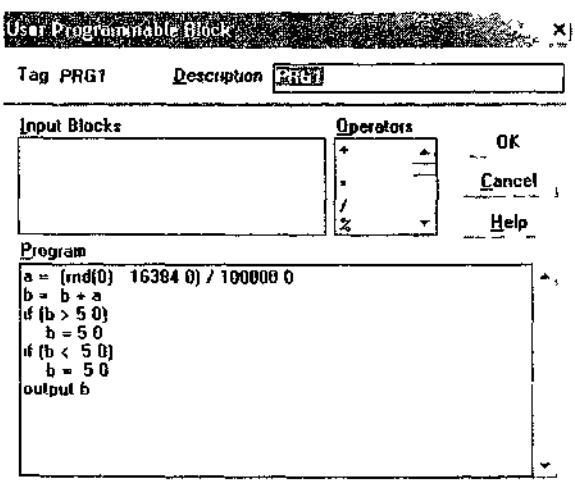


Рис 7 6 Содержимое программного модуля PRG1

На рис. 7.7 приведен пример окна настройки для модуля прямоугольного индикатора.

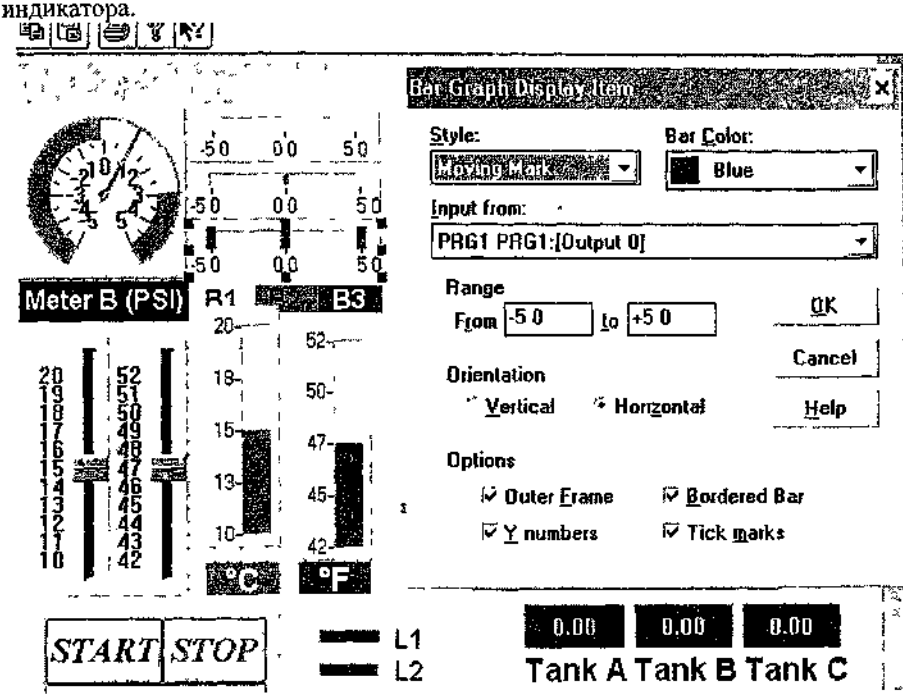


Рис 7.7 Окно настройки прямоугольного индикатора

Окно настраивает прямоугольный индикатор на отображение значения выходов модуля PRG1.

Таким образом, приведенные примеры показывают, что Genie позволяет в наглядном виде задавать структуру системы, описывать, как блоки системы получают данные от объектов управления и других блоков, как они эти данные обрабатывают и отображают на экране операторского терминала

Кроме продуктов компании Advantech хорошо известны системы SCADA Genesis32 фирмы Iconics, российская система SCADA TraceMode компании AdAstra. Скриншот системы TraceMode приведен на рис 7 8

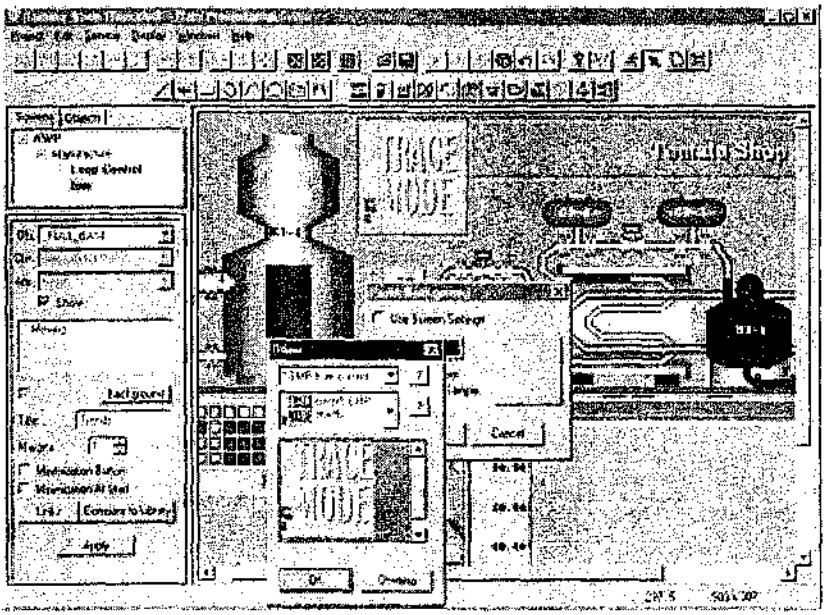


Рис 7.8. Проект в SCADA TraceMode

Контрольные вопросы и задания

1. Каковы отличительные особенности ОС РВ?
2. На сколько частей делится обработка прерывания в ОС РВ?
3. Каковы преимущества объектно-ориентированных ОС РВ?
4. Какие ОС РВ вы знаете? Охарактеризуйте их.
5. Можно ли использовать ОС Linux в задачах РВ? Если можно, то ка ким образом?
6. Подходят ли ОС семейства Windows NT для задач РВ? С какими огра ничениями?
7. Что такое системы SCADA? Каковы их основные характеристики?

Глава 8. Примеры систем реального времени

В этой главе рассматриваются примеры СРВ различного назначения.

8.1. Дефектоскопия в металлургии

Ультразвуковая дефектоскопия [39] широко используется при определении качества продукции на металлургических предприятиях. Дефектоскоп позволяет определить наличие трещин и внутренних полостей в стенках труб. По количеству дефектов трубы надлежащим образом сортируются или отбраковываются.

Приблизительная схема установки приведена на рис. 8.1. Установка состоит из промышленного дефектоскопа, блока сопряжения дефектоскопа с компьютером, персонального компьютера, регистрирующего поступающую информацию.

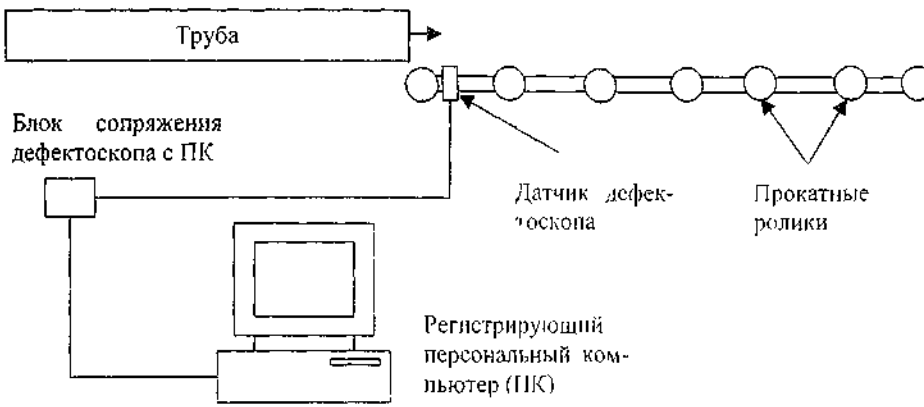


Рис. 8.1. Схема дефектоскопической установки

Труба движется по прокатному стану посредством валиков с постоянной скоростью. Датчик дефектоскопа производит проверку трубы и выдачу сигналов о наличии или отсутствии дефектов дискретным образом каждые 400 мкс. Блок сопряжения дефектоскопа с компьютером интерпретирует переданную дефектоскопом информацию, формирует байт специального формата, который поступает в компьютер через интерфейс RS232 (COM-порт) каждые 400 мкс. Этот байт содержит информацию о предыдущем такте работы дефектоскопа.

Задачей СРВ, расположенной на компьютере, является регистрация поступающих сигналов в реальном масштабе времени и отображение информации о дефектах на экране в виде модели трубы (рис. 8.2) и специальных графических символов, указывающих на наличие и тип дефектов (рис. 8.3).

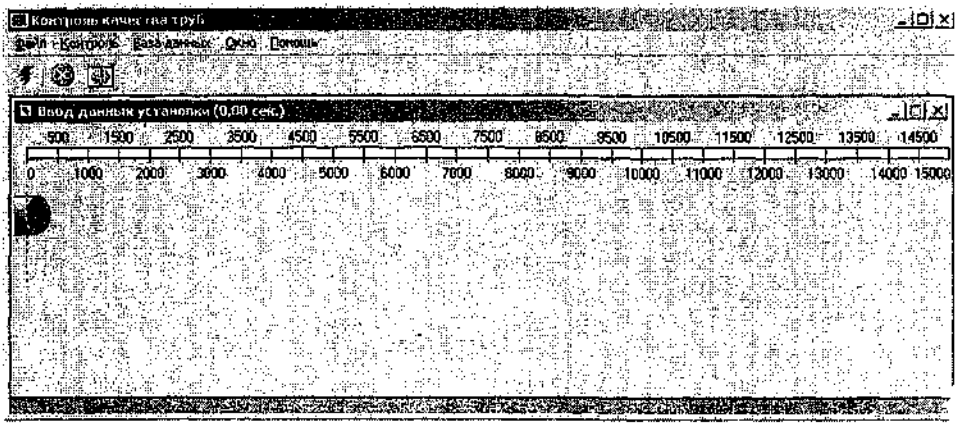


Рис. 8.2. Модель трубы с масштабной линейкой

На модели трубы (рис. 8.2) дефекты отмечаются схематично пятнами.

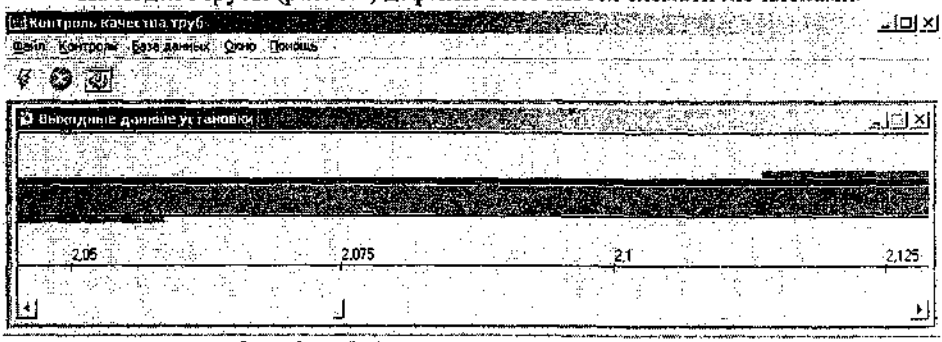


Рис. 8.3. Информация о наличии дефекта

На рис. 8.3 показана подробная информация о процессе дефектоскопии с точным указанием расстояния от начала трубы в метрах. Синим цветом отмечается синхроимпульс, красным - наличие дефекта по двум каналам дефектоскопа (сверху и снизу). Эта информация становится доступна после прогона всей трубы.

Данная система относится к мягким СРВ, поэтому в качестве операционной системы в ней используется Windows NT 4.0 Workstation.

В программном обеспечении работают два основных потока: поток вывода информации на экран и обработки сообщений Windows, а также поток приема данных от блока сопряжения через COM-порт и сохранения ее в специальном буфере ввода. Этот поток постоянно прослушивает порт в синхронном режиме (нет данных - поток ждет их появления) и после появления информации регистрирует ее, как было сказано выше.

После прогона всей трубы через дефектоскоп информация из буфера может быть подвергнута детальному анализу, как показано на рис. 8.3. Одновременно это делать нельзя, т.к. поток перерисовки будет забирать слишком много времени и могут появиться пропуски информации от дефектоскопа, что недопустимо.

Укрупненная информация о дефектах сохраняется в базе данных, и может потом быть статистически обработана и выведена в качестве отчетов, позволяющих делать заключение об общем качестве продукции, выпускаемой предприятием.

8.2. Подводная робототехника

Данный материал взят из [40].

Общепризнанно, что наиболее безопасным и эффективным путем исследования глубин Океана является использование технических средств, обеспечивающих косвенное присутствие человека под водой. Важную роль в этом играют автономные подводные роботы (АПР). С помощью АПР в настоящее время выполняются обзорно-поисковые и обследовательские работы на больших глубинах и в условиях сложного рельефа дна, подлёдные работы, прокладка оптических кабелей, обследование водозаполненных тоннелей и многие другие. Выполнение этих работ с помощью других средств крайне затруднительно или просто невозможно.

АПР представляет собой автоматический самоходный носитель исследовательской аппаратуры, способный погружаться в заданный район Океана на предельные глубины, двигаться по заданной траектории, выполнять требуемые работы и по окончанию программы возвращаться на обеспечивающее судно или береговую базу. В качестве исследовательской аппаратуры на аппарате обычно устанавливаются измерители параметров среды, фото-видеоаппаратура, обзорные гидролокаторы, геофизическая аппаратура (магнитометр, акустический профилограф, гравиметр).

Основным режимом работы АПР является движение на небольшом расстоянии от дна (от 1,5 до 10 м) с выполнением необходимых измерений. Система управления в совокупности с эхолокационной системой обеспечивает движение в достаточно сложном рельефе.

АПР работает под водой автономно, без связующего кабеля. Время непрерывной работы аппарата под водой зависит от проекта, типа его энергоисточника и может составлять от единиц до нескольких десятков часов и даже суток.

В настоящее время в АПР размещаются достаточно мощные вычислительные системы. В круг задач, решение которых потенциально возможно с помощью АПР, вошли исследовательские работы, инспекция протяженных объектов (подводных трубопроводов, кабелей), мониторинг Океана.

8.2.1. Особенности вычислительной системы автономного подводного робота

Традиционно АПР проектируются для работы на максимальных глубинах, что находит отражение в их механической конструкции. Бортовая электроника размещена в прочных контейнерах диаметром около полутора десятков сантиметров, электрически связанных между собой. Это обстоятельство серьезно ограничивает размеры конструктивов для размещения бортовых процессоров и контроллеров. В зависимости от назначения и размеров АПР состав системы управления может меняться и содержать от одного до несколько процессоров, объединенных в локальную вычислительную сеть. Для передачи большого количества информации во время работы используется бортовая сеть Ethernet, а для связи с бортовыми специализированными контроллерами - последовательный канал обмена RS-485.

Локальная сеть АПР строится, как правило, из готовых процессорных модулей индустриального назначения. В качестве базовой в настоящее время выбрана архитектура x86 (в первую очередь из-за относительной дешевизны и распространенности процессоров), а в качестве изготовителей - фирмы WinSystems и Octagon. Выбор объясняется небольшими размерами конструктивов процессорных модулей данных фирм.

8.2.2. Используемая операционная система

Особенности системы управления АПР связаны с характером размещения и мощностью процессоров, условиями их функционирования, а также с решаемыми задачами. В качестве базовой системы была выбрана ОС РВ QNX. Перечислим ее основные достоинства.

1. Минимальные требования операционной системы к вычислительным ресурсам.

Одно из основных требований к ЛВС АПР - минимальный уровень используемых вычислительных ресурсов. Данное требование имеет под собой следующие основания:

1) необходимость рассеяния тепла в замкнутом объеме при отсутствии конвекции не позволяет использовать мощные процессоры;

2) чтобы увеличить максимальную продолжительность запуска АПР требуется минимизировать энергозатраты, в том числе за счет меньшего потребления системы управления (это требование особенно важно для аппаратов с малой емкостью энергоисточника);

3) косвенной экономии электроэнергии можно добиться за счет выбора меньших габаритов прочных контейнеров для размещения процессоров, что влечет за собой уменьшение затрат на движение, и, следовательно, увеличение автономности АПР;

4) маломощные микрокомпьютеры имеют существенно меньшую стоимость.

2. Поддержка распределенных вычислений.

В зависимости от назначения АПР система управления может выполняться на сети различной конфигурации. Требуется обеспечить простоту взаимодействия подзадач, выполняющихся на различных процессорах. Кроме того, при смене оборудования отдельные устройства могут переключаться с одного процессора на другой, что должно гибко отслеживаться системой управления. QNX, благодаря встроенному механизму IPC, обеспечивает "прозрачное" взаимодействие процессов, динамически размещаемых на различных узлах сети.

3. Поддержка механизма приоритетов и системных таймеров.

Время работы модулей системы управления характеризуется различными интервалами времени. Время работы алгоритмов управления нижнего уровня (например, управление движением) измеряется десятками миллисекунд, и каждый цикл управления должен быть чётко привязан к временным меткам. В то же время рабочий цикл более высоких уровней управления может занимать секунды и даже минуты. Эти уровни не требуют столь точной привязки ко времени и являются фоном для задач нижнего уровня. Кроме того, немаловажное значение имеет такой параметр, как время переключения контекста задач (желательно минимальное). QNX обеспечивает поддержку 32 приоритетов задач, а время переключения контекста минимально среди всех известных ОС РВ.

4. Простота реализации и широта использования.

Система управления должна допускать свою реализацию существующими средствами программирования на доступных в АПР вычислительных ресурсах. Поставляемая с QNX среда разработки обеспечивает реализацию любой управляющей СРВ.

8.2.3. Основные типы аппаратов

Ниже приведено несколько примеров АПР, предназначенных для разных задач. Отметим, что все вычислительные системы, использующиеся на них, относятся к классу систем жесткого РВ.

АПР МТ-98 (прототип)

Аппарат предназначен для работы на глубинах до 6000 м и ориентирован на использование в сложном рельефе (рис. 8.4). Аппарат имеет движительную установку, обеспечивающую произвольные режимы движения (в том числе, движение боком, зависание и т.п.). Для обеспечения движения вблизи препятствий используется многоканальная эхолокационная система, позволяющая различать препятствия впереди по ходу движения, а также двигаться в диапазоне высот от 0,8 до 50 м.

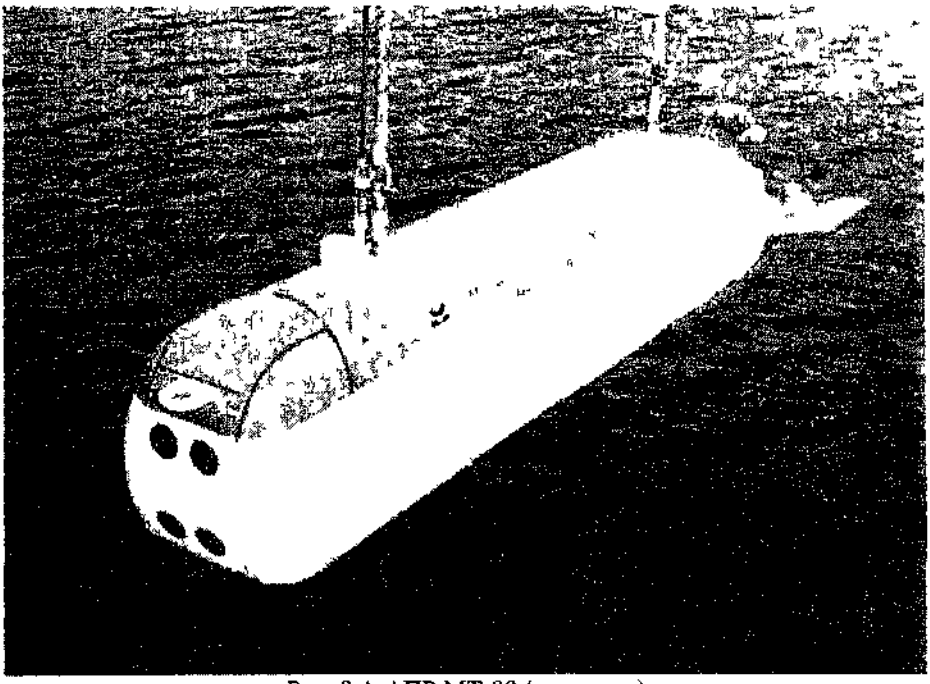


Рис 8 4 АПР МТ-98 (прототип)

Для слежения за работой АПР под водой используется гидроакустическая навигационная система Кроме того, эта же система позволяет осуществлять су первизорный контроль работы АПР (коррекцию выполняемой миссии) На по верхности для связи с аппаратом применяется радиомодем С его помощью имеет ся возможность получить по радиоканалу ключевые наборы данных для анализа загрузить новую миссию, управлять перемещением аппарата по поверхности воды Источником питания являются аккумуляторы, обеспечивающие около 20 часов эффективной работы у дна Вес аппарата составляет примерно 1000 кг и зависит от оснащения дополнительной аппаратурой

Система управления состоит из нескольких компьютеров Octagon 5066 вы полняющих различные функции и объединенных в локальную вычислительную сеть (ЛВС) посредством канала Ethernet Работа этой сети организуется с исполь зованием ОС РВ QNX К этой же сети подключается компьютер оператора, когда АПР находится на борту судна (рис 8 5) Компьютеры связаны между собой стан дартным FLEET-протоколом, что обеспечивает скорость и надежность передачи данных Распределенная структура системы управления предполагает следующее разделение функций между компьютерами ЛВС автопилот и системный контрол лер составляют основу системы управления и обеспечивают выполнение введен

ного задания, координацию работы бортовой аппаратуры и контрольно-аварийные функции; процессор обработки изображений служит для выделения объектов и прецизионной навигации.

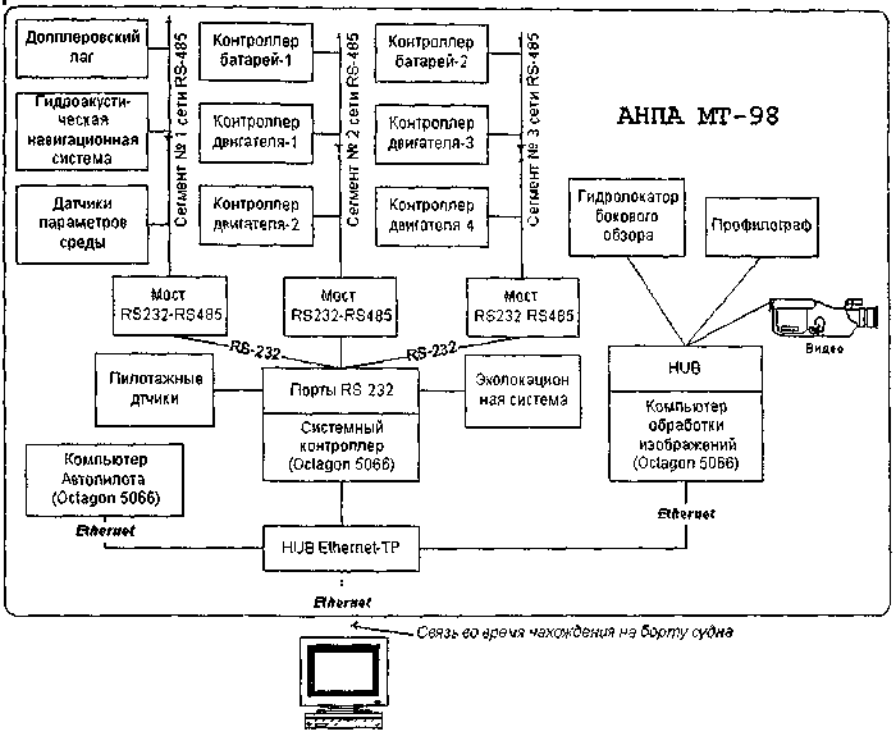


Рис. 8.5. Структура ЛВС АПР МТ-98

Характерной особенностью ЛВС МТ-98 является наличие наряду со скоростным каналом Ethernet-TP нескольких последовательных каналов обмена RS-485. На последовательных каналах размещается бортовое оборудование, оснащенное микроконтроллерами. Последние обеспечивают функции предварительной обработки данных и самодиагностики, поэтому информационный обмен с ними сведен к минимуму. Оборудование группируется вокруг каналов таким образом, чтобы выход из строя одного из них не приводил к потере работоспособности всего аппарата.

Солнечный АПР

Солнечный автономный необитаемый подводный аппарат (САНПА) предназначен для проведения длительных многосуточных океанографических измерений в океане (рис. 8.6). В процессе выполнения программы-задания аппарат пе-

риодически всплывает на поверхность для осуществления связи с оператором посредством спутниковой системы связи Oceanographic Data Link (ODL) и сети Интернет для передачи накопленных данных и возможной модификации программы-задания. В ночное время аппарат выполняет активную часть программы-задания, а в светлое время он всплывает на поверхность для зарядки аккумуляторов от расположенных на нем солнечных панелей (рис. 8.6). Находясь на поверхности, аппарат определяет свои координаты с помощью GPS. Суточный пробег аппарата в средних широтах составляет 20 - 40 км и зависит от солнечного излучения. Предельная рабочая глубина погружения САНПА равна 1000 метрам, и время непрерывной работы не ограничено.

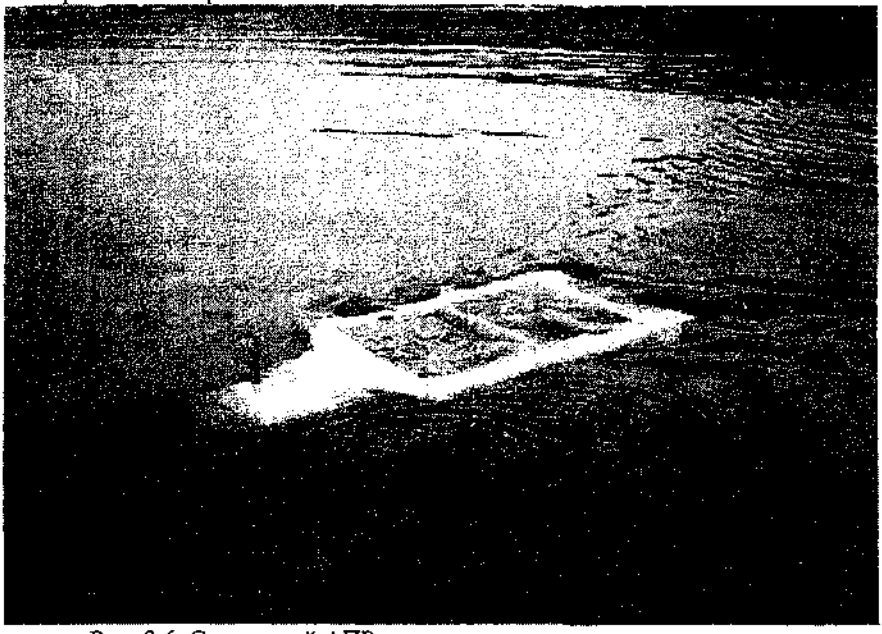


Рис. 8.6. Солнечный АПР во время подзарядки на поверхности

Система управления включает экономичный PC-совместимый промышленный компьютер Epson CARD 486-HB и набор микроконтроллеров, связанных между собой сетью на основе последовательного канала обмена RS-485.

Бортовой компьютер выполняет функции верхнего уровня системы управления. Он также обеспечивает расчет местоположения САНПА на основе GPS и датчиков бортовой системы позиционирования, выполняет сохранение собранной сенсорными системами аппарата информации, осуществляет информационный обмен с компьютером оператора и бортовыми микроконтроллерами. Компьютер функционирует на основе ОС РВ QNX.

Связь бортового компьютера с компьютером оператора осуществляется с помощью радиомодема или акустического модема При этом компьютер аппарата является удаленным узлом сети QNX Система автоматически выбирает наиболее скоростной канал из доступных в данный момент для связи с АПР

Набор микроконтроллеров реализует систему управления нижнего уровня Данная система включает систему зарядки аккумуляторов от солнечных панелей, систему энергообеспечения, блок управления маршевым движителем и двигателями рулей высоты

Автономно-привязной аппарат TSL

Подводный аппарат TSL (рис 8 7) проектировался для выполнения обзорно-поисковых работ на шельфе на глубине до 200 метров, визуального обследования подводной части причальных сооружений, донных конструкций или затонувших объектов, протяженных водозаполненных тоннелей или емкостей, а также для инспектирования подводных трубопроводов

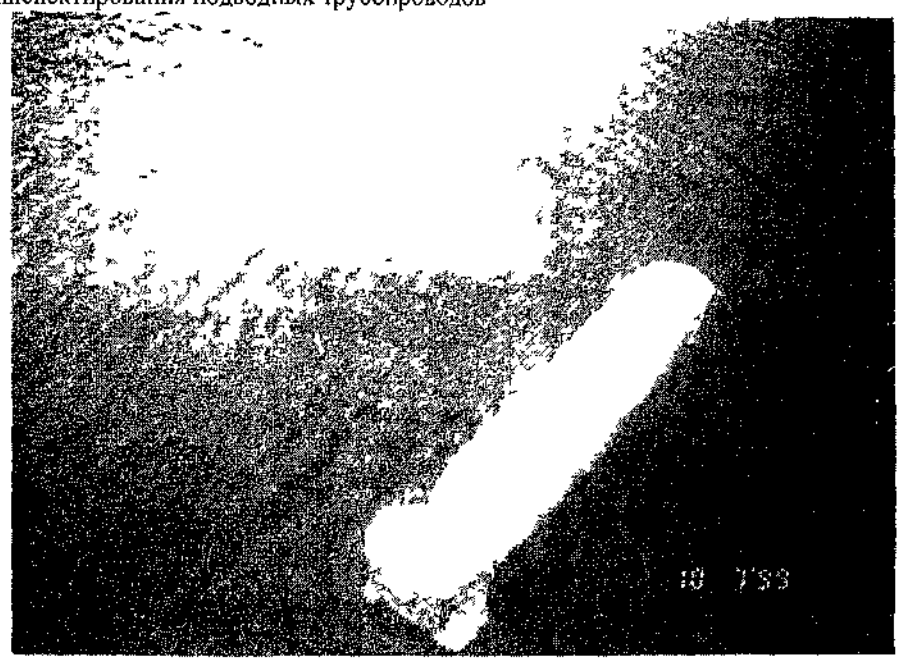


Рис 8 7 Автономно-привязной аппарат TSL перед погружением

Аппарат имеет на борту аккумуляторы и энергетически автономен Особенностью аппарата является его способность функционировать как в автономном так и в супервизорном режиме В последнем случае оператор имеет возможность управлять аппаратом и наблюдать телевизионное изображение от видеокамеры

располагающейся на аппарате, в реальном времени. Связь между компьютером оператора и вычислительной системой ПА TSL в этом случае осуществляется через оптоволоконный кабель (поддерживается канал обмена RS232). Кабель располагается на специальной катушке в корме аппарата и по мере движения свободно выматывается (максимальная длина кабеля до 5 км.).

Управление аппаратом осуществляется командами высокого уровня, поступающими с пульта оператора. Следует отметить, что набор этих команд стандартизован и используется также в системах управления автономных аппаратов (САНПА и МТ-98). В автономном режиме работы TSL волоконно-оптическая связь отсутствует, а источником команд является загруженная в автопилот программа-задание.

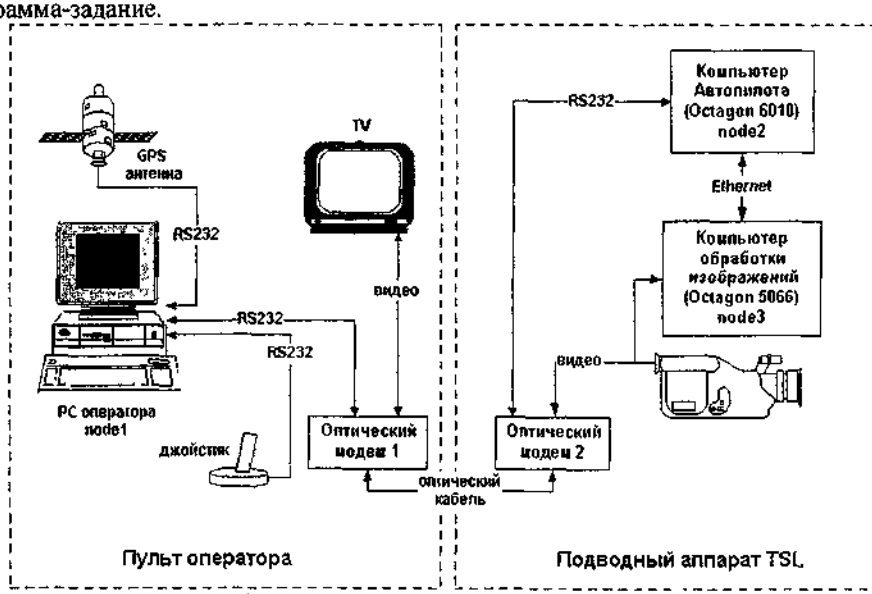


Рис. 8.8. Структура подводного аппарата TSL.

Система управления TSL является распределенной и включает бортовую систему управления на основе автопилота и пульт управления оператора. Компьютер оператора и бортовые компьютеры объединены в сеть на базе операционной системы QNX (рис. 8.8). Обмен данными между процессами производится с использованием встроенного механизма обмена сообщениями QNX.

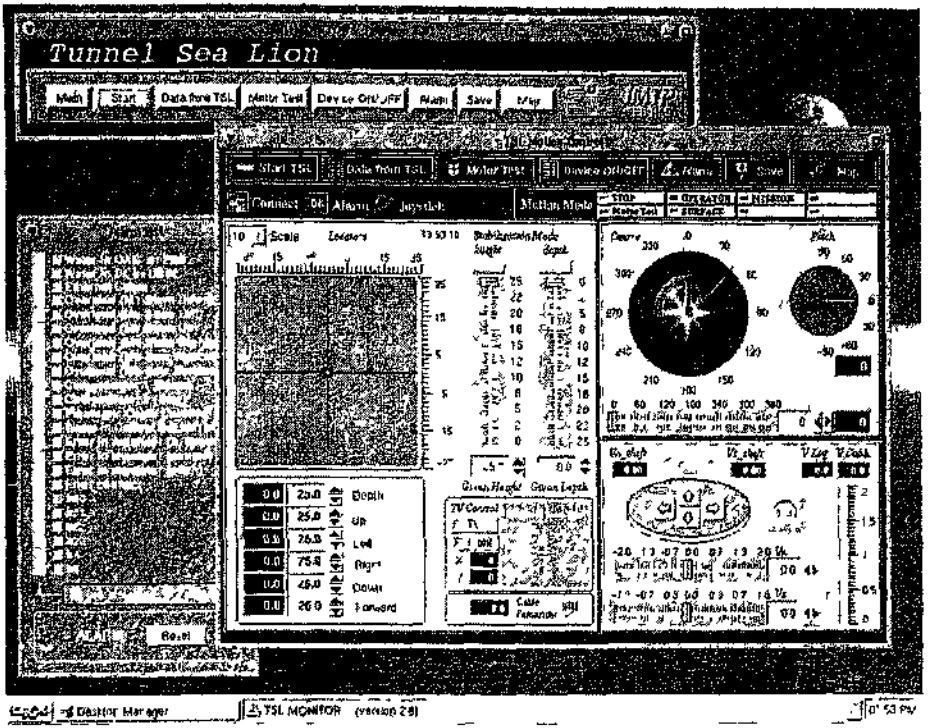


Рис 8 9 Интерфейс оператора подводного аппарата TSL

Ввод управляющей информации осуществляется оператором посредством разработанного для этих целей интерфейса пользователя с помощью клавиатуры, мыши или джойстика В основном окне данного интерфейса выводятся данные измерений пилотажных датчиков, осуществляется ввод целевых значений параметров движения подводного аппарата, а также производится переключение режимов работы бортового оборудования Результаты измерений отображаются в цифровом виде, а также в виде шкал и стрелочных указателей Программа работает в графической оболочке Photon 1 14 и была разработана с помощью Photon Application Builder 1 14 На рис 8 9 в качестве примера показано несколько окон интерфейса пользователя подводного аппарата TSL

8.3. Система управления дорожным движением "Старт"

За последние годы в Москве резко возросла интенсивность транспортного движения С этим сталкиваются практически все крупные города мира, и без применения компьютерных средств управления транспортными потоками эту проблему решить практически невозможно Подобная система должна

- уметь оценивать интенсивность потоков транспорта в различных районах города;
- управлять длительностью сигналов светофоров для создания "зеленых волн", с помощью которых поток автомобилей, не останавливаясь лишний раз на перекрестках, двигается в нужных направлениях;
- оценив интенсивность потока транспорта, управлять его скоростью и, возможно, направлением с помощью "интеллектуальных" знаков - указателей или прямых директив водителям на световых табло;
- управлять движением потоков транспорта с учетом времени суток;
- уметь оценивать экологическую обстановку в районе управления и предпринимать определенные усилия по ее улучшению;
- уметь понимать команды человека-оператора и передавать их на дорожные контроллеры перекрестков;
- уметь принимать команды постовых-милиционеров с улиц для оперативного управления движением транспорта с уличного перекрестка;
- круглосуточно и без перерывов работать с беспрецедентной надежностью.

В Москве ряд районов оборудован подобной системой, представляющей собой сложный иерархический комплекс управления движением транспорта. Эта система получила название "СТАРТ" [41].

На нижнем уровне системы расположены средства управления светофорной сигнализацией - дорожные контроллеры и детекторы транспорта - сенсоры, собирающие информацию о параметрах транспортных потоков. Дорожные контроллеры и детекторы транспорта расположены в зонах перекрестков и функционируют непосредственно в уличных условиях. Каждый дорожный контроллер может работать автономно (локальный режим) по заданной программе или в режиме системы, транслируя команды центра управления. передавая их на светофорную сигнализацию и управляемые дорожные знаки.

В центре управления, представляющем верхний уровень управления, установлена сеть станций, работающих под управлением операционной системы Solaris. Для хранения данных в системе использована СУБД Informix. Верхний уровень предусматривает кроме автоматического, еще и оперативное диспетчерское управление, при котором персонал получает информацию от управляющего вычислительного комплекса; речевую информацию через средства связи; телеметрию от подсистемы телевизионного надзора за движением, выводимую на полиэкран видеоконтрольных устройств; данные от коллективных средств отображения информации на базе "видеостены".

Информация с дорожных контроллеров и детекторов транспорта попадает в коммутатор, который представляется районной подсистемой "СТАРТ-КВИН". Подсистема связана с перекрестками проводными линиями связи, а с центром управления по оптоволоконной сети с использованием протокола TCP/IP.

Подсистема "СТАРТ-КВИН" построена на базе промышленного VMEbus-контроллера. Ее ядро - интеллектуальный встраиваемый компьютер MVME-172 компании Motorola Computer. Рабочее место оператора реализовано на базе стандартного персонального компьютера, установленного в центре телеавтоматического управления ГИБДД УВД Москвы.

Из центра управления в подсистему "СТАРТ-КВИН" загружается локальная база данных и необходимые программные модули, сохраняемые в памяти MVME172. Во время работы в центр управления возвращается информация для анализа и отображения, а также принимаются команды операторов.

В качестве устройств ввода/вывода для связи с дорожными контроллерами используются мезонинные контроллеры IndustryPack от компании SBS Technologies Modular I/0, а для цифрового ввода/вывода применяется IP-Digital48 - 48-иканальный недорогой контроллер с прекрасными параметрами надежности (наработка на отказ - более 1 млн. часов). Каждый дорожный контроллер подсоединен к подсистеме "СТАРТ-КВИН" линиями связи, по которым производится ввод телеметрической информации и вывод информации телеуправления. Таким образом, один мезонинный контроллер IP-Digital48 управляет шестнадцатью дорожными контроллерами, что соответствует 16-ти перекресткам дорожной системы Москвы.

В процессе работы "СТАРТ-КВИН" ежесекундно осуществляет следующие операции:

- опрашивает все подключенные к нему дорожные контроллеры;
- на основании полученной информации, используя локальную базу данных и текущие алгоритмы координации, вырабатывает управляющие воздействия на светофоры и прочее периферийное оборудование перекрестков;
- производит вывод управляющей информации на дорожные контроллеры;
- производит обмен информацией с управляющим вычислительным комплексом системы "СТАРТ";
- выводит информацию на локальный пульт оператора и обрабатывает введенные от него команды.

Управляет работой комплекса ОС РВ OS-9, которая предоставляет стандартные средства для:

- организации работы прикладных процессов в соответствии с заданными временными интервалами при обменах с периферийным оборудованием;
- организации информационного взаимодействия между процессами системы;
- организации обменов с пультом оператора комплекса путем поддержки DDE-протокола обмена;

- управления дисковыми накопителями (UNIX-подобная файловая система);
- организации и управления обменами с контроллерами ввода/вывода.

Пульт оператора комплекса "СТАРТ" представляет собой персональный компьютер, работающий под управлением Windows NT и соединенный каналом связи с VME-контроллером. На экране пульта в реальном масштабе времени отображается оперативная обстановка на перекрестках. Пульт предназначен также для управления за движением транспортных потоков в обслуживаемом районе. SCADA система InTouch, используемая для сбора, обработки и отображения информации, позволяет оператору-технологу выполнить графическую привязку подсистемы "СТАРТ-КВИН" к реальному району управления, а также скорректировать схемы перекрестков в случае изменения состава их оборудования, порядка размещения дорожных контроллеров и т.п.

Система "СТАРТ" относится к классу мягких СРВ.

Контрольные вопросы и задания

1. Какая операционная система используется в СРВ для дефектоскопии? Почему?
2. Почему ОС QNX хорошо подходит для вычислительных систем АПР?
3. Почему системы управления АПР относятся к СРВ жесткого типа?
4. С чем связана необходимость автоматического управления дорожным движением?
5. Опишите архитектуру системы "СТАРТ".

Заключение

В настоящем учебном пособии приведена лишь основная информация, связанная с внутренним устройством, функционированием, применением и проектированием СРВ. Безусловно, мир СРВ гораздо шире и многообразнее. Такие системы находят приложения в большом количестве повседневных задач, причем, некоторые из них уже немыслимы без компьютерного управления в реальном масштабе времени.

Авторы надеются, что полученной информации будет достаточно, чтобы самостоятельно разобраться в вопросах реализации и построения той или иной СРВ, а также иметь общее представление о подобных вычислительных системах. Такие знания являются обязательными для современного специалиста в области программного обеспечения и информационных технологий.

Приложение. Задания для самостоятельной работы

Ниже приведены задания для самостоятельной работы, которые могут быть использованы как лабораторный практикум по курсу "Системы реального времени".

1. Лабораторная работа №1. Организация параллельных вычислительных процессов с помощью системного таймера

Формулировка задачи

В условиях задачи синхронизации "Обедающие философы" (формулировку задачи см. в п. 3.7) написать с использованием средств операционной системы MS-DOS программу, обеспечивающую параллельное существование процессов-философов и ВЫТЕСНЯЮЩУЮ (см. п. 3.2.3) многозадачность с помощью подпрограммы обработки прерывания (см. п. 3.3) от аппаратного таймера. Должно осуществляться квантование времени процессора между процессами-философами. Вопросы синхронизации процессов в рамках данной работы не рассматривать.

Детали реализации

Использование аппаратного прерывания от таймера (прерывание 08h) позволяет реализовать механизмы вытесняющей многозадачности и квазипараллельного исполнения подпрограмм-философов.

Подпрограмма-обработчик прерывания в данном случае выступает в роли диспетчера задач, который вызывается приблизительно 18 раз в секунду (именно столько раз происходит прерывание таймера). Каждый вызов обработчика заканчивается переключением с текущего процесса-философа на следующего. Причем для текущего процесса необходимо сохранение в служебной структуре, называемой блоком управления процессом, его персональной информации - контекста (например, это может быть состояние регистров процессора) и адреса команды в его теле, на которой произошло прерывание. Для следующего процесса, наоборот, происходит восстановление его состояния и передача управления на ту команду, на которой он был прерван ранее. Таким образом, процессы-философы не знают, что их несколько и развиваются так, как будто каждый из них является единственным.

При старте программы необходимо соответствующим образом заполнить начальную информацию в блоках управления процессами (контексты каждого процесса и т.п.). Затем необходимо установить новый вектор прерывания, соответ-

ствующий адресу процедуры-диспетчера. После этого можно запускать параллельное исполнение процессов-философов.

Отметим, что для языков высокого уровня процедура сохранения контекста может быть изменена. Например, для продукции фирмы Borland (Borland Pascal, Borland C++) существует специальное ключевое слово interrupt, которое указывается при описании подпрограммы обработчика прерывания. Такое ключевое слово заставляет компилятор при входе в подпрограмму сохранять почти все регистры процессора (напомним, что CS, IP и AF - флаги - сохраняет сам процессор) в стеке. Соответственно, при выходе из подпрограммы перед вызовом команды iret компилятор восстанавливает все сохраненные в стеке регистры. Следовательно, достаточно выделить каждому процессу свою область памяти под стек и записать туда начальные значения, а затем только лишь изменять адрес стека (пару регистров SS:SP), что приведет к автоматическому сохранению старого контекста и восстановлению нового.

Подобную схему сохранения контекста в стеке можно реализовать и на языке ассемблера.

2. Лабораторная работа №2. Семафорная синхронизация параллельных процессов

Формулировка задачи

Опираясь на программу, разработанную в рамках лабораторной работы №1, разработать программу, реализующую семафорное решение задачи "Обедающие философы" (алгоритм см. в п. 3.7). Механизм семафоров необходимо запрограммировать самостоятельно.

Детали реализации

Необходимо реализовать механизм семафоров и использовать его в алгоритме, описанном в п. 3.7. По алгоритму требуются N+1 двоичных семафоров, где N - количество философов. Отметим, что алгоритм полностью исключает активное ожидание в любом его проявлении, т.е. если процессу не может быть предоставлен ресурс (семафор), он приостанавливается до тех пор, пока этот ресурс не освободится. Программа должна это учитывать и не передавать управление приостановленному процессу.

3. Лабораторная работа №3. Метод Хабермана обхода тупиков.

Формулировка задачи

Реализовать программно метод Хабермана обхода тупиков (см. п. 3.10.3).

Детали реализации

На входе программы должна быть информация о времени работы процессов и количестве требуемых на каждом отрезке времени ресурсов по аналогии с примерами, разобранными в п. 3.10.3. Также задается количество ресурсов в системе, векторы b и c (см. описание метода Хабермана). При условии реализуемости начального состояния выходом должна быть безопасная последовательность, а также таблица занятия ресурсов процессами во времени, т.е. расписание исполнения процессов.

4. Лабораторная работа №4. Средства синхронизации и коммуникации процессов в операционной системе при обеспечении работы в реальном времени

Формулировка задачи

Реализовать программную модель объекта управления и СРВ, управляющую этим объектом с учетом внешних возмущений.

Детали реализации

Программа состоит из нескольких основных модулей: модуль имитации объекта управления, модуль управления объектом, пользовательский модуль, модуль внешнего воздействия.

Каждый из модулей является законченной независимой конструкцией, которая имеет свои входы и выходы, алгоритм преобразования входной информации в выходную и работает параллельно другим модулям. Кроме того, каждый модуль должен иметь средства привязки внутреннего времени системы к времени реальному.

Приблизительная схема системы приведена на рис. П1.1.

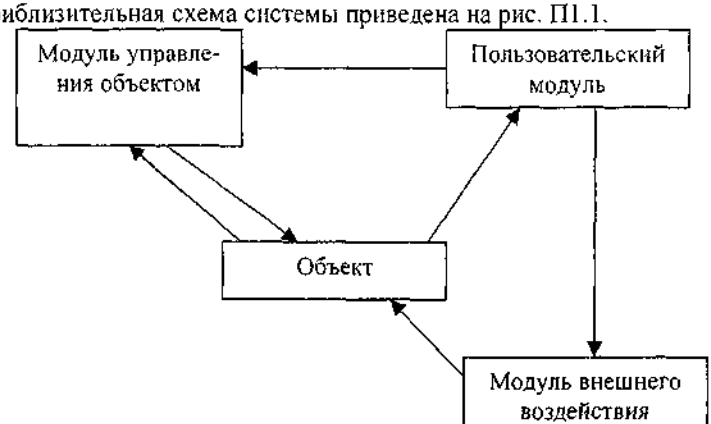


Рис. П1.1. Схема модели СРВ

Модуль имитации объекта реализует физическую модель объекта, рассчитывая его координаты, угол поворота и т.п. с использованием соответствующего математического аппарата (например, дифференциальных уравнений).

Модуль управления объектом получает информацию о текущем состоянии объекта и выдает соответствующее управляющее воздействие, которое затем подставляется в уравнения объекта.

Пользовательский модуль отвечает за отображение информации о состоянии объекта и, реагируя на команды пользователя, позволяет контролировать модули управления объектом и внешнего воздействия. Например, управление можно отключить или включить, а для внешнего воздействия можно задать его интенсивность или какие-то другие параметры.

Информация от модуля объекта поступает в пользовательский модуль, который отображает текущее состояние объекта на экране.

Как уже отмечалось ранее, каждый модуль функционирует параллельно с другими и, по сути, зависит только от получаемых им входных данных от других модулей или от пользователя. Иными словами, должны быть созданы специальные параллельные подпрограммы (программы), реализующие функциональность каждого модуля. Также должна быть предусмотрена возможность обмена информацией между ними (например, по схеме на рис. П1.1).

Работа выполняется с использованием средств распараллеливания и синхронизации операционной системы Windows. В качестве объекта управления рассматривается математический маятник, в котором нить заменена невесомой жесткой сцепкой (рис. П1.2), с уравнением

$$\frac{d^2\varphi}{dt^2} - \frac{1}{T_0^2}\sin\varphi = U,$$

где φ - угол отклонения маятника от вертикали;

$T_0 = \sqrt{\frac{l}{g}}$, причем $T_0\sqrt{2}$ - время падения маятника из верхней точки (l - длина маятника, g - ускорение свободного падения около поверхности Земли);

U - управляющее воздействие (включает корректировку от модуля управления и внешнее воздействие).

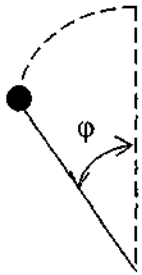


Рис. П1.2. Схема движения маятника

Если U=0 и φ=0, то маятник находится в состоянии покоя в верхнем положении. Таким образом, достаточно небольшого внешнего воздействия, чтобы вывести его из этого состояния. Задача модуля управления - удерживать маятник в верхнем состоянии вне зависимости от внешних воздействий.

Заметим, что по желанию возможна реализация своего объекта управления, отличного от рассмотренного выше.

5. Лабораторная работа №5. Изучение средств синхронизации и межпроцессного взаимодействия языка Ада

Формулировка задачи

Реализовать с использованием средств языка Ада одну из предложенных ниже задач.

Варианты задач

Поиск в файле

Реализовать параллельный поиск в файле, в том числе и по маске, например: "s??fgt" или "s*fgt". Поиск распараллеливается путём передачи разным задачам различных фрагментов файла с перекрытием. Перекрытие необходимо для того, чтобы успешно искать строки, попавшие на стык двух фрагментов.

Сортировка слиянием

Алгоритмы слияния основываются на процедуре слияния двух уже отсортированных отрезков массива в один. При слиянии двух отрезков происходит формирование нового отсортированного массива следующим образом: просматриваются все элементы одного отрезка и другого с первого и до последнего; если очередной элемент первого отрезка меньше элемента второго, то он попадает в массив-результат и указатель элемента первого отрезка увеличивается, в противном случае аналогичные действия производятся со вторым отрезком массива.

При параллельной сортировке весь процесс разбивается на итерации. На первой итерации массив разбивается на фрагменты, общее количество которых пре-

вышает количество Ада-задач в два раза. Все фрагменты сортируются любым алгоритмом сортировки. Затем каждая пара фрагментов сливается соответствующей задачей, после чего результат передается основной задаче для повторного распределения. Все следующие итерации идентичны первой за исключением уменьшения работающих Ада-задач вдвое на каждой из итераций. Последняя пара фрагментов может быть слита одной задачей.

Шифрование методом замены (моноалфавитная подстановка)

В этом методе символы шифруемого текста заменяются символами, взятыми из одного (одноалфавитная или моноалфавитная подстановка) алфавита. Самой простой разновидностью является прямая замена, когда буквы шифруемого сообщения заменяются другими буквами того же самого или некоторого другого алфавита. Таблица замены может иметь следующий вид:

Символы шифруемого текста	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Заменяющие символы	S P X L R Z I M A Y E D W T B G V N J O C F H Q U K

Произведение матриц

Произведением AB матрицы A(mxn)=(a_{ij}) на матрицу B(nxk)=(b_{ij}) является матрица C(mxk)=(c_{ij}), где

$$c_{ij} = \sum_{d}^{n} a_{id} \cdot b_{dj} ,$$

где i=1,2,…,m, j=1,2,…,k.

Для распараллеливания вычислений между L задачами за каждым из них закрепляются m*k/L элементов результирующей матрицы. Например, при L=m за i-м клиентом закрепляется i-тая строка матрицы C. i=1..m.

Предметный указатель

А

Б

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

P

С

Т

У

Ф

Ц

Ч

Я

Библиографический список

1. Liu J.W.S., Real-Time Systems, Prentice-Hall, Inc., Upper Saddle River, 2000, ISBN 0-13-099651-3, 610 p., p. 2.
2. Гэри М, Джонсон Д. Вычислительные машины и труднорешаемые задачи. - М.: Мир, 1982. - 416 с.
3. Graham, R. Bounds for multiprocessing timing anomalies. SIAM Journal on Applied Mathematics 17 (1969), pp. 263-269.
4. MIL-STD-2167A. - www2.umassd.edu/SWPI/DOD/MIL-STD-2167A/DOD2167A.html.
5. Laplante P.A., Real-time systems design and analysis. An engineer's handbook. 2nd ed., IEEE Press, NY, 1997, ISBN 0-7803-3400-0, 361 p.
6. How to draw Jackson system development (JSD) diagrams. - http://www.smartdraw.com/resources/centers/software/jsd.htm
7. Jackson M. Software requirements&Specifications, Addison-Wesley, 1995, ISBN 0201877120.
8. How to draw data flow diagrams. - http://www.smartdraw.com/resources/centers/software/dfd.htm.
9. Warnier-Orr notation. - http://www.courses.vcu.edu/INFO465-gs/notation.htm.
10. How to draw UML diagrams. - http://www.smartdraw.com/resources/centers/uml.
11. Fawler M., Scott K. UML Distilled Second Edition. A brief guide to the Standard Object Modeling Language. Addison-Wesley, Massachusetts, 2000. - 185 p.
12. Stevens P., Pooley R. Using UML. Software Engineering with Objects and Components. - Addison-Wesley, Harlow, 2000. - 256 p.
13. Larman C. Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design. - Prentice Hall PTR, Upper Saddle River,1998. - 507 p.
14. Рамбо Дж., Якобсон А., Буч Г. UML: специальный справочник. - СПб.: Питер, 2002. - 656 с.
15. UML resource center - http://www.rational.com/uml/resources/documentation/index.jsp.
16. How to draw ROOM diagrams . - http://www.smartdraw.com/resources/centers/software/room.htm.
17. Selic B. Requirements specification using executable models. - http://www.cis.upenn.edu/~lee/99cis642/papers/require.pdf.
18. Wyke E. Investigation of models for real-time systems: AIDA through UML and ROOM. - http://www.snart.org/docs/exjobb2000/Erik_Wyke_2000.pdf.
19. Selic B. An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems. - http://www.cis.upenn.edu/~lee/99cis642/papers/efficient.pdf.
20. Миренков Н.Н. О синхронизации процессов. - Новосибирск: Ин-т математики СО АН СССР, 1978, препринт ОВС-04. - 16 с.

21 Eisenberg M A , McGuire M R Further comments on Dijkstra's concurrent programming control problem Comm ACM, 1972, 15 (11)
22 Bacon J , Harris T Operating systems concurrent and distributed software design, 2003, Addison-Wesley, Harlow, 877 p ISBN 0-321-11789-1
23 Lamport L A new solution of Dijkstra's concurrent programming problem, Comm ACM, 1974, 17(8)
24 Burns A , Wellings A Real-Time Systems and Programming Languages - Addison-Wesley, 2001, 738 p ,ISBN 0 201 72988 1
25 Ada Reference Manual ANSI-RM-xx v23, 83-02-11
26 Master outline of Lovelace lessons - http //www dwheeler com/lovelace/master htm
27 Ada Reference Manual - http //www adaic org/standards/95lrm/html/RM-TOC html
28 Priority ceiling locking - http //www grammatech com/rm95html-1 0/rm9x-D-03 html#6
29 Гавва А Адское программирование - http //www ada-ru org/V-0 4w/index html
30 Mazidi A M , Mazidi J G The 80x86 IBM PC and Compatible Computers (Volumes 1&2) Assembly Language, Design, and Interfacing 3rd Edition, Prentice Hall, Upper Saddle River, 1995, 984 p ISBN 0-13-016568-9
31 Baker A , Lozano J The Windows 2000 Device Driver Book A Guide for Programmers 2nd Edition, 2001 Prentice Hall PTR Upper Saddle River New Jersey, 07458, ISBN 0-13-020431-5, 446 p
32 Сорокина С , Тихонов А Щербаков А Программирование драйверов и систем безопасности СПб, BHV-Петербург Москва, Издатель Молгачева С В , 2002, 256 р , ISBN 5-94157-263-8
33 Вахалия Ю UNIX изнутри - СПб Питер 2003 - 844 с
34 Ядро ОС Linux - http / www citforum ru operating_systems linux_khg index shtml
35 ОС QNX - реальное время реальная философия - http //www rts-ukraine com/QNXArticles/RealTimeRealPhilosophy htm
36 Shedding light on the Mars rover malfunction - http //www embedded com/showArticle jhtml?articleID=17800188
37 RT-Linux - http //info kiev kareha ru Linux/rt/rtlinux shtml
38 Рипол И Линукс реального времени KU (KURT) - http //www tldp org/linuxfocus Russian July 1998/article56 html
39 Разработка, изготовление и установка системы компьютерного управления дефектоскопом труб - Х д НИР №12431 с ОАО "Таганрогский радиотехнический университет, 2000
40 Ваулин Ю В , Инзарцев А В Применение ОС QNX в подводной робототехнике - http //www swd ru/qnx/support/literature/vote/006 html
41 Старт-КВИН - подсистема управления дорожным движением - http //www rtsoft ru/products/Start-KVIN

Чефранов Александр Георгиевич
Троценко Роман Владимирович

Учебное пособие

Проектирование систем реального времени

Ответственный за выпуск Троценко Р.В.
Редактор Надточий З.И.
Корректор Чиканенко Л.В.

Лр № 020565 от 23.06.1997 г . Подписано к печати
Формат 60x84 1/16. Бумага офсетная.
Офсетная печать.
Усл. п.л. - 14,2. Уч.-изд. л. - 13,8.
Заказ №208 Тираж 150 экз.

"С"

Издательство Таганрогского технологического института ЮФУ
ГСП 17А, Таганрог, 28, Некрасовский, 44
Типография Таганрогского технологического института ЮФУ
ГСП 17А, Таганрог, 28, Энгельса, 1