Издательство Знание

МАТЕМАТИКА КИБЕРНЕТИКА

Новое в жизни, науке, технике

Подписная научно-популярная серия

Издается ежемесячно с 1967 г.

Б. Г. Сушков

ЭВМ управляет экспериментом

1987/9

Новое
в жизни,
науке,
технике

Подписная
научно-
популярная
серия

9/1987

Издаётся
ежемесячно
с 1967 г.

МАТЕМАТИКА
КИБЕРНЕТИКА

Б. Г. Сушков

ЭВМ УПРАВЛЯЕТ ЭКСПЕРИМЕНТОМ

(Вычислительные системы реального времени)

ОГЛАВЛЕНИЕ

Издательство
«Знание»
Москва,
1987

ББК 32.97
С 89

СУШКОВ Борис Григорьевич – кандидат физико-математических наук, заведующий сектором Вычислительного центра АН СССР, лауреат премии Совета Министров СССР. Основные научные интересы – математическое моделирование управляющих систем, системное программирование, вычислительные системы реального времени.

Рецензент: Натан А. А., доктор технических наук, профессор.

Сушков Б. Г.
С89. ЭВМ управляет экспериментом (Вычислительные системы реального времени).— М.: Знание, 1987. 32 с.— (Новое в жизни, науке, технике.
Сер. «Математика, кибернетика»; № 9).

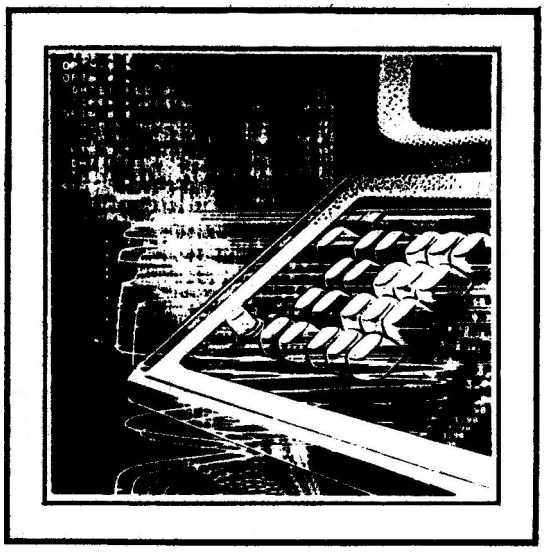
11 к.

Управление сложными производственными экспериментами, контроль за быстропротекающими явлениями невозможны без применения вычислительных систем реального времени.

Брошюра рассказывает о задачах. методах проектирования и реализации таких систем, особенностях языков программирования высокого уровня для составления программ реального времени. Изложение поясняется на основе простых примеров.

Рассчитана на лекторов, преподавателей и слушателей народных университетов.

2404000000 ББК 32 97



ВВЕДЕНИЕ

"Эксперимент всегда служил основой научных достижений человечества, особенно в области естественных наук. Шло время, усложнялись эксперименты, увеличивалось количество данных, которые получались в результате их проведения, всё труднее становилось анализировать их, делать выводы. Возникла необходимость в появлении целого научного направления — методов обработки результатов эксперимента.

Представим себе испытательный полёт современного авиалайнера, снабжённого сотнями датчиков, измеряющих давление набегающего потока воздуха в различных местах конструкций, характеристики работы двигателя, деформации силовых элементов под действием аэродинамических и других нагрузок и т. д. Что делать с этим непрерывно меняющимся потоком чисел, являющихся результатами эксперимента? Как вообще проводить эксперимент?

Можно действовать разными способами, например, связать каждый датчик с некоторым прибором, допустим стрелочным или осциллографом, который будет непрерывно показывать изменение сигнала, получаемого от соответствующего датчика. За показаниями каждого прибора наблюдают специалисты, отмечая наиболее интересные для целей эксперимента моменты, обмениваясь мнениями между собой и, быть может, вмешиваясь (по возможности) в течение эксперимента, давая те или иные рекомендации проводящим его операторам, наземным службам и пилоту. По завершении эксперимента специалисты по различным 'системам самолёта, наблюдавшие за экспериментом, могут собраться, обсудить получившиеся результаты, наметить следующие эксперименты, вынести какие-то суждения о качестве конструкции. При таком способе проведения эксперимента его информативность, т. е. количество новых сведений, получаемых об испытываемой конструкции, существенно зависит от многих субъективных факторов, таких, как квалификация участников, их умение быстро принимать решение в случае изменения обстановки и тому подобное.

Не исключено, что в этих условиях некоторые важные явления, например, такие, как возникновение кратковременных критических нагрузок в силовых элементах конструкции, могут быть просто не замечены.

Рассмотрим другой вариант, более надёжный. Одновременно с наблюдением за ходом эксперимента со стороны специалистов ведётся достаточно подробная фиксация результатов эксперимента с помощью тех или иных записывающих устройств, например самописцев, магнитных накопителей, с тем чтобы по окончании его произвести более тщательный, более подробный анализ измерений. Отметим, что таким образом в настоящее время и проводится большинство подобных экспериментов. Последующая обработка результатов эксперимента — это те или иные расчёты. Рост сложности объектов экспериментирования, увеличение количества данных, использование в расчётах сложных математических моделей требуют всё больших объёмов проводимых вычислений, и здесь на помощь человеку приходит ЭВМ со своими колоссальными возможностями в проведении расчётов.

Но эксперимент не всегда течёт сам по себе, человеку необходимо вмешиваться в его ход, контролировать, управлять экспериментом. Он в состоянии сделать это, если проводимые опыты просты и протекают достаточно медленно.

Однако не всегда дело обстоит таким образом. Потоки данных могут быть столь велики и меняться так быстро, что надеяться на их хоть какой-то анализ экспериментаторами в темпе поступления невозможно. Значит, невозможно и управлять ходом опыта. Эксперимент может стать неконтролируемым, малоэффективным, возможно опасным.

Если расчёты предшествуют эксперименту или производятся после его проведения с целью обработки полученных данных, то мощность современных ЭВМ позволяет надеяться, что с этими расчётами можно справиться, используя традиционные методы организации и управления вычислительными процессами. Однако при проведении, вычислений с помощью ЭВМ в темпе, определяемом некоторыми внешними по отношению к ней процессами, появляются специфические проблемы в управлении самими вычислениями и в описании этих вычислений программами. Возникающие здесь задачи составляют содержание большого раздела современного программирования — программирование в реальном времени,— которому и посвящена данная публикация.

В первой главе рассказывается об общей схеме обработки эксперимента в реальном времени, приводится определение вычислительной системы реального времени (ВСРВ), излагаются требования к их надёжности.

Во второй главе проводится формализация понятия «вычислительный процесс» и рассматривается решение задач, связанных с корректным выделением ресурсов системы для выполнения параллельных вычислений.

Третья глава посвящена рассмотрению специального класса ВСРВ — детерминированных систем. Априорное знание временных характеристик вычислительных процессов в таких системах позволяет организовать их работу в реальном времени по заранее составленному расписанию.

В четвёртой главе рассматриваются вопросы реализации программного обеспечения ВСРВ, приводится описание средств автоматизации программирования детерминированных систем.

ГЛАВА 1.

ОБРАБОТКА ЭКСПЕРИМЕНТА В РЕАЛЬНОМ ВРЕМЕНИ

Наибольшей эффективности эксперимента можно достичь, если обрабатывать данные в темпе их поступления, в процессе проведения самого эксперимента. Высокая производительность ЭВМ позволяет ожидать, что все интересующие нас явления не останутся незамеченными, а в случае отклонения процесса от заранее запланированного удастся вовремя сделать нужные коррективы. Не исключено, что выработать управляющие сигналы сможет одна из программ, ведущих обработку данных эксперимента. Таким образом, ЭВМ оказывается включённой в контур управления экспериментом.

В самом общем виде последовательность обработки данных эксперимента может быть представлена схемой, приведённой на рис. 1. От датчиков экспериментальной установки сигналы различной физической природы в определённые моменты времени, задаваемые устройством опроса датчиков, поступают на вход преобразователя сигнал-код. На его выходных регистрах формируются

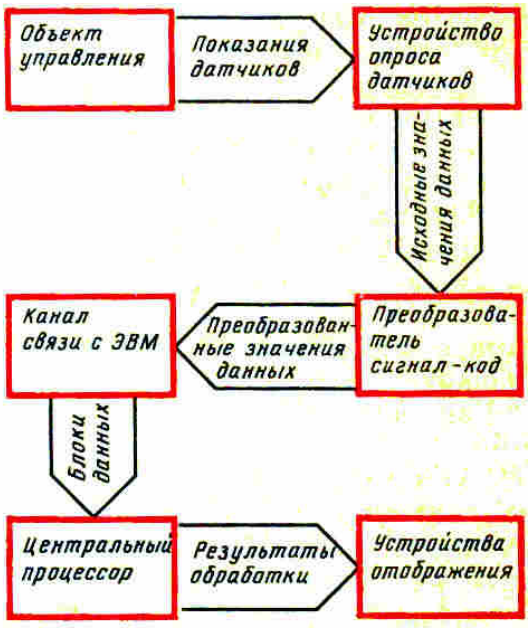


Рис. 1. Общая схема обработки экспериментальных данных в режиме реального времени.

значения соответствующих физических величин в виде цифровых записей, готовых к обработке на ЭВМ, С помощью каналов ЭВМ эти данные поступают на обработку в оперативную память, причём измерения, относящиеся к одному и тому же моменту опроса датчиков, группируются в один кадр. Каналы ЭВМ обладают особенностью, в силу которой выгодно как можно большее число кадров объединить в единый блок для передачи в оперативную память. Дело в том, что непосредственно пересылка данных осуществляется за очень короткое время, необходимое для прохождения электрическим сигналом расстояния от выходных регистров аналого-цифровых преобразователей до регистров оперативной памяти. Значительно большее время требуется на вычисление - адресов, указывающих, откуда взять данные и куда их поместить. Кроме того, если центральный процессор ЭВМ в момент прихода блока данных выполнял команды некоторой программы, то необходимо осуществить прерывание этих вычислений, запомнить состояние прерванной программы для последующего продолжения её работы и предоставить процессор в распоряжение программы, контролирующей приём новой порции данных, т. е. программы, вычисляющей нужные адреса и размещающей данные в памяти ЭВМ в требуемой последовательности. Чтобы уменьшить «накладные расходы» на связь ЭВМ с внешним миром, сеансы связи выгодно делать реже, а для это го следует укрупнить блоки данных, собирая кадры, относящиеся к разным моментам времени, в единую группу.

Следовательно, даже просто приём данных эксперимента требует согласования работы ЭВМ с некоторым процессом, протекающим вне ЭВМ. Действия вычислительной системы следует согласовать с процессом выработки данных экспериментальной установкой. Оказавшись в оперативной памяти ЭВМ, данные должны быть подвергнуты обработке программами, вычисляющими различные характеристики изучаемого в эксперименте явления. Некоторые результаты этих вычислений должны оперативно выводиться на различные устройства отображения — алфавитно-цифровые и графические дисплеи, печатающие устройства. Другие — запоминаться для последующего, более тщательного анализа на магнитных дисках и лентах, третьи — использоваться в качестве входных значений для других программ, вычисляющих новые характеристики и управляющие сигналы для воздействия на экспериментальную установку. Ясно, что время, отпущенное на выполнение этих расчётов, зависит от скорости протекания процессов в наблюдаемом эксперименте. Таким образом, действия вычислительной системы должны быть синхронизированы с внешними процессами, и мы приходим к выводу о том, что для контроля эксперимента в темпе его проведения ЭВМ должна работать в режиме реального времени.

Здесь следует сказать, что такой режим работы вычислительных систем характерен не только для обработки эксперимента в темпе проведения, но используется и во многих других случаях. Управление гибкими автоматизированными производствами, роботами, транспортными системами, атомными реакторами и другими мощными энергетическими установками, оперативный контроль за состоянием пациента во время операции, управление и контроль работы систем современных самолётов и космических ракет с помощью бортовых и наземных вычислительных средств — всё это примеры использования ЭВМ в режиме реального времени.

Вычислительная система реального времени — это «система, которая управляет внешними объектами, получая информацию, обрабатывая её и возвращая результаты достаточно быстро для того, чтобы воздействовать на функционирование внешних объектов почти в тот же момент времени» [1].

Итак, главной отличительной чертой вычислительной системы реального времени (ВСРВ) является необходимость синхронизации выполнения вычислений с событиями вне ЭВМ. Что здесь особенного, может спросить читатель, ЭВМ всегда работает во взаимодействии с некоторым внешним процессом (таким процессом могут быть, например, действия оператора, устанавливающего колоду перфокарт в читающее устройство машины или посылающего с помощью

клавиатуры различные команды для исполнения ЭВМ). Да и выдачу результатов вычислений на печатающее устройство или экран дисплея оператор ожидает через определённое время.

Ответ на поставленный вопрос заключается в следующем. Если перфокарты, находящиеся в кармане устройства ввода, один раз начнут вводиться сразу же, а в другой — спустя несколько минут (причиной этому может служить, например, отсутствие свободного места в буфере задач, уже решаемых ЭВМ), то мы не в праве усомниться в работоспособности системы. Точно так же, если исполнение введённой с клавиатуры команды задержится на одну-две минуты, это вызовет, быть может, раздражение, но отнюдь не будет свидетельствовать о неправильной работе ЭВМ. И результаты расчётов, особенно очень больших программ, программисты привыкли ждать часами (а иной раз даже сутками). В системе же реального времени задержка в выдаче некоторой команды управления, например, для совершающего экспериментальный полёт самолёта, скорее всего будет равносильна отказу в работе. Учитывая тяжесть последствий сбоев в работе системы управления подобными процессами, становится очевидным, что надёжность — главное требование при создании систем реального времени.

Основные причины неправильной работы вычислительной системы — это сбои аппаратуры и ошибки программирования. За десятилетия, прошедшие со времени появления первых ЭВМ, придумано много способов борьбы со сбоями в работе аппаратуры. Используются всё более надёжные электронные системы, реализующие операции ЭВМ, надёжное кодирование передаваемых по каналам связи сообщений, во многих случаях позволяющее автоматически исправить возникающие в процессе передачи ошибки, дублирование функций наиболее ответственных блоков (порой многократное), автоматическое обнаружение некоторых ошибок функционирования ЭВМ, самотестирование, повторный запуск процессов в случае возникновения ошибок. Вследствие этого надёжность аппаратуры в целом растёт, хотя проблема борьбы со сбоями аппаратуры ещё далека от своего полного решения.

Успехи борьбы с ошибками программирования гораздо скромнее — программы составляют люди, и их способность ошибаться не уменьшается, несмотря на прогресс в развитии вычислительной техники. Конечно, средства автоматизации программирования, такие, как. компиляторы с языков высокого уровня, отладчики, автоматизированные редакторы текстов, позволяют гораздо быстрее обнаруживать и исправлять допущенные программистом ошибки, но и сами программы сейчас становятся всё более сложными и изощрёнными. Программы, работающие в реальном времени, являются, пожалуй, самым трудным разделом современного программирования: к ошибкам, свойственным другим видам программ, добавляются специфические виды ошибок, присущие именно программам реального времени. Прежде всего следует сказать об ошибках синхронизации параллельных процессов, существующих в ВСРВ. Рассмотрим подробнее возникающие здесь задачи и способы их решения.

ГЛАВА 2. ВЗАИМОДЕЙСТВИЕ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ В СИСТЕМАХ РЕАЛЬНОГО ВРЕМЕНИ

2.1. Параллелизм в работе вычислительных систем

Проблема синхронизации параллельных процессов не нова, разработчики операционных систем вынуждены были заниматься ею давно, как только возникла идея, что производительность ЭВМ можно увеличить, совмещая во времени действия разных блоков вычислительной машины, например, центрального процессора, выполняющего арифметические и логические операции над данными, и устройств ввода-вывода, осуществляющих обмены информацией с внешней памятью. Появление многопроцессорных ЭВМ заставило задуматься над тем, как организовать взаимодействие друг с другом нескольких процессоров, если таковое потребуется. В сетях ЭВМ, где несколько часто весьма

различающихся по своим характеристикам вычислительных устройств работают параллельно, обмениваясь сообщениями и данными через линии связи, проблема синхронизации процессов, заключающаяся в организации такого управления работой процессов и устройств, чтобы к моменту обмена данными каждый из участников обмена находился в соответствующем состоянии, становится ещё труднее. Чтобы лучше понять природу возникающих проблем и способы их решения, надо выделить то общее, что присуще разнообразным явлениям в ВСРВ, в которых на первый план выступает задача синхронизации параллельно развивающихся во времени процессов. Лучше всего это сделать, построив формальную (или математическую) модель явления. Это позволит строго сформулировать задачу синхронизации и описать алгоритм её решения.

2.2. Понятие процесса в ВСРВ

Слово «процесс» часто встречалось в предыдущих разделах. Считается, что это наиболее употребляемый и наименее точно определённый термин, используемый в работах из области вычислительных систем [2].

Понятие «процесс» охватывает все последовательности событий, происходящих в ЭВМ. Изменение во время работы состояний аппаратуры, составляющей вычислительную систему, изменение содержимого как отдельных ячеек, так и всей памяти ЭВМ, работающей согласно введённой программе, изменение текста самой программы во время её выполнения — всё это процессы.

Существует много формальных определений понятия процесса. Мы рассмотрим одно из них, обладающее достаточной общностью, чтобы охватить большинство явлений в ВСРВ. Это определение предложено Хорнингом и Ренделлом [3].

Назовём процессом тройку (S, F, I). Здесь S — набор переменных, характеризующих состояние процесса. Он называется набором переменных состояния. Состояние задано, когда все переменные принимают конкретные значения. Символом F обозначена функция действия. Если процесс в момент времени t находится в состоянии q_t, то в момент времени t+1 будет находиться в состоянии q_{t+1}, определяемом равенством

$$q_{t+1} = F(q_t). \qquad (1)$$

Считается, что время изменяется дискретно, квантами постоянной величины, длительность которых принимается за единицу. Такое время в данной модели отражает дискретность событий, происходящих в ЭВМ. Что происходит внутри кванта времени, не существенно для описания логики работы вычислительного устройства. Нас интересуют только состояния q_t в конце каждого такта.

Особая группа состояний, называемая начальными состояниями, обозначена символом I.

Содержательно, на примере программного процесса можно считать, что S — это память ЭВМ, q_t — совокупность двоичных кодов, содержащихся в ячейках памяти в момент времени t, F — выполняемая программа, а I — содержимое памяти ЭВМ в начальный момент времени 0, непосредственно перед выполнением программы.

Если два процесса не имеют общих переменных, то они называются независимыми. Работа (т. е. последовательность состояний) каждого из них не влияет на работу другого. Если же общие переменные у двух процессов существуют, то эти переменные называются разделяемыми. Разделяемые переменные служат средством общения процессов друг с другом.

Для того чтобы выполняться (т. е. менять состояния согласно функции действия), процессам требуются ресурсы вычислительной системы — это аппаратные ресурсы (процессоры, различные виды памяти, устройства ввода и вывода, каналы и т. п.) и информационные ресурсы (стандартные программы, компиляторы и т. п.). Ресурсы также могут быть разделяемыми, т. е. использоваться поочерёдно разными процессами. Ресурс, который в каждый момент времени может быть использован только одним процессом, называется *критическим*. Примером такого ресурса может служить единственный процессор ЭВМ или устройство печати. Если несколько процессов обращаются к одному и тому

же критическому ресурсу, то ясно, что они должны получить его в разное время. Таким образом, в работе процесса появляются критические участки — последовательности состояний, в которых процессу требуется критический ресурс. Чтобы исключить одновременное выполнение критических участков несколькими процессами, необходимо синхронизировать их работу. Механизм синхронизации должен удовлетворять двум требованиям: во-первых, если несколько процессов готовы выполнять свои критические участки, то по крайней мере одному из них такая возможность должна быть предоставлена, во-вторых, в течение всего времени нахождения процесса в своём критическом участке ни один другой процесс не может войти в свой критический участок.

Иллюстрацией понятия «критический участок» может служить рис. 2, изображающий два кольцевых маршрута движения поездов, имеющих общий участок, на котором поезда движутся навстречу друг другу. В подобной ситуации термин «критический участок» отнюдь не производит впечатления сгущения красок. В этом реальном случае можно надеяться, что относительно небольшая скорость движения поездов (по сравнению со временем срабатывания сигнализации, например, железнодорожных семафоров) позволит избежать катастрофы, своевременно открывая и закрывая вход в критический участок. Кроме того, простота ситуации (2 процесса, 1 критический участок) позволяет проиграть в уме все варианты, возникающие в процессе управления движением. Для того чтобы лучше понять природу появления в подобных задачах трудностей, читателю предлагается самостоятельно

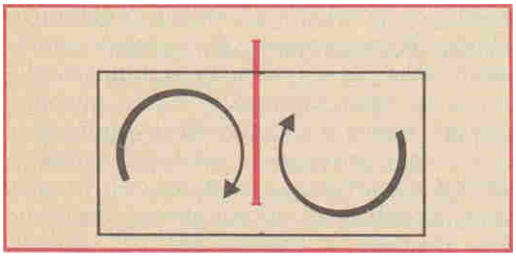


Рис. 2. Критический участок двух кольцевых маршрутов.

придумать систему сигнализации, обеспечивающую безопасное движение поездов по путям и направлениям на рис. 3. Каждый поезд должен бесконечное число раз повторять свой маршрут. Анализируя предлагаемое решение, проверьте: пройдёт ли каждый поезд хотя бы раз по своему маршруту; не будут ли какие-либо поезда бесконечно долго стоять перед критическими участками, ожидая, когда пройдёт встречный; что произойдёт, если один из поездов остановится, например, из-за аварии, вне критического участка; не окажется ли система в «тупике», когда ни один из поездов не имеет права двинуться вперёд и для приведения системы в движение кому-то нужно давать задний ход.

После разбора этого примера вообразите картину, когда процессов очень много (например, сотни), разделяемых ресурсов десятки, а скорости процессов разнятся на несколько порядков, как, например, скорость работы центрального процессора ЭВМ и скорость работы устройства ввода данных с перфоленты. Именно с такими ситуациями приходится иметь дело разработчикам операционных систем реального времени, поскольку главной функцией операционной системы является распределение ресурсов между процессами. Рассмотрим различные способы решения возникающих проблем.

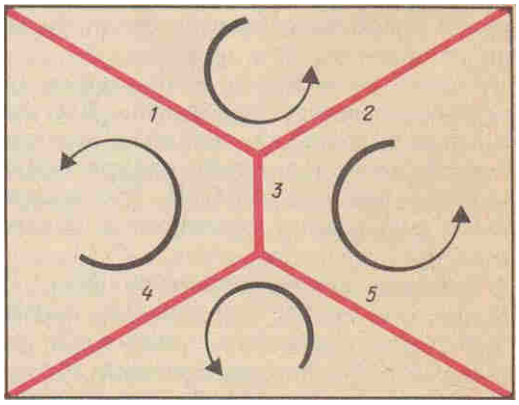


Рис. 3. Четыре кольцевых маршрута: 1, 2, 3, 4, 5 — критические участки

2.3. Решение проблемы «критического участка»

Прежде чем приступить к решению предложенных проблем синхронизации, уточним наши средства описания процессов. Ядром такого описания является построение функции действия F. Если процессы просты, например, значения переменных состояния в последующий момент легко выражаются с помощью арифметических действий через значения в предшествующий момент, то можно воспользоваться описанием функции действия в виде формул (1) из предыдущего раздела. Для более сложных процедур последовательного изменения значений переменных различной природы в программировании известны более удобные и выразительные средства — языки программирования. В дальнейшем мы воспользуемся средствами языка ПЛ/1, несколько расширив множество его конструкций, для того чтобы задавать параллельно выполняющиеся во времени процессы. Все вновь вводимые понятия и наиболее важные из существующих объектов языка ПЛ/1 будут поясняться по мере появления. Заметим, что существуют языки программирования, специально предназначенные для составления программ реального времени или для моделирования параллельных процессов. Можно назвать такие языки программирования, как Модула, Симула, RTL/2, наконец, Ада [4]. Но нас будут интересовать не конкретные программы тех или иных процессов, а иллюстрации к алгоритмам решения задачи синхронизации. Для этих целей вполне достаточно выразительных средств языка ПЛ/1, которые кажутся проще и легче воспринимаемыми при первоначальном знакомстве.

С использованием выбранного языка проблема критического участка иллюстрируется с помощью рис. 4 [2]. Здесь представлены два процесса, обозначенные метками ПРОЦЕСС 1 и ПРОЦЕСС 2. Тела (тексты) обоих процессов заключены в операторные скобки **parbegin** и **parend**, означающие, что процессы выполняются параллельно. Первый процесс состоит в бесконечном повторении двух групп операторов. На факт бесконечного повторения первого процесса указывает конструкция вида **do while** (B). Эта конструкция языка предписывает выполнять последующие операторы до тех пор, пока истинно логическое условие B. Поскольку в первом процессе в качестве логического выражения используется логическая константа **true** (истина), не меняющаяся при выполнении процесса, то последующие операторы (вплоть до второй операторной скобки **end**, первая соответствует операторной скобке **begin**) должны выполняться бесконечное число раз. Первая группа операторов заключена в простые операторные скобки **begin** и **end** и выполняется последовательно. При выполнении этих операторов требуется некоторый критический ресурс. Соответственно название этой группы — КРИТИЧЕСКИЙ УЧАСТОК 1. Остальные операторы первого процесса также выполняются последовательно (друг относительно друга, но параллельно с операторами второго процесса!) и названы «ОСТАВШАЯСЯ ЧАСТЬ ПРОЦЕССА 1». Они не требуют для своего выполнения критического ресурса. Второй процесс устроен подобным образом.

Если уточнить макет представленной на рис. 4 программы, поместив в критические участки операторы, требующие критического ресурса (например, обращающиеся к одному и тому же дисководу ЭВМ или читающие и записывающие информацию в одном и том же для обоих процессов разделе памяти), а оставшиеся части процессов составить из других операторов языка ПЛ/1 и начать выполнять эту программу, например, на некоторой двухпроцессорной ЭВМ, то нет никакой гарантии, что процессы одновременно не окажутся в критических

```
parbegin
ПРОЦЕСС 1:   do while(true);
             begin
               критический участок 1;
             end;
             оставшаяся часть процесса 1:
        end;
ПР0ЦЕСС2:    do while (true);
             begin
               критический участок 2;
             end;
             оставшаяся часть процесса 2;
        end
parend
```

Рис. 4.

участках. Следовательно, нужно согласовать работу процессов, сделать их зависимыми, Для этого вводятся разделяемые переменные ПЕРЕКЛЮЧАТЕЛЬ 1 и ПЕРЕКЛЮЧАТЕЛЬ 2, принимающие одно из двух значений — **true** или **false** (истина или ложь). Множество значений этих переменных задано описанием в первой строке программы на рис. 5 (также заимствованной из [2]).

Теперь каждый процесс, прежде чем войти в свой критический участок, проверяет значение переключателя другого процесса. Если проверяемое значение равно **true**, то процесс «отказывается» от вхождения в свой критический участок и спустя некоторое время, определяемое скоростью выполнения действий, заданных оператором цикла

ЦИКЛ 1: **do while** (ПЕРЕКЛЮЧАТЕЛЬ 2);
end

(для первого процесса), повторяет проверку. Если при очередной проверке первый процесс обнаруживает, что значение переменной ПЕРЕКЛЮЧАТЕЛЬ 2 равно **false**, то он устанавливает значение переменной ПЕРЕКЛЮЧАТЕЛЬ 1 равным **true**, предупреждая второй процесс о своих действиях, и входит в критический участок. После выхода из критического участка 1 процесс присваивает переменной ПЕРЕКЛЮЧАТЕЛЬ 1 значение **false** («Я освободил критический ресурс»,— сообщает он второму процессу) и выполняет

```
Begin boolean ПЕРЕКЛЮЧАТЕЛЬ1,
ПЕРЕКЛЮЧАТЕЛЬ2;
    ПЕРЕКЛЮЧАТЕЛЬ1:= false;
    ПЕРЕКЛЮЧАТЕЛЬ2:= false;
 parbegin
   ПРОЦЕСС1: do while (true);
   ЦИКЛ1:          do while (ПЕРЕКЛЮЧАТЕЛЬ2);
                   end;
                   ПЕРЕКЛЮЧАТЕЛЬ1: = true;
                   /* критический участок 1 */
                   ПЕРЕКЛЮЧАТЕЛЬ1: = false;
                   /* оставшаяся часть процесса 1 */
                end;
   ПРОЦЕСС 2: do while(true);
   ЦИКЛ2:          do while (ПЕРЕКЛЮЧАТЕЛЬ1);
                   end;
                   ПЕРЕКЛЮЧАТЕЛЬ2: = true;
                   /* критический участок 2 */
                   ПЕРЕКЛЮЧАТЕЛЬ2: = false;
                   /* оставшаяся часть процесса 2
                   */
                end;
 parend
end
```

Рис. 5.

оставшуюся часть процесса 1. Второй процесс действует аналогично первому.

Это решение проблемы критического участка для двух параллельных процессов, кажущееся на первый взгляд очевидным, на самом деле попросту небезопасно. Вспомним, что Скорости процессов произвольны, и пусть процесс 2 работает во много раз быстрее процесса 1. Может случиться, что в интервале времени от момента, когда процесс 1 обнаружит, выполняя операторы цикла 1, что ПЕРЕКЛЮЧАТЕЛЬ 2 имеет значение **false**, до момента, когда он установит значение ПЕРЕКЛЮЧАТЕЛЬ 1 **true**, процесс 2 снова войдёт в критический участок, успев быстро выполнить свою оставшуюся часть. Оба процесса одновременно окажутся в своих критических участках.

Ещё одна попытка решения этой задачи представлена на рис. 6. Легко убедиться в безопасности этого варианта. Действительно, в тот момент, когда процесс 1 обнаруживает, что C2 = 0 и он может войти в свой критический участок (прекратив ожидание ЦИКЛ 1), выполнены условия:

1) значение C1 = 1 уже установлено и не изменится, пока первый процесс не выйдет из критического участка;

2) поскольку C2 = 0, ПРОЦЕСС 2 находится вне своего критического участка и не сможет войти в него, пока выполняется равенство C1 = 1, т. е. до выхода первого процесса из своего критического участка.

```
Begin integer     c1, c2;
                  c1 := 0;
                  c2 := 0;
   parbegin
     ПРОЦЕСС 1:  do while (true);
                        c1 := 1;
            ЦИКЛ1:      do while (c2 = 1);
                        end;
                        критический участок 1;
                        c1 := 0;
                        оставшаяся часть процесса 1
                     end;
     ПР0ЦЕСС2:  do while(true);
                        c2 := 1;
          ЦИКЛ2: do     while(c1 = 1);
                        end;
                        критический участок 2;
                        c2 := 0;
                        оставшаяся часть процесса 2
                     end
   parend
end
```

Рис. 6.

Однако это решение также не приемлемо — оно содержит опасность взаимной блокировки процессов. Если в то время, когда ПРОЦЕСС 1 выполняет оператор присваивания C1 := 1, но ещё не осуществил проверку C2 = 0, ПРОЦЕСС 2 выполнит оператор присваивания C2 := 1, то оба процесса начнут выполнять операторы ожидания ЦИКЛ 1 и ЦИКЛ 2 при Одновременном выполнении соотношений C1 = 1 и C2 = 1. Их ожидание разрешения войти в критический участок затянется до бесконечности.

Правильное решение предложено Деккером и представлено на рис. 7. В этом алгоритме процессы используют три разделяемые переменные. Когда ПРОЦЕСС 1 хочет войти в свой критический участок, он устанавливает значение переменной C1, равное единице, а C2 = 1, когда ПРОЦЕСС 2 намеревается войти в свой критический участок. Третья разделяемая переменная ОЧЕРЕДЬ указывает, кому предстоит выполнить свой критический участок в случае, если оба процесса намерены это сделать одновременно.

```
Begin   integer  c1, c2, ОЧЕРЕДЬ;
                 c1 :=0; c2 :=0; 0ЧЕРЕДЬ :=1;
 parbegin
  ПРОЦЕСС1: begin  c1 :=1;
                   do while (c2 = 1);
                   if ОЧЕРЕДЬ = 2
                       then  begin  c1: = 0;
                          do while (ОЧЕРЕДЬ =2);
                          end;
                          c1 :=1
                       end
                   end;
                   критический участок процесса 1;
                   c1 :=0;
                   ОЧЕРЕДЬ :=2;
                   оставшаяся часть процесса 1
                 end;
  ПРОЦЕСС2: begin  c2 :=1;
                   do while (c1 = 1);
                   if ОЧЕРЕДЬ = 1
                       then  begin  c2: = 0;
                          do while (ОЧЕРЕДЬ =1);
                          end;
                          c2 :=1
                       end
                   end;
                   критический участок процесса 2;
                   c2 :=0;
                   ОЧЕРЕДЬ :=1;
                   оставшаяся часть процесса 2
                 end;
 parend
end
```

Рис. 7.

Правильность этого решения доказана в работе [5]. В работах [6, 7] решение Деккера обобщено на случай произвольного числа процессов, имеющих общий критический ресурс.

Решение Деккера чрезвычайно остроумно, в чём можно убедиться, проделав вручную все предлагаемые действия при различных соотношениях скоростей первого и второго процессов. Но оно не лишено недостатков.

Во-первых, разделяемые переменные являются средством низкого уровня с точки зрения программиста (как, например, язык ассемблера по сравнению с каким-либо языком программирования высокого уровня). Следовательно, использование их в сложных ситуациях, когда имеется несколько критических ресурсов и много процессов, оказывается затруднительным, появляются многочисленные ошибки, отладка, и сопровождение системы удорожается.

Во-вторых, ожидание процессов перед входом в критический участок в алгоритме Деккера является активным — процессы всё время проверяют значения разделяемых переменных, а значит, используют для этого какие-то ресурсы вычислительной системы.

2.4. Семафоры

Итак, желательно построить такие средства для синхронизации процессов, которые обладают большей наглядностью, общностью и исключают активное ожидание. В [5] для решения задачи синхронизации предлагается использовать семафоры.

Семафор S — это целочисленная переменная, над которой в процессах разрешается выполнять лишь две операции, обозначаемые обычно буквами P и V.

Если процесс выполняет операцию $P(S)$, то значение S уменьшается на единицу. Кроме того, если $S \geq 0$, то процесс продолжает работу, а если $S < 0$, то процесс останавливается, помещается в очередь вместе с другими заблокированными этим семафором процессами. Работу он сможет продолжить лишь после того, как некоторый другой, активный процесс выполнит операцию $V(S)$.

Если процесс выполнит операцию $V(S)$, значение S увеличивается на единицу, и процесс продолжает работу. Кроне того, если $S < 0$, то один из блокированных процессов удаляется из очереди перед семафором S и становится активным.

Важнейшее свойство операций P и V — неделимость. Если один из процессов начал выполнять операцию, то для других процессов этот семафор недоступен до завершения этой операции.

На рис. 8 представлено решение проблемы критического участка с использованием семафора [2]. Роль этого семафора выполняет целая переменная СВОБОДНЫЙ, принимающая вначале значение 1. Если S =1, то это значит, что оба процесса находятся вне своих критических участков. Равенство S =0 означает, что один из процессов вошёл в критический участок, а если $S = -1$, то один из процессов вошёл в критический участок, а второй ожидает своей очереди перед семафором S. Как только находящийся в критическом участке процесс выйдет из него и завершит операцию V(S), второй процесс войдёт в критический участок, предварительно выполнив операцию P(S). Неделимость операций P и V гарантирует безопасность решения.

Простота и наглядность предлагаемого решения по сравнению с алгоритмом Деккера очевидны. Исчезло активное ожидание. Процессы не выполняют никаких действий,

```
Begin   integer  СВОБОДНЫЙ ;
    СВОБОДНЫЙ := 1;
    parbegin
     ПРОЦЕСС1: begin
                    do while(true);
                    начало процесса 1;
                    P(СВОБОДНЫЙ);
                    критический участок 1;
                    V(СВОБОДНЬЙ);
                    оставшаяся часть процесса 1
                    end
                  end;
     ПРОЦЕСС2: begin
                    do while(true);
                    начало процесса 2;
                    P(СВОБОДНЫЙ);
                    критический участок 2;
                    V(СВОБОДНЬЙ);
                    оставшаяся часть процесса 2
                    end
                  end;
parend
end
```

Рис. 8.

находясь в очереди у семафора. Какова цена этих преимуществ? Необходимо иметь семафоры, т. е. возможность выполнения неделимых операций P и V над целочисленными переменными. Кроме того, операционная система должна содержать средства, позволяющие активизировать и останавливать процессы. Если на время ожидания желательно изъять некоторые ресурсы у заблокированного процесса, например процессор, то необходимо уметь восстанавливать нужное состояние ресурса после активизации бывшего процесса-хозяина.

Обычно семафоры стараются реализовать аппаратурой вычислительной системы, с тем чтобы сократить время их работы, т. е. время выполнения операций P и V. Аппаратные средства используют и при реализации функций операционной системы, связанных с блоки ровной и активизацией процессов.

Критическими ресурсами являются. не только устройства ЭВМ, но и сами разделяемые переменные и различные сообщения, которыми процессы обмениваются друг с другом.

Рассмотрим так называемую проблему «поставщик – потребитель».

Пусть ПОСТАВЩИК — это процесс, который готовит и посылает порции информации другому процессу, называемому ПОТРЕБИТЕЛЬ. ПОТРЕБИТЕЛЬ получает порции информации и обрабатывает их. Например, ПОСТАВЩИК — это программа, которая вырабатывает некоторые значения, группирует их в блоки стандартного размера и отправляет процессу ПОТРЕБИТЕЛЬ. Тот получаемые сообщения разбивает на строки и печатает их на устройстве печати.

Поскольку скорости процессов могут различаться, более того, меняться со временем, для обмена данными между ПОСТАВЩИКОМ и ПОТРЕБИТЕЛЕМ устанавливается буфер, состоящий из некоторого фиксированного количества ячеек, в каждую из которых помещается ровно одно сообщение. ПОСТАВЩИК, посылая сообщение, помещает его в хвост очереди из занятых ячеек, а ПОТРЕБИТЕЛЬ каждый раз берёт для обработки сообщение, находящееся в первой из заполненных ячеек. При выполнении такого обмена процессам необходимо

взаимодействовать друг с другом. Например, ПОСТАВЩИК может посылать сообщения в буфер, пока там есть свободные ячейки. Послав сообщение, он должен уменьшить на единицу значение переменной, в которой хранится число свободных ячеек. ПОТРЕБИТЕЛЬ может брать сообщения из буфера, пока есть, хотя бы одна заполненная ячейка, и, взяв его, должен увеличить на единицу значение числа свободных ячеек. Следовательно, наши процессы имеют по меньшей мере одну разделяемую переменную.

Назовём эту переменную СЧЕТЧИК и рассмотрим следующую возможную последовательность событий во взаимодействии процессов ПОСТАВЩИК и ПОТРЕБИТЕЛЬ:

ПОСТАВЩИК считывает значение переменной СЧЕТЧИК в некоторую свою локальную переменную ТЕМП 1, уменьшает значение ТЕМП 1 на единицу;

ПОТРЕБИТЕЛЬ считывает СЧЕТЧИК в локальную переменную ТЕМП 2 и увеличивает значение ТЕМП 2 на единицу;

ПОСТАВЩИК посылает сообщение в буфер, а переменной СЧЕТЧИК присваивает значение переменной ТЕМП 1;

ПОТРЕБИТЕЛЬ получает сообщение из буфера и присваивает переменной СЧЕТЧИК значение ТЕМП 2.

Хотя количество свободных ячеек не изменилось, значение переменной СЧЕТЧИК уменьшилось на единицу. При других последовательностях событий переменная СЧЕТЧИК может не изменить своё значение, а может увеличиться на единицу. Этот простой пример показывает, что даже учёт числа свободных ячеек в буфере требует синхронизации работы процессов. Аналогичная проблема возникает при передаче сообщений от процесса ПОСТАВЩИК к процессу ПОТРЕБИТЕЛЬ. Если ПОСТАВЩИК посылает сообщение в буфер как раз в то время, когда ПОТРЕБИТЕЛЬ получает оттуда последнее сообщение, то в еле дующем цикле один из процессов может обратиться к буферу с неправильным номером ячейки.

Решение проблемы «поставщик — потребитель» приведено на рис. 9, взятом попрежнему из [2] В этом алгоритме используются три семафора. Переменная "НАЛИЧИЕ — семафор, указывающий число свободных ячеек в буфере, значение семафора ЗАПОЛН равно числу заполненных ячеек, посланных потребителю, Семафор ВЗАИСК (ВЗАимное ИСКлючение) принимает лишь три значения: 1, 0, –1. Этот семафор не позволяет обоим процессам одновременно изменять значения номеров ячеек — той, в которую в данный момент помещается, и той, из которой считывается сообщение. ПОСТАВЩИК содержит один критический участок, который образован группой операторов «послать сообщение», критический участок процесса ПОТРЕБИТЕЛЬ — это операторы, названные «получить сообщение». До начала работы процессов семафору НАЛИЧИЕ присваивается значение, равное числу свободных ячеек буфера. Работа алгоритма проста, нежелательных явлений, связанных с неправильной адресацией, не возникает.

Идея неделимых операций над семафорами, развиваясь, приводит к понятию монитора [8, 9]. Монитором называется набор процедур и информационных массивов, которым в каждый момент может пользоваться лишь один процесс. В тело монитора удобно собрать все разделяемые процессами объекты, такие, как разделяемые переменные, семафоры. Кроме того, мониторы должны

```
Begin  integer  НАЛИЧИЕ, ЗАПОЛН, ВЗАИСК;
       НАЛИЧИЕ := количество свободных буферов;
       ЗАПОЛН := 0;
       ВЗАИСК : = 1;
   parbegin
       ПОСТАВЩИК:
          begin
             do while(true);
                приготовить сообщение;
                P(НАЛИЧИЕ);
                P(ВЗАИСК);
                послать сообщение;
                V(ЗАПОЛН);
                V(ВЗАИСК);
             end
       end
       ПОТРЕБИТЕЛЬ:
          begin
             do while(true);
                P(ЗАПОЛН);
                P(ВЗАИСК);
                получить сообщение;
                V(НАЛИЧИЕ);
                V(ВЗАИСК);
                обработать сообщение;
             end
       end
   parend
end
```

Рис. 9.

обладать возможностью блокировать и активизировать процессы, Поскольку в мониторе сосредоточены все семафоры, которые с целью синхронизации останавливают и запускают процессы, то остановки процессов при обращении к монитору происходят, вообще говоря, по разным причинам — один процесс задерживается одним семафором, другой — другим. Поэтому когда монитор блокирует некоторый процесс, переводит его в состояние ожидания, то он указывает условие (например, положительность значения некоторого семафора *S*), при котором блокировка процесса снимается. Когда это условие выполняется, то в функцию монитора входит выработка соответствующего сигнала, и перевод одного из процессов, стоящих в очереди ожидающих выполнения этого условия, в активное состояние.

Решение задач синхронизации при использовании монитора достигает ещё большего единообразия. Разработчики ВСРВ аппаратными или программными средствами обеспечивают неделимость обращения к монитору, и все авторы прикладных процессов получают возможность организовать синхронизацию работы, используя монитор.

Следует упомянуть ещё один механизм синхронизации процессов, получивший название «рандеву*. Суть этого подхода заключается в том, что передача данных от одного процесса к другому и их синхронизация рассматриваются как единая неразделимая деятельность. Модель этого механизма предложена Хоаром и Хансеном [10, 11]. Когда процесс А намерен передать данные процессу В, оба процесса должны быть готовы установить связь, выдав соответствующие заявки на передачу данных и их получение. Если один из процессов выдаст свою заявку раньше другого, то он должен приостановиться, до тех пор пока другой процесс не выдаст свою заявку. Осуществив таким образом синхронизацию, процессы производят обмен данными и продолжают свою деятельность. В идеале предполагается, что передача данных в момент синхронизации происходит мгновенно. Исходя из этого при реализации механизма «рандеву» не предполагается наличие системного буфера, и поэтому передача данных между, асинхронными процессами может потребовать создания дополнительного буферного процесса. Это несколько искажает элегантность решения проблемы синхронизации с помощью «рандеву».

Тем не менее такой механизм считается более удобным и естественным для использования, особенно в многопроцессорных системах с разделёнными ресурсами. Именно организация «рандеву» является средством синхронизации процессов в языке Ада.

2.5. Тупики при распределении ресурсов в вычислительных системах

В связи с распределением ресурсов вычислительной системы между процессами возникает ещё одна трудная и интересная проблема — проблема тупиков. В отличие от критических участков тупики представляют опасность и в том случае, когда два процесса не являются «истинно параллельными*. Например, тупик может возникнуть, когда однопроцессорная ЭВМ выполняет две независимые программы в режиме мультипрограммирования, т. е. обе программы одновременно представлены в оперативной памяти и поочерёдно получают в своё распоряжение процессор на некоторое время. Положим, что объём оперативной памяти ЭВМ составляет 1024 страницы, максимальная потребность каждой программы — 1000 страниц, память выделяется по запросам, которые программы делают по ходу выполнения, и каждая программа может в любой момент времени запросить в дополнение к уже выделенным ей страницам новый объём памяти в пределах заявленного ею в начале выполнения максимального объёма (1000 страниц). Тогда, если первая программа запросит и получит 400 страниц, затем столько же — вторая, свободными в системе останутся 224 страницы. Если следующие запросы в обеих программах требуют по 300 страниц памяти, то их удовлетворить нечем. Система попадает в тупик, без постороннего вмешательства процессы неопределённо долго будут ждать поступления ресурсов, ни у одного из

них нет возможности пройти через второй запрос объёма памяти Тупики возникают при обращении параллельных процессов к нескольким видам ресурсов. Например, два параллельных процесса, выполняющие такие последовательности операций при обращении к семафорам S1 и S2 могут оказаться

ПРОЦЕСС 1: begin... P(S1); P(S2);... end
ПРОЦЕСС 2: begin... P(S2); P(S1);... end,

в тупике, если начальные значения семафоров равны 1, и каждый процесс выполнит своё обращение к семафору до последующего обращения к семафору другого процесса.

Существуют две стратегии борьбы с тупиковыми ситуациями — при первой тупики обнаруживаются и ликвидируются, при второй — предотвращаются. Понятно, что обнаружение тупиков может осуществляться при участии человека, однако этот путь вряд ли приемлем для ВСРВ, ведущих обработку некоторого быстрого и ответственного эксперимента. Существуют алгоритмы автоматического анализа сложившейся ситуации в распределении ресурсов с целью обнаружения тупика. В обоих случаях ликвидация тупика производится перераспределением ресурсов, чтобы дать возможность завершиться некоторым процессам и освободить ресурсы для других процессов. Как правило, при этом приходится уничтожать некоторые процессы (с тем чтобы потом запустить их заново).

Предотвращение тупиков основывается на информации о максимальной потребности процесса в ресурсах и учёте уже выделенных. Простейшая стратегия предотвращения тупиков заключается в выделении процессу сразу максимально требуемого им количества всех ресурсов. Ясно, что при такой стратегии ресурсы могут быть отданы процессу задолго до того, как они действительно понадобятся.

Более сложный механизм предотвращения тупиков основан на использовании при каждом запросе со стороны процесса очередной порций некоторого ресурса специальных алгоритмов, проверяющих, возможна ли в результате удовлетворения запроса тупиковая ситуация.

Если возникающая ситуация небезопасна, то запрос отклоняется.

Одна из подобных процедур, так называемый «алгоритм банкира» [5], приведена на рис. 10. В этом алгоритме каждому процессу присвоен номер i ($1 \le i \le N$). В качестве распределяемого ресурса фигурируют некоторые устройства, общее количество которых обозначено ОБЩУСТР. Текущее число свободных устройств хранится в переменной СВОБУСТР. Максимальная потребность в ресурсе процесса с номером с задаётся элементом массива МАКС [i] В каждый момент времени распределение ресурса по процессам характеризуется двумя массивами целых чисел. Массив ВЫДЕЛУСТР [1:N] содержит количество устройств, уже выделенных каждому процессу. В массиве ОСТАТОК [1:N] хранятся полагающиеся каждому процессу числа невостребованных единиц ресурса. Кроме того, с каждым процессом связана булевская переменная ЗАВЕРШСОМН [i] ($1 \le i \le N$). В начальный момент неизвестно, закончится ли хотя бы один процесс, т. е. ЗАВЕРШСОМН [i] = true для всех i.

Алгоритм включается в работу каждый раз, когда один из процессов запрашивает очередную порцию устройств. Не проводя фактического выделения процессу этих устройств, система увеличивает соответствующую переменную ВЫДЕЛУСТР [i] на требуемое число единиц и выполняет «алгоритм банкира». После выполнения первых шести строк программы определяется число

```
СВОБУСТР := ОБЩУСТР;
for  i :=1  step 1  until N  do
   begin СВОБУСТР:= СВОБУСТР – ВЫДЕУСТР[i];
         ЗАВЕРШСОМ [i] := true;
         ОСТАТОК [i] :=МАКС[i] – ВЫДЕЛУСТР[i];
   end;
ПРИЗНАК := true;
do while (ПРИЗНАК);
 ПРИЗНАК := false;
   for i:=1 step 1 until N do
     begin
       if (ЗАВЕРШСОМ[i] and ОСТАТОК[i]<=СВОБУСТР)
       then begin
          ЗАВЕРШСОМ [i] := false;
          СВОБУСТР:=СВОБУСТР+ ВЫДЕЛУСТР [i];
          ПРИЗНАК := true
       end
     end
end;
if  СВОБУСТР = ОБЩУСТР
   then   состояние системы безопасное
   else   состояние системы небезопасное
```

Рис. 10.

остающихся нераспределёнными устройств и максимальные запросы, которые могут быть сделаны процессами впоследствии.

Последующие действия алгоритма проверяют, хватит ли свободных устройств при условии, что все процессы запросят полностью причитающиеся им остатки ресурса. Если находится хотя бы один процесс, для которого ОСТАТОК [*i*] < СВОБУСТР, то предполагается, что он выполняется до конца и освобождает выделенные ему устройства. Они прибавляются к числу свободных устройств, и ищется следующий процесс, с которым можно поступить аналогично. Если в результате такой имитации выяснится, что все процессы имеют возможность завершиться (в этом случае число свободных устройств примет значение общего числа устройств), состояние системы в результате выделения дополнительного ресурса процессу, выдавшему запрос, объявляется безопасным и запрос удовлетворяется. В противном случае запрос отклоняется, и сделавший его процесс ждёт более благоприятного момента. Своим названием этот алгоритм обязан тому обстоятельству, что его действия напоминают поведение осторожного банкира, подсчитывающего, сможет ли он удовлетворить запрос своего клиента об увеличении текущего займа и оказаться в будущем платёжеспособным, если все клиенты потребуют выдать обещанные им займы хотя бы по очереди.

Тупики принадлежат к числу хорошо формализуемых проблем при проектировании вычислительных систем и управляющих ими операционных систем. В частности, весьма содержательные результаты и мощные алгоритмы анализа ситуации на возможность тупиков получены при описании этой проблемы на языке теории графов и с использованием моделирования вычислительных систем сетями Петри [2].

ГЛАВА. 3. ДЕТЕРМИНИРОВАННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

3.1. Случайность и детерминированность в контролируемых физических процессах

Рассмотренные нами механизмы управления параллельными вычислительными процессами обладают значительной универсальностью, они пригодны для решения задач синхронизации процессов, если скорости последних неизвестны и могут варьироваться в самых широких пределах. Такая ситуация имеет место при использовании ВСРВ для контроля экспериментов с самыми разнообразными системами, когда поведение последних плохо предсказуемо, сильно зависит от случайных факторов.

В качестве примера можно взять исследование надёжности функционирования какого-нибудь сложного современного механизма, если по ходу эксперимента могут наблюдаться различные, порой совершенно неожиданные отклонения от планируемого течения процесса — аварии, выходы из строя отдельных частей испытуемой системы. Возможно, что в каждом таком случае в работу должны включаться новые программы реального времени, которые в случайные моменты начнут обращаться к ресурсам ВСРВ, в том числе критическим. Чтобы избежать ошибочной работы системы обработки и управления подобными физическими процессами, вызываемой неправильными действиями с разделяемыми объектами и ресурсами, необходимо оградить последние средствами защиты в виде семафоров, мониторов и т. д.

Аналогичным образом приходится поступать и при реализации интерактивных (допускающих непосредственное взаимодействие пользователя с ВСРВ) систем с большим числом пользователей. Количество и типы запросов и моменты их поступления в вычислительную систему являются случайными величинами. Конкуренция нескольких процессов за владение одними и теми же ресурсами становится частым событием, и если не приняты соответствующие меры, возможны тупики, одновременные входы в критические

участки, ссылки на неправильные адреса переменных и другие нежелательные явления.

Однако в целом ряде случаев, как при обработке экспериментов в темпе их проведения, так и при других использованиях вычислительных средств в режиме реального времени, течение контролируемых физических процессов может быть предсказано достаточно точно для того, чтобы можно было с уверенностью планировать проведение соответствующих сопровождающих расчётов. Во всяком случае, даже в процессах, имеющих случайный характер, как правило, существуют участки с детерминированным поведением. Тогда появляется возможность применить для управления вычислительными процессами в ВСРВ более точные методы планирования. В качестве примеров таких детерминированных процессов можно привести управление автоматической производственной линией, упоминавшиеся во введении испытательные полёты самолётов, обычно состоящие из одного или нескольких заранее намеченных режимов полёта, в которых определённая информация о функционировании различных систем обрабатывается точно известными программами реального времени.

3.2. Математические модели детерминированных ВСРВ

Итак, мы приходим к выводу, что контроль детерминированных физических процессов (или их отдельных детерминированных участков) может проводиться с помощью особого вида ВСРВ — детерминированных ВСРВ (иногда их называют также вычислительными системами жёсткого реального времени).

При проектировании таких систем ведущее значение получает простой принцип: предварительная подготовка вычислений, проводимых в реальном времени, должна быть максимальна. Конечно, этот принцип разумен при проектировании программного обеспечения любых ВСРВ, а не только детерминированных. Но именно в последних последовательное его применение приводит к наибольшему эффекту. Согласно этому принципу основным инструментом анализа и синтеза программного обеспечения ВСРВ становится расчёт его функционирования в различных рабочих режимах. Целью такого расчёта является составление расписания, указывающего, когда и какой процедуре программного обеспечения должны выделяться ресурсы ВСРВ при выполнений конкретной программы реального времени

Расчёт расписания работы ВСРВ осуществляется с помощью математических моделей. Эти модели рассмотрены во многих работах (см., например, [12]). Как правило, модели ВСРВ строятся в виде систем обслуживания $S = (M, N, G, T, R, F)$. Поясним смысл отдельных компонент в этом обозначении. Символом M обозначено множество параллельно работающих приборов $\{m_1, m_2, ..., m_k\}$. В вычислительной системе можно считать, что каждый прибор — это арифметико-логический процессор либо процессор ввода-вывода.

Множества работ, которые должны быть выполнены на этих приборах, обозначается N. Каждой работе присваивается свой номер, так что $N = \{1, 2, ..., n\}$ или $N = \{1, 2, ...\}$ в зависимости от того, конечно или бесконечно число работ. Содержательно под работой можно понимать выполнение некоторой процедуры, программы, системной функции, операции обмена с внешним устройством и т. д.

Работы не являются независимыми, некоторые из них, прежде чем начать выполняться, должны дожидаться окончания других работ по той, например, причине, что ожидаемые работы поставляют входные данные для ждущей работы. Эти отношения предшествования для совокупности работ N задаются графом G. Направленная дуга от вершины i этого графа к вершине j означает, что работа с номером j не может выполняться до окончания работы i. На рис. 11 приведён пример графа предшествований для конкретной совокупности работ 1, 2, ..., 10. Заметим, что такой граф не должен иметь ориентированного цикла, т. е. направленного пути, начинающегося и заканчивающегося в одной и той же вершине. Понятно, что такой цикл был бы порочным кругом — любая работа, в него входящая, прежде чем начать выполняться, должна дожидаться своего окончания. Кроме того, наше отношение предшествования обладает транзитивностью:

если работа i предшествует работе j, а работа j – работе l, то работа i предшествует работе l. Можно конечно, во всех таких случаях провести соответствующие направленные дуги и получить вместо графа G его так называемое транзитивное замыкание не внося, по существу, новой информации о порядке выполнения работ, такие дуги во много раз увеличивают объём описания графа. Поэтому обычно при постановке задач планирования сети работ наоборот из графа предшествований удаляются все «транзитивные» дуги, остаются только дуги, означающие, что работа i непосредственно предшествует работе j. Отношение, заданное таким графом является отношением строгого частичного порядка (на множестве работ), а сам граф называется графом редукции этого отношения.

Составить расписание работ невозможно, не имея информации о времени выполнения каждой работы. Обычно считается заданным время t_{qp} выполнения работы q на приборе p. Совокупность этих времён образует матрицу T из N строк и M столбцов. Получить точные значения этих величин удаётся отнюдь не всегда, часто приходится довольствоваться лишь оценками длительностей работ. Если приборы системы обслуживания одинаковые, то вместо матрицы I рассматривается вектор $T = \{t_1, t_2, \ldots, t_k\}$, где t_i означает длительность i-й работы.

Если для выполнения работы недостаточно предоставления ей прибора, а требуются другие ресурсы, то потребность в них задаётся матрицей R. Элемент r_{qp} этой матрицы равен количеству ресурса с номером p, необходимого для выполнения работы q.

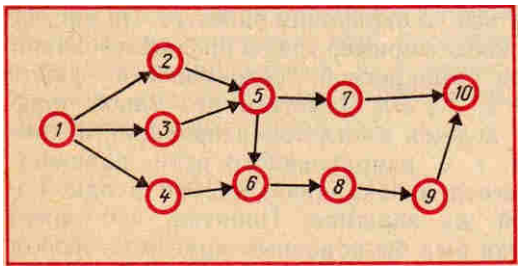


Рис. 11. Граф предшествований для совокупности работ (1, 2, …, 10)

Наконец, символом F в рассматриваемой системе обслуживания обозначается критерий качества составляемого расписания. Применительно к ВСРВ рассматриваются обычно два вида критериев.

Первый вид выделяет задачи на быстродействие. Требуется найти расписание т. е. распределение задач по процессорам, и порядок предоставления им других ресурсов, такие, чтобы время за которое завершатся все работы, было минимальным.

Второй вид критериев связан с задачами, рассматриваемыми в связи с проектированием ВСРВ, которые называется задачами с директивными сроками Для едой работы задана пара чисел b_i, f_i, где b_i — момент поступления работы на обслуживание (раньше этого времени она не может начать выполняться), а f_i — директивный срок окончания (к этому сроку работа обязательно должна завершиться). Расписание, которое требуется найти, называется допустимым.

Расписанием для системы обслуживания $S = (M, N, G, T, R, F)$ называется совокупность кусочно-постоянных непрерывных слева функций $P(t) = \{p_1(t), p_2(t), \ldots. p_k(t)\}$, определённых на интервале планирования $(0, \tau)$ и принимающего целочисленные значения из множества $N = \{1, 2, \ldots, n\}$. Равенство $p_i(t) = j$, $(1 \le i \le k, \quad 1 \le j \le n)$ означает, что в момент времени t прибор с номером i выполняет работу j.

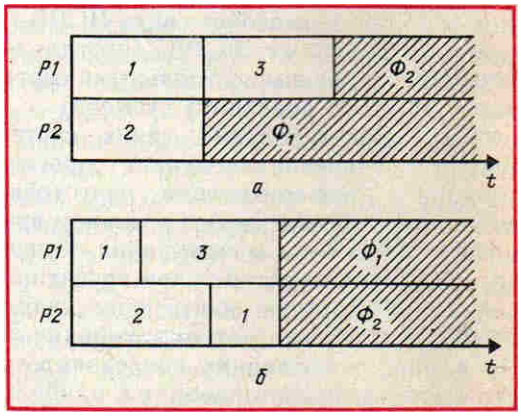


Рис. 11. Общее время вычислений. Расписание без прерываний (*а*) и с прерываниями (*б*)

Аналогично можно определить расписание использований каждого ресурса, представляющего отдельные устройства, например, канал, устройство ввода, печатающее устройство. Расписание использований памяти можно представить таким же способом, понимая равенство $p_i(t) = j$ как определение события «в момент времени t раздел памяти с номером i предоставлен работе j». Если при моделировании нас не интересуют конкретные номера разделов памяти, предоставленных отдельным работам, а важно только выделенное количество памяти, то расписание изображается набором из n функций того же вида, но равенство $p_i(t) = j$ означает, что работа i в момент t использует j единиц памяти.

Удобный способ графического изображения расписания (за исключением последнего варианта расписания для памяти) иллюстрируется рис. 12. Число горизонтальных полос равно числу используемых процессоров, горизонтальные линии изображают оси времени. В каждом отрезке горизонтальной полосы пишется номер работы, которой в это время предоставлен данный процессор. Если процессор простаивает, то соответствующий участок горизонтальной полосы заштрихован и помечен символом $\varnothing$ с индексом для возможных ссылок на этот участок.

Существенным моментом, который обязательно оговаривается при постановке задачи планирования, является возможность прерывания работ и продолжения их через некоторый промежуток времени, возможно, на другом приборе. Если же прерывания запрещены, то работа i, начавшись в момент времени t на некотором приборе, выполняется непрерывно и завершается в момент $t+t_i$ на этом же приборе.

Важность наличия или отсутствия прерываний хорошо иллюстрируется следующим простым примером. Рассмотрим систему обслуживания $S = (\{1, 2\}, \{1, 2, 3\}, \varnothing, \{1, 1, 1\}, \varnothing, F)$. Используются два идентичных прибора (процессора) с номерами 1, 2. Работы 1, 2 и 3 имеют единичную длительность каждая, для них не задано отношение предшествования (граф G не имеет дуг, обозначение $G=\varnothing$) и не требуется других ресурсов ($R=\varnothing$). Решается задача на быстродействие, содействующий критерий обозначен F. На рис. 12 изображено одно из решений этой задачи при невозможности прерываний работ (а) и при наличии прерываний (б). Если работу 1 выполнять в два этапа на двух процессорах, то общее вредя выполнения работ сокращается до 1,5 единиц по сравнению с 2 единицами времени, когда прерывания запрещены.

3.3. О методах решения задач планирования работ в детерминированных ВСРВ

Методы решения оптимизационных задач, минимизирующие время выполнения всех работ, берут своё начало от алгоритмов анализа сетей [13]. Основным понятием здесь является критический путь, т. е. ориентированный путь от начальной работы до заключительной, имеющий наибольшую протяжённость. Так, если для сети на рис. 11 предположить, что все работы имеют одинаковую длительность, существуют два критических пути: один из них включает последовательность вершин 1, 2, 5, 6, 8, 9, 10, другой — 1, 3, 5, 6, 8, 9, 10. Ясно, что вся сеть работ не может быть выполнена за время меньшее, чем длина критического пути. Так или иначе многие методы решения задачи на быстродействие сводятся к нахождению таких расписаний, при которых составляющие критический путь работы выполняются за наименьшее время. В дальнейшем развитие этих методов привело к появлению новой математической дисциплины — теории расписаний [14]. В рамках этой теории классы задач, сводящихся к поиску расписаний, значительно расширяются.

Основными критериями оптимальности расписаний для детерминированных вычислительных процессов считаются следующие (см. [15], где можно найти дальнейшие ссылки):

1) минимизация времени выполнения всех работ;

2) минимизация количества требуемых процессоров;

3) минимизация среднего времени окончания выполнения работ;

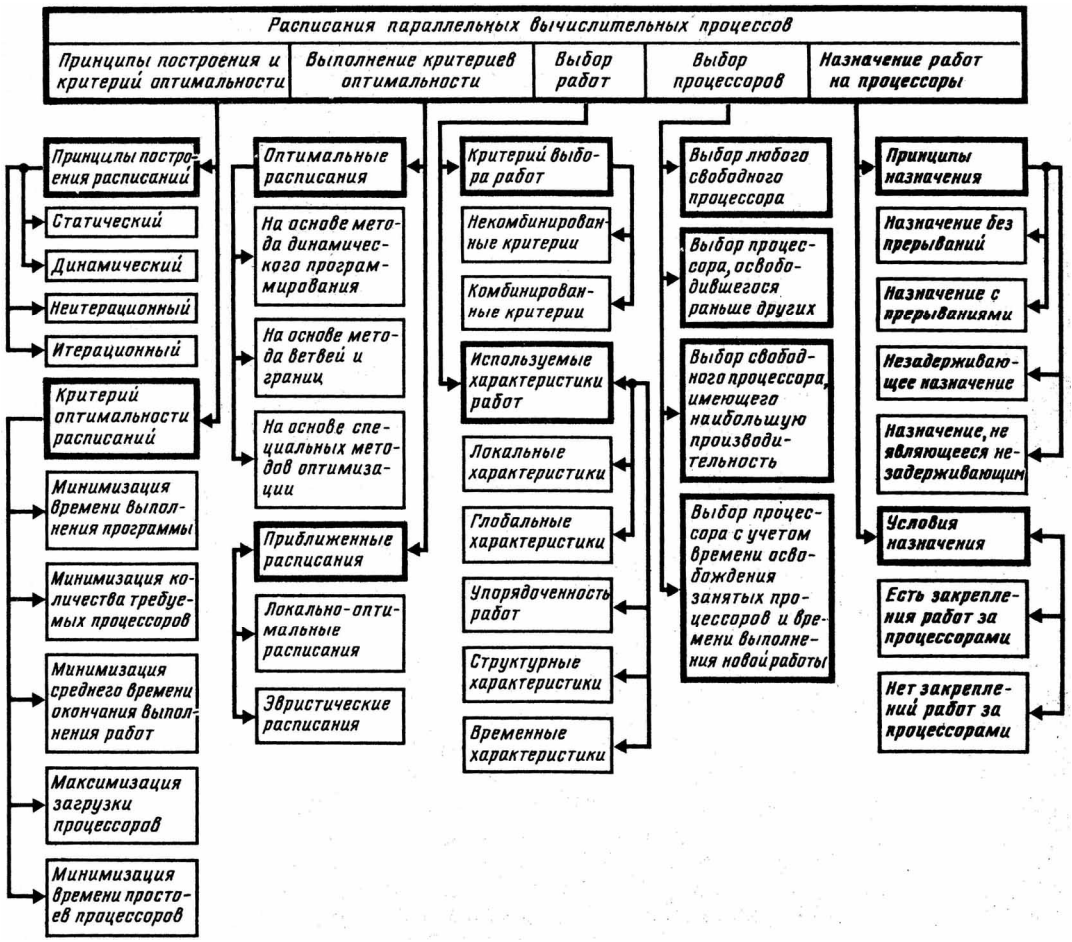
4) максимизация загрузки процессоров;

5) минимизация времени простоев процессоров.

При проектировании программного обеспечения детерминированных ВСРВ последние два критерия используются сравнительно редко, их место занимают определённые выше задачи с директивными сроками.

Большинство рассматриваемых в теории расписаний задач оказываются *NP*-полными [14]. Для решения нескольких тысяч задач из этого класса в настоящее время существуют лишь алгоритмы, осуществляющие перебор вариантов, число которых экспоненциально зависит от размера описания задачи. Вместе с тем ни для одной из этих задач не доказано, что для её решения невозможен полиномиальный алгоритм. Более того, если полиномиальный алгоритм можно построить хотя бы для одной *NP*-полной задачи, то это будет означать, что полиномиальные алгоритмы существуют для всех задач из этого класса. Правда, степени полиномов, оценивающих сложность решения задачи этими алгоритмами, окажутся различными для разных задач и в принципе могут быть как угодно большими. Поэтому для практики проектирования детерминированных ВСРВ большое значение имеет разработка эффективных способов построения приближённых решений задач теории расписаний.

Даже краткий обзор методов решения задач теории расписаний и связанных с ними вопросов теории сложности дискретных задач уведёт нас далеко за



рамки возможностей данной публикации. Поэтому мы ограничимся таблицей (рис. 13), достаточно полно отображающей классификацию применяемых рас писаний. В этой таблице расписания для параллельных вычислительных процессов различаются по принципам их построения, критериям оптимальности и признаку точного или приближённого выполнения критерия, по способам выбора и назначения процессоров для выполнения работ [15]. Эти характеристики дают возможность хотя бы частично судить о многообразии и сложности алгоритмов построения расписаний и о возможности их конструирования.

Проблематика теории расписаний составляет обширный, весьма интересный и постоянно развивающийся раздел современной дискретной математики. Читателям, заинтересованным в серьёзном ознакомлении с этой областью, рекомендуются уже цитированные выше работы [12–15]. Кроме того, читателей издания «Новое в жизни, науке, технике» (серия «Математика, кибернетика») в 1988 г. ожидает знакомство с публикацией В. С. Танаева, посвящённой алгоритмам построения расписаний.

1.4. Аномалии многопроцессорных приоритетных расписаний

Теперь вспомним, что для построения расписания выполнения сети программ реального времени в качестве исходной информации требуется задание времён выполнения отдельных работ, составляющих сеть.

Определение времени выполнения программ или системных функций является самостоятельной и не простой задачей. Трудности, возникающие при измерении или оценке времени выполнения программного модуля процессором вызываются, в главном, двумя обстоятельствами. Во-первых, это зависимость времени выполнения модуля от исходных энных, во-вторых — зависимость этого времени от конфигурации и состояния аппаратных и операционных средств СРВ. По этим причинам во многих случаях приходится ограничиваться приближёнными оценками времени выполнения, использовать при реализации программы реального времени расписания, рассчитанные на наихудший случай.

Заметим, что оценка потребности программы в других ресурсах ВСРВ; как правило, менее затруднительна и может быть получена непосредственно при составлении модуля либо с использованием различных приёмов трассировки программ. Тем не менее при оценке потребности программы в других ресурсах отклонения от истинных значений вполне вероятны хотя бы из-за наличия условных переходов в программе. В зависимости от исходных данных и других обстоятельств некоторые части программы могут вообще не выполняться при конкретном прогоне.

Эти отклонения от результатов измерений программ приводят иногда к такому поведению расписаний, которое на первый взгляд кажется парадоксальным. Особенно часто такие явления возникают в случае использования так называемых приоритетных (или списочных) расписании для многопроцессорных детерминированных ВСРВ. Такие расписания полностью определяются предварительным упорядочиванием множества работ *N*. Этот порядок задаётся списком приоритетов. Как только при исполнении программы реального времени процессор заканчивает выполнение некоторой рабе ты, он сразу начинает выполнять готовую к исполнению работу, имеющую наибольший приоритет. Готовыми считаются работы, для которых завершено выполнение всех предшественников и свободно требуемое количество других ресурсов. Работы выполняются без прерываний. Считается, что просмотр списка приоритетов производится мгновенно.

В качестве первого примера рассмотрим систему обслуживания $S = (\{m_1, m_2, m_3\}, \{1, 2, 3, 4, 5, 6, 7, 8, 9), G, T, \varnothing, F)$. Процессоры идентичны. Граф предшествования G для этой системы изображён на рис. 14, а. Там же возле номера каждой работы проставлено время её выполнения. Символ F обозначает критерий оптимальности расписания — минимум времени завершения всей совокупности работ {1, 2, 3, ..., 9}. На рис. 14, б представлено оптимальное расписание, соответствующее списку приоритетов $L = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. На рисунках 15–18 показано, как меняется расписание

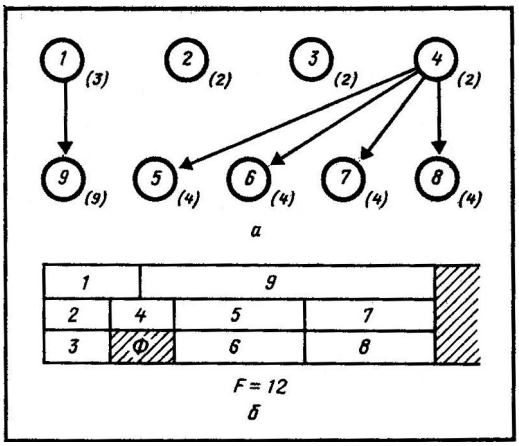


Рис. 14. Сеть работ (а) и оптимальное расписание (б); m=3, L=(1, 2, 3, 4, 5, 6, 7, 8, 9)

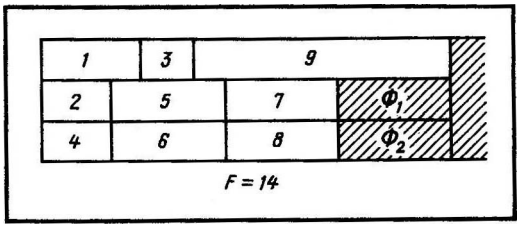


Рис. 15. Изменение списка приоритетов. Здесь L^1=(1, 2, 3, 4, 5, 6, 7, 8, 9)

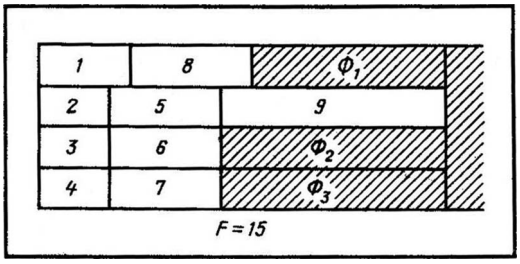


Рис. 16. Эффект увеличения числа процессоров: m заменено на $m' = 4$

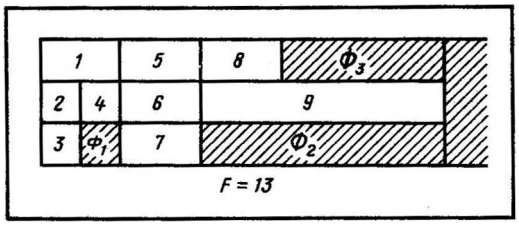


Рис. 17 Эффект уменьшения времени выполнения

при изменении L, M, T и O. Во всех случаях время завершения всей программы увеличивается, что, по-видимому, не соответствует интуитивным ожиданиям.

Может показаться, что увеличение F происходит из-за неудачного выбора' списка приоритетов. Пример на рис. 19 показывает, что подобные аномалии могут быть

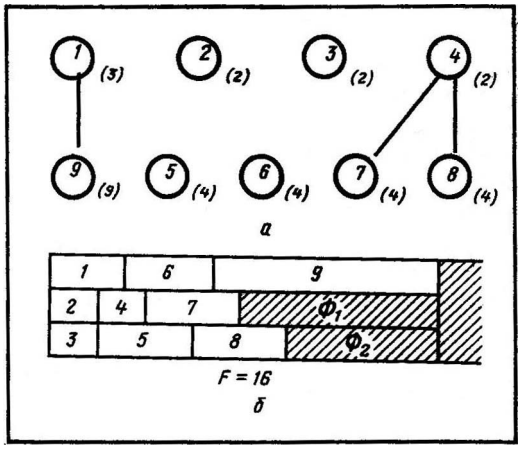


Рис. 18. Эффект ослабления ограничений предшествования: граф G заменяется на граф G': m=3, L=(1, 2, 3, 4, 5, 6, 7, 8, 9)

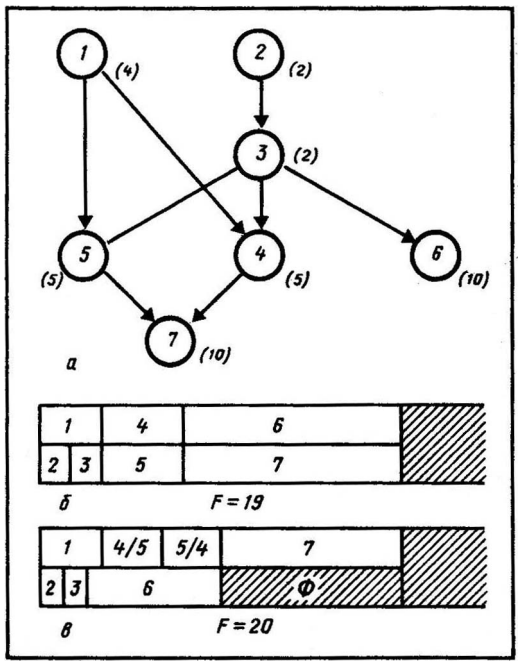


Рис. 19. Сеть работ (а); оптимальное расписание работ: (б); t_i заменены на $t'_j = t_i - 1$ (в)

свойством, присущим самой сети работ. Для модели двухпроцессорной системы обслуживания, представленной на этом рисунке, уменьшение на единицу времени выполнения каждой работы приводит к увеличению общего времени по крайней мере на единицу (рис. 19, в) по сравнению с оптимальным расписанием (рис. 19, б) при любой последовательности приоритетов. Чтобы избежать увеличения времени выполнения, в этом случае необходимо запланировать простой второго процессора после выполнения работ 2 и 3. Приоритетные расписания не допускают таких действий, Тем не менее этот тип расписаний часто используется при реализации ВСРВ благодаря своей простоте. Аномальные изменения длины расписаний, как показывает следующий результат, по счастью, ограниченны.

Рассмотрим две системы обслуживания $S_1 = (M, N, G, T, \varnothing, F)$ и $S_2 = (M', N', G', T', \varnothing, F')$. Процессоры идентичны, и для каждой системы построено приоритетное расписание. Обозначим через m число процессоров в первой системе, m' — во второй. Предположим также, что граф предшествований G' есть часть графа G, т. е. ограничения на порядок следования работ во второй системе слабее, чем в первой. Пусть каждая компонента вектора T' не превосходит соответствующую компоненту вектора T — во второй системе каждая работа выполняется быстрее, чем в первой. Пусть τ и τ' — времена завершения всех работ N согласно первому и второму расписаниям соответственно. При сделанных предположениях $\tau'/\tau \le 1+(m'-1)$. Эта формула показывает, что аномальное увеличение времени выполнения сети работ из-за изменения параметров системы обслуживания не может быть более чем в два раза.

На рис. 20 приведён пример системы с m идентичными процессорами, где изменение времён выполнения работ происходит согласно формуле

$$t_i' = \begin{cases} t_i - \varepsilon & \text{для} \quad 1 \le i \le m-1, \\ i_1 & \text{в противном случае} \end{cases}$$

(ε — достаточно малое положительное число). Сравнивая оптимальное расписание для первой системы (рис. 20, б) с расписанием для второй при том же списке приоритетов, находим соотношение времён завершений

$$\frac{\tau'}{\tau} = \frac{2m-1+\varepsilon}{m+2} \longrightarrow 2-\frac{1}{m} \quad \text{при} \quad \varepsilon \longrightarrow 0.$$

Этот пример показывает, что аномальное двукратное увеличение времени счёта при изменении параметров системы асимптотически (при большом числе процессоров) достижимо.

Дальнейшие примеры того же типа читатель найдёт в [12], откуда позаимствованы привёденные в этом разделе формулы и описания аномальных эффектов в приоритетных расписаниях для многопроцессорных систем.

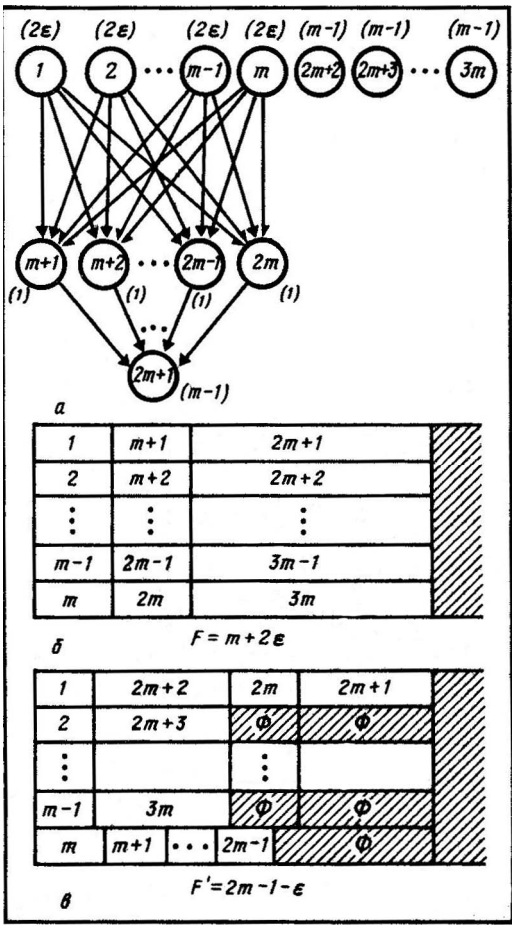


Рис. 20. Сеть работ (а); оптимальное расписание (б); увеличение общего времени выполнения работ при уменьшении длительности отдельных работ на ε (в)

3.5. Динамическое распределение памяти в детерминированных ВСРВ

Наше знакомство с задачами составления расписаний для детерминированных ВСРВ показывает их чрезвычайную сложность, возможность неожиданных эффектов. Особенно трудными являются задачи, в которых распределению между работами согласно расписанию подлежат вместе с процессорами другие ресурсы — оперативная память, устройства, различные информационные ресурсы, например трансляторы, сервисные программы, блоки данных.

В этих условиях возрастает роль приближённых алгоритмов построения расписаний. Разнообразие подходов к конструированию приближённых алгоритмов столь же велико, как и количество типов задач теории расписаний.

В этом разделе мы рассмотрим один из способов приближённого решения задач теории расписаний для системы $S = (M, N, G, T, R, F)$, где множество R дополнительных ресурсов включает в себя оперативную память, состоящую из отдельных страниц, количество которых мы обозначим C, и одного или нескольких каналов, позволяющих пересылать содержимое оперативной памяти во внешнюю память и обратно. Время пересылки одной страницы считаем постоянным и обозначим символом τ.

Сущность предлагаемого приближённого подхода заключена в следующем. Сначала решается задача нахождения оптимального (или допустимого, в зависимости от вида критерия F) расписания для рассматриваемой системы без учёта дополнительных ресурсов. В результате решения этой задачи находятся интервалы времени, в которые выполняются отдельные работы. Следовательно, в течение этих интервалов программы, описывающие работы, должны находиться в оперативной памяти. Затем решается проблема динамического распределения памяти, которая, в свою очередь, может быть изложена в следующем виде. Имеется набор программ и данных, одновременно не помещающихся в оперативной памяти, часть их хранится во внешней памяти. Заданы интервалы времени, в которые каждая программа должна находиться в оперативной памяти. Возникает задача подкачки каналами запрашиваемых программ, т. е. задача динамического распределения оперативной памяти, решение которой заключается в построении расписания, удовлетворяющего заданным директивным срокам нахождения программ в памяти и учитывающего пропускную способность каналов.

Таким образом, осуществляется декомпозиция исходной задачи на две более простые. Если в результате рассмотрения задачи динамического распределения памяти не удаётся найти допустимое расписание, то приходится возвращаться к построению допустимого расписания для распределения работ по процессорам в надежде, что для нового варианта такого расписания найдётся подходящее расписание использования дополнительных ресурсов — каналов и оперативной памяти.

Задача динамического распределения оперативной памяти при имеющемся расписании центрального процессора с учётом ограниченной пропускной способности каналов подробно исследована в [16], где показано, что, если допустимое расписание работы каналов и распределения памяти имеется, то оно может быть построено следующим образом. Первые C страниц из последовательности запросов на память загружаются в оперативную память заблаговременно, до наступления момента использования первой работающей в реальном времени программы. Вспомним, что расписание выполнения всех работ уже имеется. Повторные загрузки одной и той же страницы на этом этапе не производятся. Затем рассматривается следующий запрос. Если он относится к странице, которая уже находится в оперативной памяти, то алгоритм переходит к обработке последующего запроса. Если же страницы, которая требуется в очередном рассматриваемом запросе, в оперативной памяти нет, то производится её загрузка на место находящейся в памяти такой страницы, обращение к которой в последовательности запросов произойдёт в будущем позже других загруженных страниц. При этом работа канала по выполнению этой загрузки планируется как можно раньше. Если устанавливается, что нет свободного канала, который может начать загрузку не позднее,

чем за время τ до начала использования требуемой страницы, значит, допустимого расписания не существует. Такой же вывод следует сделать и в ситуации, когда канал свободен, но ещё не окончилось время использования уже загруженных в память *C* страниц. Корректность этого алгоритма доказана, численные эксперименты подтвердили его эффективность.

ГЛАВА 4. РЕАЛИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ВСРВ

4.1. Языки программирования в реальном времени

Предыдущее рассмотрение убеждает нас в особой трудности составления программы реального времени, необходимости повышенных требований к надёжности такого программирования и наличии специфических проблем, не имеющих аналогов при составлении обычных программ, не используемых в режиме реального времени. С другой стороны, потребность в ВСРВ становится всё ощутимей.

Естественный выход из этого положения — автоматизация самого процесса проектирования и реализации ВСРВ. Как уже говорилось в первой главе, применительно к программированию автоматизация означает прежде всего создание удобного и выразительного языка, позволяющего составлять программы лаконичные, ясные и простые для пользователя, обрабатывая и выполняя которые, ЭВМ все сложные проблемы, связанные с наличием синхронизации процессов, обнаружением ошибок, тупиковых ситуаций, решит без участия человека, автоматически (или почти автоматически). Все необходимые для этого средства создаются при реализации компилятора для реализуемого языка программирования и являются универсальными сразу для всего класса однотипных задач. Конечно, из-за универсальности может происходить некоторая потеря эффективности выполняемых программ. Но, как показывает опыт использования языков программирования высокого уровня, снижение эффективности с лихвой окупается повышением производительности труда программистов и повышением надёжности программ.

Перечень требований, к языку программирования в реальном времени включает в себя как требования, обычно предъявляемые к языкам программирования, так и ряд специфических требований.

К первой группе требований относятся следующие возможности [4]:

1) развитая система типов употребляемых в языке объектов, возможности определения программистом своих собственных типов и операций над ними. Эти возможности прежде всего повышают надёжность программирования, поскольку чёткая типизация всех объектов позволяет осуществить большое число проверок правильности употребления переменных;

2) наличие удобных структур управления выполнением операторов, таких, как **if** (если), **case** (вариант), **while** (пока), **for** (для);

3) блочная структура программы, позволяющая группировать операторы в блоки с локально определёнными переменными;

4) возможность использования процедур, функций, подпрограмм;

5) обеспечение модульности программирования, т. е. возможности разбиения сложной программы на простые части. Модуль должен иметь средства для указания объектов, экспортируемых из модуля и импортируемых в модуль. Модули могут быть вложенными в другие модули или подпрограммы.

Сформулируем кратко дополнительные требования, которым должен удовлетворять входной язык ВСРВ.

Прежде всего язык должен содержать возможность организации параллельно выполняемых процессов. Поскольку выполнение операторов программы зависит от времени, язык программирования должен предоставлять пользователю средства для задания директивных сроков начала и (или) окончания различных вычислений. Эти сроки можно задавать в виде явных моментов времени либо указывая соответствующие события (например, поступление очередного блока данных на входные регистры). Кроме того, входной язык должен давать возможность описывать как обстоятельства синхронизации вычислительных процессов с событиями вне

ВСРВ и друг с другом, так и условия смены режимов обработки входной информации.

В настоящее время в полной мере всем перечисленным требованиям удовлетворяет, пожалуй, лишь язык программирования Ада [4]. Однако этот язык в силу своей универсальности чрезвычайно труден для реализации и освоения. Поэтому естественно наличие других языков для программирования в реальном времени, в которых за счёт отказа от выполнения части формулированных требований удаётся достигнуть упрощения и сделать их удобными для программирования отдельных типов ВСРВ.

В качестве примера рассмотрим входной Язык ЯРВ-1 системы автоматизированного программирования в реальном времени, предложенной в Вычислительном центре АН СССР [17]. Система предназначена для автоматизированной генерации объектных программ в детерминированных ВСРВ, реализуемых на ЭВМ серии ЕС. Каждая такая программа получает на вход приходящие извне кадры входной информации (быть может, сгруппированные в блоки), осуществляет их обработку и посылает результаты отображения на различные устройства отображения. Выполнение программы реального времени (программы РВ) происходит под управлением специального монитора — управляющей программы реального времени (УПРВ).

УПРВ является программной надстройкой операционной системы используемой ЭВМ и в описываемой здесь конкретной системе существенно использует супервизор реального времени ЕС ЭВМ, специально предназначенный для управления параллельными вычислительными процессами. Если ВСРВ базируется на отличных от ЕС ЭВМ аппаратных средствах, то возможно, что она будет заметно отличаться от предлагаемого варианта, хотя главные её функции вряд ли изменятся.

Язык ЯРВ-1, опуская несущественные для дальнейшего детали, можно описать следующим образом.

Основным элементарным объектом языка является модуль — обычный для многих языков программирования оператор процедуры. Все процедуры, участвующие в образовании модулей, составляются заранее и помещаются в системную библиотеку процедур вместе с необходимыми спецификациями формальных параметров. Фактические параметры модуля могут быть либо РВ-параметрами, либо константами, простыми переменными, идентификаторами с индексами и т. д. РВ-параметры имеют следующий вид:

<РВ-параметр> → (<имя РВ-параметра>; <поставщик>; <время>).

Компонента (поставщик) однозначно указывает модуль программы, в котором вычисляется запрашиваемое модулем-потребителем значение параметра с данным именем. Как будет видно из дальнейшего описания языка, модули могут повторяться в программе, входить в более сложные объекты языка. Поэтому эта компонента, в свою очередь, состоит из нескольких полей.

Компонента (время) определяет момент времени, в который необходимо взять требуемое значение параметра. В этом разделе может быть указано некоторое показание таймера ЭВМ, и тогда потребителю будет передано последнее имеющееся к этому моменту значение параметра. Кроме того, в системе предусмотрены другие способы задания времени, основанные на фиксации моментов выполнения указанных программистом циклических процессов, например, поступления входного блока кадров или неоднократно повторяющегося выполнения определённой цепочки модулей.

Другие виды фактических параметров здесь не обсуждаются, поскольку они аналогичны параметрам процедур известных языков программирования.

Из модулей может быть составлен следующий по иерархии сложности объект языка — цикл реального времени:

<цикл РВ> → (<имя цикла РВ>, <период>, <фаза>, <тело цикла РВ>).

Компонента (период) с помощью одного из указанных выше способов отсчёта времени задаёт интервал, через который будет повторяться выполнение заданных модулей, составляющих тело цикла РВ. Допускается вместо указания периода задание абсолютных значений моментов

времени (в том числе не обязательно образующих арифметическую прогрессию), в которые будет выполняться тело цикла. Тело цикла РВ это список модулей, РВ-параметры которых могут подставляться из блоков входной информации, из модулей других циклов РВ и из модулей данного цикла.

Поименованная совокупность циклов РВ образует безусловное задание (простое задание):

<безусловное задание> → **begin** <имя задания>; <список циклов РВ> **end**.

Все циклы реального времени, входящие в одно простое задание, должны рассматриваться как выполняющиеся параллельно во времени. Если модули некоторого цикла требуют на свой вход данные, поставляемые другим циклом, то необходимая задержка организуется системой.

В ходе обработки данных может потребоваться изменение режима обработки. Такая возможность предусматривается в языке реального времени введением условного задания:

<условное задание> **begin** <имя задания>; <параметры>; <кадр>; <блок>; <список условий>; <таблица условного задания> **end**

В разделе (параметры) перечисляются имена всех переменных, используемых в данном условном задании. Затем предложение <кадр> задаёт те имена, которые связаны с параметрами, приходящими от объекта. При необходимости выполнения некоторых контрольных действий кадровые параметры могут сопровождаться описанием формата. Метапеременная <блок> — это целое число, указывающее, сколько кадров содержится в одном входном блоке. Затем <список условий> перечисляет параметры, являющиеся элементарными условиями, т. е. переменными, принимающими лишь два значения — 0 или 1. Каждый такой параметр сообщает о выполнении или невыполнении некоторого отношения над значениями других параметров. Предполагается, что библиотека процедур системы содержит процедуры, обращения к которым с соответствующими фактическими параметрами образуют модули, вычисляющие значения условий.

Описанные таким образом условия образуют вектор условий. Каждая его компонента является элементарным условием. Таблица условного задания ставит в соответствие каждому значению вектора условий имя некоторого условного или безусловного задания, описанного в данной программе РВ. Каждый раз, когда УПРВ получает управление, текущий вектор условий сравнивается со значениями левой части таблицы условий и в зависимости от результатов этого сравнения либо продолжается текущая работа, либо система приступает к выполнению нового условного или простого задания, имя которого указано в соответствующей строке таблицы.

Наконец, программа реального времени представляет собой совокупность условных и простых заданий, все таблицы условий которой не содержат ссылок на неописанные задания:

<программа РВ> → <имя программы>; <задания>.

4.2. Предварительная обработка программ реального времени для детерминированных ВСРВ

Предварительная обработка программы реального времени, составленной на входном языке, решает следующие задачи:

а) измерение (оценка) времени выполнения каждого модуля процессором;

б) количественная оценка других ресурсов (таких, как память, устройства и т. п.), необходимых для выполнения модуля;

в) объединение модулей в группы, каждая из которых представляет единицу, управляемую УПРВ (в частном случае такая группа может состоять из единственного модуля);

г) составление допустимого расписания работы сформированных программных единиц;

д) генерация объектного кода для всех программных единиц.

Объединение модулей в группы диктуется обычно необходимостью использования при составлении УПРВ штатных операционных средств ЭВМ, таких, например, как операционные системы или супервизоры реального времени. Подробное описание этой части предварительной обработки программы реального времени, вызванное использованием в

УПРВ средств супервизора реального времени ЕС ЭВМ, можно найти в [17]. Здесь мы лишь отметим, что программной единицей, управляемой супервизором реального времени ЕС ЭВМ, является процесс — объект более высокого уровня по сравнению с модулем, подпрограммой или процедурой: Использование процессов в качестве объектов управления УПРВ позволяет избежать при её реализации трудоёмкого и детального программирования средств низкого уровня управления программными вычислениями, таких, как инициаторы, терминаторы, семафоры, уже реализованные в супервизоре реального времени ЕС ЭВМ.

Автоматизированный расчёт расписаний для функционирования ВСРВ осуществляется с помощью математических моделей таких систем. Эти модели рассмотрены в предыдущей главе. Особенностью задач построения расписаний при проектировании реальных ВСРВ является необходимость одновременного рассмотрения в одной системе обслуживания как приборов, допускающих прерывания (например, центральный процессор ЭВМ), так и приборов, работающих без прерываний (каналы ввода- вывода). В такой постановке задача на выполнение директивных сроков рассмотрена в [18].

Методы генерации объектного кода при компиляции программ реального времени незначительно отличаются от методов и приёмов при реализации других языков программирования высокого уровня.

В заключение раздела приведём схему предварительной обработки программ реального времени (рис. 21), заметив, что в случае отсутствия приемлемого расписания из третьего блока возможен возврат во второй — новый вариант компоновки управляемых программных единиц из модулей может оказаться удачнее.

4.3. Управляющая программа системы реального времени

Блок-схема управляющей программы реального времени (УПРВ) и схема её взаимодействия с другими частями ВСРВ приведены на рис. 22. Там же показано распределение оперативной памяти ВСРВ при выполнении программы РВ

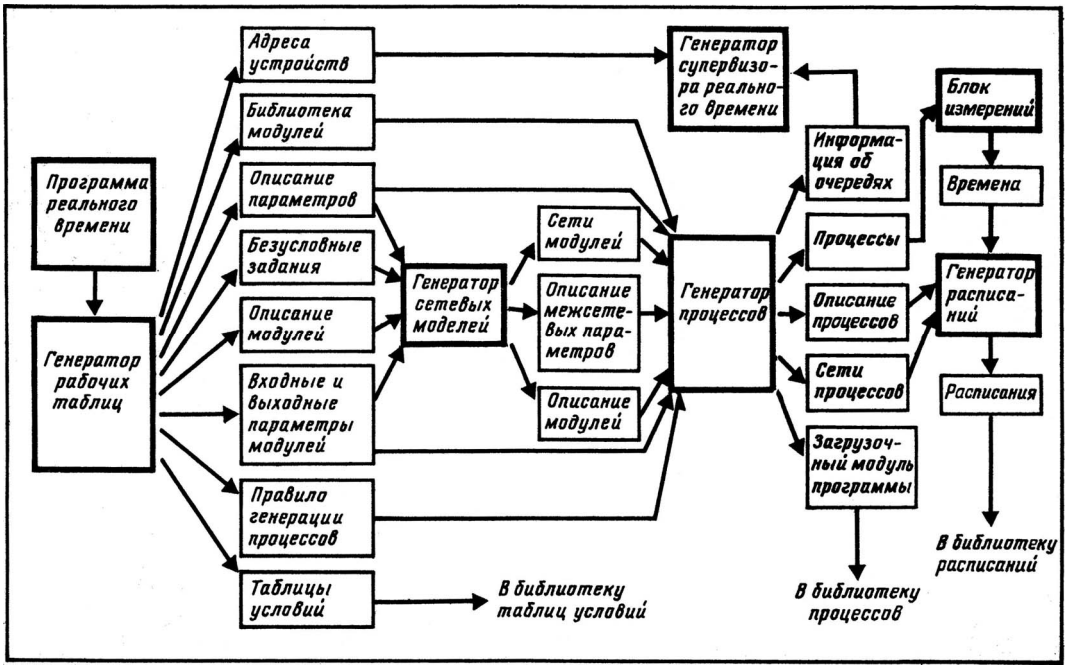


Рис. 21. Схема предварительной обработки программы реального времени

Основные функции УПРВ следующие:

а) связь с оператором;

б) приём информации от контролируемого объекта и передача её для обработки программе РВ;

в) активизация управляемых программных единиц программы РВ согласно расписанию;

г) выделение необходимых ресурсов активным программным единицам;

д) переключение заданий в зависимости от условий.

Как уже отмечалось, УПРВ по сравнению с системой предварительной обработки программ реального времени в большей степени зависит от аппаратуры и используемых штатных операционных средств ВСРВ. Однако структура УПРВ и функции составляющих её блоков могут быть описаны независимо от машинной реализации.

УПРВ состоит из трёх блоков: блока связи с контролируемым объектом, интерпретатора команд, ведущего диалог УПРВ с операторской консолью, и диспетчера, контролирующего выполнение расписания.

УПРВ получает управление в тех случаях, когда поступает очередной блок входных данных от объекта, оканчивается работа одной из управляемых программных единиц или вводится команда с консоли оператора.

Интерпретатор команд находится в ожидании команд с консоли оператора. Он осуществляет ввод этих команд, выполнение и выдачу соответствующих сообщении на консоль.

Блок связи с объектом целесообразно реализовать как обработчик прерывания, возникающего в момент сеанса связи с объектом. Его функцией является приём данных от объекта и передача их в области, доступные программе реального времени.

Задачей диспетчера является активизация согласно расписанию необходимой управляемой единицы программы РВ. Эту функцию диспетчер осуществляет каждый раз, когда УПРВ получает управление. Он же производит смену выполняемых заданий при изменении условий, выделяет ресурсы, необходимые для выполнения этого задания, и распределяет их между отдельными программными единицами задания.

Типичные действия, осуществляемые как интерпретатором команд, так и диспетчером, заключаются в загрузке активизируемых заданий в память, их активизации, остановке, удалении из памяти. Разница заключается в том, что диспетчер осуществляет эти функции в моменты времени, предписанные расписанием, а интерпретатор — подчиняясь командам с консоли оператора. Наличие этих двух управляющих процессов в УПРВ позволяет с её помощью выполнять программы как в режиме жёсткого реального времени, так и в интерактивном режиме, в диалоге с оператором.

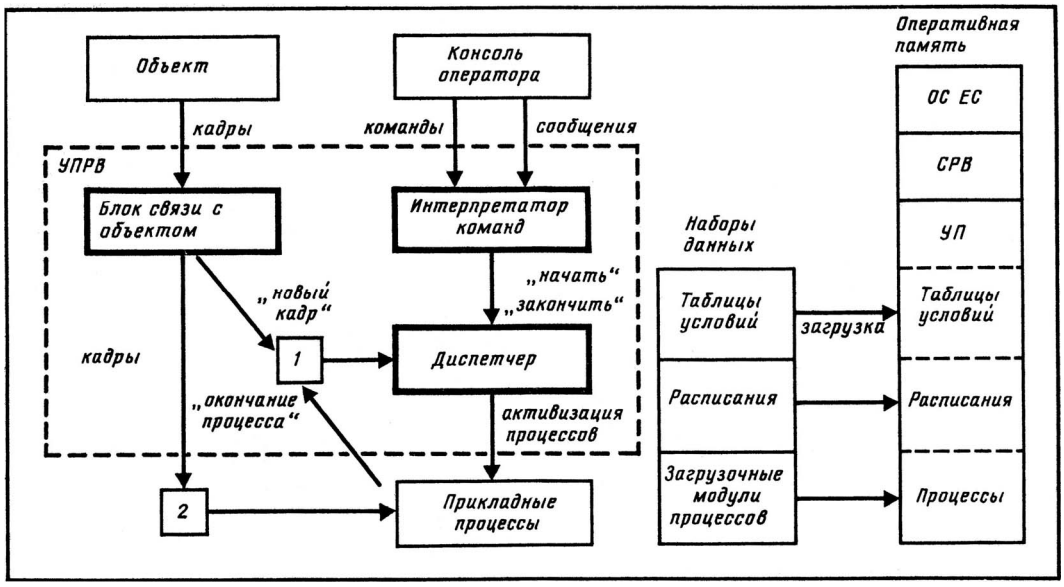


Рис. 22. Схема управляющей программы детерминированной ВСРВ

Реализация УПРВ для ЕС ЭВМ с использованием средств супервизора реального времени ЕС ЭВМ и ОС ЕС ЭВМ описана в [17].

4.4. Контроль данных в ВСРВ

В конечном счёте автоматизация программирования ВСРВ призвана сделать их максимально надёжными, исключив прежде всего такие сложные специфические ошибки, как отсутствие правильной синхронизации, возникновение тупиков, запаздывание результатов расчётов по отношению к установленным директивным срокам. Именно на это направлена предварительная обработка программы РВ, изложенная в этой главе.

Но под контролем УПРВ будут работать цепочки объединённых в процессы модулей, которые тоже могут содержать ошибки (горький программистский фольклор утверждает, что в каждой программе есть хотя бы одна ошибка). Их последствия могут вызвать самые непредсказуемые результаты. Известен, например, случай срыва важного космического эксперимента по наведению установленного на спутнике зеркала на луч лазера, расположенного на горной вершине, из-за того что в программе для ЭВМ, управляющей эволюциями спутника, высота вершины была задана не в футах, как предполагалось, а в милях. Подобных примеров можно привести много. Сосредоточим своё внимание на способах обнаружения ошибок и устранения их последствий.

Мы будем говорить о контроле данных в ВСРВ, понимая под ним не только проверку соответствия данных их спецификациям, которые обычно задают форму представления этих данных в ЭВМ, а в более развитых языках программирования и другие характеристики, такие, например, как диапазон значений. Более глубокий и содержательный контроль — это контроль выполнения свойств объектов предметной области, а не спецификаций данных прикладных программ. Объектами могут быть, например, двух- или трёхмерная матрица давления, вектор скорости, плотность потока и т. д., причём исчерпывающие характеристики свойств объекта может давать специалист предметной области, а не прикладной программист, реализующий алгоритм обработки данных.

Итак, основная задача контроля может быть решена созданием единых средств контроля, использующих все имеющиеся сведения об объекте и ориентированных на работу специалистов предметной области. Сведения о физических параметрах объектов могут представляться в самой разнообразной форме, не всегда приемлемой для ввода в ЭВМ, Опыт использования автоматизированных систем контроля данных показывает, что достаточно общим видом контроля является проверка выполнения определённых логических (булевых) выражений над множествами значений параметров исследуемого физического объекта. Например, одно из таких соотношений может означать следующее: если высота полёта самолёта больше 8 км, то температура за бортом отрицательная. На формальном языке, введя соответствующие обозначения, в соответствие этому факту может быть поставлено логическое выражение:

$B = (H < 8) \rightarrow (T < 0)$.

Здесь $P_1 = (H < 8)$ — предикат, принимающий два значения — «истина» или «ложь», можно обозначить их для краткости 0 и 1. Выражение $P_2 = (T < 0)$ — второй предикат. Стрелка означает операцию импликации — если истинно P_1, то истинно P_2. Выражение B тоже является предикатом, принимающим значение «истина» или «ложь» в зависимости от P_1 и P_2 согласно таблице значений для операции «импликация» на рис. 23. Любое логическое выражение над несколькими предикатами может быть записано в виде так называемой дизъюнктивной нормальной формы (ДНФ), где используются только логические операции: — (отрицание), $\vee$ (логическое «или», дизъюнкция) и $\wedge$ (логическое «и», конъюнкция). Так выражение B в виде ДНФ принимает вид

ДНФ $(B) = \overline{P_1} \vee P_2$.

Таким образом, задача контроля данных практически во всех случаях программирования в реальном времени может быть сведена к вычислению значения «истина» или «ложь» соответствующих логических функций. Эти проверки могут осуществляться специальными модулями программы РВ, включёнными в циклы реального времени, уже имеющиеся или специально для этих модулей составленные. Напомним, что аналогичным образом осуществляется вычисление компонент вектора условий, управляющего выполнением условного задания программы РВ, Следовательно, для программирования реакции ВСРВ на обнаруженную в процессе контроля ошибку могут быть использованы средства языка ЯРВ-1.

В связи с необходимостью неоднократного вычисления логических функций в режиме реального времени возникает задача отыскания такого представления функции, когда время вычисления её минимально. Среди классических методов преобразования булевых функций наиболее распространён и изучен метод построения и преобразования дизъюнктивных нормальных форм.

Минимизируя представления ДНФ, мы освобождаемся от избыточности, сохраняй логическую эквивалентность исходной и преобразованной форм. ДНФ — очень удобны, так как они допускают использование линейных алгоритмов для их вычисления.

p_1	p_2	$p_1 \rightarrow p_2$
0	0	1
0	1	1
1	0	0
1	1	1

Рис. 23. Таблица истинности для импликации

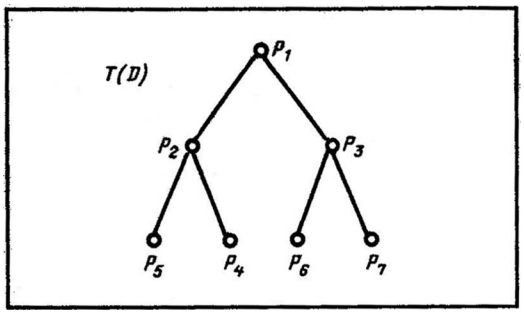


Рис. 24. Дерево проверок, эквивалентное ДНФ

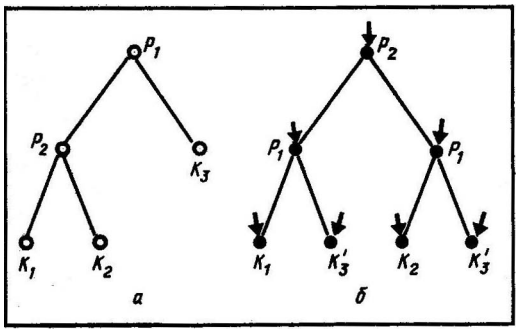


Рис. 25. Применение преобразования вращения относительно вершин P_1 и P_2 к дереву проверок ДНФИ: а) сокращённое изображение исходного дерева, K_1, K_2, K_3 — поддеревья, эквивалентные фрагментам ДНФ D; б) результат вращения, K'_3 не зависит от P_2

Кратко работу линейного алгоритма можно описать как вычисление значения конъюнкции до первого нулевого значения одного из её термов и переход к следующей конъюнкции, если значение предыдущей не равно единице. Алгоритм выдаёт значение «единица», если нашлась конъюнкция, принимающая на проверяемом наборе предикатов значение «единица».

В качестве примера рассмотрим ДНФ

$$D = P_4\overline{P}\ P_2 \vee \overline{P}_2 P_5 \overline{P}_1 \vee P_6 P_1 \overline{P}_3 \vee P_1 P_3 P_7.$$

Для краткости знак конъюнкции ∧ опущен.

На наборе {1, 0, 1, 1, 0, 0, 1} линейный алгоритм вычислит все семь предикатов.

В работе [19] предлагается представлять булевую функцию, указывающую принадлежность значений контролируемых параметров заданным областям во множестве их значений, с помощью двоичных деревьев. При этом вершинами дерева являются предикаты, а дуги задают переходы, так что выполняется переход по левой дуге, если предикат равен нулю, и по правой, если предикат равен единице. В качестве примера приведём дерево проверок $T(D)$ для ДНФ (рис. 24).

Этот пример наводит на мысль о возможности построения дерева проверок по произвольной ДНФ и даже по произвольной булевой функции для сокращения времени её вычисления. Оказывается, что по любой булевой функции можно построить не одно, а целое множество деревьев, реализующих данную булевую функцию. Деревья эти могут различаться, высотой и количеством вершин, оставаясь логически эквивалентными.

Введём два преобразования, позволяющих построить класс эквивалентных деревьев (Ts) по данной булевой функции. Представители класса Ts обладают различными характеристиками, в частности разной высотой. Преобразование S определяется рекурсивным алгоритмом и строит один элемент класса Ts:

$S(F) — T(F)$, $T \in Ts$.

Для преобразования деревьев из Ts вводится вращение $R(T, P_1, P_2)$, которое определено для двух соседних вершин дерева P_1 и P_2, лежащих на одной ветви. После его применения две соседние вершины двоичного дерева меняются местами. Однако преобразование R не является локальным из-за требования сохранения логической эквивалентности деревьев из Ts. Меняется вид других фрагментов дерева. Приведём результат однократного вращения дерева $T(D)$ относительно вершин P_1 и P_2 (K_1, K_2, K_3 — поддеревья, эквивалентные фрагментам ДНФ) (рис. 25).

Доказано, что система преобразований $\{S, R\}$ полна, т. е. она позволяет построить все деревья из Ts и только их. Следовательно, для каждого представителя Ts, полученного применением S, можно указать последовательность R_1, ..., R_k такую, что дерево $T(F)$ сведётся к T^* оптимальному дереву после преобразования

$R_1(R_2(\ldots (R_k (S(F)) \ldots \ldots))) = T^*$.

Алгоритмы нахождения T^* по своей природе переборные, однако размер перебора можно значительно уменьшить. Для этого используются эвристические методы выбора очередной вершины в рекурсивном алгоритме S построения начального дерева $T(F)$.

ЗАКЛЮЧЕНИЕ

Использование ЭВМ — один из решающих факторов научно-технического прогресса. Трудно представить развитие науки без точных расчётов изучаемых явлений с помощью всё совершенствующихся математических моделей. Планирование экономики, учёт, оперативное управление народным хозяйством, повышение качества проектирования новых образцов техники, информационно-справочные службы — всё это примеры областей нашей деятельности, где ЭВМ подтвердили свою эффективность. Благодаря их применению человек создаёт всё более совершенные машины, использует новые источники энергии, разумно осваивает территорию планеты и космическое пространство. Управление процессами в современной технологии, контроль скоротечных явлений, где сложному анализу подлежит в условиях дефицита времени большое число факторов,— также область, где ЭВМ незаменимы.

Если демонстрацией лишь части трудных проблем, возникающих при создании вычислительных систем реального времени для управления процессами, и способов их решения удалось вселить в читателя надежду, что и здесь ЭВМ — надёжный помощник, то автор считает, что достиг своей цели.

ЛИТЕРАТУРА

1. *Мартин Дж.* Программирование для вычислительных систем реального времени – М.: Наука, 1975. – 359 с.

2. *Цикритзис Д., Бернстайн Ф.* Операционные системы. – М.: Мир, 1977. – С. 336.

3. *Horning J., Rendell R.* Process Structuring. – ACM Comput. Surveys, 5, – № 1. P. 5–30.

4. *Янг С.* Алгоритмические языки реального времени. Конструирование и разработка. – М.: Мир, 1985. – С. 400.

5. *Дейкстра Е.* Взаимодействие последовательных процессов // Языки программирования / Под ред. *Женюи Ф.* – М.: Мир, 1972, – С. 9–86.

6. *Dijkstra E.W.* Solution to a Problem in Concurrent Programming Control. – CACM, 1965, 8, № 9. P. 569.

7. *Knuth D.* Additional Comments on a Problem in Concurrent .Programming Control. – CACM, 1966, 9. – № 5. P. 321–322.

8. *Dijkstra E.W.* Hierarchical Ordering of Sequential Processes. – Acta Informat., 1 – P. 115–138.

9. *Hoare C.A.R., Perrot R.H.* Operating Systems Techniques. – N.Y.: Academic Press, 1972.

10. *Hoare C.A.R.* Communicating Sequential Processes. – CACM, 1976, 21. №8 – P. 666–677.

11. *Hansen P.B.* Distributed Processes: Concurrent Programming Concept. – CACM, 1978, 21. – № 11. – P. 934-941.

12. Теория расписаний и вычислительные системы / Под ред. *Коффмана Э.Г.* – М.: Наука, 1984. – С. 334.

13. *Филипс Д., Гарсиа-Диас А.* Методы анализа сетей./ Пер. с англ. под ред. *Сушкова Б. Г.* – М.: Мир, 1984. – С. 496.

14. *Танаев В.С., Гордон В.С., Шафранский Я.М.* Теория расписаний. Одностадийные системы. – М.: Наука, 1984. – С. 381.

15. *Головкин Б.А.* Расчёт характеристик и планирование параллельных вычислительных процессов. — М.: Радио и связь, 1983. – С. 272.

16. *Логинова И.В., Сушков Б.Г.* Динамическое распределение памяти в системах реального времени при имеющемся расписании использования центрального процессора // Теория и реализация систем реального времени – М.: Изд. ВЦ АН СССР, 1984. – С. 49–69.

17. Теория и реализация систем реального времени. – М.: Изд. ВЦ АН СССР, 1984. – С. 104.

18. *Буланже Д.Ю., Сушков Б.Г.* Алгоритмы управления вычислительными системами жёсткого реального времени. // Изв. АН СССР Техн. кибернетика, 1982. – № 6.

19. *Самыловский А.И., Сушков Б.Г.* Построение характеристической функции невыпуклого множества и связанная с этим задача квадратичного целочисленного программирования // ЖВМ и МФ, 1978. – Т. 18. – № 2. – С. 322–337.

Научно-популярное издание

СУШКОВ Борис Григорьевич

ЭВМ УПРАВЛЯЕТ ЭКСПЕРИМЕНТОМ

(Вычислительные системы реального времени)

Гл. отраслевой редактор Л. А. Ерлыкин
Редактор Г. Г. Карвовский
Мл. редактор Н. А. Васильева
Художник Л. П. Ромасенко
Худож. редактор М. А. Бабичева
Техн. редактор И. Е. Жаворонкова
Корректор Л. С. Соколова
ИБ № 8669

Сдано в набор 20.07.87. Подписано к печати 17.08.87
Т–00643. Формат бумаги 70×100$^{1}/_{16}$. Бумага кн. журнальная. Гарнитура литературная. Печать офсетная. Усл. печ. л. 2,60. Усл. кр. отт. 5,52. Уч.-изд. л. 3,10. Тираж 36 120 экз. Заказ 1996. Цена 11 коп.
Издательство «Знание». 101835, ГСП Москва, Центр, проезд Серова, д. 4. Индекс заказа 874309.
Ордена Трудового Красного Знамени Чеховский полиграфический комбинат ВО «Союзполиграфпром» Государственного комитета СССР по делам издательств, полиграфии и книжной торговли.
142300, г. Чехов Московской области