СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Сергей Сорокин

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Уверенностью можно сказать, что ссылки на красивое словосочетание «реальное время» стали общим местом на различных семинарах, конференциях и в специализированной печати. С неменьшей уверенностью можно сказать, что смысл этого термина трактуется специалистами по-разному в зависимости от области их профессиональных интересов, от того, являются они теоретиками или практиками, и даже просто от личного опыта и круга общения.

В этой статье мы сконцентрируемся на рассмотрении данного вопроса применительно к цифровой вычислительной технике, используемой в системах управления и сбора данных. Основное внимание будет уделено программному обеспечению, так как оно является наиболее слабым звеном в системах реального времени. Многопроцессорные системы для простоты рассматриваться не будут. Статья не претендует на исчерпывающее изложение предмета и является скорее заметками на тему основных понятий и терминологии в этой области.

ТАК ЧТО ЖЕ ТАКОЕ РЕАЛЬНОЕ ВРЕМЯ

Если попытаться дать короткое определение, то

1. Система называется системой реального времени, если правильность ее функционирования зависит не только от логической корректности вычислений, но и от времени, за которое эти вычисления производятся. То есть для событий, происходящих в такой системе, то, КОГДА эти события происходят, так же важно, как логическая корректность самих событий.

2. Говорят, что система работает в реальном времени, если ее быстродействие адекватно скорости протекания физических процессов на объектах контроля или управления. Так как окружающий нас мир весьма многообразен, здесь уместно добавить, что имеются в виду именно те процессы, которые непосредственно связаны с функциями, выполняемыми конкретной системой реального времени. То есть система управления должна собрать данные, произвести их обработку в соответствии с заданными алгоритмами и выдать управляющие воздействия за такой промежуток времени, который обеспечивает успешное решение поставленных перед системой задач.

Из приведенных определений следует несколько интересных выводов.

Во-первых, практически все системы промышленной автоматизации являются системами реального времени.

Во-вторых, принадлежность системы к классу систем реального времени никак не связана с ее быстродействием. Например, если ваша система предназначена для контроля уровня грунтовых вод, то даже выполняя измерения с периодичностью один раз за полчаса, она будет работать в реальном времени.

Исходные требования к времени реакции системы и другим временны́м параметрам определяются или техническим заданием на систему, или просто логикой ее функционирования. Например, шахматная программа, думающая над каждым ходом более года, работает явно не в реальном времени, так как шахматист скорее всего не доживет до конца партии. Однако точное определение «приемлемого времени реакции» не всегда является простой задачей, а в системах, где одним из звеньев служит человек, подвержено влиянию субъективных факторов. Впрочем, человек – это своеобразная вычислительная машина, а мы договорились многопроцессорных конфигураций не рассматривать.

Интуитивно понятно, что быстродействие системы реального времени должно быть тем больше, чем больше скорость протекания процессов на объекте контроля и управления. Чтобы оценить необходимое быстродействие для систем, имеющих дело со стационарными процессами, часто используют теорему Котельникова, из которой следует, что частота дискретизации сигналов должна быть как минимум в 2 раза выше граничной частоты их спектра.

При работе с широкополосными по своей природе переходными процессами (транзиент-анализ) часто применяют быстродействующие АЦП с буферной памятью, куда с необходимой скоростью записывается реализация сигнала, которая затем анализируется и/или регистрируется вычислительной системой. При этом требуется закончить всю необходимую обработку до следующего переходного процесса, иначе информация будет потеряна. Подобные системы иногда называют системами квази-реального времени.

Принято различать системы «жесткого» и «мягкого» реального времени. Читатель наверное догадался, что эти различия не связаны с органолептическими свойствами систем. Тогда что же это такое?

1. Системой «жесткого» реального времени называется система, где неспособность обеспечить реакцию на какие-либо события в заданное время является отказом и ведет к невозможности решения поставленной задачи.

Последствия таких отказов могут быть разные, от пролива драгоценной влаги на линии по розливу алкогольных напитков до более крупных неприятностей, если, например, вовремя не сработала система аварийных блокировок атомного реактора.

Многие теоретики ставят здесь точку, из чего следует, что время реакции в «жестких» системах может составлять и секунды, и часы, и недели. Однако большинство практиков считают, что время реакции в системах «жесткого» реального времени должно быть все-таки минимальным. Идя на поводу у практиков, так и будем считать. Разумеется, однозначного мнения о том, какое время реакции свойственно «жестким» системам, нет. Более того, с увеличением быстродействия микропроцессоров это время имеет тенденцию к уменьшению, и если раньше в качестве границы называлось значение 1 мс, то сейчас, как правило, называется время порядка 100 мкс.

2. Точного определения для «мягкого» реального времени не существует, поэтому будем считать, что сюда относятся все системы реального времени, не попадающие в категорию «жестких».

Так как система «мягкого» реального времени может не успевать ВСЁ делать

ВСЕГДА в заданное время, возникает проблема определения критериев успешности (нормальности) ее функционирования. Вопрос этот совсем не простой, так как в зависимости от функций системы это может быть максимальная задержка в выполнении каких-либо операций, средняя своевременность отработки событий и т. п. Более того, эти критерии влияют на то, какой алгоритм планирования задач является оптимальным. Вообще говоря, системы «мягкого» реального времени проработаны теоретически далеко не до конца.

ЯДРА И ОПЕРАЦИОННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Чтобы быстрее перейти к делу, примем как очевидные следующие моменты.

1. Когда-то операционных систем совсем не было.
2. Через некоторое время после их появления возникло направление ОС РВ.
3. Все ОС РВ являются многозадачными операционными системами. Задачи делят между собой ресурсы вычислительной системы, в том числе и процессорное время.

Четкой границы между ядром (Kernel) и операционной системой нет. Различают их, как правило, по набору функциональных возможностей. Ядра предоставляют пользователю такие базовые функции, как планирование и синхронизация задач, межзадачная коммуникация, управление памятью и т. п. Операционные системы в дополнение к этому имеют файловую систему, сетевую поддержку, интерфейс с оператором и другие средства высокого уровня.

По своей внутренней архитектуре ОС РВ можно условно разделить на монолитные ОС, ОС на основе микроядра и объектно-ориентированные ОС. Графически различия в этих подходах иллюстрируются рисунками 1, 2, 3. Преимущества и недостатки различных архитектур достаточно очевидны, поэтому подробно мы на них останавливаться не будем.

Пользователь, напуганный перспективой изучать новую операционную систему, может здесь вполне резонно спросить: «А нельзя ли вообще обойтись без всей этой заумной канители?»

Если отвечать на этот вопрос односложно, то да, МОЖНО. Однако ответ на вопрос о том, когда это НУЖНО делать, остается, конечно, за читателем. Материалы во врезке к статье, возможно, дадут некоторую пищу к размышлениям на эту тему.

Задачи, процессы, потоки

Существуют различные определения термина «задача» для многозадачной ОС РВ. Мы будем считать задачей набор операций (машинных инструкций), предназначенный для выполнения логически законченной функции системы. При этом задача конкурирует с другими задачами за получение контроля над ресурсами вычислительной системы.

Принято различать две разновидности задач: процессы и потоки. Процесс представляет собой отдельный загружаемый программный модуль (файл), который, как правило, во время исполнения имеет в памяти свои независимые области для кода и данных. В отличие от этого потоки могут пользоваться общими участками кода и данных в рамках единого программного модуля.

Хорошим примером многопоточной программы является редактор текста WORD, где в рамках одного

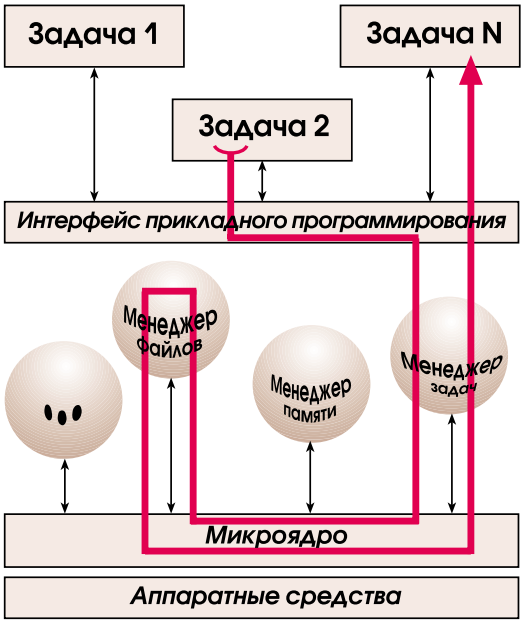

Рис. 2. ОС РВ на основе микроядра

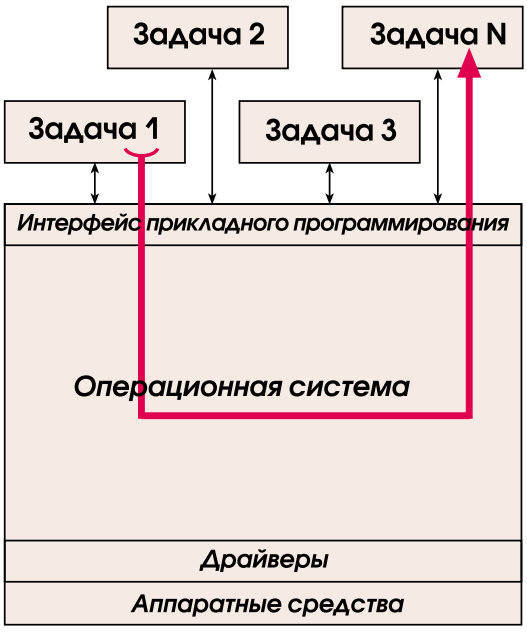

Рис. 1. ОС РВ с монолитной архитектурой

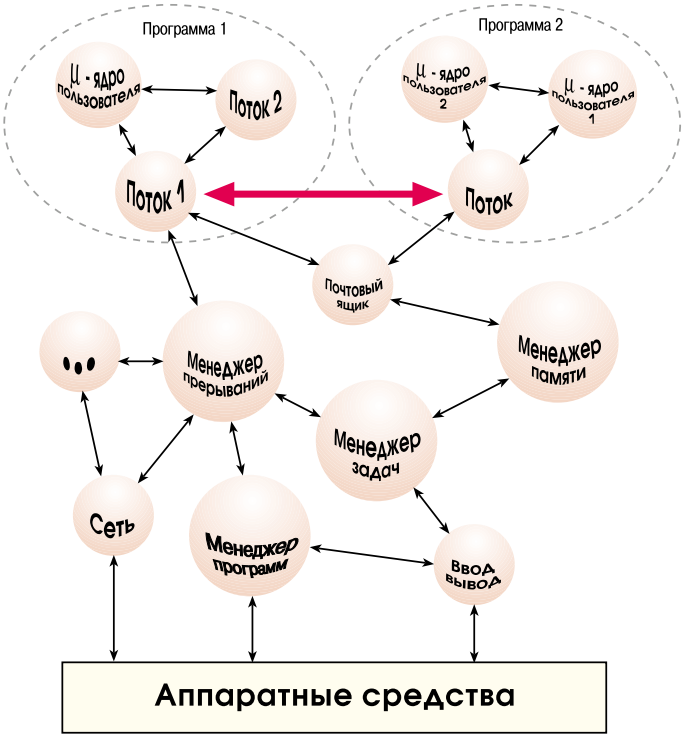

Рис. 3. Объектно-ориентированная ОС РВ

приложения может одновременно происходить и набор текста, и проверка правописания.

Преимущества потоков

1. Так как множество потоков способно размещаться внутри одного EXE-модуля, это позволяет экономить ресурсы как внешней, так и внутренней памяти.
2. Использование потоками общей области памяти позволяет эффективно организовать межзадачный обмен сообщениями (достаточно передать указатель на сообщение). Процессы не имеют общей области памяти, поэтому ОС должна либо целиком скопировать сообщение из области памяти одной задачи в область памяти другой (что для больших сообщений весьма накладно), либо предусмотреть специальные механизмы, которые позволили бы одной задаче получить доступ к сообщению из области памяти другой задачи.
3. Как правило, контекст потоков меньше, чем контекст процессов, а значит, время переключения между задачами-потоками меньше, чем между задачами-процессами.
4. Так как все потоки, а иногда и само ядро РВ размещаются в одном EXE-модуле, значительно упрощается использование программ-отладчиков (debugger).

Недостатки потоков

1. Как правило, потоки не могут быть подгружены динамически. Чтобы добавить новый поток, необходимо провести соответствующие изменения в исходных текстах и перекомпилировать приложение. Процессы, в отличие от потоков, подгружаемы, что позволяет динамически изменять функции системы в процессе ее работы. Кроме того, так как процессам соответствуют отдельные программные модули, они могут быть разработаны различными компаниями, чем достигается дополнительная гибкость и возможность использования ранее наработанного ПО.
2. То, что потоки имеют доступ к областям данных друг друга, может привести к ситуации, когда некорректно работающий поток способен испортить данные другого потока. В отличие от этого процессы защищены от взаимного влияния, а попытка записи в «не свою» память приводит, как правило, к возникновению специального прерывания по обработке «исключительных ситуаций».

Реализация механизмов управления процессами и потоками, возможность их взаимного сосуществования и взаимодействия определяются конкретным ПО РВ.

ОСНОВНЫЕ СВОЙСТВА ЗАДАЧ

Как правило, вся важная, с точки зрения операционной системы, информация о задаче хранится в унифицированной структуре данных - управляющем блоке (Task Control Block, TCB). В блоке хранятся такие параметры, как имя и номер задачи, верхняя и нижняя границы стека, ссылка на очередь сообщений, статус задачи, приоритет и т. п.

Приоритет – это некое целое число, присваиваемое задаче и характеризующее ее важность по сравнению с другими задачами, выполняемыми в системе. Приоритет используется в основном планировщиком задач для определения того, какая из готовых к работе задач должна получить управление. Различают системы с динамической и статической приоритетностью. В первом случае приоритет задач может меняться в процессе исполнения, в то время как во втором приоритет задач жестко задается на этапе разработки или во время начального конфигурирования системы.

Контекст задачи – это набор данных, содержащий всю необходимую информацию для возобновления выполнения задачи с того места, где она была ранее прервана. Часто контекст хранится в TCB и включает в себя такие данные, как счетчик команд, указатель стека, регистры CPU и FPU и т. п. Планировщик задач в случае необходимости сохраняет контекст текущей активной задачи и восстанавливает контекст задачи, назначенной к исполнению. Такое переключение контекстов и является, по сути, основным механизмом ОС РВ при переходе от выполнения одной задачи к выполнению другой.

Состояние (статус) задачи. С точки зрения операционной системы, задача может находиться в нескольких состояниях. Число и название этих состояний различаются от одной ОС к другой. По-видимому, наибольшее число состояний задачи определено в языке Ada. Тем не менее практически в любой ОС РВ загруженная на выполнение задача может находиться, по крайней мере, в трех состояниях.

1. Активная задача – это задача, выполняемая системой в текущей момент времени.
2. Готовая задача – это задача, готовая к выполнению и ожидающая у планировщика своей «очереди».
3. Блокированная задача – это задача, выполнение которой приостановлено до наступления определенных событий. Такими событиями могут быть освобождение необходимого задаче ресурса, поступление ожидаемого сообщения, завершение интервала ожидания и т. п.

Пустая задача (Idle Task) – это задача, запускаемая самой операционной системой в момент инициализации и выполняемая только тогда, когда в системе нет других готовых для выполнения задач. Пустая задача запускается с самым низким приоритетом и, как правило, представляет собой бесконечный цикл «ничего не делать». Наличие пустой задачи предоставляет операционной системе удобный механизм отработки ситуаций, когда нет ни одной готовой к выполнению задачи.

Многократный запуск задач. Как правило, многозадачные ОС позволяют запускать несколько копий одной и той же задачи. При этом для каждой такой копии создается свой TCB и выделяется своя область памяти. В целях экономии памяти может быть предусмотрено совместное использование одного и того же исполняемого кода для всех запущенных копий. В этом случае программа должна обеспечивать повторную входимость (реентерабельность). Кроме того, программа не должна использовать временные файлы с фиксированными именами и должна корректно осуществлять доступ к глобальным ресурсам.

Реентерабельность (повторная входимость) означает возможность без негативных последствий временно прервать выполнение какой-либо функции или подпрограммы, а затем вызвать эту функцию или подпрограмму снова. Частным проявлением реентерабельности является рекурсия, когда тело подпрограммы содержит вызов самой себя. Классическим примером нереентерабельной системы является DOS, а типичной причиной нереентерабельности служит использование глобальных переменных. Предположим, что у нас есть функция, реализующая низкоуровневую запись на диск, и пусть она использует глобальную переменную write_sector, которая устанавливается в соответствии с параметром, передаваемым этой функции при вызове. Предположим теперь, что Задача А вызывает эту функцию с параметром 3, то есть хочет записать данные в сектор номер 3. Допустим, что когда переменная write_sector уже равна 3, но сама запись еще не произведена, выполнение Задачи А прерывается и начинает выполняться Задача В, которая вызывает ту же функцию, но с аргументом 10. После того как запись в сектор номер 10 будет произведена, управление рано или поздно вернется к Задаче А, которая продолжит работу с того же места. Однако, так как переменная write_sector имеет теперь значение 10, данные Задачи А, предназначавшиеся для сектора номер 3, будут вместо этого записаны в

сектор номер 10. Из приведенного примера видно, что ошибки, связанные с нереентерабельностью, трудно обнаружить, а последствия они могут вызвать самые катастрофические.

ПЛАНИРОВАНИЕ ЗАДАЧ

Важной частью любой ОС РВ является планировщик задач. Несмотря на то, что в разных источниках он может называться по-разному (диспетчер задач, супервизор и т. п.), его функции остаются теми же: определить, какая из задач должна выполняться в системе в каждый конкретный момент времени. Самым простым методом планирования, не требующим никакого специального ПО и планировщика как такового, является использование **циклического алгоритма** в стиле round robin:

```
void main (void)
{
        for (;;) {
                task0();
                task1();
                task2();
                /* и т. д. */
        }
}
```

Каждая «задача», представляющая собой отдельную подпрограмму, выполняется циклически. При этом надо придерживаться следующих правил:

1. Подпрограммы не должны содержать циклов ожидания в стиле
```
while (TRUE) {
        if (switch_up()) {
                lamp_off();
                break;
                }
        }
```
2. Подпрограммы должны выполнять свою работу как можно быстрее, чтобы дать возможность работать следующей подпрограмме.
3. При необходимости подпрограмма может сохранять свое окружение и текущие результаты, чтобы в следующем цикле возобновить работу с того же места.

Можно отметить следующие преимущества циклического алгоритма.

1. Простота использования и прозрачность для понимания.
2. Если исключить из рассмотрения прерывания, система полностью детерминирована. Задачи всегда вызываются в одной и той же последовательности, что позволяет достаточно просто произвести анализ «наихудшего случая» и вычислить максимальную задержку.
3. Минимальные размеры кода и данных. Кроме того, в отличие от алгоритмов с вытеснением, для всех задач необходим только один стек.
4. Отсутствуют ошибки, обусловленные «гонками».

К недостаткам циклического алгоритма можно отнести отсутствие приоритетности и очередей. К тому же задачи вызываются независимо от того, должны ли они в данный момент что-либо делать или нет, а на прикладного программиста ложится максимальная ответственность за работоспособность системы.

Перейдем теперь к другому широко используемому алгоритму планирования. Речь пойдет о режиме **разделения времени**. Существуют различные реализации в рамках этого алгоритма, и некоторые западные специалисты даже различают такие в общем-то идентичные для нас понятия, как time-slicing и time-sharing. Как правило, алгоритм реализуется следующим образом: каждой задаче отводится определенное количество квантов времени (обычно кратно 1 мс), в течение которых задача может монопольно занимать процессорное время. После того как заданный интервал времени истекает, управление передается следующей готовой к выполнению задаче, имеющей наивысший приоритет. Та, в свою очередь, выполняется в течение отведенного для нее промежутка времени, после чего все повторяется в стиле round robin. Легко заметить, что такой алгоритм работы может привести к определенным проблемам. Представим, что в системе работают 7 задач, 3 из которых имеют высокий приоритет, а 4 – низкий. Низкоприоритетные задачи могут никогда не получить управление, так как три высокоприоритетные задачи будут делить все процессорное время между собой. Единственную возможность для низкоприоритетных задач получить управление предоставляет ситуация, когда все высокоприоритетные задачи находятся в блокированном состоянии.

Для решения этой проблемы применяется прием, получивший название **равнодоступность** (fairness). При этом реализуется принцип адаптивной приоритетности, когда приоритет задачи, которая выполняется слишком долго, постепенно уменьшается, позволяя менее приоритетным задачам получить свою долю процессорного времени. Равнодоступность применяется главным образом в многопользовательских системах и редко применяется в системах реального времени.

Кооперативная многозадачность – это еще один алгоритм переключения задач, с которым широкие массы компьютерной общественности знакомы по операционной системе Windows 3.x. Задача, получившая управление, выполняется до тех пор, пока она сама по своей инициативе не передаст управление другой задаче. По сути это продолжение идеологии round robin, и нет нужды объяснять, почему алгоритм кооперативной многозадачности в чистом виде мало применяется в системах реального времени.

Приоритетная многозадачность с вытеснением – это, по-видимому, наиболее часто используемый в ОС РВ принцип планирования. Основная идея состоит в том, что высокоприоритетная задача, как только для нее появляется работа, немедленно прерывает (вытесняет) низкоприоритетную. Другими словами, если какая-либо задача переходит в состояние готовности, она немедленно получает управление, если текущая активная задача имеет более низкий приоритет. Такое «вытеснение» происходит, например, когда высокоприоритетная задача получила ожидаемое сообщение, освободился запрошенный ею ресурс, произошло связанное с ней внешнее событие, исчерпался заданный интервал времени и т. п.

Заканчивая рассмотрение основных принципов планирования задач, необходимо отметить, что тема эта далеко не исчерпана. Диапазон систем реального времени весьма широк, начиная от полностью статических систем, где все задачи и их приоритеты заранее определены, до динамических систем, где набор выполняемых задач, их приоритеты и даже алгоритмы планирования могут меняться в процессе функционирования. Существуют, например, системы, где каждая отдельная задача может участвовать в любом из трех алгоритмов планирования или их комбинации (вытеснение, разделение времени, кооперативность).

В общем случае алгоритмы планирования должны соответствовать критериям оптимальности функционирования системы. Однако, если для систем «жесткого» реального времени такой критерий очевиден: «ВСЕГДА и ВСЁ делать вовремя», то для систем «мягкого» реального времени это может быть, например, минимальное «максимальное запаздывание» или средневзвешенная своевременность завершения операций. В зависимости от критериев оптимальности могут применяться алгоритмы планирования задач, отличные от рассмотренных. Например, может оказаться, что планировщик должен анализировать момент выдачи критичных по времени управляющих воздействий и запускать на выполнение ту задачу, которая отвечает за ближайшие из них (алго-

ритм earliest deadline first, EDF).

Необходимо отметить, что в одной вычислительной системе могут одновременно сосуществовать задачи и «жесткого», и «мягкого» реального времени, и что только одна из этих задач, обладающая наивысшим приоритетом, может быть по-настоящему детерминированной.

Не стоит особо увлекаться приоритетами. Если система нормально работает, когда все задачи имеют одинаковый приоритет, то и слава Богу. Если нет, то можно присвоить высокий приоритет «критической» задаче, и низкий приоритет всем остальным. Если у вас больше одной «критической» задачи, при недостаточном быстродействии системы имеет смысл рассмотреть многопроцессорную конфигурацию или, отказавшись от ПО РВ, перейти к простому циклическому алгоритму.

Как правило, разработчики стараются свести свою систему реального времени к наиболее простым конфигурациям, характерным для систем «жесткого» реального времени, иногда даже в ущерб эффективности использования вычислительных ресурсов. Причина понятна: сложные динамические системы весьма трудно анализировать и отлаживать, поэтому лучше заплатить за более мощный процессор, чем иметь в будущем проблемы из-за непредвиденного поведения системы. В связи с этим большинство существующих систем реального времени представляют собой статические системы с фиксированными приоритетами. Часто в системе реализуется несколько «режимов» работы, каждый из которых имеет свой набор выполняемых задач с заранее заданными приоритетами. Значительная часть особо ответственных систем по-прежнему реализуется без применения коммерческих ОС РВ вообще.

СИНХРОНИЗАЦИЯ ЗАДАЧ

Хотя каждая задача в системе, как правило, выполняет какую-либо отдельную функцию, часто возникает необходимость в согласованности (синхронизации) действий, выполняемых различными задачами. Такая синхронизация необходима, в основном, в следующих случаях.

1. Функции, выполняемые различными задачами, связаны друг с другом. Например, если одна задача подготавливает исходные данные для другой, то последняя не выполняется до тех пор, пока не получит от первой задачи соответствующего сообщения. Одна из вариаций в этом случае – это когда задача при определенных условиях порождает одну или несколько новых задач.
2. Необходимо упорядочить доступ нескольких задач к разделяемому ресурсу.
3. Необходима синхронизация задачи с внешними событиями. Как правило, для этого используется механизм прерываний, с которым читатель, безусловно, знаком.
4. Необходима синхронизация задачи по времени. Диапазон различных вариантов в этом случае достаточно широк, от привязки момента выдачи какого-либо воздействия к точному астрономическому времени до простой задержки выполнения задачи на определенный интервал времени. Для решения этих вопросов в конечном счете используются специальные аппаратные средства, называемые таймером.

Давайте рассмотрим все четыре случая более подробно.

Связанные задачи

Взаимное согласование задач с помощью сообщений является одним из важнейших принципов операционных систем реального времени. Способы реализации межзадачного обмена отличаются большим разнообразием, что не в последнюю очередь приводит к обилию терминов в этой области. Можно встретить такие понятия, как сообщение (message), почтовый ящик (mail box), сигнал (signal), событие (event), прокси (proxy) и т. п. Если, читая описание какой-либо ОС РВ, вы встретите уже знакомое название, не спешите делать выводы. Даже один и тот же термин может для разных ОС РВ обозначать разные вещи. Чтобы не запутаться, мы будем в дальнейшем называть сообщениями любой механизм явной передачи информации от одной задачи к другой (такие объекты, как семафоры, можно отнести к механизму неявной передачи сообщений).

Объем информации, передаваемой в сообщениях, может меняться от 1 бита до всей свободной емкости памяти вашей системы. Во многих ОС РВ компоненты операционной системы, так же как и пользовательские задачи, способны принимать и передавать сообщения. Сообщения могут быть асинхронными и синхронными. В первом случае доставка сообщений задаче производится после того, как она в плановом порядке получит управление, а во втором случае циркуляция сообщений оказывает непосредственное влияние на планирование задач. Например, задача, пославшая какое-либо сообщение, немедленно блокируется, если для продолжения работы ей необходимо дождаться ответа, или если низкоприоритетная задача шлет высокоприоритетной задаче сообщение, которого последняя ожидает, то высокоприоритетная задача, если, конечно, используется приоритетная многозадачность с вытеснением, немедленно получит управление.

Иногда сообщения передаются через отведенный для этого буфер определенного размера («почтовый ящик»). При этом, как правило, новое сообщение затирает старое, даже если последнее не было обработано.

Однако наиболее часто используется принцип, когда каждая задача имеет свою очередь сообщений, в конец которой ставится всякое вновь полученное сообщение. Стандартный принцип обработки очереди сообщений по принципу «первым вошел, первым вышел» (FIFO) не всегда оптимально соответствует поставленной задаче. В некоторых ОС РВ предусматривается такая возможность, когда сообщение от высокоприоритетной задачи обрабатывается в первую очередь (в этом случае говорят, что сообщение наследует приоритет пославшей его задачи).

Иногда полезным оказывается непосредственное управление приоритетом сообщений. Представим, что задача послала серверу (драйверу) принтера несколько сообщений, содержащих данные для печати. Если теперь задача хочет отменить всю печать, ей надо послать соответствующее сообщение с более высоким приоритетом, чтобы оно встало в очередь впереди всех посланных ранее заданий на печать.

Сообщение может содержать как сами данные, предназначенные для передачи, так и указатель на такие данные. В последнем случае обмен может производиться с помощью разделяемых областей памяти, разделяемых файлов и т. п.

Общие ресурсы

Трудно переоценить важность правильной организации взаимодействия различных задач при доступе к общим ресурсам. Хорошей аналогией может служить обед в многодетной крестьянской семье прошлого века. Едокам (задачам) не разрешалось одновременно лезть ложками в общую миску (ресурс). Нарушители порядка могли получить от отца семейства (супервизора) ложкой по лбу.

Ресурс – это общий термин, описывающий физическое устройство или область памяти, которые могут одновременно использоваться только одной задачей. Процессорное время тоже представляет собой своеобразный конкурентно используемый ресурс вычислительной системы. Примером физи-

ческих устройств могут служить клавиатура, дисплей, дисковый накопитель, принтер и т. п. Представим, например, что несколько задач пытаются одновременно выводить данные на принтер. На распечатке в результате ничего, кроме странной мешанины символов, мы не увидим. В качестве другого примера рассмотрим ситуацию, когда в бортовом компьютере мирно летящего самолета МИГ-29 среди прочих работают две задачи. Одна из них, взаимодействуя с радиолокационной системой, выдает удаление и направление до цели, а другая задача использует эти данные для пуска ракет класса «воздух-воздух». Не исключено, что первая задача, записав в глобальную структуру данных удаление до цели, будет прервана второй задачей, не успев записать туда направление до цели. В результате вторая задача считает из этой структуры ошибочные данные, что может привести к неудачному пуску со всеми вытекающими отсюда неприятными последствиями. Прервись первая задача чуть позже, и все было бы нормально. Упомянутые здесь проблемы обусловлены **времязависимыми ошибками** (time dependent error), или «гонками» и характерны для многозадачных ОС, применяющих алгоритмы планирования с вытеснением (кстати, системы с разделением времени также относятся к категории «вытесняющих»).

Приведенный пример показывает, что ошибки, обусловленные «гонками», а) характерны для работы с любыми ресурсами, доступ к которым имеют несколько задач, и б) происходят только в результате совпадения определенных условий, а потому с трудом обнаруживаются на этапе отладки.

Вот возможные пути решения проблемы.

1. Не использовать алгоритмы планирования задач с вытеснением. Это решение, правда, не всегда приемлемо.
2. Использовать специальный сервер ресурса, то есть задачу, ответственную за упорядочивание доступа к ресурсу. В этом случае запрос на изменение значения глобальных данных посылается этому серверу в виде сообщения. Аналогичный подход применим и для физических устройств. Так, например, задача может послать данные на печать в виде сообщения, направленного к серверу принтера.
3. Запретить прерывания на время доступа к разделяемым данным. Кардинальное решение, которое, впрочем, не приветствуется в системах реального времени.
4. Использовать для упорядочивания доступа к глобальным данным семафоры. Наиболее часто применяемое решение, которое, впрочем, может привести в некоторых случаях к «инверсии приоритетов».

Последний пункт стоит прокомментировать подробнее, поскольку понятие «семафор» встречается первый раз.

Семафор – это как раз то средство, которое часто используется для синхронизации доступа к ресурсам. В простейшем случае семафор представляет собой байтовую переменную, принимающую значение 0 или 1. Задача, перед тем как использовать ресурс, захватывает семафор, после чего остальные задачи, желающие использовать тот же ресурс, должны ждать, пока семафор (ресурс) освободится. Существуют также так называемые счетные семафоры, где семафор представляет собой счетчик. Пусть к системе подключено три принтера. Семафор, отвечающий за доступ к функциям печати, инициализируется со значением 3, а затем каждый раз, когда какая-либо задача запрашивает семафор для осуществления печати, его значение уменьшается на 1. После завершения печати задача освобождает семафор, в результате чего значение последнего увеличивается на 1. Если текущее значение семафора равно 0, то ресурс считается недоступным, и задачи, запрашивающие печать, должны ждать, пока не освободится хотя бы один принтер. Таким образом может производиться синхронизация доступа множества задач к группе из 3 принтеров. Так как по своей сути семафор также представляет собой глобальную переменную, все неприятности, которые упоминались ранее в связи с самолетом МИГ-29, по идее, должны поджидать нас и здесь. Однако, так как работа с семафорами происходит на уровне системных вызовов, программист может быть уверен, что разработчики операционной системы обо всем заранее позаботились.

Проникнувшись сознанием того, насколько опасно изменять глобальные переменные в условиях, когда все вокруг так и норовят друг друга вытеснить, читатель, наверно, не удивится, что участки кода программ, где происходит обращение к разделяемым ресурсам, называются **критическими секциями**.

Так как процессы обычно не имеют доступа к данным друг друга, а ресурсы физических устройств, как правило, управляются специальными задачами-серверами (драйверами), наиболее типична ситуация, когда «гонки» за доступ к глобальным переменным устраивают различные потоки, исполняемые в рамках одного программного модуля. Для того чтобы гарантировать, что критическая секция кода исполняется в каждый момент времени только одним потоком, используют механизм взаимоисключающего доступа, или попросту **мутексов** (Mutual Exclusion Locks, Mutex). Практически мутекс представляет собой разновидность семафора, который сигнализирует другим потокам, что критическая секция кода кем-то уже выполняется.

Критическая секция, использующая мутекс, должна иметь определенные суффиксную и префиксную части. Например:

```
int global_counter;
void main (void)
{
mutex_t mutex;
(
/* И все это лишь для того, чтобы увеличить глобальную переменную на единицу.*/
mutex_init (& mutex, USYNC, NULL);
mutex_lock (& mutex);
global_counter++;
mutex_unlock (& mutex);
}
```

Если мутекс захвачен, то поток, пытающийся войти в критическую секцию, блокируется. После того как мутекс освобождается, один из стоящих в очереди потоков (если таковые накопились) разблокируется и получает возможность доступа к глобальному ресурсу.

Думаю, на этом рассмотрение средств синхронизации доступа к общим ресурсам можно закончить, хотя, разумеется, множество тем осталось за скобками. Например, в WIN32 используется, в числе прочего, специальная разновидность мутексов под названием Critical Section Object. Необходимо также помнить, что, кроме ОС, имеющих WIN32 или POSIX API, существует большое число ни с чем не совместимых ОС, поэтому наличие средств синхронизации и особенности их реализации должны рассматриваться отдельно для каждой конкретной ОС РВ.

А вот возможные неприятности в борьбе за ресурсы.

Смертельный захват (Deadlock). В народе побочные проявления этой ситуации называются более прозаично – «зацикливание» или «зависание». А причина этого может быть достаточно проста – «задачи не поделили ресурсы». Пусть, например, Задача А захватила ресурс клавиатуры и ждет, когда освободится ресурс дисплея, а в это время Задача В также хочет пообщаться с пользователем и, успев захватить ресурс дисплея, ждет теперь, когда освободится клавиатура. Так и будут задачи ждать друг друга до второго потопа, а пользователь будет в это время смотреть на пустой экран и ругать последними словами яйцеголовых

программистов, которые не смогли сделать нормально работающую систему. В таких случаях рекомендуется придерживаться тактики «или все, или ничего». Другими словами, если задача не смогла получить все необходимые для дальнейшей работы ресурсы, она должна освободить всё, что уже захвачено, и, как говорится, «зайти через полчаса». Другим решением, которое уже упоминалось, является использование серверов ресурсов.

Инверсия приоритетов (Priority inversion). Как уже отмечалось, алгоритмы планирования задач (управление доступом к процессорному времени) должны находиться в соответствии с методами управления доступом к другим ресурсам, а всё вместе – соответствовать критериям оптимального функционирования системы. Эффект инверсии приоритетов является следствием нарушения гармонии в этой области. Ситуация здесь похожа на «смертельный захват», однако сюжет закручен еще более лихо. Представим, что у нас есть высокоприоритетная Задача А, среднеприоритетная Задача В и низкоприоритетная Задача С. Пусть в начальный момент времени Задачи А и В блокированы в ожидании какого-либо внешнего события. Допустим, получившая в результате этого управление Задача С захватила Семафор А, но не успела она его отдать, как была прервана Задачей А. В свою очередь, Задача А при попытке захватить Семафор А будет блокирована, так как этот семафор уже захвачен Задачей С. Если к этому времени Задача В находится в состоянии готовности, то управление после этого получит именно она, как имеющая более высокий, чем у Задачи С, приоритет. Теперь Задача В может занимать процессорное время, пока ей не надоест, а мы получаем ситуацию, когда высокоприоритетная Задача А не может функционировать из-за того, что необходимый ей ресурс занят низкоприоритетной Задачей С.

Синхронизация с внешними событиями

Известно, что применение аппаратных прерываний является более эффективным методом взаимодействия с внешним миром, чем метод опроса. Разработчики систем реального времени стараются использовать этот факт в полной мере. При этом можно проследить следующие тенденции:

1. Стремление обеспечить максимально быструю и детерминированную реакцию системы на внешнее событие.
2. Старание добиться минимально возможных периодов времени, когда в системе запрещены прерывания.
3. Подпрограммы обработки прерываний выполняют минимальный объем функций за максимально короткое время. Это обусловлено несколькими причинами. Во-первых, не все ОС РВ обеспечивают возможность «вытеснения» во время обработки подпрограмм прерывания. Во-вторых, приоритеты аппаратных прерываний не всегда соответствуют приоритетам задач, с которыми они связаны. В-третьих, задержки с обработкой прерываний могут привести к потере данных.

Как правило, закончив элементарно необходимые действия, подпрограмма обработки прерываний генерирует в той или иной форме сообщение для задачи, с которой это прерывание связано, и немедленно возвращает управление. Если это сообщение перевело задачу в разряд готовых к исполнению, планировщик в зависимости от используемого алгоритма и приоритета задачи принимает решение о том, необходимо или нет немедленно передать управление получившей сообщение задаче. Разумеется, это всего лишь один из возможных сценариев, так как каждая ОС РВ имеет свои особенности при обработке прерываний. Кроме того, свою специфику может накладывать используемая аппаратная платформа.

Синхронизация по времени

Совсем не так давно (лет 20 назад) аппаратные средства, отвечающие в вычислительных системах за службу времени, были совершенно не развиты. В те приснопамятные времена считалось достаточным, если в системе генерировалось прерывание с частотой сети переменного тока. Те же, кто не знал, что частота сети в США 60 Гц, а не 50, как у нас, постоянно удивлялись тому, что системное время в RSX-11M никогда не бывает правильным. Программисты для получения задержек по времени часто использовали программные циклы ожидания и, разумеется, пользователи таких программ получали массу сюрпризов при попытке их переноса на следующее поколение компьютеров с более высокими тактовыми частотами. Слава Богу (или научно-техническому прогрессу), сейчас любой мало-мальски приличный компьютер имеет часы/календарь с батарейной поддержкой и многофункциональный таймер (а то и несколько) с разрешением до единиц микросекунд.

Как правило, в ОС РВ задается эталонный интервал (квант) времени, который иногда называют тиком (Tick) и который используется в качестве базовой единицы измерения времени. Размерность этой единицы для разных ОС РВ может быть разной, как, впрочем, разными могут быть набор функций и механизмы взаимодействия с таймером. Функции по работе с таймером используют для приостановки выполнения задачи на какое-то время, для запуска задачи в определенное время, для относительной синхронизации нескольких задач по времени и т. п. Если в программе для ОС РВ вы увидите операнд delay (50), то, скорее всего, это обозначает, что в этом месте задача должна прерваться (блокироваться), а через 50 мс возобновить свое выполнение, а точнее, перейти в состояние готовности. Все это время процессор не простаивает, а решает другие задачи, если таковые имеются. Множество задач одновременно могут запросить сервис таймера, поэтому если для каждого такого запроса используется элемент в таблице временных интервалов, то накладные расходы системы по обработке прерываний от аппаратного таймера растут пропорционально размерности этой таблицы и могут стать недопустимыми. Для решения этой проблемы можно вместо таблицы использовать связный список и алгоритм так называемого дифференциального таймера, когда во время каждого тика уменьшается только один счетчик интервала времени.

Для точной синхронизации таймера вычислительной системы с астрономическим временем могут применяться специальные часы с подстройкой по радиосигналам точного времени или навигационные приемники GPS, которые позволяют воспользоваться атомными часами на борту орбитальных космических аппаратов, запущенных по программе Navstar.

ТЕСТИРОВАНИЕ

Прежде чем устанавливать вашу систему реального времени на не менее реальном объекте, рекомендуется проверить ее работоспособность с помощью интенсивных тестов. Это особенно важно для сложных динамических систем. Во время такого тестирования желательно смоделировать наиболее неприятные и «тяжелые» режимы работы, аварийные ситуации и т. п. При умозрительном анализе простых систем следует осторожно относиться к рекламной информации разработчиков ОС РВ, которые из коммерческих соображений показывают, как правило, параметры для «лучшего случая». Например, если речь идет о максимальном времени обработки прерывания, необходимо в первую очередь понять, а что, собственно, подразумевается под этим временем:

МОЖНО ЛИ ОБОЙТИСЬ БЕЗ ОС РВ?

Как любую вычислительную систему можно создать только из элементов 2И-НЕ, так и все, что может делать ОС РВ, реализуемо и без нее. Тем не менее, давайте все-таки попробуем разобраться, когда ОС РВ реально нужна, а когда нет.

Предположим, нам надо не реже 10 раз в секунду опросить три переключателя и в зависимости от их положения включить или выключить соответствующий насос. Программа может выглядеть следующим образом:

```
void main (void)
{
int i;
for (;;) {
        for (i=0;i<3;i++) {
                if (switch_was_changed(i)) change_Pump(i);
                }
        }
}
```

Заметьте, что никаких проверок по времени не производится, так как даже самые медленные процессоры могут сканировать переключатели чаще 10 раз в секунду. Программа адекватна поставленной задаче.

Если требуется опрашивать переключатели ровно 10 раз в секунду, программа может быть изменена:

```
for (;;) {
        if (100msec_passed) {
                for (i=0;i<3;i++) {
                        if (switch_was_changed(i) change_Pump(i);
                        }
                100msec_passed=0;
                }
        }
```

Глобальная переменная 100msec_passed устанавливается в «1» каждые 100 мс с помощью подпрограммы обработки прерываний от таймера. Теперь допустим, что нам дополнительно нужно каждые 200 мс измерять давление и открывать вентиль, если давление больше 20 атм. Если при открытом вентиле давление падает ниже 15 атм, вентиль необходимо закрыть. Для выполнения этой задачи в тело цикла может быть добавлен следующий фрагмент:

```
if (200msec_passed) {
        switch (valve_status) {
                case CLOSED:
                        if (pressure_value()>20){
                                open_valve();
                                valve_status=OPEN;
                                }
                case OPEN:
                        if (pressure_value()<15){
                                close_valve();
                                valve_status=CLOSED;
                                }
                }
        200msec_passed=0;
        }
```

Глобальная переменная 200msec_passed устанавливается в 1 каждые 200 мс. Так как тело цикла for (;;) стало большим, удобно вынести функции в отдельные подпрограммы и переписать основную программу следующим образом:

```
for (;;) {
        process_pump_switches();
        process_pressure_regulation();
}
```

По мере того как добавляются новые «задачи», их можно оформлять отдельными подпрограммами и включать соответствующие вызовы в тело главной программы. Однако по мере добавления новых функций время выполнения основного цикла увеличивается, в результате чего может наступить момент, когда требование о сканировании переключателей 10 раз в секунду перестанет выполняться.

Давайте еще немного усложним задачу. Пусть система должна отображать тренды на основе пакетов данных, получаемых через быстродействующий последовательный порт. В этом случае основная программа может выглядеть как

```
for (;;) {
        process_pump_switches();
        process_pressure_regulation();
        show_trend();
}
```

Если функция show_trend вызывается с запаздыванием, то пакет данных может быть потерян. Решением этой проблемы может быть вынос коммуникационных функций из show_trend и организация очереди, куда при обработке прерываний последовательного порта помещается очередной заполненный пакет для последующей обработки с помощью show_trend.

Приведенный здесь пример иллюстрирует циклический (round robin) механизм выполнения задач, вполне подходящий для многих применений. Однако этот же пример показывает, что по мере роста числа и сложности функций, которые необходимо выполнять, наличие стандартных средств организации параллельного выполнения задач, работы с таймером, межзадачного обмена информацией и т. п. могут существенно повысить производительность программистов и уменьшить число ошибок. Именно здесь могут проявить свои положительные стороны ядра и операционные системы реального времени, как раз такие средства и предлагающие.

В общем случае решение о применении какого-либо коммерческого ПО реального времени зависит от множества факторов, в том числе и от таких, как время, отпущенное на разработку, наличие и квалификация специалистов, объемы финансирования проекта и т. п.

а) время от возникновения запроса на прерывание до передачи управления по вектору прерывания;

б) или включая время сохранения контекста текущей задачи и передачи управления подпрограмме обработки прерывания;

в) или дополнительно к этому еще и время до завершения подпрограммы обработки прерывания и передачи сообщения связанной с прерыванием задаче;

г) или дополнительно к этому время до момента, когда эта задача наконец получит управление (в предположении, что она является наиболее приоритетной) и начнет реальную обработку события.

Безотносительно к тому, какой вариант рассматривается, необходимо помнить, что

1) если наряду с разработанными вами программами используется программное обеспечение третьих фирм, вы не застрахованы от того, что там не встретятся участки кода, где прерывания запрещены;

2) практически любая ОС РВ имеет в своих недрах участки такого кода. Нам остается только надеяться, что разработчики ОС старались делать их как можно меньше;

3) всё ядро ОС РВ или его участки могут быть «невытесняемыми»;

4) интеллектуальные контроллеры ввода/вывода типа SCSI могут инициировать в системе различные служебные операции, которые способны отразиться на ее характеристиках;

5) многое зависит от применяемой системы кэширования.

Поэтому, если ваше событие произошло в «неподходящее» время, реальные показатели быстродействия могут сильно отличаться от рекламируемых.

Перефразируя классика, самое время воскликнуть: «Тестировать, тестировать и еще раз тестировать!!!»

ЗАКЛЮЧЕНИЕ

Хотя, как мы увидели, ОС РВ предоставляют много полезных и удобных средств для написания программ, основной груз ответственности лежит на плечах рядового труженика-программиста. Ведь стоит ему переместить переменную из категории локальных в категорию глобальных и забыть надлежащим образом оформить критические секции, как станут падать космические корабли, взрываться нефтеперерабатывающие заводы, источать радиоактивность атомные станции. Так воспоем гимн этому великому труженику. Крепче держать знамя социалистического соревнования за победу над инверсией приоритетов. ●