**Планирование системы реального времени.**

Обычно одно или несколько физических устройств, не имеющих отношения к компьютеру, генерируют входные сигналы, а компьютер в определенный промежуток времени должен соответствующим образом на них реагировать.

В системах реального времени, в которых главным критерием эффективности является обеспечение временных характеристик вычислительного процесса, планирование имеет особое значение. Любая система реального времени должна реагировать на сигналы управляемого объекта в течение заданных временных ограничений. Необходимость тщательного планирования работ облегчается тем, что в системах реального времени весь набор выполняемых задач известен заранее.

При разработке алгоритмов планирования для систем реального времени необходимо учитывать, какие последствия в этих системах возникают при несоблюдении временных ограничений. Если эти последствия катастрофичны, как, например, для системы управления полетами или атомной электростанцией, то операционная система реального времени, на основе которой строится управление объектом, называется жесткой (hard). Если же последствия нарушения временных ограничений не столь серьезны, то есть сравнимы с той пользой, которую приносит система управления объектом, то система являетсямягкой (soft)системой реального времени. Примером мягкой системы реального времени является система резервирования билетов.

Системы реального времени:

• Жёсткие(Превышение дедлайна недопустимоУправление движением, технологическими процессами)

• Гибкие(Нарушение графика нежелательно, но допустимоПотоковое видео)

События в системах реального времени

• Периодические

• Непериодические

**Потоки. Способы реализации потоков.**

Традиционная операционная система определяет для каждого процесса свое адресное пространство, кроме того у каждого процесса имеется единственный поток управления. Но довольно часто возникают ситуации, когда для выполнения задачи требуется несколько параллельных потоков управления в одном и том же адресном пространстве.

Существует несколько способов реализации потоков:в пространстве пользователя,в ядре,смешанная реализация.

При реализации потоков в ядре, ядро знает о существовании потоков и управляет ими. Все запросы, которые могут блокировать поток, реализуются как системные запросы, что требует значительно больших временных затрат, чем вызов процедуры системы поддержки исполнения программ.

При реализации в пользовательском пространстве - поместить весь набор потоков в пользовательское пространство. Ядро операционной системы не имеет никакой информации о таком наборе. В таком случае ядро управляет с его точки зрения обычными процессами, которые состоят из одного потока. Самым очевидным преимуществом такой реализации считается то, что набор потоков может быть реализован в не поддерживающей потоки операционной системе.

**Прерывания. Типы прерываний.**

Прерывание - это прекращение выполнения текущей команды или текущей последовательности команд для обработки некоторого события специальной программой - обработчиком прерывания, с последующим возвратом к выполнению прерванной программы

Классы прерываний 1. Внешние / аппаратные (Возникают асинхронно выполнению команд• Маскируемые• Немаскируемые) 2. Внутренние / исключения (Синхронно или в аварийной ситуации) 3. Программные (Вызов обработчика)

**Межпроцессорное взаимодействие. Средства межпроцессорного взаимодействия.**

Цели кооперации процессов• Повышение скорости работы(Один ждёт события, пока второй работает)• Совместное использование данных(Общая БД, разделяемые файлы)• Модульная конструкция системы(Программы-серверы микроядерной ОС)• Удобство работы пользователя(Редактор+отладчик)

Процессы, которые влияют на поведение друг друга путем обмена информацией, принято называть кооперативными или взаимодействующими процессами, в отличие от независимых процессов, не оказывающих друг на друга никакого воздействия.

**Сигнальные средства связи.**

• Передается минимальное количество информации – вплоть до одного бита • Степень воздействия минимальна. • Адресат должен ожидать сигнал и знать, что он означает

**Критическая секция.**

Критическая секция – это часть программы, исполнение которой может привести к возникновению race condition для определенного набора программ. Чтобы исключить эффект гонок по отношению к некоторому ресурсу (критическому ресурсу), необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом.

**Механизм обработки прерываний.**

опрашиваемый способ 1. На ЦПУ поступает сигнал о прерывании 2. ЦПУ производит опрос устройств 3. По найденному устройству определяется номер (вектор) прерывания 4. По вектору определяется адрес обработчика

+: простота аппаратной реализации, гибкость с т.з. устройств. -: длительный цикл опроса устройств, ЦПУ занимается непрофильной работой

векторный способ 1. На ЦПУ поступает сигнал о прерывании и вектор прерывания 2. По вектору определяется адрес обработчика

+: высокая скорость, обработка начинается сразу. -: сложность подключения одинаковых устройств, требующих одного и того же вектора  необходимо дополнительное конфигурирование

векторно-опрашиваемый способ 1. Все устройства относятся к одному из 15 IRQL 2. На ЦПУ поступает сигнал о прерывании в виде номера IRQL 3. ЦПУ производит опрос устройств данного IRQL 4. По найденному устройству определяется номер (вектор) прерывания 5. По вектору определяется адрес обработчика

+: на одном уровне IRQ мало устройств (обычно от 1 до 3), опрос устройств короткий; при этом сохраняется гибкость подключения устройств без конфигурирования. -: опрос, хоть и короткий, всё равно нужен

контроллер прерываний 1. Все устройства относятся к одному из 15 IRQL 2. КП – доп.устройство 3. Cигнал о прерывании в виде номера IRQL поступает на КП 4. КП производит опрос устройств данного IRQL 5. По найденному устройству определяется номер (вектор) прерывания 6. IRQL и вектор прерывания КП передаёт на ЦПУ 7. По вектору определяется адрес обработчика

+: опрос короткий и не всегда нужен; процессор вообще не опрашивает, а имеет сразу вектор. -: необходимо дополнительное устройство – КП

**Системные вызовы.**

Системный вызов — обращение прикладной программы к ядру операционной системы для выполнения какой-либо операции.

Ядро ОС исполняется в привилегированном режиме работы процессора. Для выполнения межпроцессной операции или операции, требующей доступа к оборудованию, программа обращается к ядру, которое, в зависимости от полномочий вызывающего процесса, исполняет либо отказывает в исполнении такого вызова.

С точки зрения программиста, системный вызов обычно выглядит как вызов подпрограммы или функции из системной библиотеки. Однако системный вызов, как частный случай вызова такой функции или подпрограммы, следует отличать от более общего обращения к системной библиотеке, поскольку последнее может и не требовать выполнения привилегированных операций.

Как правило, система предоставляет библиотеку или API, которые находятся между обычным приложением и ОС. Такая библиотека предоставляет программисту удобный интерфейс для работы с ОС в виде интерфейсных функций. В заголовочном файле ядра Linux sys/syscall.h доступна функция syscall, позволяющая непосредственно выполнять системные вызовы. Системными вызовами являются: open, read, write, close, wait, exec, fork, exit и kill. Многие современные ОС имеют сотни системных вызовов.

Системные вызовы могут быть сгруппированы в пять больших категорий:Управление процессами(load,execute)Работа с файлами(open, close,read, write)Управление устройствами(request device, release device)Работа с информацией(get/set time or date)Связь, коммуникация(send, receive messages)

**Логическая организация взаимодействия.**

Логическая организация взаимодействия: адресация• прямая• непрямая• симметричная прямая• асимметричная, валентность(валентность связи – максимальное количество процессов, которые могут одновременно воспользоваться данной связью)

направленность • Однонаправленная (симплексная) Передача только в одном направлении • Двунаправленная (полудуплексная) Поочередная передача в обе стороны • Двунаправленная (дуплексная) Одновременная передача в обе стороны

**Взаимоисключение. Условие Бернстайна.**

Про недетерминированный набор программ (и активностей вообще) говорят, что он имеет race condition (состояние гонки , состояние состязания) Устранение race condition = эксклюзивное право доступа к данным для каждого процесса Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется взаимоисключением (mutual exclusion).

Условия Бернстайна Если для двух данных активностей P и Q: пересечение W(P) и W(Q) пусто, пересечение W(P) с R(Q) пусто, пересечение R(P) и W(Q) пусто, тогда выполнение P и Q детерминировано.

+: детерминированность выполнения. -: слишком жёсткие условия, фактически невзаимодействующие активности

**Канальные средства связи.**

• В общем случае – передача неструктурированного потока байт • Взаимодействие по логическим линиям связи, предоставленным ОС • Объём передаваемой информации ограничен пропускной способностью • Буфер нулевой емкости или отсутствует. Никакая информация не может сохраняться на линии связи. • Буфер ограниченной емкости. Размер буфера равен n, то есть линия связи не может хранить до момента получения более чем n единиц информации. • Буфер неограниченной емкости. Теоретически это возможно, но практически вряд ли реализуемо.

[программные каналы]• Программный канал (PIPE) реализует потоковую модель ввода/вывода • PIPE создаётся в памяти, где именно – знает только сам процесс (переменная с адресом) • Дочерние наследуют это знание – использовать PIPE могут только генеалогически связанные процессы

[именованные каналы]• Именованный канал (FIFO) размещается в файловой системе и имеет имя • По имени FIFO доступен любому процессу • FIFO является файлом специального типа, который имеет только имя, но не имеет содержимого в файловой системе – отображается в область памяти

[сообщения]• Сообщения накладывают на поток данных структуру • Сообщение может включать заголовок, тело (необязательно), хвостовик (необязательно) • Заголовок описывает: тип сообщения, адрес получателя, состав тела данных и т.п.

**Алгоритмы синхронизации.**

Алгоритм Деккера - первое известное корректное решение проблемы взаимного исключения. Если два процесса пытаются перейти в критическую секцию одновременно, алгоритм позволит это только одному из них, основываясь на том, чья в этот момент очередь. Если один процесс уже вошёл в критическую секцию, другой будет ждать, пока первый покинет её. Это реализуется при помощи использования двух флагов (индикаторов "намерения" войти в критическую секцию) и переменной turn (показывающей, очередь какого из процессов наступила).

Алгоритм Петерсона — программный алгоритм взаимного исключения потоков исполнения кода. Перед тем как начать исполнение критической секции кода (то есть кода, обращающегося к защищаемым совместно используемым ресурсам), поток должен вызвать специальную процедуру (назовем ее EnterRegion) со своим номером в качестве параметра. Она должна организовать ожидание потока своей очереди входа в критическую секцию. После исполнения критической секции и выхода из нее, поток вызывает другую процедуру (назовем ее LeaveRegion), после чего уже другие потоки смогут войти в критическую область. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу.

Алгоритм булочной. Каждый вновь прибывающий процесс получает метку с номером. Процесс с наименьшим номером метки обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут метки с разными номерами.

**Требования к алгоритмам синхронизации.**

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения.

2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.

3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях - Условие взаимоисключения

4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго -Условие прогресса

5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз - Условие ограниченного ожидания

**Семафоры. Концепция семафоров.**

Сем-ы (semaphore) - это осн. метод синхр-ии. Он, в сущ-ти, явл. наиб. общим методом синхр-ции процессов. В класс. определении семафор предст. собой целую пер-ную, знач. которой больше нуля, то есть просто счетчик. Обычно семафор инициализируется в начале программы 0 или 1. Семафоры, которые могут принимать лишь значения 0 и 1, называются двоичными.

Семафоры Дейкстры. Семафор – это целая неотрицательная переменная, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P и V. Proberen P(S): пока S == 0 процесс блокируется; S = S – 1; Verhogen V(S): S = S + 1; Мьютекс – двоичный семафор (0 или 1)

**Алгоритмы синхронизации. Запрет прерываний. Переменная замок. Строгое чередование. Флаги готовности. Алгоритм Патерсона. Команда Test-and-Set. Команда Swap.**

Пренебрежение вопросами синхронизации в многопоточной системе может привести к неправильному решению задачи или даже к краху системы.

Запрет прерываний. while (some condition) { запретить все прерывания -> critical section -> разрешить все прерывания -> remainder section }

Сбой в критической секции приведет к неработоспособности системы (с запрещёнными прерываниями) Применяется как пролог и эпилог к критическим секциям внутри самой операционной системы

Переменная-замок. shared int lock = 0; while (some condition) { while(lock); lock = 1; critical section lock = 0; remainder section }

Нарушено условие взаимоисключения: неатомарный пролог приводит к попаданию в критическую секцию двух процессов

Строгое чередование. shared int turn = 0; while (some condition) { while(turn != i); critical section turn = 1-i; remainder section } Взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, P0, ... Не удовлетворяет условию прогресса. Например, если значение turn равно 1, и процесс P0 готов войти в критический участок, он не может сделать этого, даже если процесс P1 находится в remainder section

Флаги готовности. shared int ready[2] = {0, 0}; while (some condition) { ready[i] = 1; while(ready[1-i]); critical section ready[i] = 0; remainder section } Нарушает условие прогресса: при одновременном выполнении неатомарного пролога с чередованием оба процесса бесконечно ждут друг друга на входе в критическую секцию.

Алгоритм Петерсона — программный алгоритм взаимного исключения потоков исполнения кода. Перед тем как начать исполнение критической секции кода (то есть кода, обращающегося к защищаемым совместно используемым ресурсам), поток должен вызвать специальную процедуру (назовем ее EnterRegion) со своим номером в качестве параметра. Она должна организовать ожидание потока своей очереди входа в критическую секцию. После исполнения критической секции и выхода из нее, поток вызывает другую процедуру (назовем ее LeaveRegion), после чего уже другие потоки смогут войти в критическую область. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу.

Команда Test-and-Set. int Test\_and\_Set (int \*target){ int tmp = \*target; \*target = 1; return tmp; } shared int lock = 0; while (some condition) { while(Test\_and\_Set(&lock)); critical section lock = 0; remainder section }

Команда Swap. void Swap (int \*a, int \*b){ int tmp = \*a; \*a = \*b; \*b = tmp; } shared int lock = 0; int key; while (some condition) { key = 1; do Swap(&lock,&key); while (key); critical section lock = 0; remainder section }

**Механизмы синхронизации.**

При паралл. работе нескол. процессов или потоков могут возник. конфликты при исп-ии разд-мых рес-ов. Такие ресурсы наз. критическими. Для избежания данного явл-я применяют разл. механизмы взаимоисключ., т. е. один процесс при работе в КС блок-ет другой. Мех-мы синхр-ции — высокоуровневые вар-ты реализации синхр-ции с исп-ем средств ОС. Достоинством явл. Возм-ость учета вр-ни ожид-ия входа в КС. Реализ-ся программно или с исп-ием аппаратных средств (семафоры, мониторы, сообщения). Можно показать, что в рамках одной вычислительной системы, когда процессы имеют возможность использовать разделяемую память, все механизмы синхронизации эквивалентны. Т. е. любые два из механизмов могут быть реализованы на базе третьего.

**Решение проблемы производителя и потребителя.**

Одной из типовых задач, требующих синхронизации, является задача producerconsumer (производитель-потребитель). Пусть два потока обмениваются информацией через буфер ограниченного размера. Производитель добавляет информацию в буфер, а потребитель извлекает ее оттуда. Задача "Производители-Потребители" заключается в обеспечении согласованного доступа нескольких потоков к разделяемому циклическому буферу.

Корректное решение должно удовлетворять следующим условиям: потоки выполняются параллельно; одновременно в критической секции, связанной с каждым критическим ресурсом, должно находиться не более одного потока; потоки должны завершить работу в течение конечного времени; потоки должны корректно использовать операции с циклическим буфером.

Решение задачи "Производитель-Потребитель", использующее семафоры Semaphore Access = 1;//управляет доступом к разделяемым данным Semaphore Empty = n; // количество пустых записей Semaphore Full = 0; // количество заполненных записей Producer(){ P(Empty); // ждем появления свободной записи P(Access); // получаем доступ к указателям <записываем значение в запись> V(Access); // завершили работу с указателями V(Full); // сигнализируем о появлении заполненной записи } Consumer(){ P(Full); // ждем появления заполненной записи P(Access); // получаем доступ к указателям <извлекаем данные из записи> V(Access); // завершили работу с указателями V(Empty); // сигнализируем о появлении свободной записи <используем данные из записи> }

**Мониторы, мьютексы, сообщения.**

Мониторы Хоара • Монитор – тип данных в ЯВУ (аналог класса в ООП), содержит переменные, определяющие его состояние, и функции-методы. • В любой момент времени только один процесс может быть активен, т. е. находиться в состоянии готовность или исполнение, внутри данного монитора • Условные переменные – переменные монитора, над которыми могут производиться две примитивные операции: • wait – процесс блокируется на данной переменной • signal – процесс, заблокированный на данной переменной, разблокируется

Мьютексы. Мьютекс – двоичный семафор, обычно используемый для организации согласованного доступа к неделимому общему ресурсу. Мьютекс может принимать значения 1 (свободен) и 0 (занят). Операции над мьютексами • acquire(mutex) – уменьшить (занять) мьютекс • release(mutex) – увеличить (освободить) мьютекс • tryacquire(mutex) – часто реализуемая неблокирующая операция, выполняющая попытку уменьшить (занять) мьютекс

Мьютексы в конкретных реализациях могут иметь дополнительные свойства • Запоминание владельца – освободить мьютекс может только поток, захвативший его • Рекурсивность – поток может многократно захватить мьютекс (вызывать aquire()); для освобождения мьютекса поток должен соответствующее число раз вызвать release() • Наследование приоритета – поток, захвативший мьютекс, временно наследует максимальный из приоритет потоков, ждущих освобождения данного мьютекса

Сообщения –способ межпроцессного и межпоточного взаимодействия, позволяющий потокам подавать сигналы друг другу и обмениваться данными. Используется для организации взаимодействия потоков, выполняющихся на различных узлах сети.

Типы доставки сообщений. Асинхронный. поток, посылающий сообщение, инициирует процесс доставки сообщения, после чего продолжает свою работу. Синхронный. поток, пославший сообщение, дожидается подтверждения его получения принимающим потоком.

Адресация. Прямая. send(P, message) – послать сообщение message процессу P, receive(Q, message) – получить сообщение message от процесса Q. Непрямая. send(A, message) – послать сообщение message в буфер A, receive(A, message) – получить сообщение message из буфера A