Процессы и потоки. Межпроцессное и межпоточное взаимодействие. Часть 2

Лекция

Ревизия: 0.2

Последняя модификация: Vlad Kovtun
© NRJETIX 2000 - 2010
Дата последней модификации: 15.10.2014 22:29:00

История изменений

09.02.2010 – Версия 0.1.Первичный документ. Ковтун В.Ю.

19.07.2014 – Версия 0.2.Замена рисунков. Ковтун В.Ю.

Содержание

Лекция 3. Процессы и потоки. Межпроцессное и межпоточное взаимодействие. Часть 2

Вопросы

1. Межпроцессное взаимодействие.
2. Межпроцессное взаимодействие в распределенных системах.
3. Классические проблемы межпроцессного взаимодействия.

Межпроцессное и межпоточное взаимодействие

Процессам часто бывает необходимо взаимодействовать между собой. Например, в конвейере ядра выходные данные первого процесса должны передаваться второму и т. д. по цепочке. **Межпроцессное взаимодействие** (от англ. Inter-Process Communication или сокр. IPC) - набор способов обмена данными между множеством потоков в одном или более процессах.

Ситуации, когда приходится процессам взаимодействовать:

- Передача информации от одного процесса другому
- Контроль над деятельностью процессов (например: когда они борются за один ресурс)
- Согласование действий процессов (например: когда один процесс поставляет данные, а другой их выводит на печать. Если согласованности не будет, то второй процесс может начать печать раньше, чем поступят данные).

Два вторых случая относятся и к потокам. В первом случае у потоков нет проблем, т.к. они используют общее адресное пространство.

Состояние состязания

В некоторых ОС процессы, работающие совместно, могут сообща использовать некое общее хранилище данных. Каждый из процессов может считывать из общего хранилища данных и записывать туда информацию. Это хранилище представляет собой участок в основной памяти (возможно, в структуре данных ядра) или файл общего доступа. Местоположение совместно используемой памяти не влияет на суть взаимодействия и возникающие проблемы. Рассмотрим межпроцессное взаимодействие на простом, но очень распространенном примере: спулер печати. Если процессу требуется вывести на печать файл, он помещает имя файла в специальный каталог спулера. Другой процесс, демон печати, периодически проверяет наличие файлов, которые нужно печатать, печатает файл и удаляет его имя из каталога.

Представьте, что каталог спулера состоит из большого числа сегментов, пронумерованных 0, 1, 2, ..., в каждом их которых может храниться имя файла. Также есть две совместно используемые переменные: `out`, указывающая на следующий файл для печати, и `in`, указывающая на следующий свободный сегмент. Эти две переменные можно хранить в одном файле (состоящем из двух слов), доступном всем процессам. Пусть в данный момент сегменты с 0 по 3 пусты (эти файлы уже напечатаны), а сегменты с 4 по 6 заняты (эти файлы ждут своей очереди на печать). Более или менее одновременно процессы А и Б решают поставить файл в очередь на печать. Описанная ситуация схематически изображена на Рис. 1.

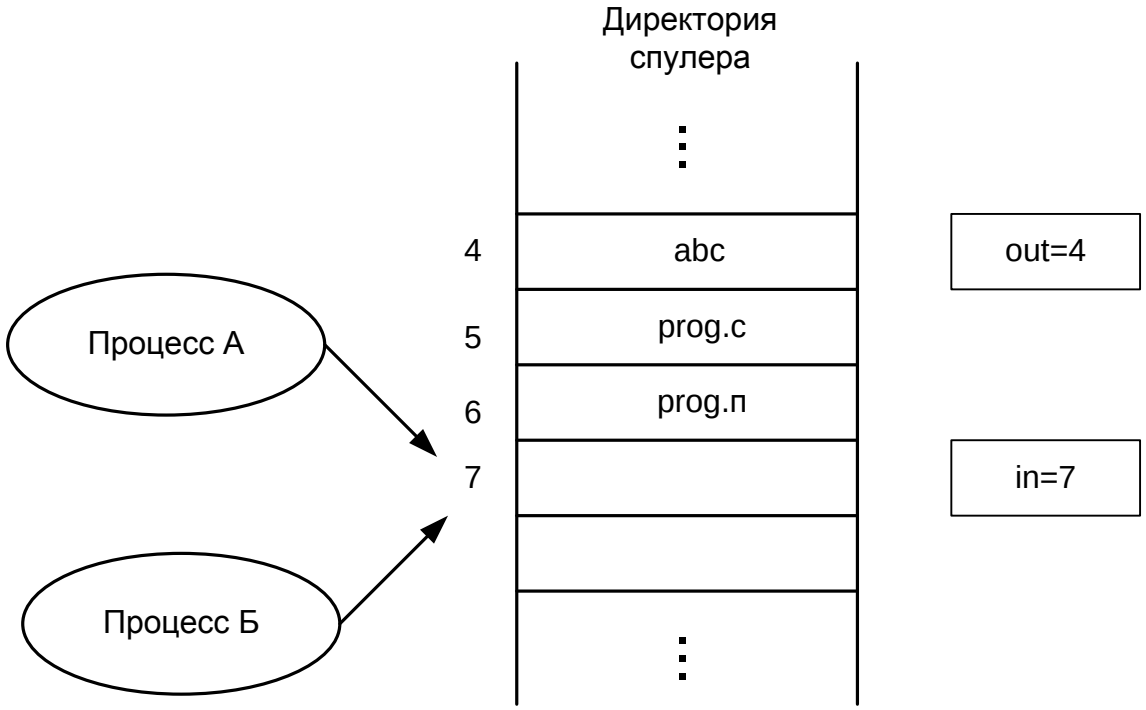

Рис. 1. Процесс А и Б пытаются получить доступ к одному и тому же адресу памяти

Процесс А считывает значение (7) переменной `in` и сохраняет его в локальной переменной `next_free_slot`. После этого происходит прерывание, по таймеру, и CPU переключается на процесс Б. Процесс Б, в свою очередь, считывает значение переменной `in` и сохраняет его (опять 7) в своей локальной переменной `next_free_slot`. В данный момент оба процесса считают, что следующий свободный сегмент — седьмой.

Процесс Б сохраняет в каталоге спулера имя файла и заменяет значение `in` на 8, затем продолжает заниматься своими задачами, не связанными с печатью.

Наконец управление переходит к процессу А, и он продолжает с того места, на котором остановился. Он обращается к переменной `next_free_slot`, считывает ее значение и записывает в седьмой сегмент имя файла (разумеется, удаляя при этом имя файла, записанное туда процессом Б). Затем он заменяет значение `in` на 8 (`next_free_slot + 1=8`). Структура каталога спулера не нарушена, так что демон печати не заподозрит ничего плохого, но файл процесса Б не будет напечатан. Ситуации, в которых два (и более) процесса считывают или записывают данные одновременно и конечный результат зависит от того, какой из них был первым, называются **состояниями состязания**.

Критические области

Как избежать **состязания**? Основным способом предотвращения проблем в этой и любой другой ситуации, связанной с совместным использованием памяти, файлов и чего-либо еще, является **запрет одновременной записи и чтения разделенных данных более чем одним процессом**. Говоря иными словами, необходимо **взаимное исключение**. Это означает, что в тот момент, когда один процесс использует разделенные данные, другому процессу это делать будет запрещено. Проблема, описанная в предыдущем параграфе, возникла из-за того, что процесс Б начал работу с одной из совместно используемых переменных до того, как процесс А ее закончил. Выбор подходящей **примитивной операции, реализующей взаимное исключение, является серьезным моментом разработки ОС**.

Проблему исключения состояний состязания можно сформулировать на абстрактном уровне. Некоторый промежуток времени процесс занят внутренними расчетами и другими задачами, не приводящими к состояниям состязания. В другие моменты времени процесс обращается к совместно используемым данным или выполняет какое-то другое действие, которое может привести к состязанию. Часть программы, в которой есть обращение к совместно используемым данным, называется **критической**

областью или **критической секцией**. Если удастся избежать одновременного нахождения двух процессов в критических областях, можно избежать состязаний.

Несмотря на то, что это требование исключает состязание, его недостаточно для правильной совместной работы параллельных процессов и эффективного использования общих данных. **Для этого необходимо выполнение четырёх условий** (Условия избегания состязания и эффективной работы процессов):

1. Два процесса не должны одновременно находиться в критических областях.

2. В программе не должно быть предположений о скорости или количестве CPU.

3. Процесс, находящийся вне критической области, не может блокировать другие процессы.

4. Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

В абстрактном виде требуемое поведение процессов представлено на Рис. 2. Процесс А попадает в критическую область в момент времени Т1. Чуть позже, в момент времени Т2, процесс Б пытается попасть в критическую область, но ему это не удается, поскольку в критической области уже находится процесс А, а два процесса не должны одновременно находиться в критических областях. Поэтому процесс Б временно приостанавливается, до наступления момента времени Т3, когда процесс А выходит из критической области. В момент времени Т4 процесс В также покидает критическую область, и мы возвращаемся в исходное состояние, когда ни одного процесса в критической области не было.

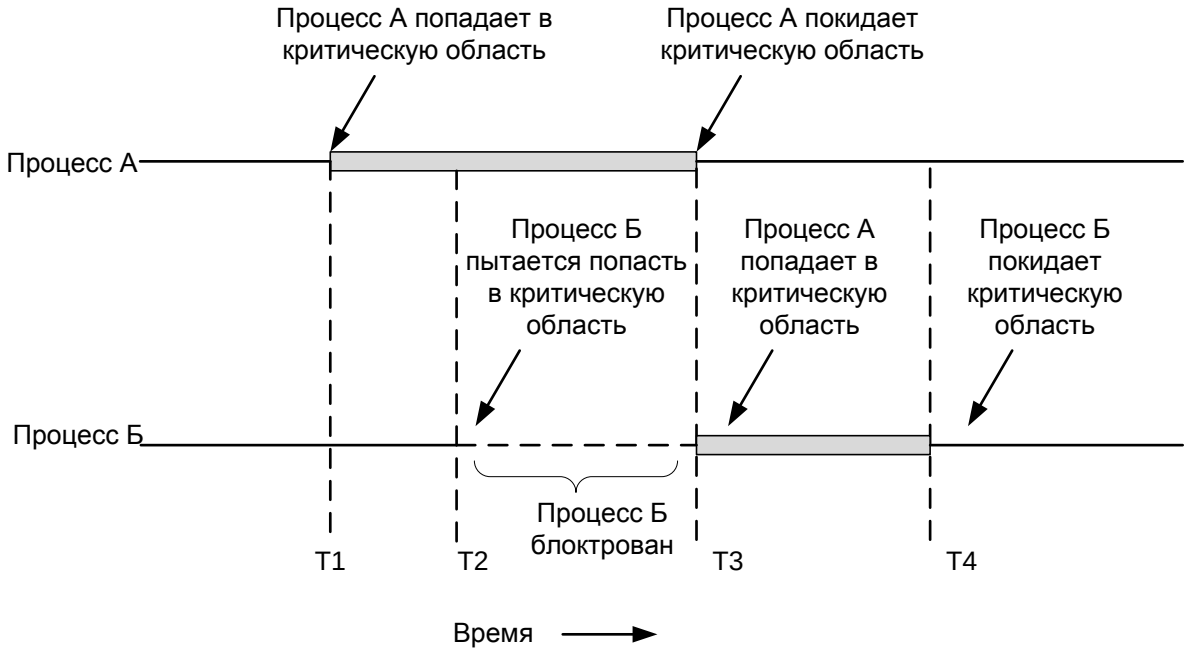

Рис. 2. Взаимное исключение с использованием критических секций

В ОС Windows предусмотрен целый ряд функций для управления критическими секциями, т.е. доступ внутрь критической секции может быть выполнен лишь единственным потоком:

- DeleteCriticalSection
- EnterCriticalSection
- InitializeCriticalSection
- InitializeSectionAndSpinCount
- LeaveCriticalSection
- SetCriticalSectionSpinCount
- TryEnterCriticalSection

Взаимное исключение с активным ожиданием

Рассмотрим различные способы реализации взаимного исключения с целью избежать вмешательства в критическую область одного процесса при нахождении там другого и связанных с этим проблем.

Запрещение прерываний

Самое простое решение состоит в запрещении всех прерываний при входе процесса в критическую область и разрешение прерываний по выходе из области. Если прерывания запрещены, невозможно прерывание по таймеру. Поскольку CPU переключается с одного процесса на другой только по прерыванию, отключение прерываний исключает передачу CPU другому процессу. Таким образом, запретив прерывания, процесс может спокойно считывать и сохранять совместно используемые данные, не опасаясь вмешательства другого процесса.

Пример исходного кода на языке Ассемблера для маскирования аппаратного прерывания:

```
;---маскирование 6-го бита регистра маски прерываний
MOV AL, 01000000B ;маскируем бит 6
OUT 21H, AL ;посылаем в регистр маски прерываний
;---сброс маски
MOV AL,0 ; пустая маска
OUT 21H,AL ;очищаем IMR в конце программы
```

Команды языка Ассемблера:

- `cli` – сбросить флаг прерываний `if` в 0.
- `sti` – установить флаг прерываний `if` в 1.

И все же было бы неразумно давать пользовательскому процессу возможность запрета прерываний. Представьте себе, что процесс отключил все прерывания и в результате какого-либо сбоя не включил их обратно. ОС на этом может закончить свое существование. К тому же в multi-CPU системе запрещение прерываний повлияет только на тот CPU, который выполнит инструкцию `disable`. Остальные CPU продолжат работу и сохранят доступ к разделенным данным.

С другой стороны, для ядра характерно запрещение прерываний для некоторых команд при работе с переменными или списками. Возникновение прерывания в момент, когда, например, список готовых процессов находится в неопределенном состоянии, могло бы привести к состоянию состязания. **Итак, запрет прерываний бывает полезным в самой ОС, но это решение неприемлемо в качестве механизма взаимного исключения для пользовательских процессов.**

Переменные блокировки

Теперь попробуем найти программное решение. Рассмотрим одну совместно используемую **переменную блокировки**, изначально равную 0. Если процесс хочет попасть в критическую область, он предварительно считывает значение переменной блокировки. Если переменная равна 0, процесс изменяет ее на 1 и входит в критическую область. Если же переменная равна 1, то процесс ждет, пока ее значение сменится на 0. Таким образом, 0 означает, что ни одного процесса в критической области нет, а 1 означает, что какой-либо процесс находится в критической области.

К сожалению, у этого метода те же проблемы, что и в примере с каталогом спулера. Представьте, что один процесс считывает переменную блокировки, обнаруживает, что она равна 0, но прежде, чем он успевает изменить ее на 1, управление получает другой

процесс, успешно изменяющий ее на 1. Когда первый процесс снова получит управление, он тоже заменит переменную блокировки на 1 и два процесса одновременно окажутся в критических областях.

Можно подумать, что проблема решается повторной проверкой значения переменной, прежде чем заменить ее, но это не так. Второй процесс может получить управление как раз после того, как первый процесс закончил вторую проверку, но еще не заменил значение переменной блокировки.

В ОС Windows предусмотрен целый ряд функций для выполнения операций над переменными, которые позволяют выполнять их как атомарные, т.е. доступ к переменной может быть выполнен лишь единственным потоком.

Название функции начинается с Interlocked:

- InterlockedCompareExchange (64)
- InterlockedCompareExchangeAcquire (64)
- InterlockedCompareExchangePointer
- InterlockedCompareExchangeRelease (64)
- InterlockedDecrement (64)
- InterlockedIncrement (64)
- etc.

Строгое чередование

Третий метод реализации взаимного исключения иллюстрирован на Рис. 3. Этот фрагмент программного кода, как и многие другие, написан на С.

```
while (TRUE)  {
        while (turn !=0)            /*loop*/;
        critical_region () ;
        turn = 1;
        noncritical_region () ;
}
```

```
while (TRUE)  {
        while ( turn !=0)  /*loop*/ ;
        critical_region () ;
        turn=0;
        noncritical_region ();
}
```

Рис. 3. Предлагаемое решение проблемы критической области: процесс 0 (а); процесс 1 (б)

На Рис. 3 целая переменная `turn`, изначально равная 0, отслеживает, чья очередь входить в критическую область. Вначале процесс 0 проверяет значение `turn`, считывает 0 и входит в критическую область. Процесс 1 также проверяет значение `turn`, считывает 0 и после этого входит в цикл, непрерывно проверяя, когда же значение `turn` будет равно 1. Постоянная проверка значения переменной в ожидании некоторого значения называется **активным ожиданием**. Подобного способа **следует избегать**, поскольку он является **бесцельной тратой времени процессора**. Активное ожидание используется только в случае, когда есть уверенность в небольшом времени ожидания. Блокировка, использующая активное ожидание, называется спин-блокировкой. Функции в ОС Windows: InitializeSectionAndSpinCount и SetSectionAndSpinCount реализуют такой механизм, который имеет **смысл лишь для многопроцессорных (многоядерных) систем**.

Когда процесс 0 покидает критическую область, он изменяет значение `turn` на 1, позволяя процессу 1 попасть в критическую область. Предположим, что процесс 1 быстро покидает свою критическую область, так что оба процесса теперь находятся вне критической области, и значение turn равно 0. Теперь процесс 0 выполняет весь цикл быстро, выходит из критической области и устанавливает значение `turn` равным 1. В этот момент значение `turn` равно 1, и оба процесса находятся вне критической области.

Неожиданно процесс 0 завершает работу вне критической области и возвращается к началу цикла. Но войти в критическую область он не может, поскольку значение `turn` равно 1 и процесс 1 находится вне критической области. Процесс 0 зависнет в своем цикле `while`, ожидая, пока процесс 1 изменит значение `turn` на 0. **Получается, что метод поочередного доступа к критической области не слишком удачен, если один процесс существенно медленнее другого.**

Эта ситуация **нарушает третье из сформулированных нами условий**: один процесс блокирован другим, не находящимся в критической области. Возвратимся к примеру с каталогом спулера: если заменить критическую область процедурой считывания и записи в каталог спулера, процесс 0 не сможет послать файл на печать, поскольку процесс 1 занят чем-то другим.

Фактически этот метод требует, чтобы два процесса попадали в критические области строго по очереди. Ни один из них не сможет попасть в критическую область (например, послать файл на печать) два раза подряд. Хотя этот алгоритм и исключает состояния состязания, его нельзя рассматривать всерьез, поскольку он нарушает третье условие успешной работы двух параллельных процессов с совместно используемыми данными.

Алгоритм Петерсона

Датский математик Деккер (T. Dekker) был первым, кто разработал программное решение проблемы взаимного исключения, не требующее строгого чередования.

В 1981 году Петерсон (G. L. Peterson) разработал существенно более простой алгоритм взаимного исключения. С этого момента алгоритм Деккера стал считаться устаревшим. Алгоритм Петерсона, представленный в Листинг 2. 1, состоит из двух процедур, написанных на ANSI C.

Листинг 2. 1. Решение Петерсона для взаимного исключения

```
#define FALSE 0
#define TRUE  1
#define N     2                    //Количество процессов
int turn:                          //Чья сейчас очередь?
int interested[N]:                 //Все переменные изначально равны 0 (FALSE)
void enter_region(int process);    //Процесс 0 или 1
{
int other:                         //Номер второго процесса
other=1-process;                   //Противоположный процесс
interested[process]=TRUE;          //Индикатор интереса
turn=process;                      //Установка флага
while(turn==process && interested[other]==TRUE);  //Пустой оператор
}
void leave_region(int process)     //process: процесс, покидающий критичную область
{
interested[process]=FALSE;         //Индикатор выхода из критической области
}
```

Прежде чем обратиться к совместно используемым переменным (то есть перед тем, как войти в критическую область), процесс вызывает процедуру `enter_region` со своим номером (0 или 1) в качестве параметра. Поэтому процессу, при необходимости, придется подождать, прежде чем входить в критическую область. После выхода из критической области процесс вызывает процедуру `leave_region`, чтобы обозначить свой выход и тем самым разрешить другому процессу вход в критическую область.

Рассмотрим работу алгоритма более подробно. Исходно оба процесса находятся вне критических областей. Процесс 0 вызывает `enter_region`, задает элементы массива и устанавливает переменную `turn` равной 0. Поскольку процесс 1 не заинтересован в попадании в критическую область, процедура возвращается. Теперь, если процесс 1 вызовет `enter_region`, ему придется подождать, пока `interested[0]` примет значение `FALSE`, а это произойдет только в тот момент, когда процесс 0 вызовет процедуру `leave_regian`, чтобы покинуть критическую область.

Представьте, что оба процесса вызвали `enter_region` практически одновременно. Оба сохранят свои номера в `turn`. Сохранится номер того процесса, который был вторым, а

предыдущий номер будет утерян. Предположим, что вторым был процесс 1, так что значение `turn` равно 1. Когда оба процесса дойдут до оператора `while`, процесс 0 войдет в критическую область, а процесс 1 останется в цикле и будет ждать, пока процесс 0 выйдет из критической области.

Команда TSL

Рассмотрим решение, требующее участия аппаратного обеспечения. Многие компьютеры, особенно разработанные с расчетом на несколько CPU, имеют команду

```
TSL RX, LOCK
```

(Test and Set Lock — проверить и заблокировать), которая действует следующим образом. В регистр RX считывается содержимое слова памяти `lock`, а в ячейке памяти `lock` сохраняется некоторое ненулевое значение. Гарантируется, что операция считывания слова и сохранения неделима — другой процесс не может обратиться к слову в памяти, пока команда не выполнена. CPU, выполняющий команду TSL, блокирует шину памяти, чтобы остальные CPU не могли обратиться к памяти.

Воспользуемся командой TSL. Пусть совместно используемая переменная `lock` управляет доступом к разделенной памяти. Если значение переменной `lock` равно 0, любой процесс может изменить его на 1 и обратиться к разделенной памяти, и затем изменить его обратно на 0, пользуясь обычной командой `move`.

Как использовать эту команду для взаимного исключения? Решение приведено в Листинг 2. 2. Здесь представлена подпрограмма из четырех команд, написанная на типичном ассемблере. Первая команда копирует старое значение `lock` в регистр и затем устанавливает значение переменной равное 1. Потом старое значение сравнивается с нулем. Если оно ненулевое, значит, блокировка уже была установлена и проверка начинается сначала. Рано или поздно значение окажется нулевым (это означает, что процесс, находившийся в критической области, вышел из нее), и подпрограмма возвращается, установив блокировку. Программа просто помещает 0 в переменную `lock`. Специальной команды CPU не требуется.

Листинг 2. 2. Вход и выход из критической области с помощью команды TSL

```
enter_region:
        TSL REGISTER,LOCK;  //значение LOCK копируется в регистр, значение
                            переменной устанавливается равной 1
        GMP REGISTER,#0;    //старое значение LOCK сравнивается с нулем
        JNE enter_region;   //если оно ненулевое, значит блокировка уже была
                            установлена, поэтому цикл завершается
        RET                 //возврат к вызывающей программе, процесс вошёл в
                            критическую область
leave_region:
       MOVE LOCK,#0;        //сохранение 0 в переменной LOCK
RET
```

Одно решение проблемы критических областей теперь очевидно. Прежде чем попасть в критическую область, процесс вызывает процедуру `enter_region`, которая выполняет активное ожидание вплоть до снятия блокировки, затем она устанавливает блокировку и возвращается. По выходу из критической области процесс вызывает процедуру `leave_region`, помещающую 0 в переменную `lock`. Как и во всех остальных решениях проблемы критической области, для корректной работы процесс должен вызывать эти процедуры своевременно, в противном случае взаимное исключение не удастся.

Описание команд CPU архитектуры 0x86, реализующих описанный механизм, описан в приложении А.

Примитивы межпроцессного взаимодействия

Оба решения — Петерсона и с использованием команды TSL — корректны, но они обладают одним и тем же недостатком: **использованием активного ожидания**. В

сущности, оба они реализуют следующий алгоритм: перед входом в критическую область процесс проверяет, можно ли это сделать. Если нельзя, процесс входит в тугой цикл, ожидая возможности войти в критическую область.

Этот алгоритм не только **бесцельно расходует время процессора**, но, кроме этого, он может иметь некоторые неожиданные последствия. Рассмотрим два процесса: H, с высоким приоритетом, L, с низким приоритетом. Правила планирования в этом случае таковы, что процесс H запускается немедленно, как только он оказывается в состоянии ожидания. В какой-то момент, когда процесс L находится в критической области, процесс H оказывается в состоянии ожидания (например, он закончил операцию ввода-вывода). Процесс H попадает в состояние активного ожидания, но поскольку процессу L во время работающего процесса H никогда не будет предоставлено время CPU, у процесса L не будет возможности выйти из критической области, и процесс H навсегда останется в цикле - **проблема инверсии приоритета**.

Теперь рассмотрим некоторые **примитивы межпроцессного взаимодействия**, применяющиеся вместо циклов ожидания, в которых лишь напрасно расходуется время CPU. Эти примитивы блокируют процессы в случае запрета на вход в критическую область. Одной из простейших является пара примитивов `sleep` и `wakeup`. Примитив `sleep` — системный запрос, в результате которого вызывающий процесс блокируется, пока его не запустит другой процесс. У запроса `wakeup` есть один параметр — процесс, который следует запустить. Также возможно наличие одного параметра у обоих запросов — адреса ячейки памяти, используемой для согласования запросов ожидания и запуска.

Проблема производителя и потребителя

В качестве примера использования этих примитивов рассмотрим **проблему производителя и потребителя**, также известную как проблема **ограниченного буфера**. Два процесса совместно используют буфер ограниченного размера. Один из них, производитель, помещает данные в этот буфер, а другой, потребитель, считывает их оттуда. (Можно обобщить задачу на случай `m` производителей и `n` потребителей, но мы рассмотрим случай с одним производителем и одним потребителем, поскольку это существенно упрощает решение.)

Трудности начинаются в тот момент, когда производитель хочет поместить в буфер очередную порцию данных и обнаруживает, что буфер полон. Для производителя решением является ожидание, пока потребитель полностью или частично не очистит буфер. Аналогично, если потребитель хочет забрать данные из буфера, а буфер пуст, потребитель уходит в состояние ожидания и выходит из него, как только производитель положит что-нибудь в буфер и разбудит его.

Это решение кажется достаточно простым, но оно приводит к **состояниям состязания**, как и пример с каталогом спулера. Нам нужна переменная `count` для отслеживания количества элементов в буфере. Если максимальное число элементов, хранящихся в буфере, равно `N`, **программа производителя должна проверить**, не равно ли `N` значение `count` прежде, чем поместить в буфер следующую порцию данных. Если значение `count` равно `N`, то производитель уходит в состояние ожидания; в противном случае производитель помещает данные в буфер и увеличивает значение `count`.

Код программы потребителя прост: сначала проверить, не равно ли значение `count` нулю. Если равно, то уйти в состояние ожидания; иначе забрать порцию данных из буфера и уменьшить значение `count`. Каждый из процессов также должен проверять, не следует ли активизировать другой процесс, и в случае необходимости проделывать это. Программы обоих процессов представлены в Листинг 2. 3.

Листинг 2. 3. Проблема производителя и потребителя с неустранимым состоянием соревнования

```
#define N 100                                   //Максимальное количество элементов в буфере
int count = 0;                                  //Текущее количество элементов в буфере
void producer(void)
{
        int item;
```

```
        while (TRUE) {                    //Повторять вечно
        item = produceitem();             //Сформировать следующий элемент
        if (count == N) sleep();          //Если буфер полон, уйти в состояние ожидания
        insert item(item);                //Поместить элемент в буфер
        count = count +1;                 //Увеличить количество элементов в буфере
        if (count == 1) wakeup(consumer); //Был ли буфер пуст?
}}
void consumer(void)
{
         int item;
         while (TRUE) {                   //Повторять вечно
         if (count == 0) sleep();         //Если буфер пуст, уйти в состояние ожидания
         item = remove_item( );           //Забрать элемент из буфера
         count = count - 1;               //Уменьшить счетчик элементов в буфере
         if (count == N - 1) wakeup(producer); //Был ли буфер полон?
         consume item(item);              //Отправить элемент на печать
}}
```

Процедуры `insert_item` и `remove_item` помещают элементы в буфер и извлекают их оттуда.

Теперь давайте вернемся к состоянию состязания. Его возникновение возможно, поскольку доступ к переменной `count` не ограничен. Может возникнуть следующая ситуация: буфер пуст, и потребитель только что считал значение переменной count, чтобы проверить, не равно ли оно нулю. В этот момент планировщик передал управление производителю, производитель поместил элемент в буфер и увеличил значение `count`, проверив, что теперь оно стало равно 1. Зная, что перед этим оно было равно 0 и потребитель находился в состоянии ожидания, производитель активизирует его с помощью вызова `wakeup`.

Но потребитель не был в состоянии ожидания, так что сигнал активизации пропал впустую. Когда управление перейдет к потребителю, он вернется к считанному когда-то значению `count`, обнаружит, что оно равно 0, и уйдет в состояние ожидания. Рано или поздно производитель наполнит буфер и также уйдет в состояние ожидания. Оба процесса так и останутся в этом состоянии.

Суть проблемы в данном случае состоит в том, **что сигнал активизации, пришедший к процессу, не находящемуся в состоянии ожидания, пропадает**. Если бы не это, проблемы бы не было. Быстрым решением может быть добавление бита ожидания активизации. Если сигнал активизации послан процессу, не находящемуся в состоянии ожидания, этот бит устанавливается. Позже, когда процесс пытается уйти в состояние ожидания, бит ожидания активизации сбрасывается, но процесс остается активным. Этот бит исполняет роль копилки сигналов активизации.

Несмотря на то, что введение бита ожидания запуска спасло положение в этом примере, легко сконструировать ситуацию с несколькими процессами, в которой одного бита будет недостаточно. Мы можем добавить еще один бит, или 8, или 32, но это не решит проблему.

В ОС Windows, частичным решением данной проблемы является использование связного списка с блокировками:

- InitializeSListHead
- InterlockedFlushSList
- InterlockedPopEntrySList
- InterlockedPushEntrySList
- QueryDepthSList

Семафоры

В 1965 году Дейкстра (E. W. Dijkstra) предложил использовать целую переменную для подсчета сигналов запуска, сохраненных на будущее. Им был предложен новый тип переменных, так называемые **семафоры**, значение которых может быть нулем (в случае отсутствия сохраненных сигналов активизации) или некоторым положительным числом, соответствующим количеству отложенных активизирующих сигналов.

Дейкстра предложил две операции, `down` и `up` (обобщения `sleep` и `wakeup`). Операция `down` сравнивает значение семафора с нулем. Если значение семафора больше нуля, операция `down` уменьшает его (то есть расходует один из сохраненных сигналов активации) и просто возвращает управление. Если значение семафора равно нулю, процедура `down` не возвращает управление процессу, а процесс переводится в состояние ожидания. Все операции проверки значения семафора, его изменения и перевода процесса в состояние ожидания выполняются **как атомарное действие**. Тем самым гарантируется, что после начала операции ни один процесс не получит доступа к семафору до окончания или блокирования операции. Элементарность операции чрезвычайно важна для разрешения проблемы синхронизации и предотвращения состояния состязания.

Операция `up` увеличивает значение семафора. Если с этим семафором связаны один или несколько ожидающих процессов, которые не могут завершить более раннюю операцию `down`, один из них выбирается системой (например, случайным образом) и ему разрешается завершить свою операцию `down`. Таким образом, после операции `up`, примененной к семафору, связанному с несколькими ожидающими процессами, значение семафора так и останется равным 0, но число ожидающих процессов уменьшится на единицу. Операция увеличения значения семафора и активизации процесса тоже неделима. Ни один процесс не может быть блокирован во время выполнения операции `up`, как ни один процесс не мог быть блокирован во время выполнения операции `wakeup` в предыдущей модели.

Решение проблемы производителя и потребителя с помощью семафоров

Как показано в листинге 2.4, проблему потерянных сигналов запуска можно решить с помощью семафоров. Очень важно, чтобы они были реализованы неделимым образом. Стандартным способом является реализация операций `down` и `up` в виде системных запросов, с запретом ОС всех прерываний на период проверки семафора, изменения его значения и возможного перевода процесса в состояние ожидания. Поскольку для выполнения всех этих действий требуется всего лишь несколько команд CPU, запрет прерываний не приносит никакого вреда. Если используются несколько CPU, каждый семафор необходимо защитить переменной блокировки с использованием команды `TSL`, чтобы гарантировать одновременное обращение к семафору только одного CPU. Необходимо понимать, что использование команды `TSL` принципиально отличается от активного ожидания, при котором производитель или потребитель ждут наполнения или опустошения буфера. Операция с семафором займет несколько микросекунд, тогда как активное ожидание может затянуться на существенно больший промежуток времени.

Листинг 2. 4. Решение проблемы производителя и потребителя с помощью семафоров

```
#define N 100  //Количество сегментов в буфере
typedef int semaphore;          //семафоры – особый тип целочисленных переменных
semaphore mutex = 1;                    //Контроль доступа в критическую область
semaphore empty = N;                    //Число пустых сегментов буфера
semaphore full =0;                      //Число полных сегментов буфера
void producer(void) {
        int item;
        while (TRUE) {                  //TRUE - константа, равная 1
        item = produce item();          //Создать данные, помещаемые в буфер
        down(&empty);                   //Уменьшить счетчик пустых сегментов буфера
        down(&mutex);                   //Вход в критическую область
```

```
        insert item(item);                  //Поместить в буфер новый элемент
        up(&mutex);                         // Выход из критической области
        up(&full);                          //Увеличить счетчик полных сегментов буфера
}}
void consumer(void)
{
         int item;
         while (TRUE) {                     //Бесконечный цикл
         down(&full);                       //Уменьшить число полных сегментов буфера
         down(&mutex);                      //Вход в критическую область
         item = remove_item();              //Удалить элемент из буфера
         up(&mutex);                        //Выход из критической области
         up(&empty);                        //Увеличить счетчик пустых сегментов буфера
         consume item(item);                //Обработка элемента
}}
```

В представленном решении используются три семафора: один для подсчета заполненных сегментов буфера (`full`), другой для подсчета пустых сегментов (`empty`), а третий предназначен для исключения одновременного доступа к буферу производителя и потребителя (`mutex`). Значение счетчика `full` исходно равно нулю, счетчик `empty` равен числу сегментов в буфере, а `mutex` равен 1. Семафоры, исходное значение которых равно 1, используемые для исключения одновременного нахождения в критической области двух процессов, называются двоичными семафорами. Взаимное исключение обеспечивается, если каждый процесс выполняет операцию `down` перед входом в критическую область и `up` после выхода из нее.

В системах, использующих семафоры, естественным способом скрыть прерывание будет связать с каждым устройством ввода-вывода семафор, исходно равный нулю. Сразу после запуска устройства ввода-вывода управляющий процесс выполняет операцию `down` на соответствующем семафоре, тем самым входя в состояние блокировки. В случае прерывания обработчик прерывания выполняет `up` на соответствующем семафоре, переводя процесс в состояние готовности. В такой модели процедуры обработки прерываний следующий шаг заключается в выполнении `up` на семафоре устройства, чтобы следующим шагом планировщик смог запустить программу, управляющую устройством. Разумеется, если в этот момент несколько процессов находятся в состоянии готовности, планировщик может выбрать другой, более значимый процесс.

В примере, представленном в Листинг 2. 4, семафоры использовались двумя различными способами. Это различие достаточно значимо, чтобы сказать о нем особо. Семафор `mutex` используется для реализации взаимного исключения, то есть для исключения одновременного обращения к буферу и связанным переменным двух процессов.

Остальные семафоры использовались для синхронизации. Семафоры `full` и `empty` необходимы, чтобы гарантировать, что определенные последовательности событий происходят или не происходят. В нашем случае они гарантируют, что производитель прекращает работу, когда буфер полон, а потребитель прекращает работу, когда буфер пуст.

В ОС Windows для управления семафорами используются следующие функции:

- CreateSemaphore
- OpenSemaphore
- ReleaseSemaphore

Мьютексы

Иногда используется упрощенная версия семафора, называемая **мьютексом** (`mutex`, сокращение от mutual exclusion — взаимное исключение). `Mutex` не способен считать,

он может лишь управлять взаимным исключением доступа к совместно используемым ресурсам или кодам. Реализация mutex проста и эффективна, что делает использование mutex особенно полезным в случае потоков, действующих только в пространстве пользователя.

Mutex — переменная, которая может находиться в одном из двух состояний: блокированном или неблокированном. Поэтому для описания mutex требуется всего один бит, хотя чаще используется целая переменная, у которой 0 означает неблокированное состояние, а все остальные значения соответствуют блокированному состоянию. Значение мьютекса устанавливается двумя процедурами. Если поток (или процесс) собирается войти в критическую область, он вызывает процедуру mutex_lock. Если мьютекс не заблокирован (то есть вход в критическую область разрешен), запрос выполняется и вызывающий поток может попасть в критическую область.

Напротив, если мьютекс заблокирован, вызывающий поток блокируется до тех пор, пока другой поток, находящийся к критической области, не выйдет из нее, вызвав процедуру mutex_unlock. Если мьютекс блокирует несколько потоков, то из них случайным образом выбирается один.

Мьютексы легко реализовать в пользовательском пространстве, если доступна команда TSL. Код программы для процедур mutex_lock и mutex_unlock в случае потоков на уровне пользователя представлен в Листинг 2. 5.

Листинг 2. 5. Реализация mutex_lock и mutex_unlock

```
mutex_lock:
        TSL REGISTER.MUTEX |Старое значение мьютекста копируется в регистр:
                           устанавливается новое значение 1
        CMP REGISTER.#0    |Сравнение старого значения с нулем
        JZE ok             |Если старое значение было нулём, мьютекст был
                           блокирован. Возврат
        CALL thread_yield  |Мьютекст занят, управление передаётся другому потоку
        JMP mutex_lock     |Повторить попытку позже
ok: RET                    |Возврат, вход в критическую область

mutex_unlock:
        MOVE MUYEX.#0      |Устанавливается значение мьютекста 0
        RET                |Возврат
```

Процедура mutex_lock похожа на процедуру enter_region в Листинг 2. 2, но с одним существенным отличием. Если процедуре enter_region не удается войти в критическую область, она продолжает в цикле проверять наличие блокировки (активное ожидание). В конце концов, время, отведенное этому процессу, кончается, и планировщик передаст управление другому процессу. Раньше или позже процесс, заблокировавший вход в критическую область, освобождает его.

В случае потоков, ситуация кардинально меняется, поскольку нет прерываний по таймеру, останавливающих слишком долго работающие потоки. (**В современных ОС, например Windows, поток – единица планирования в планировщике, поэтому данное утверждение не распространяется на Windows**) Поток, пытающийся получить доступ к семафору и находящийся в состоянии активного ожидания, зациклится навсегда, поскольку он не позволит предоставить CPU другому потоку, желающему снять блокировку.

В этой ситуации mutex_lock ведет себя по-другому. Если войти в критическую область невозможно, mutex_lock вызовет thread_yeld, чтобы предоставить CPU другому потоку. Активного ожидания здесь нет. При следующем запуске поток снова проверит блокировку.

Поскольку вызов thread_yeld является всего лишь обращением к планировщику потоков в пространстве пользователя, он выполняется очень быстро. Следовательно, ни mutex_lock, ни mutex_unlock не требуют обращений к ядру. Синхронизация потоков

на уровне пользователя происходит полностью в пространстве пользователя, с применением процедур, состоящих всего из нескольких команд CPU.

Система мьютексов, является только скелетом набора запросов. Программное обеспечение часто требует реализации разнообразных возможностей, и примитивы синхронизации не являются исключением. Например, в некоторых реализациях пакета потоков поставляется вызов `mutex_trylock`, который либо предоставляет доступ к критической области, либо возвращает код ошибки, но в любом случае мгновенно возвращает управление, то есть не заставляет поток ждать. Этот запрос дает потоку возможность выбора в случае наличия альтернативы простому ожиданию.

Следует обратить внимание на случай: потокам в пользовательском пространстве нет проблемы доступа потоков к мьютексу, поскольку у всех потоков общее адресное пространство. Тем не менее, в большинстве предыдущих моделей ОС, в частности в алгоритме Петерсона и семафорах, молчаливо предполагалось, что несколько процессов имеют доступ к совместно используемому участку памяти, пусть содержащему одно слово. Если адресные пространства процессов несовместны, как мы постоянно утверждали, как они могут совместно использовать переменную `turn` в алгоритме Петерсона, или семафоры, или общий буфер?

На этот вопрос существует два ответа:

- Во-первых, некоторые из совместно используемых структур данных, скажем, семафоры (**именованные семафоры**), могут храниться в ядре с доступом только через системные запросы. Этот подход решает проблему.
- Во-вторых, большинство современных ОС (включая UNIX и Windows) предоставляют возможность совместного использования процессами некоторой части адресного пространства. В этом случае возможно разделение буфера и других структур данных. В крайнем случае, можно совместно использовать файл.

Если два или больше процессов разделяют частично или полностью адресные пространства, различие между процессами и потоками частично размывается, но тем не менее все равно остается. Два процесса с общим адресным пространством все равно обладают разными открытыми файлами, аварийными таймерами и прочими характеристиками, присущими процессам, в то время как два потока, разделяющие адресное пространство, разделяют и все остальное. И в любом случае несколько процессов, совместно использующих адресное пространство, никогда не будут столь же эффективны, как потоки на уровне пользователя, поскольку управление потоками всегда происходит через ядро.

В ОС Windows существуют следующие функции для управления мьютексами:

- CreateMutex
- OpenMutex
- ReleaseMutex

Мониторы

Межпроцессное взаимодействие с применением семафоров выглядит довольно просто, не правда ли? Взгляните внимательнее на порядок выполнения процедур `down` перед помещением или удалением элементов из буфера в листинге 2.4. Представьте себе, что две процедуры `down` в программе производителя поменялись местами, так что значение `mutex` было уменьшено раньше, чем `empty`. Если буфер был заполнен, производитель блокируется, установив `mutex` на 0. Соответственно, в следующий раз, когда потребитель обратится к буферу, он выполнит `down` с переменной `mutex`, равной 0, и тоже заблокируется. Оба процесса заблокированы навсегда.

Вышеизложенная ситуация показывает, с какой аккуратностью нужно обращаться с семафорами. Одна маленькая ошибка, и все останавливается. Поскольку такие ошибки приводят к абсолютно невоспроизводимым и непредсказуемым состояниям состязания, взаимоблокировкам и т. п.

Чтобы упростить написание программ, в 1974 году Хоар (Hoare) и Бринч Хансен (Brinch Hansen) предложили примитив синхронизации более высокого уровня, называемый монитором. Их предложения несколько отличались друг от друга, как мы увидим дальше. **Монитор** — набор процедур, переменных и других структур данных, объединенных в особый модуль или пакет. Процессы могут вызывать процедуры монитора, но у процедур, объявленных вне монитора, нет прямого доступа к внутренним структурам данных монитора.

Реализации взаимных исключений способствует важное свойство монитора: **при обращении к монитору в любой момент времени активным может быть только один процесс**. Мониторы являются структурным компонентом языка программирования, поэтому компилятор знает, что обрабатывать вызовы процедур монитора следует иначе, нем вызовы остальных процедур. Обычно при вызове процедуры монитора первые несколько команд процедуры проверяют, нет ли в мониторе активного процесса. Если активный процесс есть, вызывающему процессу придется подождать, в противном случае запрос удовлетворяется.

Реализация взаимного исключения зависит от компилятора, но обычно используется мьютекс или бинарный семафор. Поскольку взаимное исключение обеспечивает компилятор, а не программист, вероятность ошибки гораздо меньше. В любом случае программист, пишущий код монитора, не должен задумываться о том, как компилятор организует взаимное исключение. Достаточно знать, что, обеспечив попадание в критические области через процедуры монитора, можно не бояться попадания в критическую область двух процессов одновременно.

Хотя мониторы предоставляют простой способ реализации взаимного исключения, этого недостаточно. Необходим также способ блокировки процессов, которые не могут продолжать свою деятельность. В случае проблемы производителя и потребителя достаточно просто поместить все проверки буфера на наполненность и пустоту в процедуры монитора, но как процесс заблокируется, обнаружив полный буфер?

Решение заключается во введении переменных состояния и двух операций, `wait` и `signal`. Когда процедура монитора обнаруживает, что она не в состоянии продолжать работу (например, производитель выясняет, что буфер заполнен), она выполняет операцию `wait` на какой-либо переменной состояния, скажем, `full`. Это приводит к блокировке вызывающего процесса и позволяет другому процессу войти в монитор.

Другой процесс, в примере, потребитель может активизировать ожидающего напарника, например, выполнив операцию `signal` на той переменной состояния, на которой он был заблокирован. Чтобы в мониторе не оказалось двух активных процессов одновременно, нам необходимо правило, определяющее последствия операции `signal`:

- Хоар предложил запуск «разбуженного» процесса и остановку второго.
- Бринч Хансен предложил другое решение: процесс, выполнивший `signal`, должен немедленно покинуть монитор. Иными словами, операция `signal` выполняется только в самом конце процедуры монитора.

В дальнейшем будем использовать второе решение, поскольку оно в принципе проще и к тому же легче в реализации. Если операция `signal` выполнена на переменной, с которой связаны несколько заблокированных процессов, планировщик выбирает и «оживляет» только один из них.

- Кроме этого, существует третье решение, не основывающееся на предположениях Хоара и Бринча Хансена: позволить процессу, выполнившему `signal`, продолжать работу и запустить ждущий процесс только после того, как первый процесс покинет монитор.

Переменные состояния не являются счетчиками. В отличие от семафоров они не аккумулируют сигналы, чтобы впоследствии воспользоваться ими. Это означает, что в случае выполнения операции `signal` на переменной состояния, с которой не связано ни одного блокированного процесса, сигнал будет утерян. Проще говоря, операция `wait` должна выполняться прежде, чем `signal`. Это правило существенно упрощает реализацию. На практике это правило не создает проблем, поскольку отслеживать состояния процессов при необходимости не очень трудно. Процесс, который собирается выполнить `signal`, может оценить необходимость этого действия но значениям переменных:

В Листинг 2. 6 представлена схема решения проблемы производителя и потребителя с применением мониторов, написанная на языке Java. В каждый момент времени активна только одна процедура монитора. Буфер состоит из `N` сегментов. Решение состоит из четырех классов. Внешний класс, `ProducerConsumer`, создает и запускает два потока. Второй и третий классы, `producer` и `consumer` соответственно, содержат программы производителя и потребителя. Класс `out_monitor` является монитором. Он содержит два синхронизированных потока, используемых для текущего помещения элементов в буфер и извлечения их оттуда. В отличие от предыдущих примеров, здесь приведен полный текст программ `insert` и `remove`.

Листинг 2. 6. Решение проблемы производителя и потребителя на Java

```
public class ProducerConsumer {
        static final int N=100;  //Константа, задающая размер буфера
        static producer p=new producer(); //создать экземпляр потока производителя
        static consumer c=new comsumer(); //создать экземпляр потока потребителя
        static our_monitor mon = new our_monitor(); //создать экземпляр монитора

        public static void main(String args[])  {
              p.start(): //запуск потока производителя
              c.start(): .//запуск потока потребителя
        }
        static class producer extends Thread  {
           public void run()  { //метод rub содержит программу потока
                int item;
                while (true)  { //цикл производителя
                item = produce_item();
                mon.insert(item);
                }
           }
             private int produce_item() {…} //собственное производство
         }
        static class consumer extends Theread  {
           public void run()  { //метод содержит программу потока
                int item;
                while (true)  { //цикл потребителя
                item =mon.remove();
                consumer_item(item);
                }
           }
         private void consume_item(int item)  {…} //собственное потребление
         }
         static class our_monitor  { //монитор
                 private int buffer[]=new int[N];
                 private int count=0. lo = 0. his 0; //счетчики и индексы
                 public synchronized void insert(int val)  {
                       if(count==N) go_to_sleep(); //если буфер полон, уйти в состояния
                                                   ожидания
                       buffer [hi] = val; //поместить элемент в буфер
                       hi = (hi+1)%N; //следующий сегмент, в котором будет помещён
                                      элемент
                       count = count+1; //теперь в буфере на один элемент больше
                       if(count==1) notify(); //если потребитель в состоянии ожидания,
                                              активировать его
```

```
                }
                  public synchronized int remove()  {
                        int val;
                         if(count==0) go_tosleep(); //если буфер пуст, уйти в состояние
                                                    ожидания
                         val = buffer [lo]; //забрать элемент из буфера
                         lo=(lo+1)%N; //следующий сегмент, из которого заберут
                                      элемент
                         count = count -1; //теперь в буфере на 1 элемент меньше
                         if(count==N-1) nofity(); //если производитель в состоянии
                                                  ожидания , активировать его
                         return val;
                  }
            private void go_to_sleep()  {try{wait():} catch(InterruptedException exc)  {}: }
            }
}
```

Потоки производителя и потребителя функционально идентичны соответствующим частям программы предыдущих примеров. В программе производителя есть бесконечный цикл формирования данных и помещения их в общий буфер. В коде потребителя есть бесконечный цикл с изъятием данных из общего буфера и их обработкой.

Добавление в описание метода ключевого слова **synchronized** гарантирует, что если хотя бы один поток начал выполнение этого метода, ни один другой поток не сможет выполнять другой синхронизированный (определенный как synchronized) метод из этого класса.

Интерес для нас представляет класс `out_monitor`, содержащий буфер, переменные администрирования и два метода синхронизации. Когда производитель активен в процедуре insert, потребитель не может быть активным в процедуре remove, что исключает состояние состязания. Переменная count отслеживает количество элементов в буфере, принимая значения от 0 до `N-1`. Переменная `lo` является индексом следующего сегмента буфера, из которого следует извлечь данные. Переменная `hi` является индексом следующего сегмента буфера, в который следует поместить данные. Разрешена ситуация, в которой `lo = hi`, что означает 0 или `N` элементов в буфере. Различать эти два случая можно по переменной `count`.

Синхронизированные методы в языке Java отличаются от стандартных мониторов отсутствием переменных состояния. Взамен предлагаются две процедуры, `wait` и `notify`, которые аналогичны `sleep` и `wakeup` с той лишь разницей, что они используются в синхронизированных методах, а это исключает состояния состязания. Теоретически процедура может быть прервана, для чего и служит весь окружающий се набор программ. Java требует, чтобы исключения обрабатывались явно. В данном случае просто представьте, что `go_to_sleep` описывает уход в состояние ожидания.

Благодаря автоматизации взаимного исключения применение мониторов сделало параллельное программирование значительно менее подверженным ошибкам, чем применение семафоров. Но и у мониторов тоже есть свои недостатки.

Во многих языках программирования, таких как C/C++, отсутствует поддержка мониторов – необходимо самостоятельно разрабатывать.

В .NET Framework существует альтернативный подход – использование статического класса `Monitor`, который обладает методами:

- Enter
- TryEnter
- Pulse (Signal)
- PulseAll
- Exit

Другая проблема, связанная с мониторами и семафорами, состоит в том, что они были разработаны для решения задачи взаимного исключения в системе с одним или несколькими CPU, имеющими доступ к общей памяти. Помещение семафоров в разделенную память с защитой в виде команд `TSL` может исключить состояния состязания. Эти примитивы будут неприменимы в распределенной системе, состоящей из нескольких CPU с собственной памятью у каждого, связанных локальной сетью. Вывод из всего вышесказанного следующий: семафоры являются примитивами слишком низкого уровня, а мониторы могут использоваться только в некоторых языках программирования. Примитивы не подходят и для реализации обмена информацией между компьютерами — нужно что-то другое.

Задание для самостоятельного изучения

Изучить особенности синхронизации в распределенных системах:

- Передача сообщений.
  - Разработка системы передачи сообщений.
  - Решение проблемы производителя и потребителя с помощью сообщений.
- Барьеры.

Классические проблемы межпроцессного взаимодействия:

- Проблема обедающих философов.
- Проблема читателей и писателей.
- Проблема спящего брадобрея.

За основу следует взять источник [1, 4].

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. А. Шоу. Логическое проектирование операционных систем. Пер. с англ. –М.: Мир, 1981. –360 с.

3. С. Кейслер. Проектирование операционных систем для малых ЭВМ: Пер. с англ. –М.: Мир, 1986. –680 с.

4. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

5. Microsoft Development Network. URL: http://msdn.com

Приложение А. Мультипроцессорная обработка

Введение

CPU i486 поддерживает multi-CPU работу по системной шине. CPU, работающие на системную шину, должны иметь различные полосы пропускания шины.

Multi-CPU обработка позволяет улучшить некоторые аспекты быстродействия системы. Например, система компьютерной графики может использовать CPU i860(TM) специально для быстрой обработки растровых образов, в то время как CPU i486 будет поддерживать стандартную ОС, например UNIX или Windows.

Multi-CPU системы чувствительны к двум аспектам конструкции:

- Организация непротиворечивого кеширования - Когда один CPU выполняет доступ к данным, кешируемым в другом CPU, он не должен получить неверные данные. Если CPU модифицирует данные, то все прочие обращающиеся к этим данным CPU должны получать модифицированные данные.
- Надежная связь - CPU должны иметь между собой связь, исключающую недопустимые взаимные помехи при доступе более чем одного CPU к одной и той же области памяти.

Циклы шины с захватом и псевдо-захватом шины

Хотя архитектура мульти-CPU систем может сильно отличаться, в целом все они нуждаются в надежной связи с памятью. CPU в CPU, например, обновления бита

Доступа в дескрипторе сегмента должен исключить аналогичные попытки всех прочих CPU, до тех пор, пока операция не завершится.

Также требуется надежная связь с прочими CPU. Хозяева шины должны иметь возможность надежного обмена данными. Например, бит в памяти может разделяться несколькими хозяевами шины и использоваться как сигнал того, что некоторые ресурсы, такие как переферийное устройство, находятся в состоянии ожидания. Хозяин шины может проверить этот бит, увидеть, что ресурс свободен, и изменить состояние бита. Это состояние будет указывать другим потенциальным хозяевам шины, что ресурс используется. Проблема может возникнуть в том случае, когда другой хозяин шины прочитывает этот бит в промежутке времени между тем, как первый хозяин шины прочитал бит, и моментом изменения состояния этого бита. В этом случае оба потенциальных хозяина шины будут считать, что ресурс свободен. При одновременной попытке использовать ресурс они могут повлиять друг на друга недопустимым образом.

CPU предотвращает такие ситуации, поддерживая циклы шины с захватом; во время таких циклов запросы на управление шиной игнорируются.

CPU i486 защищает целостность некоторых критических операций с памятью, воздействуя на выходной сигнал LOCK#. Чтения и записи 64-разрядных операндов и (128-разрядные) предварительные выборки команд защищаются выходом, который называется PLOCK#. За использование этих сигналов для управления доступом к памяти среди CPU отвечает разработчик аппаратного обеспечения.

CPU автоматически устанавливает один из этих сигналов во время некоторых критических операций с памятью. ПО задавать дополнительные операции с памятью, для которых требуется сигнал LOCK#.

В число средств интерфейса multi-CPU обработки общего назначения входят:

- Сигнал LOCK#, появляющийся на штырьке CPU.
- Сигнал PLOCK#, появляющийся на штырьке CPU.
- Префикс команд LOCK, позволяющий установку сигнала LOCK# ПО.
- Автоматическая установка сигнала LOCK# для некоторых видов операций с памятью.
- Автоматическая установка сигнала PLOCK# для некоторых других видов операций с памятью.