НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Курс «Современные операционные системы»

Лекция 6

Взаимодействие процессов

Москва
2016

Содержание

1.8. Передача сообщений

1.8.1. Проблемы разработки систем передачи сообщений.

1.8.2. Передача сообщений для решения задачи производителя-потребителя.

1. Межпроцессное взаимодействие

Межпроцессное взаимодействие (*InterProcess Communication* (***IPC***)) решает три основных вопроса:

1. Передача информации от одного процесса другому.
2. Совместная работа процессов без создания взаимных помех.
3. Определения правильной последовательности выполнения взаимозависимых процессов.

1.1. Состязательная ситуация

Пример. Спулер печати.

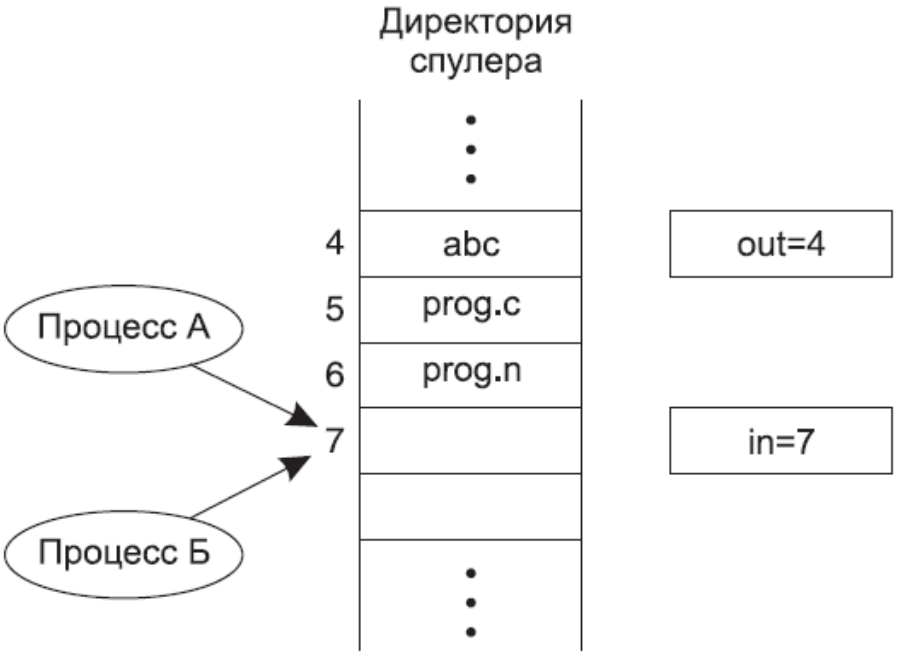
Когда процессу необходимо распечатать файл, он помещает имя этого файла в директорию спулера печати. Процесс демон принтера периодически проверяет наличие файлов для печати и, если такие файлы имеются, распечатывает их и удаляет их имена из директории.

В директории спулера имеются области памяти с номерами 0, 1, 2..., в каждой из которых может храниться имя файла. Имеются две общие переменные, которые хранятся в файле, доступном всем процессам:

`out` – следующий файл для печати,

`in` – следующая свободная область в директории.

В какой-то момент времени области от 0 до 3 пустуют (файлы уже распечатаны). Почти одновременно процессы А и Б решают, что им нужно поставить файл в очередь на печать.



Процесс А считывает значение переменной `in` и сохраняет значение 7 в локальной переменной `next_free_slot`. Сразу же после этого происходит прерывание по таймеру, планировщик процессора решает, что процесс А проработал достаточно долго, и переключается на выполнение процесса Б.

Процесс Б также считывает значение переменной `in` и также получает число 7. Он также сохраняет его в своей локальной переменной `next_free_slot`. К текущему моменту оба процесса полагают, что следующей доступной областью будет 7. Процесс Б продолжает выполняться. Он сохраняет имя своего файла в области 7 и присваивает переменной `in` обновленное значение 8. Затем он переходит к выполнению каких-нибудь других действий.

Через некоторое время выполнение процесса А возобновляется с того места, где он был остановлен. Он считывает значение переменной `next_free_slot`, видит там число 7 и записывает имя своего файла в область 7, затирая то имя файла, которое только что было в него помещено процессом Б. Затем он вычисляет `next_free_slot+1`, получает значение 8 и присваивает его переменной `in`.

В каталоге спулера нет внутренних противоречий, поэтому демон печати не заметит никаких нестыковок, но процесс Б никогда не получит вывода на печать.

Эта ***состязательная ситуация*** произошла из-за того, что процесс Б стал использовать общие переменные еще до того, как процесс А завершил работу с ними.

При отладке программы, в которой присутствует состязательная ситуация, результаты большинства прогонов могут не выявить описанного случая, который будет возникать довольно редко.

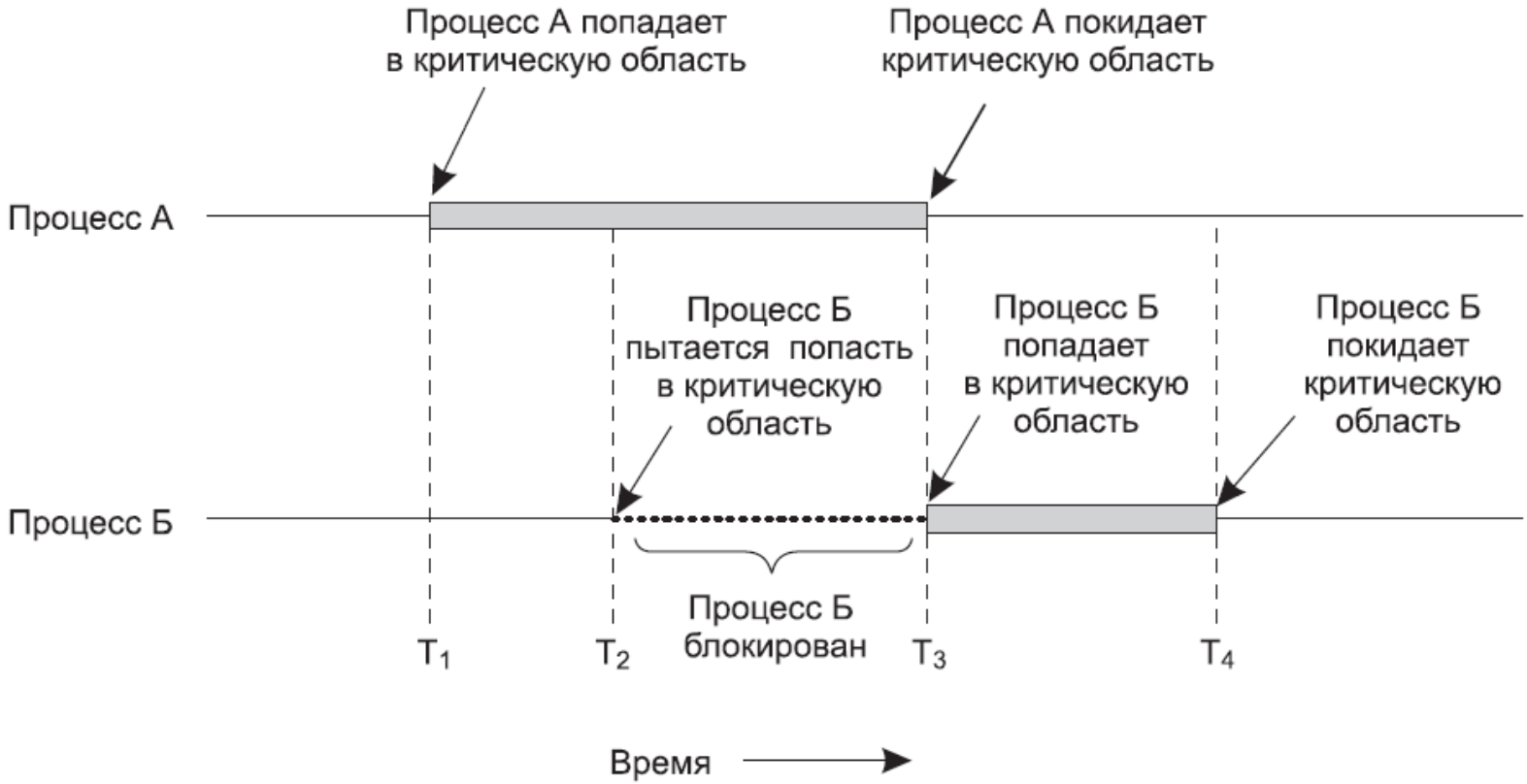
1.2. Критические области

Процесс во время своего выполнения совершает действия, приводящие к состязательным ситуациям, при этом та часть программы, в которой используется доступ к общей памяти, называется ***критической областью***.

Избежать состязательных ситуаций можно, если никакие два процесса не будут одновременно находиться в своих критических областях. Т.е. если в каждый конкретный момент времени доступ к общим данным для чтения и записи может получить только один процесс. Это достигается с помощью ***взаимного исключения*** критических областей – способа, при котором если общие данные или файл используются одним процессом, возможность их использования всеми другими процессами исключается.

Чтобы процессы правильно выстраивали совместную работу и эффективно использовали общие данные, необходимо соблюдение четырех условий:

1. Два процесса не могут одновременно находиться в своих критических областях.
2. Не должны выстраиваться никакие предположения по поводу скорости или количества центральных процессоров.
3. Никакие процессы, выполняемые за пределами своих критических областей, не могут блокироваться любым другим процессом.
4. Процессы не должны находиться в вечном ожидании входа в свои критические области.



Взаимное исключение использования критических областей

1.3. Взаимное исключение с активным ожиданием

1.3.1. Запрещение прерываний.

В однопроцессорных системах простейшим решением является запрещение всех прерываний каждым процессом сразу после входа в свою критическую область и их разрешение после выхода из нее. Однако если один из процессов выключил и не включил прерывания, это может вызвать крах всей системы.

Запрещение прерываний – очень удобное средство для самого ядра, когда оно обновляет переменные или списки. Например, когда прерывание происходит в момент изменения состояния списка готовых процессов, может сложиться состязательная ситуация. Поэтому запрещение прерываний является полезной технологией внутри самой ОС, но не подходит в качестве механизма взаимных блокировок для пользовательских процессов.

Например, для семейства Intel x86 имеются низкоуровневые команды (ассемблер):

```
CLI    | Запрет всех прерываний.
STI    | Разрешение прерываний.
```

Команда `CLI` (**Cl**ear **I**nterrupt-Enable Flag) сбрасывает флаг `IF` (**I**nterrupt **F**lag) в регистре `EFLAGS`, что запрещает процессору обрабатывать все прерывания (кроме немаскируемых, NMI – Non-Maskable Interrupts). Команда `STI` (**Se**t **I**nterrupt-Enable Flag) устанавливает флаг `IF`, что разрешает процессору обрабатывать прерывания.

В многопроцессорной системой запрещение прерываний действует только на один процессор, все остальные процессоры продолжат свою работу и смогут обращаться к общей памяти.

1.3.2. Блокирующие переменные.

Если использовать общую ***блокирующую переменную***, нулевое значение которой показывает, что ни один из процессов не находится в своей критической области, а единица — что какой-то процесс находится в своей критической области.

Исходное значение такой блокирующей переменной равно 0. Когда процессу требуется войти в свою критическую область, сначала он проверяет значение блокирующей переменной. Если оно равно 0, процесс устанавливает его в 1 и входит в критическую область. Если значение уже равно 1, процесс ожидает, пока оно не станет равно 0.

К сожалению, использование блокирующей переменной (также как в примере с каталогом спулера) не позволяет избежать возникновения состязательной ситуации.

Предположим, что процесс А считывает значение блокирующей переменной и видит, что оно равно 0. Перед тем как он сможет установить значение в 1, планировщик запускает другой процесс Б, который устанавливает значение в 1. Когда возобновляется выполнение процесса А, он также установит значение блокирующей переменной в 1, и два процесса одновременно окажутся в своих критических областях.

1.3.3. Циклическая блокировка (спинлок).

Рассмотрим следующее решение проблемы критической области.

Процесс-0	Процесс-1

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

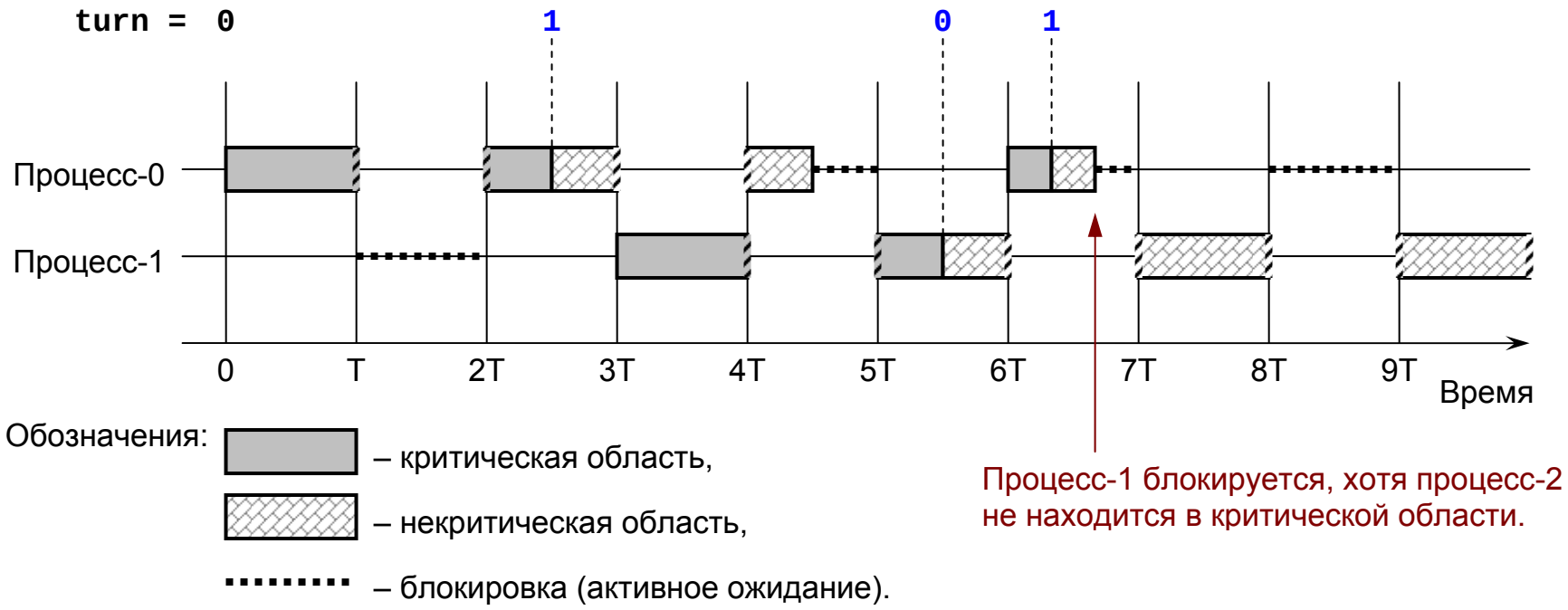
Изначально общая целочисленная переменная `turn` равна `0`. Она отслеживает, чья настала очередь входить в критическую область и проверять или обновлять общую память.

Сначала процесс-0 проверяет значение `turn`, определяет, что оно равно `0`, и входит в свою критическую область. Процесс-1 также определяет, что значение этой переменной равно `0`, из-за чего находится в коротком цикле, постоянно проверяя, когда `turn` получит значение `1`.

Постоянная проверка значения переменной, пока она не приобретет какое-нибудь значение, называется ***активным ожиданием***. Обычно его следует избегать, поскольку оно тратит впустую процессорное время. Активное ожидание используется только в том случае, если оно будет недолгим.

Блокировка, использующая активное ожидание, называется ***циклической блокировкой (спин-блокировкой, спинлок)***.

Алгоритм спин-блокировки позволяет предотвращать любые состязательные ситуации, но **нарушает третье условие**: процесс оказывается заблокированным тем процессом, который не находится в своей критической области. Это станет заметно, когда один процесс работает существенно медленнее другого.



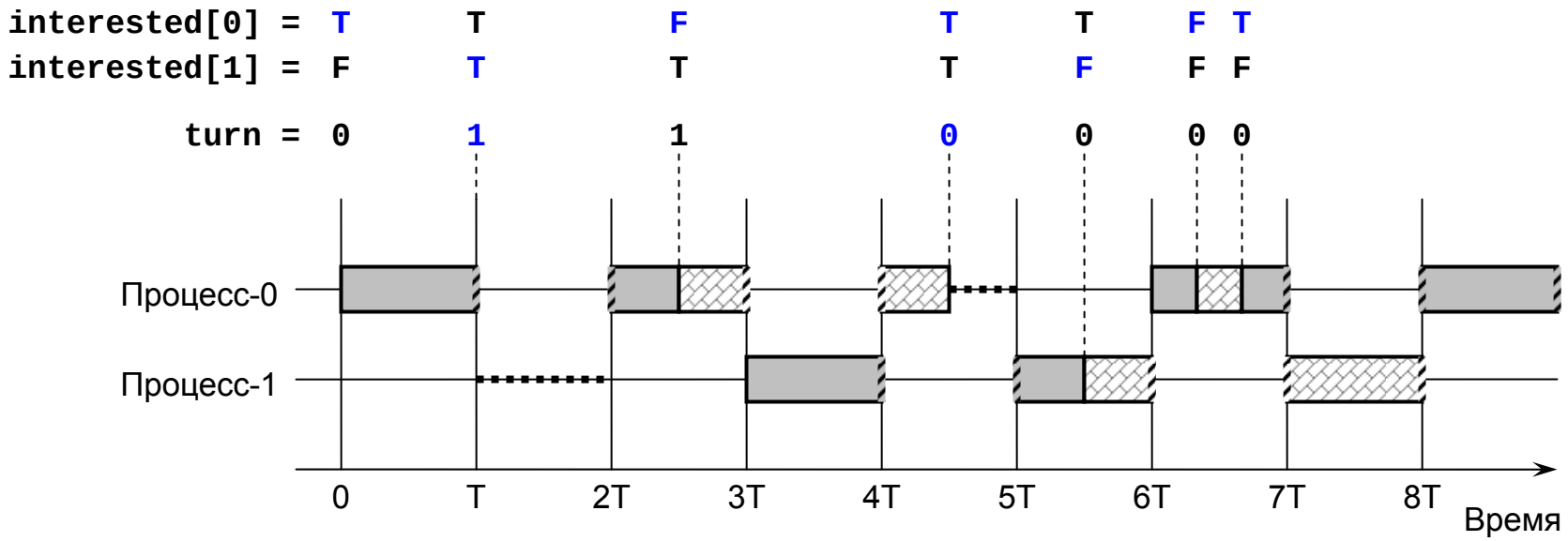
1.3.4. Алгоритм Петерсона.

Решение Петерсона (предложенное в 1981 г.), позволяющее добиться взаимного исключения не нарушая **третье условие**:

```
#define FALSE 0
#define TRUE 1
#define N 2                          /* количество процессов */
int turn;                            /* чья очередь? */
int interested[N]={FALSE,FALSE}; /* все исходные значения равны 0 (FALSE) */
void enter_region(int process) { /* process имеет значение 0 или 1 */
     int other;                     /* номер другого процесса */
     other = 1 - process;           /* противостоящий процесс */
     interested[process] = TRUE; /* демонстрация заинтересованности */
     turn = process;                /* установка флажка */
     while (turn==process && interested[other]==TRUE); /* пустой цикл */
}
void leave_region(int process) { /* процесс, покидающий критическую область */
     interested[process] = FALSE; /* признак выхода из критической области */
}
```

Перед входом в свою критическую область каждый процесс вызывает функцию `enter_region()`, передавая ей в качестве аргумента свой собственный номер процесса (`0` или `1`). Этот вызов заставляет процесс ждать, если потребуется, безопасного входа в критическую область.

После завершения работы в критической области процесс, чтобы показать это и разрешить вход в критическую область другому процессу, вызывает функцию `leave_region()`.


interested[0] = T T F T T F T
interested[1] = F T T T F F F
turn = 0 1 1 0 0 0 0
Процесс-0
Процесс-1
0
T
2T
3T
4T
5T
6T
7T
8T
Время

1.3.5. Низкоуровневая команда TSL.

Некоторые компьютеры располагают низкоуровневой командой (ассемблер)

TSL `RX,LOCK`

(`TSL` — Test and Set Lock — проверь и установи блокировку), которая считывает содержимое слова памяти `lock` в регистр `RX`, а по адресу памяти, отведенному для `lock`, записывает ненулевое значение.

При этом гарантируются ***неделимость (атомарность)*** операций чтения слова и сохранение в нем нового значения — никакой другой процесс не может получить доступ к слову в памяти, пока команда не завершит свою работу полностью. Процессор, выполняющий команду `TSL`, блокирует шину памяти, запрещая другим процессорам доступ к памяти до тех пор, пока не закончится выполнение этой команды. Для этого блокировка шины должна поддерживаться оборудованием на аппаратном уровне.

Команду `TSL` можно использовать для предотвращения одновременного входа двух процессов в их критические области (программа написана на вымышленном типовом языке ассемблера):

```
enter_region:
    TSL REGISTER,LOCK       | Копирование lock в регистр с присвоением ей 1.
    CMP REGISTER,#0         | Было ли значение lock нулевым?
    JNE enter_region        | Если оно было ненулевым, значит, блокировка
                            | уже установлена и нужно войти в цикл.
    RET                     | Возврат управления вызывающей программе;
                            | вход в критическую область осуществлен.
leave_region:
    MOVE LOCK,#0            | Присвоение переменной lock нулевого значения.
    RET                     | Возврат управления вызывающей программе.
```

Команда `CMP` сравнивает значение в регистре `REGISTER` с нулем (`#0`). Если они равны, то регистр флагов процессора `ZF` (**Z**ero **F**lag, для архитектуры Intel x86) принимает значение `1`, если не равны, то – `0`. Команда условного перехода `JNE` (**J**ump if **N**ot **E**qual) проверяет регистр флагов `ZF` и, если он равен `0` (т.е. сравниваемые значения не равны), переходит к указанной метке `enter_region`.

Как альтернатива команде `TSL` для низкоуровневой синхронизации всеми процессорами семейства Intel x86 используется команда `XCHG` (E**xch**an**g**e Register/Memory with Register):

```
enter_region:
    MOVE REGISTER,#1        | Помещение 1 в регистр.
    XCHG REGISTER,LOCK      | Обмен содержимого регистра и переменной lock.
    CMP REGISTER,#0         | Было ли значение lock нулевым?
    JNE enter_region        | Если оно было ненулевым, значит, блокировка
                            | уже установлена и нужно войти в цикл.
    RET                     | Возврат управления вызывающей программе;
                            | вход в критическую область осуществлен.
leave_region:
    MOVE LOCK,#0            | Присвоение переменной lock нулевого значения.
    RET                     | Возврат управления вызывающей программе.
```

1.3.6. Проблема инверсии приоритета.

Основной недостаток рассмотренных способов взаимного исключения — необходимость пребывания в режиме **активного ожидания**, когда процесс входит в короткий цикл, ожидая разрешения на выполнение.

Это не только приводит к **пустой трате процессорного времени**, но и может иметь проблемы при планировании выполнения процессов **с разной степенью приоритета**.

Рассмотрим два процесса: H – с высокой степенью приоритета и L – с низкой степенью приоритета. Правила планирования их работы предусматривают, что H выполняется сразу же после входа в состояние готовности.

В определенный момент, когда L находится в критической области, H входит в состояние готовности. Тогда H переходит в режим активного ожидания. Но поскольку, пока выполняется процесс H, выполнение L не планируется, у L не остается шансов выйти из своей критической области, поэтому H пребывает в **бесконечном цикле**.

Подобную ситуацию обычно называют ***проблемой инверсии приоритета***.

1.4. Приостановка и активизация

Другой способ взаимодействия процессов, которые (вместо напрасной траты процессорного времени) блокируют свою работу, пока им не разрешается входить в критическую область.

Реализуется этот способ с помощью пары системных вызовов `sleep` и `wakeup`. Системный вызов `sleep` блокирует вызывающий его процесс, который приостанавливается до тех пор, пока его не активизирует другой процесс. Активизирующий вызов `wakeup` использует один аргумент — активизируемый процесс. Дополнительно и `sleep` и `wakeup` используют еще один аргумент — адрес памяти, используемой для соотнесения вызовов `sleep` с вызовами `wakeup`.

1.4.1. Задача производителя-потребителя.

Формулировка задачи ***производителя и потребителя*** (или задачи ***ограниченного буфера***). Два процесса используют общий буфер фиксированного размера. Один из них, производитель, помещает информацию в буфер, а другой, потребитель, извлекает ее оттуда. (В общем случае можно рассматривать m производителей и n потребителей).

Когда производителю требуется поместить новую запись в уже заполненный буфер, производитель блокируется до тех пор, пока потребитель не извлечет как минимум одну запись. Также, если потребителю нужно извлечь запись из буфера, а буфер пуст, то потребитель блокируется до тех пор, пока производитель не поместит запись в буфер и не активизирует этого потребителя.

В приведенном ниже примере программы используются вызовы `sleep` и `wakeup` в виде библиотечных процедур, хотя они не являются частью стандартной библиотеки С. Процедуры `insert_item` и `remove_item` занимаются помещением записей в буфер и извлечением их оттуда.

В таком решении задачи производителя и потребителя может сложиться следующая **фатальная состязательная ситуация**.

Буфер пуст, и потребитель только что считал показания `count`, значение которого равно `0`. В этот момент планировщик временно приостанавливает процесс потребителя и возобновляет процесс производителя. Производитель помещает запись в буфер, увеличивает значение счетчика `count` (оно становится равно `1`). Поскольку только что счетчик имел нулевое значение, при котором потребитель должен находиться в заблокированном состоянии, производитель вызывает процедуру `wakeup`, чтобы активизировать выполнение процесса потребителя.

Но потребитель не находился в бездействующем состоянии, поэтому сигнал на активизацию будет утрачен. Когда подойдет очередь возобновить выполнение процесса потребителя, он проверит ранее считанное значение счетчика (равное 0) и перейдет в заблокированное состояние.

Рано или поздно производитель заполнит буфер и тоже перейдет в заблокированное состояние. И оба процесса попадут в **бесконечную блокировку**.

```
#define N 100   /* количество мест для записей в буфере */
int count = 0;  /* счетчик текущего количества записей в буфере */
void producer(void) { /* ПРОИЗВОДИТЕЛЬ */
   int item;
   while (TRUE) {
        item = produce_item(); /* производство новой записи */
        if (count==N) sleep(); /* если буфер полон, заблокироваться */
        insert_item(item);     /* добавление записи в буфер */
        count = count+1;       /* увеличение счетчика записей */
        if (count==1) wakeup(consumer); /* если буфер был пуст, активизация потребителя */
   }
}
void consumer(void) { /* ПОТРЕБИТЕЛЬ */
   int item;
   while (TRUE) {
        if (count==0) sleep(); /* если буфер пуст, заблокироваться */
        item = remove_item();  /* извлечение записи из буфера */
        count = count-1;       /* уменьшение счетчика записей */
        if (count==N-1) wakeup(producer); /* если буфер был полон, активизация производителя */
        consume_item(item); /* потребление имеющейся записи */
   }
}
```

Суть проблемы: **сигнал активизации, пришедший к процессу, не находящемуся в состоянии самоблокировки, пропадает.**

Быстро устранить проблему позволит добавление ***бита ожидания активизации***, который сохраняет сигналы активизации. Этот бит устанавливается, когда в отношении процесса, который не находится в состоянии блокировки, вызывается процедура `wakeup`. Затем, когда процесс попытается заблокироваться при установленном бите ожидания активизации, этот бит снимается, но процесс не блокируется.

Однако если в задаче имеются **три и более** процессов, то использование даже нескольких битов ожидания активизации **не решает проблему** возникновения состязания.

1.5. Семафоры

Дейкстра в 1965 г. предложил использовать ***семафор*** (semaphore) – целочисленную переменную для подсчета количества активизаций, отложенных на будущее.

Нулевое значение семафора свидетельствует об отсутствии сохраненных активизаций, положительное значение – ожидается не менее одной активизации.

Две операции с семафорами `down` и `up` обобщают работу `sleep` и `wakeup` соответственно.

Операция `down` проверяет, если значение семафора отличается от `0`, то оно уменьшает его на `1` (т.е. использует одну сохраненную активизацию) и продолжает свою работу. Если значение равно `0`, процесс приостанавливается, не завершая операцию `down`.

Проверка значения, его изменение и возможная приостановка процесса осуществляются как ***единое и неделимое (атомарное)*** действие. С началом семафорной операции никакой другой процесс не может получить доступ к семафору до тех пор, пока операция не будет завершена или заблокирована.

Операция `up` сначала увеличивает значение семафора на `1`. Затем, если предыдущее значение семафора было равно `0`, т.е. с семафором связаны один или более приостановленных процессов, способных завершить ранее начатые операции `down`, то выбирается один этих процессов (например, произвольным образом) и ему позволяется завершить его операцию `down`. Таким образом, после применения операции `up` в отношении семафора, с которым были связаны приостановленные процессы, значение семафора так и останется нулевым, но количество приостановленных процессов уменьшится на `1`. Операция увеличения значения семафора на `1` и активизации одного из процессов также является атомарной.

1.5.1. Решение задачи производителя-потребителя с помощью семафоров.

```
#define N 100                   /* количество мест для записей в буфере */
typedef int semaphore;          /* семафор — целочисленная переменная */
semaphore mutex = 1;            /* управляет доступом к критической области */
semaphore empty = N;            /* подсчитывает пустые места в буфере */
semaphore full = 0;             /* подсчитывает занятые места в буфере */
void producer(void) { /* ПРОИЗВОДИТЕЛЬ */
   int item;
   while (TRUE) {
        item = produce_item(); /* производство новой записи */
        down(&empty);          /* уменьшение счетчика пустых мест */
        down(&mutex);          /* вход в критическую область */
        insert_item(item);     /* добавление записи в буфер */
        up(&mutex);            /* выход из критической области */
        up(&full);             /* увеличение счетчика занятых мест */
   }
}
void consumer(void) { /* ПОТРЕБИТЕЛЬ */
   int item;
   while (TRUE) {

        down(&full);           /* уменьшение счетчика занятых мест */
        down(&mutex);          /* вход в критическую область */
        item = remove_item();  /* извлечение записи из буфера */
        up(&mutex);            /* выход из критической области */
        up(&empty);            /* увеличение счетчика пустых мест */
        consume_item(item);    /* потребление имеющейся записи */
   }
}
```

Атомарные операции `up` и `down` реализуются в виде системных вызовов с кратковременным запретом всех прерываний.

Если используются несколько процессоров, каждый семафор должен быть защищен переменной `lock`. Использование низкоуровневых команд `TSL` (или `XCHG`) гарантирует, что семафор в каждый момент времени задействуется только одним процессором.

В результате операция работы с семафором занимает лишь несколько микросекунд, а ожидание производителя или потребителя могло быть сколь угодно долгим.

Двоичный семафор `mutex` (от **mut**ual **ex**clusion – взаимное исключение) используется для организации взаимного исключения. Он гарантирует, что в каждый момент времени к буферу и соответствующим переменным имеет доступ только один процесс.

Семафоры `full` и `empty` используются для синхронизации. В данном случае они гарантируют, что производитель приостановит свою работу при заполненном буфере, а потребитель приостановит свою работу, если этот буфер опустеет.

1.6. Мьютексы

Мьютекс (упрощенная версия семафора) — это совместно используемая переменная, которая может находиться в одном из двух состояний: заблокированном или незаблокированном. На практике используется целое число: нуль означает незаблокированное, а все остальные значения — заблокированное состояние.

Мьютексы справляются лишь с управлением взаимным исключением доступа к общим ресурсам. Они особенно полезны для совокупности потоков, целиком реализованных в пользовательском пространстве.

Процедуры работы с мьютексами. Для работы с мьютексами используются две процедуры.

Когда потоку (или процессу) необходим доступ к критической области, он вызывает процедуру `mutex_lock`. Если мьютекс находится в незаблокированном состоянии (означающем доступность входа в критическую область), то вызов проходит удачно и вызывающий поток может войти в свою критическую область.

Если мьютекс заблокирован, то вызывающий поток блокируется до тех пор, пока поток, находящийся в критической области, не завершит свою работу и не вызовет процедуру `mutex_unlock`. Если на мьютексе заблокировано несколько потоков, то произвольно выбирается один из них.

```
mutex_lock:
     TSL REGISTER,MUTEX | Копирование мьютекса в регистр и установка его в 1.
     CMP REGISTER,#0    | Был ли мьютекс нулевым?
     JE ok              | Если мьютекс был нулевым, значит, не был заблокирован,
                        | поэтому нужно вернуть управление вызывающей программе.
     CALL thread_yield  | Мьютекс занят; пусть планировщик
                        | возобновит работу другого потока.
     JMP mutex_lock     | Повторная попытка.
ok:
     RET                | Возврат управления вызывающей программе;
                        | будет осуществлен вход в критическую область.
mutex_unlock:
     MOVE MUTEX,#0      | Сохранение в мьютексе значения 0.
     RET                | Возврат управления вызывающей программе.
```

Команда условного перехода `JE` (**J**ump if **E**qual) проверяет регистр флагов `ZF` и, если он равен `1` (т.е. сравниваемые значения равны), переходит к указанной метке `ok`. Команда `JMP` (**Jump**) безусловный переход к метке `mutex_lock`. Команда `CALL` передает управление процедуре `thread_yield` с запоминанием точки возврата.

Отличие мьютексов от предыдущего решения. Код процедуры `mutex_lock` похож на код `enter_region`, но с одной существенной разницей.

Когда процедуре `enter_region` не удается войти в критическую область, она продолжает повторное тестирование значения переменной `lock` (выполняет активное ожидание). По истечении определенного времени планировщик возобновляет работу какого-нибудь другого процесса. Рано или поздно возобновляется работа процесса, удерживающего блокировку, и он ее освобождает.

При работе `mutex_lock` с потоками (в пользовательском пространстве) отсутствует таймер, останавливающий работу слишком долго выполняющегося процесса. Поэтому поток, пытающийся воспользоваться блокировкой, находясь в состоянии активного ожидания, войдет в бесконечный цикл и

никогда не завладеет блокировкой. Поскольку он никогда не позволит никакому другому потоку возобновить выполнение и снять блокировку.

Как избежать зацикливания.

Когда процедура `mutex_lock` не может завладеть блокировкой, она вызывает процедуру `thread_yield`, чтобы уступить процессор другому потоку. Активное ожидание отсутствует. Поскольку процедура `thread_yield` представляет собой всего лишь вызов планировщика потоков в пользовательском пространстве, она работает очень быстро. Следовательно, ни `mutex_lock`, ни `mutex_unlock` не требуют никаких вызовов ядра.

Благодаря их использованию потоки, работающие на пользовательском уровне, могут синхронизироваться целиком в пространстве пользователя с использованием процедур, для реализации которых требуется совсем небольшая группа команд.

Иногда совокупности потоков предлагается вызов процедуры `mutex_trylock`, которая либо овладевает блокировкой, либо возвращает код отказа, но не вызывает блокировку. Этот вызов придает потоку возможность гибко решать, что делать дальше, если есть альтернативы простому ожиданию.

1.6.1. Мьютексы в пакете Pthreads.

Средства работы с мьютексами.

Вызов из потока	Описание
`pthread_mutex_init`	Создание мьютекса
`pthread_mutex_destroy`	Уничтожение существующего мьютекса
`pthread_mutex_lock`	Овладение блокировкой или блокирование потока
`pthread_mutex_trylock`	Овладение блокировкой или выход с ошибкой
`pthread_mutex_unlock`	Разблокирование

В дополнение к мьютексам пакет `Pthreads` предлагает второй механизм синхронизации — условные переменные. Мьютексы хороши для разрешения или блокирования доступа к критической области. Условные переменные позволяют потокам блокироваться до выполнения конкретных условий. Эти два метода практически всегда используются вместе.

Средства работы с условными переменными.

Вызов из потока	Описание
pthread_cond_init	Создание условной переменной
pthread_cond_destroy	Уничтожение условной переменной
pthread_cond_wait	Блокировка в ожидании сигнала
pthread_cond_signal	Сигнализирование другому потоку и его активизация
pthread_cond_broadcast	Сигнализирование нескольким потокам и активизация всех этихпотоков

Условные переменные (в отличие от семафоров) не запоминаются. Если сигнал отправлен условной переменной, изменения значения которой не ожидает ни один из потоков, сигнал теряется. Чтобы не потерять сигнал, программисты должны обращать особое внимание.

1.6.2. Использование потоков для решения задачи производителя-потребителя.

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* Количество производимого */

pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* используется для сигнализации */
int buffer = 0; /* буфер, используемый между производителем и потребителем */

void *producer(void *ptr) { /* производство данных */
   int i;
   for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* получение исключительного доступа к буферу */
        while (buffer != 0)
             pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* помещение записи в буфер */
        pthread_cond_signal(&condc); /* активизация потребителя */
        pthread_mutex_unlock(&the_mutex); /* освобождение доступа к буферу */
   }
   pthread_exit(0);
}
```

```
void *consumer(void *ptr) { /* потребление данных */
   int i;
   for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* получение исключительного доступа к буферу */
        while (buffer ==0 )
             pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* извлечение записи из буфера */
        pthread_cond_signal(&condp); /* активизация производителя */
        pthread_mutex_unlock(&the_mutex); /* освобождение доступа к буферу */
   }
   pthread_exit(0);
}
int main(int argc, char **argv) {
   pthread_t pro, con;
   pthread_mutex_init(&the_mutex, 0);
   pthread_cond_init(&condc, 0);
   pthread_cond_init(&condp, 0);
   pthread_create(&con, 0, consumer, 0);
   pthread_create(&pro, 0, producer, 0);
   pthread_join(pro, 0);
   pthread_join(con, 0);
   pthread_cond_destroy(&condc);
   pthread_cond_destroy(&condp);
   pthread_mutex_destroy(&the_mutex);
}
```

1.7. Мониторы

Монитор – высокоуровневый синхронизациионный примитив, поддерживаемый языком программирования для облегчения составления безошибочных программ. Язык программирования С не обладает мониторами.

Монитор представляет собой коллекцию переменных и структур данных, сгруппированных вместе в специальную разновидность модуля или пакета процедур. Процессы могут вызывать любые необходимые им процедуры, имеющиеся в мониторе, но не могут получить доступ к внутренним структурам данных монитора из процедур, объявленных за пределами монитора.

Пример монитора на псевдоязыке, поддерживающем мониторы:

```
monitor example
     integer i;
     condition c;
     procedure producer();
     ...
     end;
     procedure consumer();
     ...
     end;
end monitor;
```

В любой момент времени в мониторе может быть активен только один процесс. Т.е. **в любой момент времени может быть активна только одна процедура монитора**. Реализация взаимного исключения при входе в монитор возлагается на компилятор, а не на программиста.

Мониторы являются конструкцией языка программирования, и компилятор способен обрабатывать вызовы процедур монитора не так, как вызовы других процедур. Обычно при вызове процессом процедуры монитора вначале осуществляется проверка на активность других процессов внутри монитора. Если какой-нибудь другой процесс активен, вызывающий процесс будет приостановлен до тех пор, пока этот другой процесс не освободит монитор.

При решении задачи производителя-потребителя с помощью мониторов необходим способ, позволяющий заблокировать процессы, которые не в состоянии продолжить свою работу. Также нужно ввести условные переменные, а также две проводимые над ними операции — `wait` и `signal`.

Решение задачи производителя-потребителя на псевдоязыке с помощью мониторов.

```
monitor ProducerConsumer
     condition full, empty;
     integer count;
     procedure insert(item: integer);
     begin
          if count=N then wait(full);
          insert_item(item);
          count := count+1;
          if count=1 then signal(empty)
     end;
     function remove:integer;
     begin
          if count=0 then wait(empty);
          remove := remove_item;
          count := count-1;
          if count=N-1 then signal(full)
     end;
     count := 0;
end monitor;

procedure producer;
begin
     while true do
     begin
          item := produce_item;
          ProducerConsumer.insert(item)
     end
end;
```

```
procedure consumer;
begin
     while true do
     begin
          item := ProducerConsumer.remove;
          consume_item(item)
     end
end;
```

Когда процедура монитора обнаруживает невозможность продолжения своей работы (например, производитель обнаружил, что буфер заполнен), она осуществляет операцию `wait` в отношении условной переменной (в данном случае, `full`). Это действие приводит к блокированию процесса. Оно также позволяет войти в монитор другому процессу, которому ранее этот вход был запрещен.

Этот другой процесс (например, потребитель) может снова активизировать работу производителя, осуществив операцию `signal` в отношении той же условной переменной (`full`).

Чтобы в мониторе в одно и то же время не находились сразу два активных процесса, после осуществления операции `signal` обязательно должен произойти немедленный выход из монитора того процесса, который осуществил эту операцию `signal`. То есть операция `signal` должна фигурировать только в качестве завершающей операции в процедуре монитора.

Если операция `signal` осуществляется в отношении условной переменной, изменения которой ожидают сразу несколько процессов, активизирован будет лишь один из них, определяемый системным планировщиком.

Автоматическая организация взаимного исключения в отношении процедур монитора гарантирует следующее. Например, если производитель, выполняя процедуру внутри монитора, обнаружит, что буфер полон, он будет иметь возможность завершить операцию `wait`. При этом планировщик не может переключиться на выполнение процесса потребителя до того, как операция `wait` будет завершена. Потребителю вообще не будет позволено войти в монитор, пока не завершится операция `wait` и производитель не будет помечен как неспособный к дальнейшему продолжению работы.

1.8. Передача сообщений

В распределенной системе, состоящей из связанных по локальной сети нескольких центральных процессоров, у каждого из которых имеется собственная память, для синхронизации процессов используется ***передача сообщений***.

Этот метод взаимодействия процессов использует два примитива, `send` и `receive`, которые являются системными вызовами. Они легко могут быть помещены в библиотечные процедуры, например:

```
send(destination, &message);
receive(source, &message);
```

Вызов `send` отправляет сообщение заданному получателю, а `receive` получает сообщение из заданного источника (или из любого источника). Если доступные сообщения отсутствуют, получатель может заблокироваться до их поступления или немедленно вернуть управление с кодом ошибки.

1.8.1. Проблемы разработки систем передачи сообщений.

Системы передачи сообщений имеют немало **проблем разработки**. Вот лишь некоторые из них:

1) **Утрата сообщений** при передаче по сети. Отправитель и получатель должны договориться о том, что как только сообщение будет получено, получатель должен отправить в ответ специальное подтверждение. Если по истечении определенного интервала времени отправитель не получит подтверждение, он отправляет сообщение повторно.

Если сообщение было успешно получено, а подтверждение, возвращенное отправителю, утрачено. То отправитель заново передаст сообщение, и оно придет к получателю повторно. Чтобы получатель мог отличить новое сообщение от повторно переданного старого, всем сообщениям присваиваются последовательно идущие номера. Если получателю приходит сообщение с тем же самым номером, то это – дубликат, который можно проигнорировать.

2) Как **именовать процессы**, чтобы однозначно указывать процесс в вызовах `send` и `receive`?

3) Проблема **аутентификации**: как клиент может отличить связь с реальным файловым сервером от связи с самозванцем?

4) Если отправитель и получатель находятся на одной и той же машине, возникает проблема **производительности**. Копирование сообщений из одного процесса в другой всегда проводится медленнее, чем операции с семафорами или вход в монитор.

1.8.2. Передача сообщений для решения задачи производителя-потребителя.

Рассмотрим, как с помощью передачи сообщений и без использования общей памяти может решаться задача производителя-потребителя. Положим, что все сообщения имеют один и тот же размер и что переданные, но еще не полученные сообщения буферизуются автоматически средствами ОС.

```
#define N 100 /* N - количество мест в буфере */
void producer(void) {
     int item;
     message m; /* буфер сообщений */
     while (TRUE) {
          item = produce_item( ); /* генерация информации для помещения в буфер */
          receive(consumer, &m); /* ожидание поступления пустого сообщения */
          build_message(&m, item); /* создание сообщения на отправку */
          send(consumer, &m); /* отправка записи потребителю */
     }
}
void consumer(void) {
     int item, i;
     message m; /* буфер сообщений */
     for (i=0; i<N; i++) send(producer, &m); /* отправка N пустых сообщений */
     while (TRUE) {
          receive(producer, &m); /* получение сообщения с записью */
          item = extract_item(&m); /* извлечение записи из сообщения */
          send(producer, &m); /* возвращение пустого ответа */
          consume_item(item); /* обработка записи */
     }
}
```

Потребитель вначале посылает N пустых сообщений производителю. Как только у производителя появится запись для передачи потребителю, он берет пустое сообщение и отправляет его назад в заполненном виде. При этом общее **количество сообщений в системе остается постоянным**, поэтому они могут быть сохранены в заданном, заранее известном объеме памяти.