Межпроцессорное взаимодействие

Межпроцессорное взаимодействие

Процессам часто бывает необходимо взаимодействовать между собой.

Например, в конвейере ядра выходные данные первого процесса должны передаваться второму по цепочке.

Проблема разбивается на три пункта.

Первое: передача информации от одного процесса другому.

Второе: контроль над деятельностью процессов: как гарантировать, что два процесса не пересекутся в критических ситуациях.

Третье: касается согласования действий процессов: если процесс А должен поставлять данные, а процесс В выводить их на печать, то процесс В должен подождать и не начинать печатать, пока не поступят данные от процесса А.

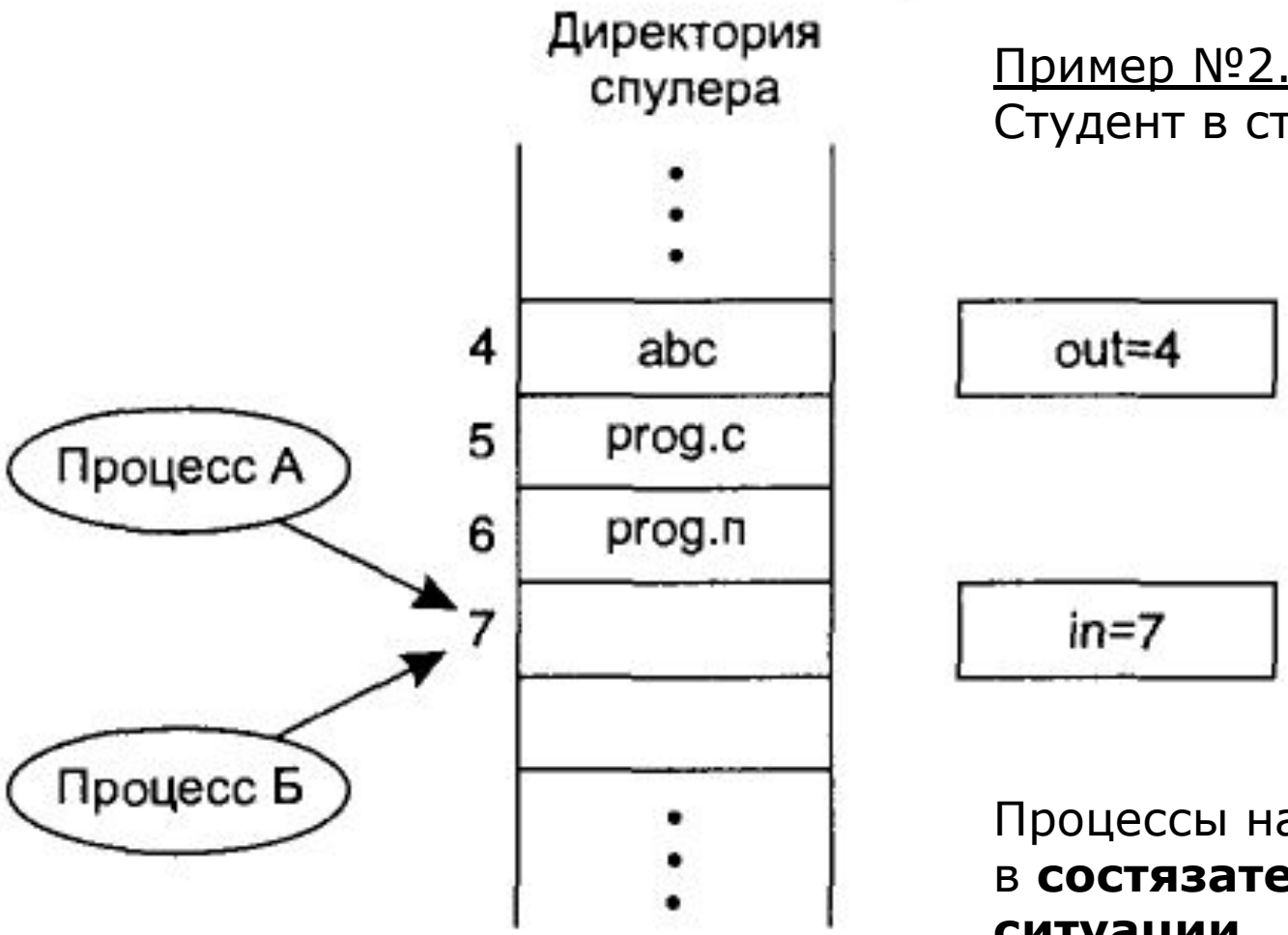
Состояние состязания

- В некоторых операционных системах 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 или файл общего доступа.

Пример: **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.

Состояние состязания (2)



Пример №2.
Студент в столовой.

Процессы находятся в **состязательной ситуации**.

Критические области. Состязания между процессами.

Основным способом **предотвращения любой ситуации**, связанной с совместным использованием памяти, файлов и чего-либо еще, **является запрет одновременной записи и чтения** разделенных данных более чем одним процессом.

Взаимное исключение: один процесс использует разделенные данные, другому процессу это делать будет запрещено.

Выбор подходящей примитивной операции, реализующей взаимное исключение, является серьезным моментом разработки операционной системы.

Критические области. Состязания между процессами.

Формулировка состояния состязания:

1. Некоторый промежуток времени процесс занят внутренними расчетами и другими задачами, не приводящими к состояниям состязания.
2. В другие моменты времени процесс обращается к совместно используемым данным или выполняет действие, которое может привести к состязанию.
3. Часть программы, в которой есть обращение к совместно используемым данным, называется **критической областью** или **критической секцией**.
4. Если удастся избежать одновременного нахождения двух процессов в критических областях, можно избежать состязаний.

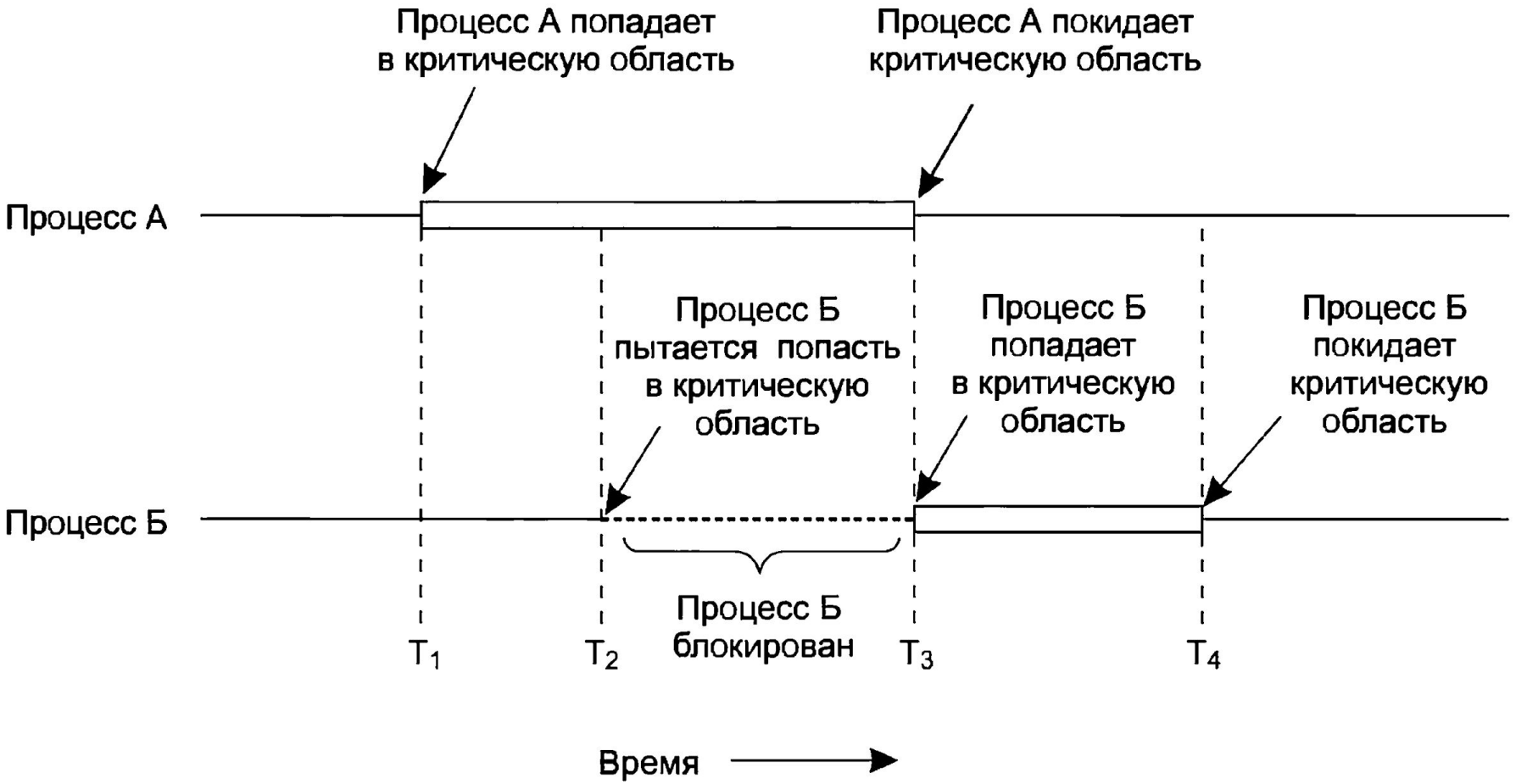
Критические области

Для правильной совместной работы параллельных процессов и эффективного использования общих данных необходимо выполнение четырех условий:

1. Два процесса не должны одновременно находиться в критических областях.
2. В программе не должно быть предположений о скорости или количестве процессоров.
3. Процесс, находящийся вне критической области, не может блокировать другие процессы.
4. Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

Критические области

- Взаимное исключение использования критических областей



Взаимное исключение с активным ожиданием

- Запрещение прерываний
- Переменные блокировки
- Строгое чередование
- Алгоритм Петерсона
- Команда TSL
- Семафоры
- Мьютексы
- Мониторы
- Передача сообщений
- Барьеры

Взаимное исключение с активным ожиданием

способы реализации

Запрещение прерываний

- Запрет всех прерываний при входе процесса в критическую область и разрешение прерываний по выходе из области.
- Если прерывания запрещены, невозможно прерывание по таймеру. Поскольку процессор переключается с одного процесса на другой только по прерыванию, отключение прерываний исключает передачу процессора другому процессу.

Запрещение прерываний (пример)

Пример: процесс пользователя отключил все прерывания и в результате какого-либо сбоя не включил их обратно. Операционная система на этом может закончить свое существование.

Пример: для ядра характерно запрещение прерываний для некоторых команд при работе с переменными или списками.

Итак, запрет прерываний бывает полезным в самой операционной системе.

Переменные блокировки

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.

Переменные блокировки (недостатки)

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

Строгое чередование

Третий метод реализации взаимного исключения.

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

а

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

б

Строгое чередование

Переменная turn=0 отслеживает, чья очередь входить в критическую область.

1. Вначале процесс 0 проверяет значение turn, считывает 0 и входит в критическую область.
2. 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**. Активное ожидание используется только в случае, когда есть уверенность в небольшом времени ожидания.

Блокировка, использующая активное ожидание, называется **спин-блокировкой**.

Алгоритм Петерсона

```
#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 критическую область */
{
       interested[process] = FALSE; /* Индикатор выхода из критической области */
}
```

Команда TSL

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

Команда TSL

```
enter_region:
        TSL REGISTER,LOCK       | значение lock копируется в регистр, значение переменной
                                  устанавливается равным 1
        CMP 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 вошел в критическую
                                  область

leave_region:
        MOVE LOCK,#0            | Сохранение 0 в переменной lock
        RET
```

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.

Примитивы межпроцессного взаимодействия. Пример.

Этот алгоритм не только бесцельно расходует время процессора, но, кроме этого, он может иметь некоторые неожиданные последствия.

Рассмотрим два процесса: H, с высоким приоритетом, и 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. Процесс H попадает в состояние активного ожидания, но поскольку процессу L во время работающего процесса H никогда не будет предоставлено процессорное время, у процесса L не будет возможности выйти из критической области, и процесс H навсегда останется в цикле. Эту ситуацию иногда называют **проблемой инверсии приоритета**.

Примитивы межпроцессного взаимодействия

Теперь рассмотрим некоторые примитивы межпроцессного взаимодействия, применяющиеся вместо циклов ожидания, в которых лишь напрасно расходуется процессорное время. 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.

Примитив **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.

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.

Семафоры.
Операции: **down** и **up.**

Операция down сравнивает значение семафора с нулем.

Если значение семафора больше нуля, операция down уменьшает его 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 единое и неделимое элементарное действие. Тем самым гарантируется, что после начала операции ни один процесс не получит доступа к семафору до окончания или блокирования операции.

Мьютексы

Иногда используется упрощенная версия семафора, называемая **мьютексом** (mutex, сокращение от mutual exclusion - взаимное исключение).

Мьютекс не способен считать, он может лишь управлять взаимным исключением доступа к совместно используемым ресурсам или кодам.

Реализация мьютекса проста и эффективна, что делает использование мьютексов особенно полезным в случае потоков, действующих только в пространстве пользователя.

Мьютексы

Мьютекс - переменная, которая может находиться в одном из двух состояний: блокированном или неблокированном.

Для описания мьютекса 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 собирается войти в критическую область, он вызывает процедуру **mutex_lock**. Если мьютекс не заблокирован, запрос выполняется и вызывающий поток может попасть в критическую область.

Мониторы

- В 1974 году Хоар (Hoare) и Бринч Хансен (Brinch Hansen) предложили примитив синхронизации более высокого уровня, называемый **монитором**.
- **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.

Передача сообщений

Этот метод межпроцессного взаимодействия использует два примитива: send и receive, которые скорее являются системными вызовами, чем структурными компонентами языка.

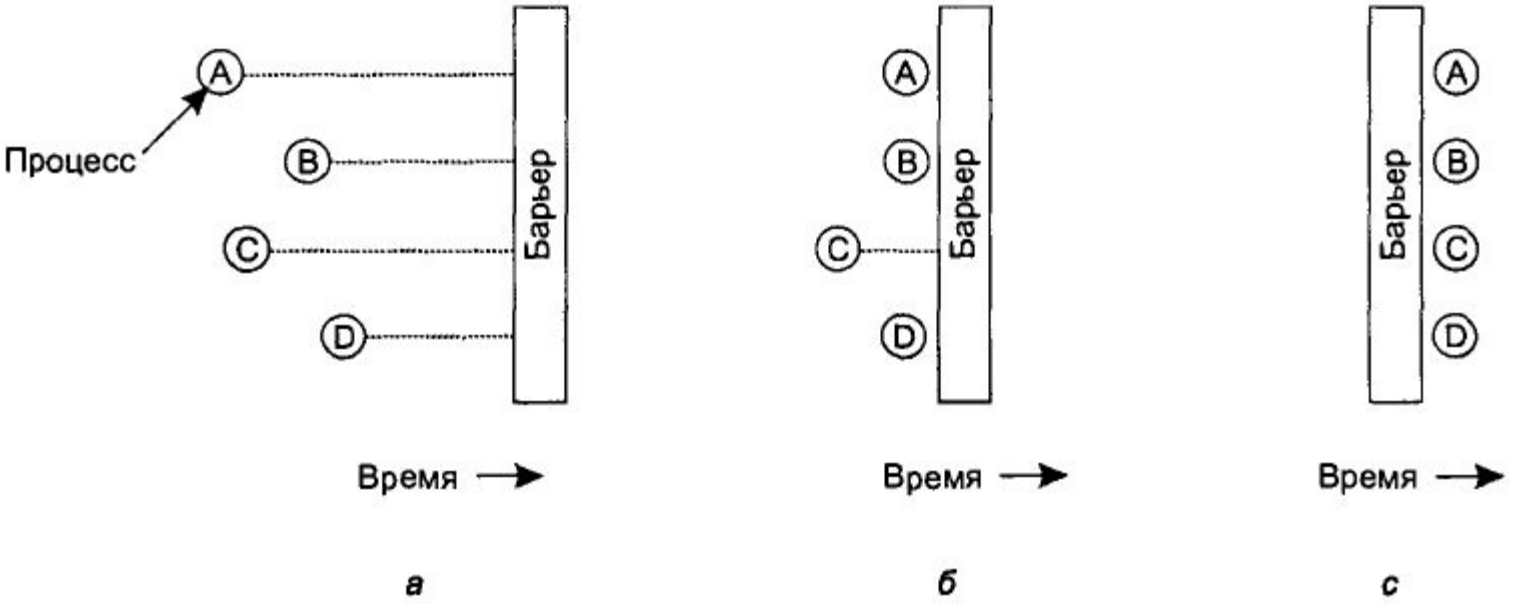
Например:

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

Первый запрос посылает сообщение заданному адресату, а второй получает сообщение от указанного источника. Если сообщения нет, второй запрос блокируется до поступления сообщения либо немедленно возвращает код ошибки.

Барьеры

Последний из рассмотренных нами механизмов синхронизации предназначался скорее для групп процессов, нежели для ситуаций с двумя процессами. Некоторые приложения делятся на фазы, и существует правило, что процесс не может перейти в следующую фазу, пока к этому не готовы все остальные процессы. Этого можно добиться, разместив в конце каждой фазы барьер.



а *б* *с*