Операционные системы

Курс лекций для гр. 4057/2

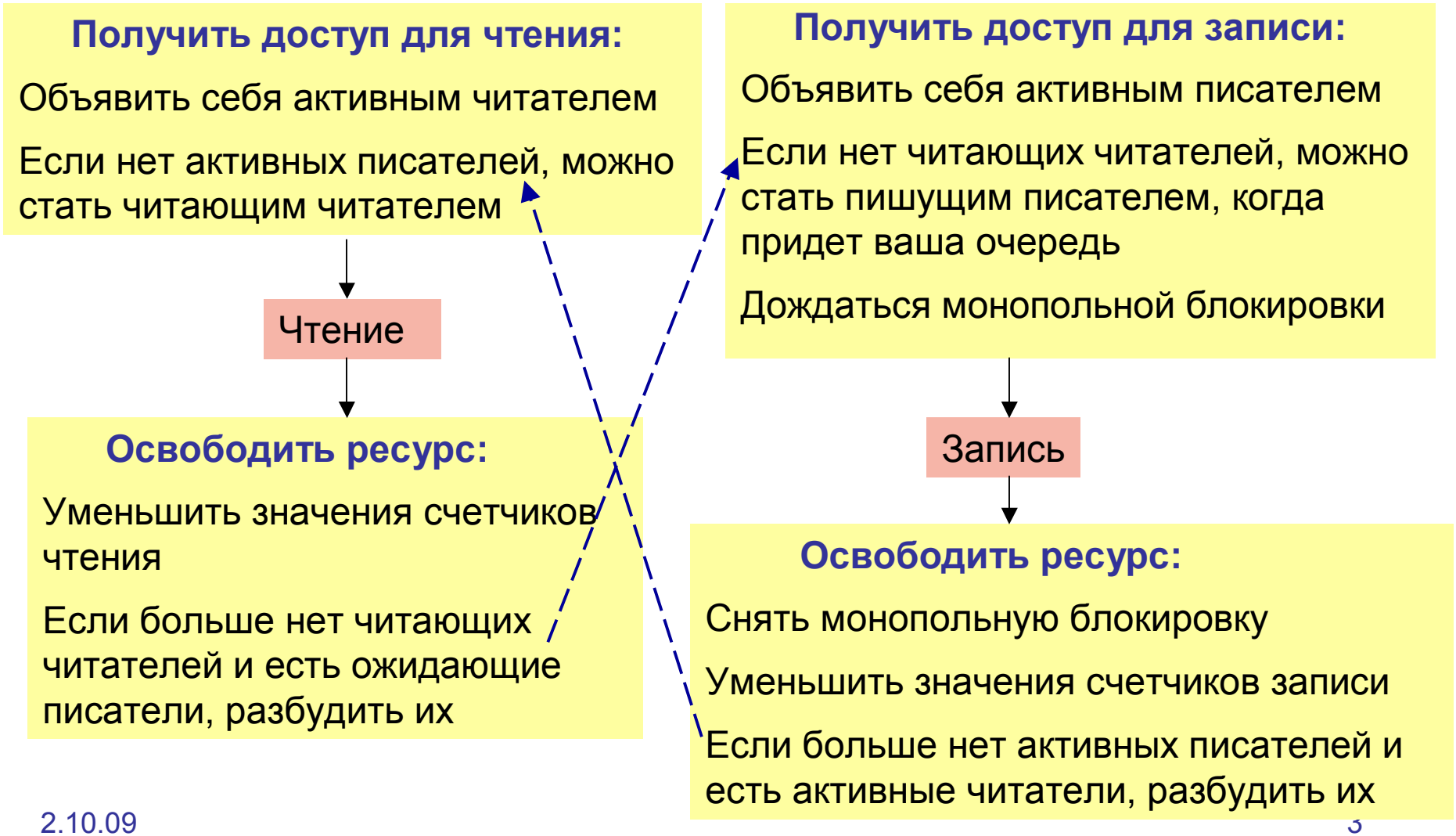
Лекция №5

Вопросы к лекции 4

1. Предположим, непосредственно перед выполнением одного (любого) из этих двух участков k = 10. Сколько (и каких) различных результатов k можно получить после выполнения обоих участков, учитывая все возможные варианты чередования во времени команд этих параллельных процессов?
2. Зачем нужно это ожидание?
3. Недостаток инструкции TS в том, что она при каждом вызове записывает значение в lock (это сильно замедляет выполнение TS в многопроцессорных системах). Как изменить код взаимного исключения (не изменяя TS) для преодоления этого недостатка?
4. Решите задачу взаимного исключения с применением инструкций Exchange и Compare&Swap, описав предварительно их семантику на псевдокоде.
5. Расширьте данное решение на случай многих потребителей и производителей с общим буфером. (Понадобится взаимное исключение работы с буфером – отдельное для потребителей и производителей.)
6. Исправьте эту ошибку.
7. Напишите псевдокод решения задачи «читатели-писатели» с приоритетом писателей: ни один читатель не должен начать чтение, если есть хоть один ждущий писатель. Используйте сильные семафоры. (Не забудьте прокомментировать объявления семафоров и глобальных переменных.)

   Подсказка: одна из возможных схем алгоритма – на следующем слайде

Возможная схема решения задачи с приоритетом писателей



Глобальные переменные кода

Семафоры

- R – new Semaphore(0) – для постановки в очередь запросов чтения, когда писатели пишут
- W – new Semaphore(0) – для постановки в очередь запросов записи, когда читатели читают
- Wmon – new Semaphore(1) – для монопольного доступа писателей к объекту
- CountAct – new Semaphore(1) – для монопольного доступа к счетчикам количеств активных читателей и писателей

Счетчики

- ar – число активных читателей
- rr – число читающих читателей
- aw - число активных писателей
- ww – число пишущих писателей (в каждый момент времени запись выполняет только один из них)

Содержание

Разнообразие примитивов синхронизации

Мьютексы (замки)

Семафоры

Критические секции

События

Переменные условий

Мониторы

Барьеры

…

Критические участки (секции)

Подобны мьютексам, но реализованы не в ядре, а в адресном пространстве процесса, поэтому работают быстрее

Функции Win API:

CreateCriticalSection
DestroyCriticalSection
EnterCriticalSection
LeaveCriticalSection

- Они могут использоваться только для взаимного исключения потоков одного и того же процесса
- Поэтому у крит. секций нет дескрипторов → они не могут передаваться от одного процесса другому

События

- объекты ядра, которые находятся в одном из двух состояний: установленном и сброшенном

- В Windows поток может ждать события, вызвав функцию WaitForSingleObject
- Если другой поток устанавливает событие вызовом SetEvent, то результат зависит от типа события:
  - если оно – сбрасываемое вручную, то все его ждущие потоки отпускаются (т.е, переходят в «готово»), а событие остается установленным, пока его не сбросят вызовом ResetEvent
  - если оно – сбрасываемое автоматически, то отпускается только один ожидающий поток, и событие тут же сбрасывается
- Есть еще ф-ция PulseEvent
  - если его никто не ждет, событие все равно сбрасывается, и значит, пропадает впустую

Переменные условий

condition variables, тж. назыв. переменными состояния

Переменная условия позволяет заблокировать выполнение потока, пока не наступит заданное событие или не будет соблюдено некоторое условие

Одновременно используются следующие три объекта:

- Переменная условия, именующая условие
- Булевская переменная (предикат), которая указывает, выполнено ли условие
- Взаимная блокировка (mutex), которая упорядочивает доступ к булевской переменной

Переменная условия работает подобно двоичному семафору, но есть отличие: прекращение ожидания не дает гарантий истинности предиката, его приходится проверять в явном виде, что делает программу устойчивее и упрощает анализ ее корректности

Мониторы

Семафоры, как и замки, не защищены от ошибочного использования:

- семафоры глобальны по отношению ко всем процессам
- нужно помнить о том, с каким критическим ресурсом связан семафор
- можно забыть вставить операцию signal

и т.д.

Более защищены от ошибок программирования *мониторы*

Монитор – это объект, в котором критические данные *инкапсулируются* в самом синхронизирующем примитиве

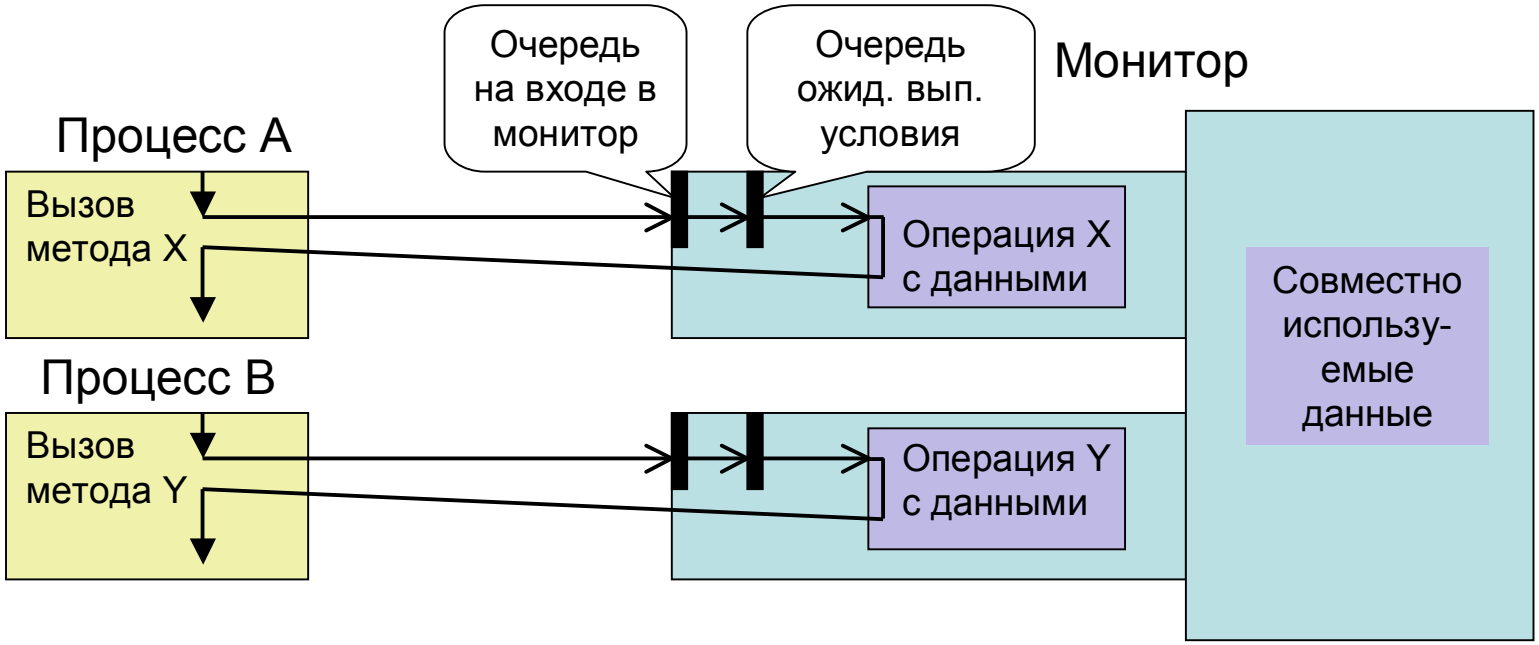
Состав монитора:

- замок для взаимного исключения доступа к инкапсулированным данным
- одно или несколько условий, представленных переменными для условной синхронизации

Взаимное исключение происходит неявно при вызове метода, а условная синхронизация программируется явно.

Мониторы обычно не примитивы ОС, а часть языка: Modula, Java, C#, Ruby, …

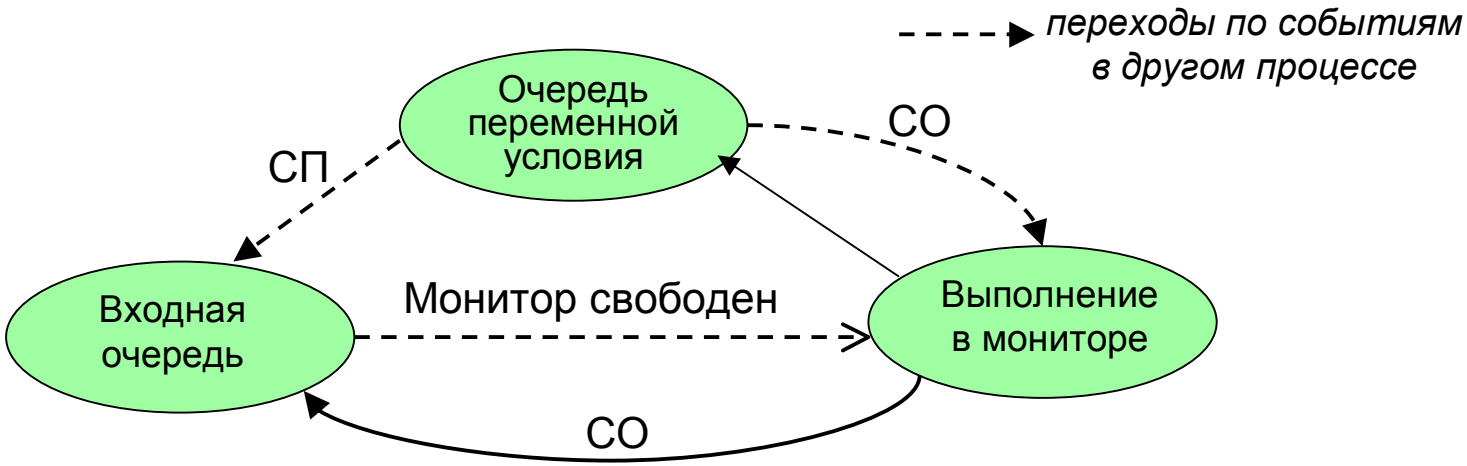
Схема монитора



- Только один процесс может быть активен в мониторе в любой момент времени
- Для блокировки процессов, которые не могут продолжить свое выполнение - две операции с переменными условий: wait и signal

Условная синхронизация в мониторе

- Процесс, вызвавший wait, блокируется, пока другой процесс не выполнит signal с этим же условием
- Операция signal деблокирует один ждущий процесс из очереди FIFO; если таких нет, то ничего не делается
- После выполнения signal возможны два варианта синхронизации:
  - Сигнализировать и продолжить (СП): просигнализировавший процесс продолжает работу, а процесс, получивший сигнал, выполняется позже
  - Сигнализировать и ожидать (СО): просигнализировавший процесс переходит в состояние ожидания, а процесс, получивший сигнал, начинает выполняться



Мониторы в языке Java

- С каждым объектом Java неявно связан монитор с возможностью *одной* монопольной блокировки и с *одной* переменной условия
- Если объявить метод объекта как `synchronized`, то он будет выполняться в режиме взаимного исключения: во время выполнения потоком такого метода никакой другой поток не сможет выполнить *ни один синхронизированный метод* данного объекта – он должен ждать в *единственной* очереди, связанной с данным объектом

Пример: счетчик с взаимоисключающим доступом – монитор без переменной условия, два эквивалентных варианта:

```
public class counter {
  private int count = 0;
  public synchronized void update(){
    count++;
  }
}
```

```
public class counter {
private int count = 0;
public void update() {
  synchronized (this) {
       count++;
    }
  }
}
```

Java: условная синхронизация потоков

Три стандартных метода для условной синхронизации (они могут вызываться только в теле метода, объявленного как synchronized):

- `wait()` - ожидание у переменной условия объекта. Этот метод снимает блокировку объекта, блокирует текущий поток и ставит его в очередь потоков, ожидающих у данного объекта
- `notify()` – деблокирует поток из очереди ожидания у данного объекта, но сам продолжает выполняться (т.е., дисциплина СП)
- `notifyall()` - подобен методу `notify()`, но вызывает пробуждение всех потоков, ожидающих у данного объекта, один из которых может получить процессор

Пример: буфер на одну порцию данных:

```
public class Buffer {
  private int value = 0
  private boolean full = false;

  public synchronized void put(int a)(
    while (full) wait();
    value = a;
    full = true;
    notifyAll();
  }
```

```
public synchronized int get(){
  int result;
  while (!full) wait();
  result = value;
  full = false;
  notifyAll();
  return result;
}
```

(Вопросы 1, 2)

Синхронизация в Unix и Linux

- Мьютексы
- Счетные семафоры
- Переменные условий

  Системные вызовы (Posix):
  - pthread _cond_init
  - pthread _cond_destroy
  - pthread _cond_wait
  - pthread _cond_signal
  - pthread_cond_broadcast // оповещение всех
- Блокировки читатели/писатели
  - rw_enter попытка захвата блокировки для чтения или записи
  - rw_exit освобождение блокировки
  - rw_tryenter неблокирующий захват
  - rw_downgrade превращение блокировки для записи в блокировку для чтения
  - rw_tryupgrade превращение блокировки для чтения в блокировку для записи
- spin locks - специальные замки для многопроцессорных систем – спин-блокировки

Синхронизация в Windows

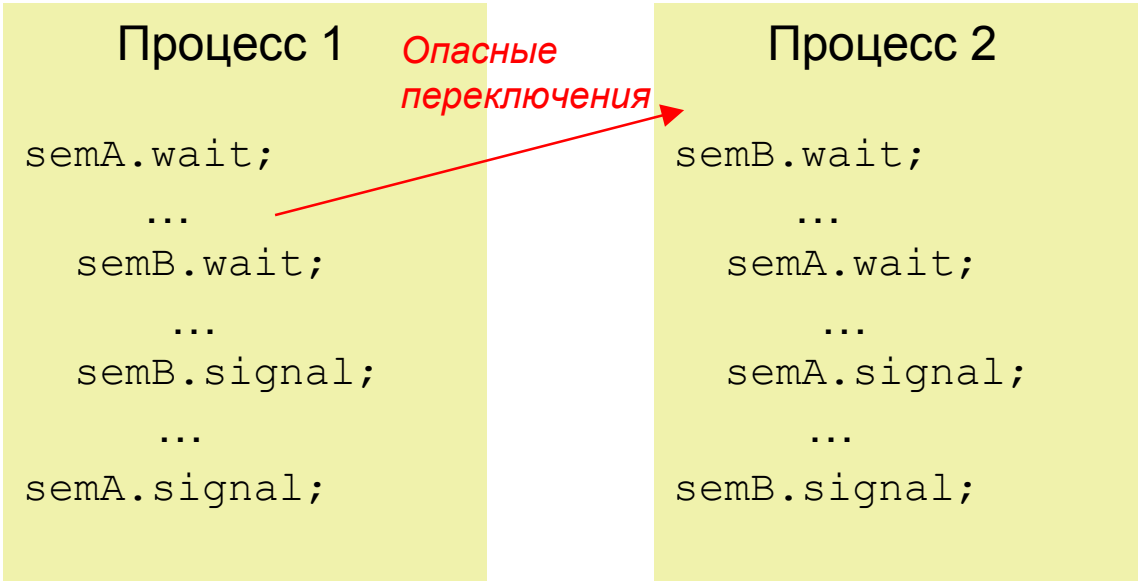
Тип объекта ядра	Определение	Когда сигнализирует	Сколько жду-щих потоков освобождаются
Процесс		Завершился последний поток	Все
Поток		Завершился поток	Все
Файл	Открытый файл или увв-вы	Завершена операция вв-вы	Все
Консоль	Буфер текстового окна	Введена информация	Один
Извещение об изменении файлов	Любые изменения файловой системы	Изменения, соотв. фильтру	Один
Мьютекс		Поток освободил мьютекс	Один
Семафор	Счетный семафор	Счетчик семафора обнулился	Все
Событие	Извещение о событии	Произошло событие	Все или один
Таймер	Счетчик времени	Истек указанный интервал времени	Все

Тупики

Тупик, или *взаимная блокировка* (*deadlock*) – ситуация невозможности продолжения двух или более процессов из-за того, что они ждут события, которое может быть сгенерировано только этими же процессами и только после их выхода из ожидания

Типичный случай - ожидание освобождения критического ресурса

Пример: бесконечные ожидания перед вложенными критическими участками:

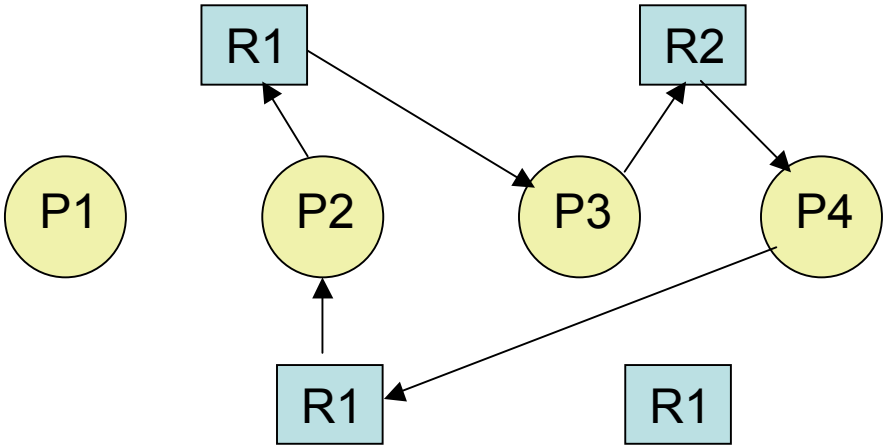


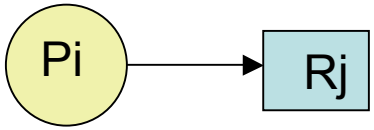
В существующих ОС нет встроенных средств обработки тупиков; это забота прикладных программистов или специального сервиса ОС - предотвращать, обнаруживать и ликвидировать тупики

Необходимые (одновременно) условия возникновения тупика

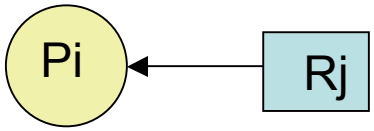
1. *Взаимное исключение:* по крайней мере один ресурс захвачен в неразделяемом режиме
2. *Захват и ожидание:* по крайней мере один процесс захватил ресурс и ждет, пока освободится другой, добавочный, ресурс, захваченный другим процессом
3. *Непрерываемость:* процесс освобождает ресурс только по своей инициативе; другой процесс или ОС не может заставить процесс освободить его
4. *Циклическое ожидание:* существует набор ждущих процессов $\{P_1, \ldots, P_n\}$, где P_i ожидает P_{i+1} $(i = 1,\ldots, n)$ и P_n ожидает P_1

Граф распределения ресурсов



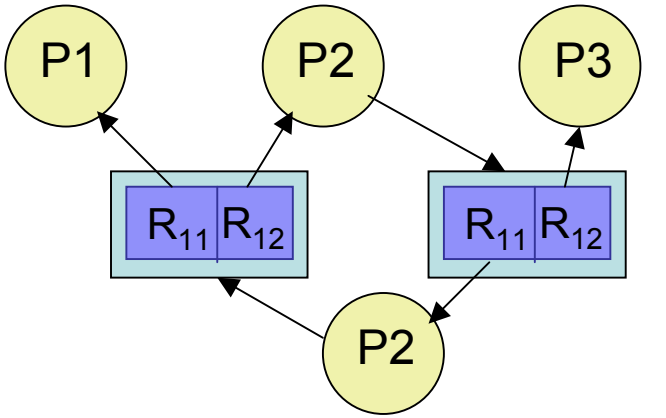


Процесс Pi запросил ресурс Rj



Процесс Pi захватил ресурс Rj

- Если этот граф не имеет циклов, никакого тупика не существует
- Если в графе есть цикл, тупик может существовать, а может и нет



Цикл в графе распределения ресурсов, не приводящий к тупику (случай множественных ресурсов одного типа)

Выявление тупика

Алгоритм поиска циклов в графе имеет сложность $O(n^2)$, где $n = |P| + |R|$. Когда его применять?

- Перед предоставлением ресурса, проверяя, не приведет ли это к циклу, т.е., при каждом запросе
- Каждый раз, когда запрос ресурса не может быть удовлетворен
- По расписанию, скажем, каждые 10 с
- Когда использование процессора падает ниже некоторого порога

Устранение тупика

- разорвать цикл одним из способов:

- убить все процессы в цикле
- убивать процессы по одному, заставляя их освобождать ресурсы
- освобождать ресурсы по одному, откатывая захвативший процесс к состоянию, предшествующему захвату ресурса
  - этот способ применяется в транзакциях баз данных

Прямое предупреждение тупиков

Обеспечить, чтобы хотя бы одно из 4-х необходимых условий не выполнялось:

1. *Взаимное исключение:* делать ресурсы не критическими. Минусы:
   - для некоторых ресурсов это невозможно, напр., для CD-ROMa
   - для переменных это приводит к возможным ошибкам одновременного доступа
2. *Захват и ожидание:*
   - запретить процессу удерживать один ресурс, когда он запросил другой
     - минус: иногда это невозможно: алгоритм вычислений требует, чтобы процесс владел одновременно несколькими несколькими критическими ресурсами
   - запрашивать и ждать все нужные процессу ресурсы сразу
     - минус: удлиняются простои из-за ожидания нескольких ресурсов сразу и падает производительность – теряется выигрыш от параллельности процессов

Прямое предупреждение тупиков (2)

3. *Непрерываемость:* если процесс запрашивает ресурс, который не может быть немедленно предоставлен ему, тогда ОС отбирает все его текущие ресурсы. Только когда все упомянутые ресурсы становятся доступными, ОС возобновляет процесс. Минусы:
   - не все ресурсы легко отобрать, прервав их использование - напр., принтер
   - опять же уменьшается производительность
4. *Циклическое ожидание:* упорядочить (перенумеровать) ресурсы и разрешить запрашивать их только в этом порядке. Т.е., если i - наибольший номер среди ресурсов, захваченных процессом, то он может запрашивать Rj только если j > i. В противном случае, если j < i, то процесс должен сперва освободить все Rk , k > j .

   (Вопрос 3)

Предупреждение тупиков путем резервирования ресурсов

- Процессы дают предварительную информацию о максимальном количестве ресурсов, которые им могут потребоваться во время выполнения
- Назовем *безопасной* такую последовательность процессов, что для каждого P_i ресурсы, которые он еще может затребовать, могут быть удовлетворены текущими доступными ресурсами плюс теми, которыми владеют все P_k , $k < i$.
- *Безопасное состояние* – такое, в котором существует безопасная последовательность процессов
- Небезопасное состояние не эквивалентно тупику, только может привести к нему, а может и нет, если некоторые процессы не используют в действительности заявленного максимального количества ресурсов

Предупреждение тупиков путем резервирования ресурсов (2)

- Когда процесс запрашивает ресурс, система дает ему его, только если новое состояние - безопасное. Иначе – процесс должен ждать, пока другие процессы не освободят достаточное количество ресурсов. Такая стратегия обеспечивает отсутствие циклического ожидания.

Алгоритмы, реализующие этот подход:

- Для уникальных ресурсов алгоритм основан на анализе графа распределения ресурсов: *Resource-Allocation Graph Algorithm*
- Для ресурсов, представленных несколькими идентичными (взаимозаменяемыми) экземплярами (напр., каналы в коммуникационной системе) - *алгоритм банкира* (Дейкстра, 1967)
  - Идея алгоритма – проверка, безопасно ли состояние, в которое перейдет система при удовлетворении очередного запроса ресурсов
  - Банкиры никогда не отдают свою наличность таким образом, чтобы это привело бы к отказу в обслуживании запросов всех клиентов банка

Заключение

- Мониторы – высокоуровневые примитивы синхронизации, в отличие от замков и семафоров
- Мониторы обычно реализуются не в ОС, а в Run-time - исполнительной системе языка программирования
- Большое разнообразие примитивов синхронизации в различных ОС (особенно в Windows) – источник сложностей при описании и сопровождении программных продуктов
  - Разнообразие следует ограничивать в стандарте кодирования команды программистов
  - Это случай, где полезен минимализм в выборе средств (пример – Java)
- Противодействие тупикам – одна из задач системных программистов, поскольку встроенных в ОС средств борьбы с ними обычно нет

Вопросы и задания

1. Почему для сигнализации здесь рекомендуется использовать метод notifyAll(), а не notify() ?
2. Напишите на Java программу класса «Данные с параллельным чтением и исключительной записью», который могут использовать читатели и писатели, для двух случаев:
   а) приоритет читателей
   б) приоритет писателей
3. Докажите, что описанный метод действительно обеспечивает отсутствие циклического ожидания.