Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Операционные системы: *аспекты параллелизма*

Синхронизация-2

Линёв А.В.

2007

Тема обсуждения

- При разработке параллельных программ необходимо решать задачу взаимного исключения
  - Можно самостоятельно реализовать низкоуровневый механизм, используя один из существующих алгоритмов
  - Можно использовать механизмы синхронизации, предоставляемые операционной системой

Высокоуровневые механизмы синхронизации

- Семафоры
- Мониторы
- Синхронные сообщения

Мы рассмотрим семафоры и мониторы

Семафоры

Семафоры

- Семафоры – примитивы синхронизации более высокого уровня абстракции, чем признаки блокировки; предложены Дийкстрой (Dijkstra) в 1968 г. в качестве компонента операционной системы THE
- Семафор - это переменная, для которой определены 2 операции: **P** и **V**
  - **P**(sem) (wait/down) ожидает выполнения условия sem > 0, затем уменьшает sem на 1 и возвращает управление
  - **V**(sem) (signal/up) увеличивает sem на 1
- Операции **P** и **V** выполняются **атомарно**

Семафоры – реализация Дийкстры…

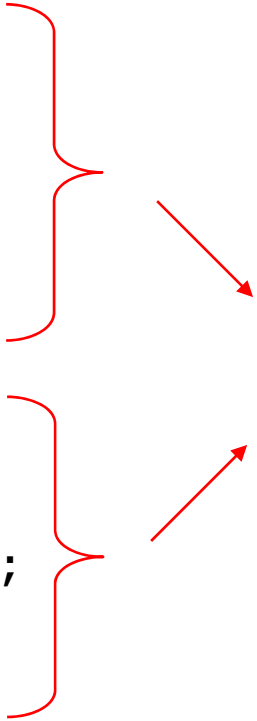
- С каждым семафором ассоциирована очередь потоков
  - при вызове потоком **P**(sem)
    - если семафор "свободен" (>0), его значение уменьшается на 1, и выполнение потока продолжается
    - если семафор "занят" (<=0), поток переводится в состояние ожидания и помещается в очередь, соответствующую данному семафору
    - запускается какой-либо другой готовый к выполнению поток
  - при вызове потоком **V**(sem)
    - если очередь потоков, ассоциированная с данным семафором, не пуста – один из них разблокируется и помещается в очередь готовых к выполнению
    - поток, вызвавший V (sem), продолжает свое выполнение
    - в противном случае (нет потоков, ожидающих освобождения семафора), значение семафора увеличивается
- Таким образом, все вызовы изменяют состояние семафора, то есть семафор сохраняет некоторую информацию о прошедших вызовах

Семафоры – реализация Дийкстры

```
typedef struct{
   int value;
   list<thread> L;
} semaphore;

P(S){
   S.value = S.value - 1;
   if( S.value < 0 ){
      add this thread to S.L;
      block;
}

V(S){
   S.value = S.value + 1;
   if( S.value <= 0 ){
      remove a thread T from S.L;
      wakeup T;
}
```



P () и V() – критические секции, должны выполняться атомарно

Виды семафоров

- **Двоичный семафор**
  - S.value может принимать значения 0 и 1, инициализируется значением 1
  - обеспечивает эксклюзивный доступ к ресурсу (например, при работе в критической секции)
  - одновременно может выполняться только один поток
- **Счетный семафор**
  - S.value инициализируется значением N = число доступных единиц ресурса
  - представляет ресурсы, состоящие из нескольких однородных элементов
  - позволяет потокам исполняться, пока есть неиспользуемые элементы

Мьютексы

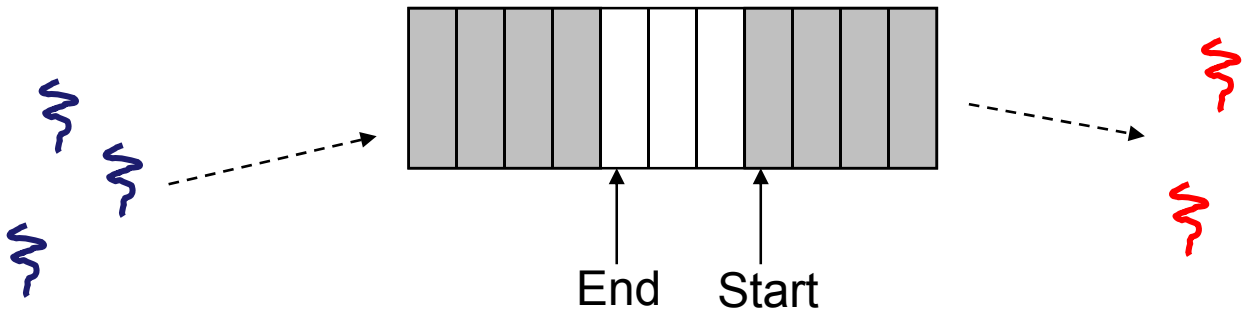
- Мьютекс – двоичный семафор, обычно используемый для организации согласованного доступа к неделимому общему ресурсу. Мьютекс может принимать значения 1 (свободен) и 0 (занят).
- Операции над мьютексами
  - acquire(mutex) – уменьшить (занять) мьютекс
  - release(mutex) – увеличить (освободить) мьютекс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

Мьютексы

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

Пример №1. Задача "Производитель-Потребитель"

- Имеется циклический буфер из N элементов
- Потоки-производители добавляют в него записи (по одной за одну операцию)
- Потоки-потребители извлекают записи (также по одной за раз)
- Потоки выполняются параллельно



Пример №1. Задача "Производитель-Потребитель"

```
Semaphore mutex = 1; // семафор, управляющий доступом к разделяемым данным
Semaphore empty = n; // количество пустых записей (после запуска все записи пусты)
Semaphore full  = 0; // количество заполненных записей
                     // (после запуска нет заполненных записей)
```

```
Producer:
   P(empty);     // ждем появления свободной записи
   P(mutex);     // получаем доступ к указателям (критические данные)
      <записываем значение в запись>
   V(mutex);     // завершили работу с указателями
   V(full);      // сигнализируем о появлении заполненной записи
```

```
Consumer:
   P(full);      // ждем появления заполненной записи
   P(mutex);     // получаем доступ к указателям (критические данные)
      <извлекаем данные из записи>
   V(mutex);     // завершили работу с указателями
   V(empty);     // сигнализируем о появлении свободной записи
      <используем данные из записи>
```

Пример №2. Задача "Читатели-писатели"

- Имеются критические данные, над которыми определены операции чтения и записи
  - Потоки-писатели изменяют данные; если поток-писатель работает с данными, они не могут использоваться другими потокам
  - Потоки-читатели не изменяют данные; если поток-читатель работает с данными, они могут быть использованы другими потоками-читателями
- Потоки выполняются параллельно

Пример №2. Задача "Читатели-писатели"

```
Semaphore RC     = 1; // управляет доступом к переменной ReadCount
Semaphore Access = 1; // управляет допуском к данным писателя или 1-го читателя
int ReadCount    = 0; // количество "активных" (читающих) читателей
```

```
Writer:
   P(Access);          // захватываем доступ к критическим данным
      <выполняем операцию записи>
   V(Access);          // освобождаем доступ к критическим данным
```

```
Reader:
   P(RC);              // получаем эксклюзивный доступ к переменной ReadCount
   ReadCount++;        // увеличиваем число активных читателей
   if( ReadCount == 1 )
      P(Access);       // если мы первые - захватываем доступ к критическим данным
   V(RC);              // освобождаем доступ к переменной ReadCount
      <выполняем операцию чтения>
   P(RC);              // получаем эксклюзивный доступ к переменной ReadCount
   ReadCount--;        // уменьшаем число активных читателей
   if( ReadCount == 0 )
      V(Access);       // если мы последние - освобождаем доступ к критическим данным
   V(RC);              // освобождаем доступ к переменной ReadCount
```

Пример №2. Задача "Читатели-писатели"

- Замечания
  - Первый появившийся читатель переходит в состояние ожидания при выполнении P(Access) в присутствии писателя; все остальные читатели переходят в состояние ожидания при выполнении вызова P(RC)
  - Если при завершении работы писателем или читателем уже присутствуют ожидающие писатели и читатели, выбор потока, захватывающего семафор, определяется реализацией семафора
  - Если потоки-читатели постоянно входят в критическую секцию, возможна задержка ("голодание", *starvation*) потоков-писателей

Реализация мьютексов – Win32…

- Атрибуты мьютекса в Win32
  - Идентификатор потока – определяет, какой поток захватил мьютекс
  - Счетчик рекурсии - сколько раз была выполнена операция захвата
- Создание мьютекса

```
HANDLE CreateMutex( PSECURITY_ATTRIBUTES psa,
          BOOL fInitialOwner, PCTSTR pszName);
```

- Открытие существующего мьютекса

```
HANDLE OpenMutex( DWORD fdwAccess,
          BOOL bInheritHandle, PCTSTR pszName);
```

- Закрытие мьютекса

```
BOOL CloseHandle(HANDLE hMutex);
```

Реализация мьютексов – Win32

- Уменьшение значения мьютекса (Захват мьютекса)

```
DWORD WaitForSingleObject( HANDLE hObject,
        DWORD dwMilliseconds);
DWORD WaitForMultipleObjects( DWORD dwCount,
        CONST HANDLE* phObjects,
        BOOL fWaitAll, DWORD dwMilliseconds);
```

- Освобождение мьютекса

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Реализация семафоров – Win32…

- Атрибуты семафора в Win32
  - максимальное число ресурсов (контролируемых семафором)
  - текущее число ресурсов
- Создание семафора

```
HANDLE CreateSemaphore( PSECURITY_ATTRIBUTE psa,
          LONG lInitialCount, LONG lMaximumCount,
          PCTRTR pszName);
```

- Открытие существующего семафора

```
HANDLE OpenSemaphore( DWORD fdwAccess,
          BOOL bInheritHandle, PCTSTR pszName);
```

- Закрытие семафора

```
BOOL CloseHandle(HANDLE hMutex);
```

Реализация семафоров – Win32

- Уменьшение значения семафора

```
DWORD WaitForSingleObject( HANDLE hObject,
        DWORD dwMilliseconds);
DWORD WaitForMultipleObjects( DWORD dwCount,
        CONST HANDLE* phObjects,
        BOOL fWaitAll, DWORD dwMilliseconds);
```

- Увеличение значения семафора

```
BOOL ReleaseSemaphore( HANDLE hSem,
   LONG lReleaseCount, PLONG plPreviousCount);
```

Реализация мьютексов – POSIX…

- Атрибуты мьютекса в POSIX
  - Рекурсивный или нерекурсивный мьютекс (определяет, может ли поток несколько раз захватить один и тот же мьютекс)
  - Разделяется ли мьютекс между процессами
  - Наследует ли поток, владеющий мьютексом, наивысший из приоритетов потоков, ожидающих данный мьютекс (требуется для решения проблемы инверсии приоритетов)
  - …

Реализация мьютексов – POSIX

- Создание мьютекса

```
int pthread_mutex_init(pthread_mutex_t *mutex,
          const pthread_mutexattr_t *attr);
pthread_mutex_t def_mutex=PTHREAD_MUTEX_INITIALIZER;
```

- Удаление мьютекса

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Уменьшение значения (захват) мьютекса

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Освобождение мьютекса

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Реализация семафоров – POSIX…

- Атрибуты семафора в POSIX
  - текущее число ресурсов
  - разделяется ли семафор между процессами
- Создание семафора

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
sem_t *sem_open(const char *name, int  oflag,
          unsigned long mode, unsigned int value,…);
```

- Открытие существующего семафора

```
sem_t *sem_open(const char *name, int  oflag,…);
```

- Удаление семафора

```
int sem_destroy(sem_t *sem);
int sem_close(sem_t *sem);
int sem_unlink(const char * name);
```

Реализация семафоров – POSIX

- Уменьшение значения семафора

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

- Увеличение значения семафора

```
int sem_post(sem_t *sem);
```

Семафоры – Выводы

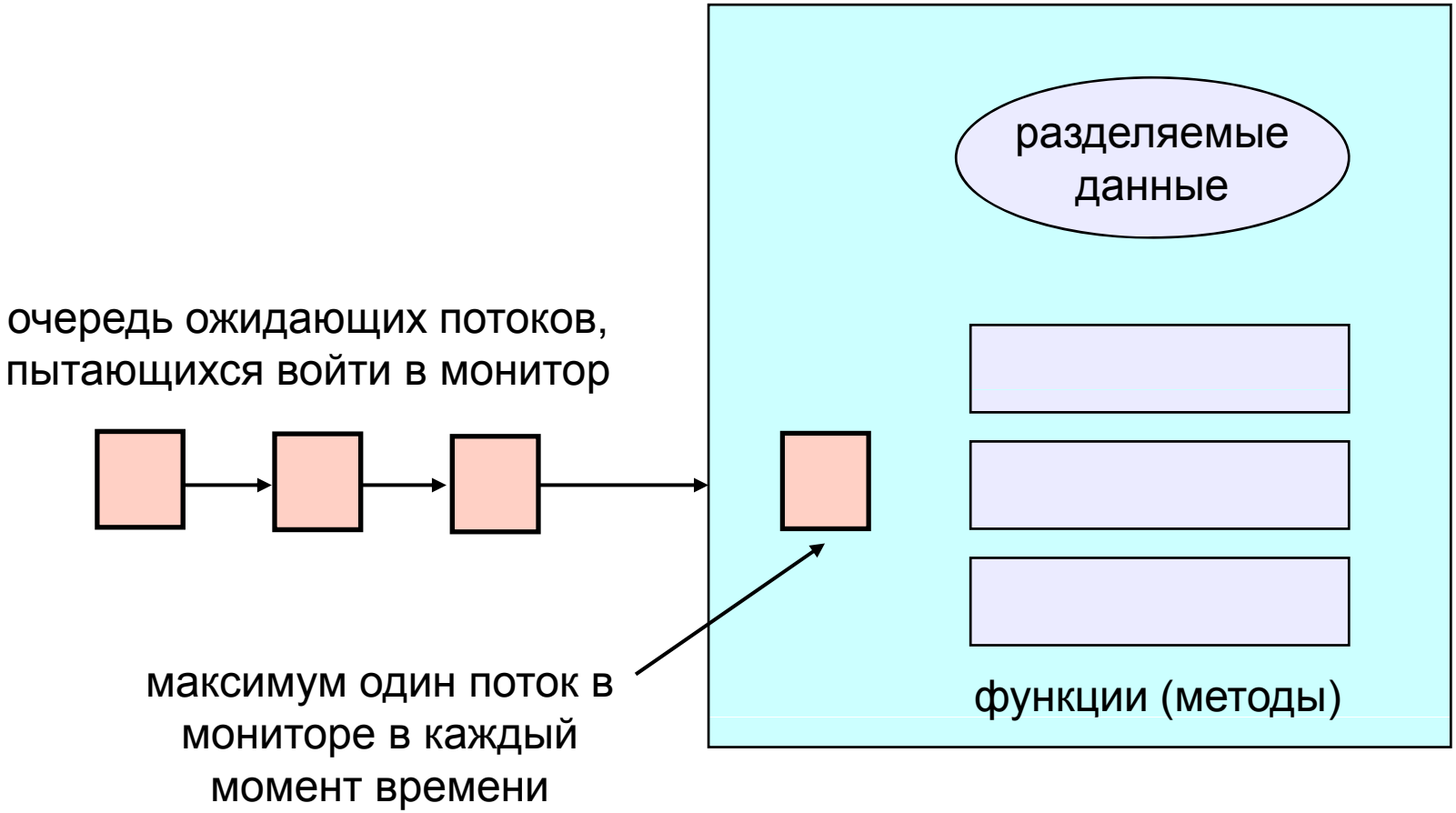
- С помощью семафоров можно решить любую классическую задачу синхронизации, но по сути, семафоры - это разделяемые глобальные переменные (что является признаком плохой структуры программы)
- Семафоры используются как для решения задачи взаимного исключения, так и для координации действий
- Отсутствует связь между семафором и данными, доступом к которым он управляет
- Нет контроля использования семафоров со стороны компилятора или операционной системы, соответственно - нет гарантий, что семафоры будут использованы правильно

Мониторы

Монитор

- **Монитор** – это конструкция языка программирования, поддерживающая управляемый доступ к разделяемым данным
- Монитор инкапсулирует
  - разделяемые критические данные
  - функции, использующие разделяемые данные
  - синхронизацию выполнения параллельных потоков, вызывающих указанные функции
- Доступ к данным, расположенным в мониторе, реализуется только посредством вызова предоставленных функций
- Код синхронизации добавляется компилятором

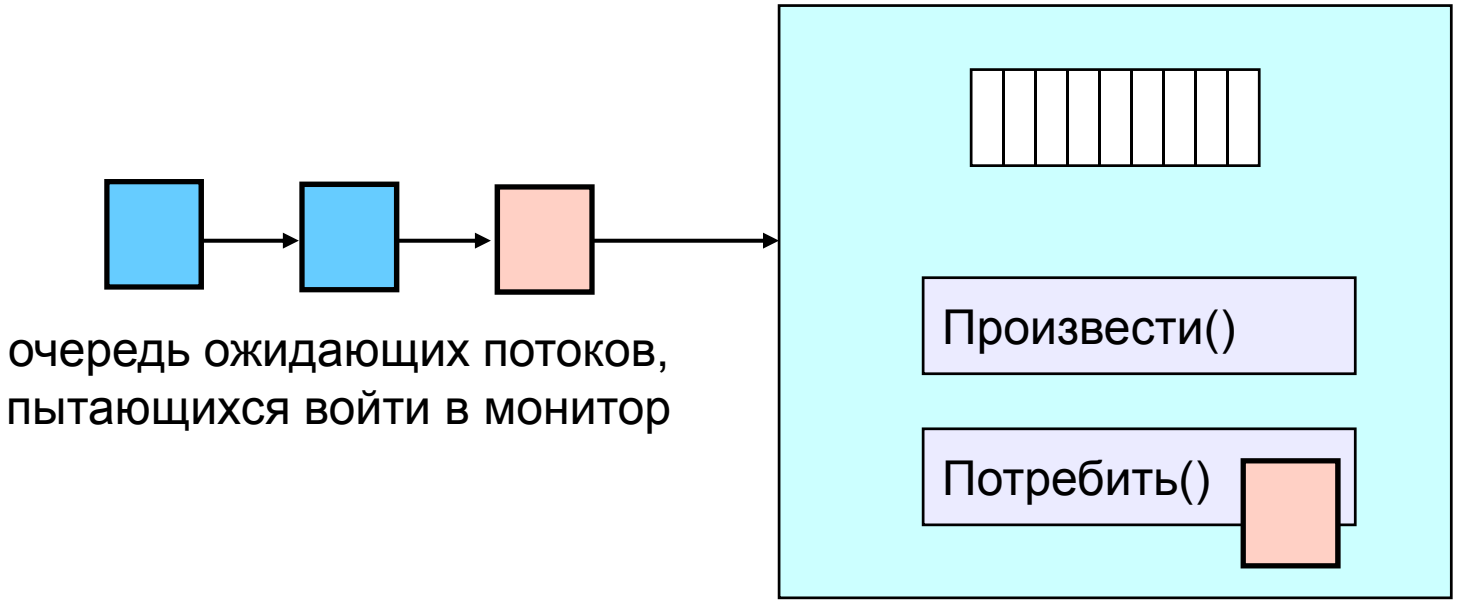
Монитор


разделяемые данные
очередь ожидающих потоков, пытающихся войти в монитор
максимум один поток в мониторе в каждый момент времени
функции (методы)

Возможности монитора

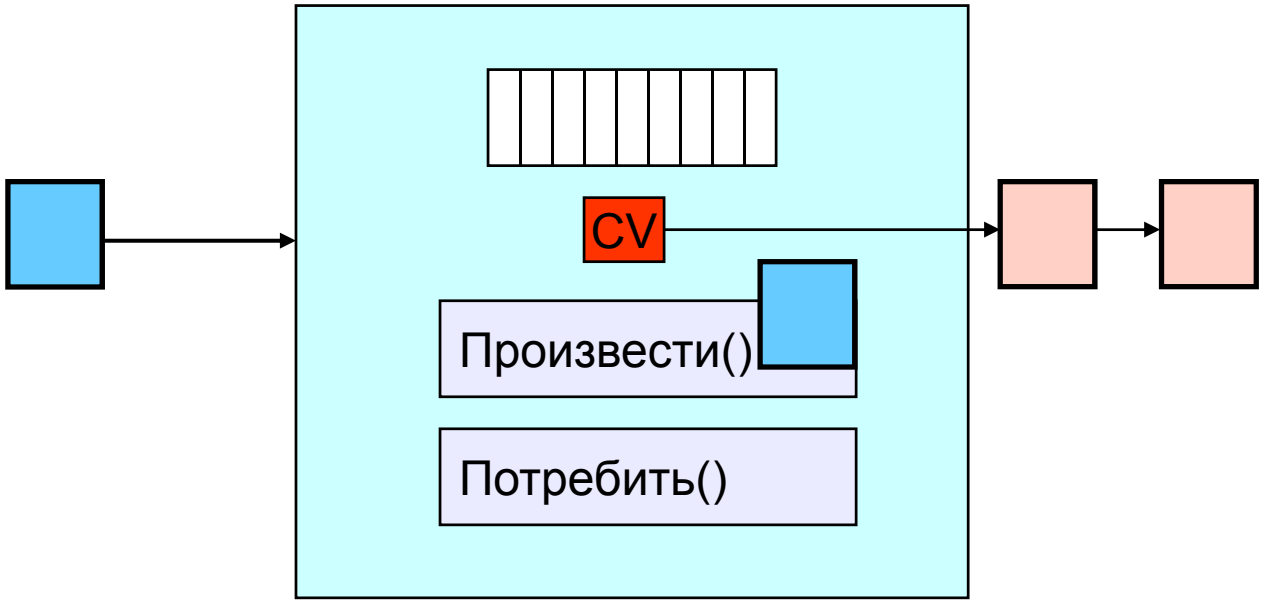
- Автоматическое взаимное исключение
  - только один поток может находиться в мониторе в любой момент времени
    - таким образом, монитор по определению решает задачу взаимного исключения – достаточно для работы с критическими данными использовать только методы монитора
  - если второй поток пытается вызвать метод монитора, он переходит в состояние ожидания до выхода из монитора первого потока
    - использование мониторов накладывает более жесткие ограничения, чем использование семафоров
    - но мониторы в большинстве случаев проще в использовании
- Однако,… есть проблема

Пример
Задача "Поставщик-потребитель"



- Заполненных записей нет
- Что получилось?
- Что делать?

Пример
Задача "Поставщик-потребитель"



CV = condition variable / условная переменная

Condition variables (Условные переменные)

- Символизируют ожидание потоком, выполняющим метод монитора, наступления некоторого события
- Требуются для мониторов, но часто реализуются даже в их отсутствие
- Над условными переменными определены три операции
  - **wait(cv)**
    - снимает блокировку монитора, после чего другой поток может войти в него
    - ожидает, пока какой-либо другой поток не подаст условный сигнал
    - как правило, условным переменным ставятся в соответствие очереди ожидания
  - **signal(cv)**
    - пробуждает максимум один ожидающий поток
    - если нет ожидающих потоков, информация о приходе сигнала теряется
  - **broadcast(cv)**
    - пробуждает все ожидающие потоки

Пример
Задача "Производитель-Потребитель"

```
Monitor Bounded_buffer {
   buffer resources[N];
   condition not_full, not_empty;

   Produce(resource x) {
      if( array “resources” is full ) // determined by a counter
         wait(not_full);
      <записываем значение “x” в запись массива “resources”>
      signal(not_empty);
   }

   Consume(resource *x) {
      if (array “resources” is empty) // determined by a counter
         wait(not_empty);
      *x = <считываем значение из массива “resources”>
      signal(not_full);
   }
}
```

Два вида мониторов

- **Мониторы Хоара**: signal(cv) означает:
  - немедленно запускается ожидающий поток
  - поток, пославший сигнал, блокируется и остается блокированным все время, пока выполняется поток, которого он вывел из состояния ожидания
    - пославший сигнал поток перед посылкой сигнала должен восстановить инварианты монитора (!) – он не должен оставлять ожидающему потоку монитор в некорректном состоянии
- **Мониторы Меса**: signal(cv) означает:
  - ожидающий поток переводится в состояние "готов к выполнению", а поток, пославший сигнал, продолжает исполнение
  - ожидавший поток запускается при выходе потока, пославшего сигнал, из монитора или его перехода в состояние ожидания
  - поток, пославший сигнал, может не восстанавливать состояние монитора вплоть до выхода из него
  - если ваш поток вышел из состояния ожидания – это всего лишь означает, что произошли какие-то изменения – при этом условие ожидания может не быть выполнено и нужно опять проверять выполнение ожидаемых условий

Мониторы Хоара vs. Мониторы Меса

- Мониторы Хоара: `if (notReady) wait(c)`
- Мониторы Меса: `while (notReady) wait(c)`
- Мониторы Меса проще использовать
  - более эффективны
  - используют меньшее количество переключений
  - непосредственно поддерживает broadcast (широковещательную рассылку сигналов)
- С мониторами Хоара меньше случайностей
  - если поток разбужен, гарантируется выполнение условия, которого он ожидал

Мониторы – Выводы

- Мониторы решают задачу взаимного исключения
- Мониторы поддерживаются на уровне языков программирования
- Мониторы обрабатываются компилятором
  - компилятор при построении кода добавляет в него вызовы функций
    - monitor entry
    - monitor exit
    - signal
    - wait

Реализация условных переменных – POSIX…

- Атрибуты условной переменной в POSIX
  - разделяется ли условная переменная между процессами
- Создание условной переменной

```
int pthread_cond_init(pthread_cond_t *cond,
      const pthread_condattr_t *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Удаление условной переменной

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Реализация условных переменных – POSIX

- Ожидание выполнения условия

```
int pthread_cond_wait(pthread_cond_t *cond,
        pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
        pthread_mutex_t *mutex,
        const struct timespec *abstime);
```

- Отправка сигнала о выполнении условия

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Заключение

- Высокоуровневые механизмы синхронизации: семафоры, мониторы и синхронные сообщения – взаимозаменяемы и позволяют организовать корректное выполнение параллельных программ
- При разработке параллельных программ можно самостоятельно реализовать низкоуровневый механизм, используя один из существующих алгоритмов, либо использовать механизмы синхронизации, предоставляемые операционной системой
- Неумелое использование средств синхронизации - источник трудно находимых ошибок и потерь производительности

Вопросы для обсуждения

- Знаете ли вы какие-либо реализации мониторов?
- Если да, каков тип этих мониторов: Хоара или Меса?

Задания для самостоятельной работы

- Известно, что семафоры, синхронные сообщения и мониторы реализуемы друг через друга. Реализуйте монитор, пользуясь семафорами или синхронными сообщениями

Литература

1. Таненбаум Э. Современные операционные системы. 2-е изд. – СПб.: Питер, 2002.
2. Рихтер Дж. Windows для профессионалов (Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows). 4-е изд. – М.: Русская Редакция; пер. с англ. – СПб.: Питер, 2001.
3. Робачевский А.М. Операционная система UNIX. – СПб.: BHV - Санкт-Петербург, 1998.

Тема следующей лекции – Взаимоблокировки (Тупики)

- Необходимые и достаточные условия возникновения тупика
- Подходы к решению проблемы тупиков