САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Параллельные вычисления

Проектирование многопоточных программ

Михаил Моисеев

Санкт-Петербург
2011

Шаблоны проектирования параллельных программ

- Шаблон проектирования определяет общие правила создания параллельных программ
  - Модель параллельных вычислений – формализм описывающий параллельную программу
  - Используемые стандарты и библиотеки функций/компонентов
- Модель параллельных вычислений
  - Примитивы параллелизма: потоки, процессы, …
  - Объекты и правила синхронизации
  - Управление выполнением, переключение контекста

Классификация шаблонов проектирования ПП

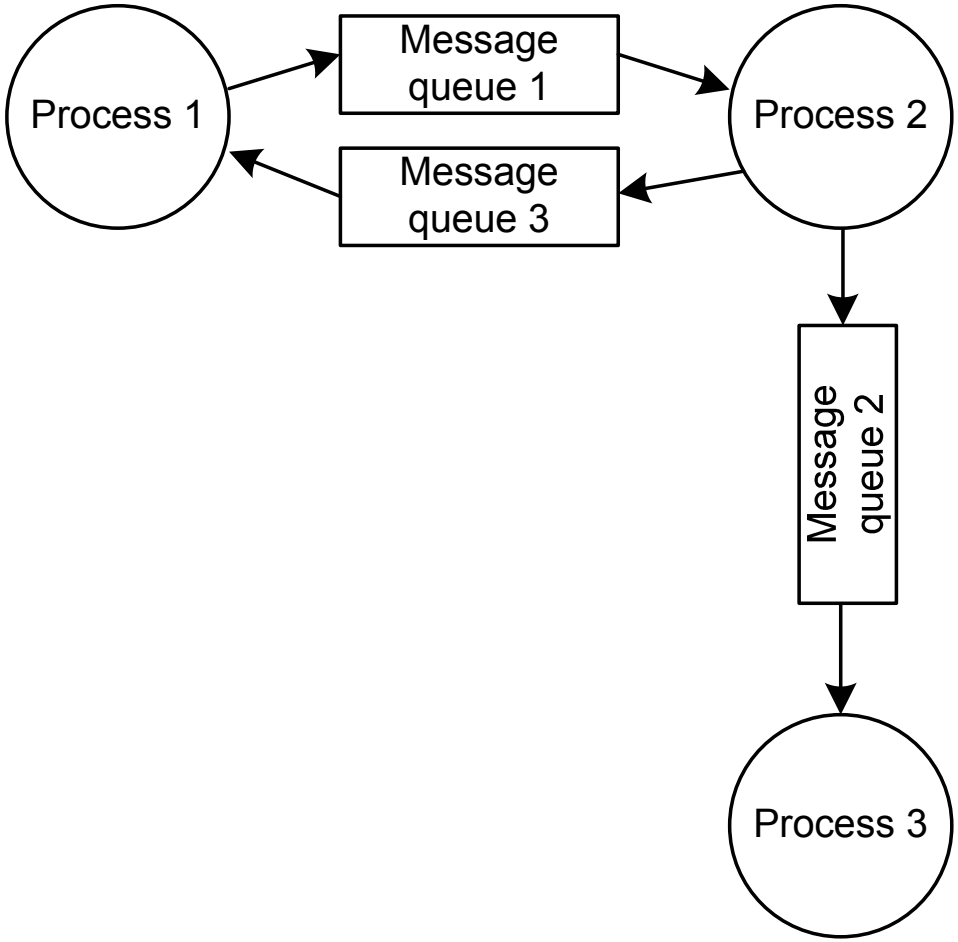
- По уровню параллелизма
  - Уровень задач(подзадач)/команд/данных
- По используемым примитивам параллелизма
  - Процессы/потоки/«зеленые» потоки
- По используемым процедурам синхронизации
- По способу управления процессами
  - Вытесняющая/невытесняющая многозадачность,
- По организации доступа к данным
  - Локальные/общие данные
- По применимости для различных платформ

Модели параллельных вычислений

- Процессы, обменивающиеся сообщениями
- Процессы, соединенные каналами связи
- Процессы/потоки с общей памятью
- ...

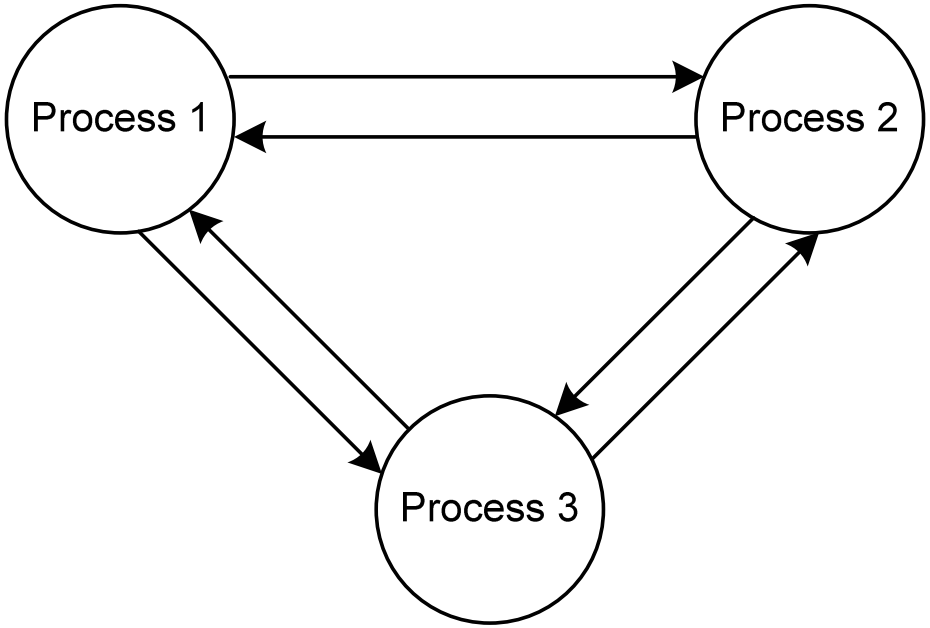
Процессы, соединенные каналами связи

- Несколько параллельно выполняющихся процессов, динамическое создание процессов
- Локальные данные процессов
- Каналы – однонаправленные очереди сообщений, связывающие пары портов разных процессов, динамическое создание каналов
- Между двумя процессами может быть произвольное число каналов



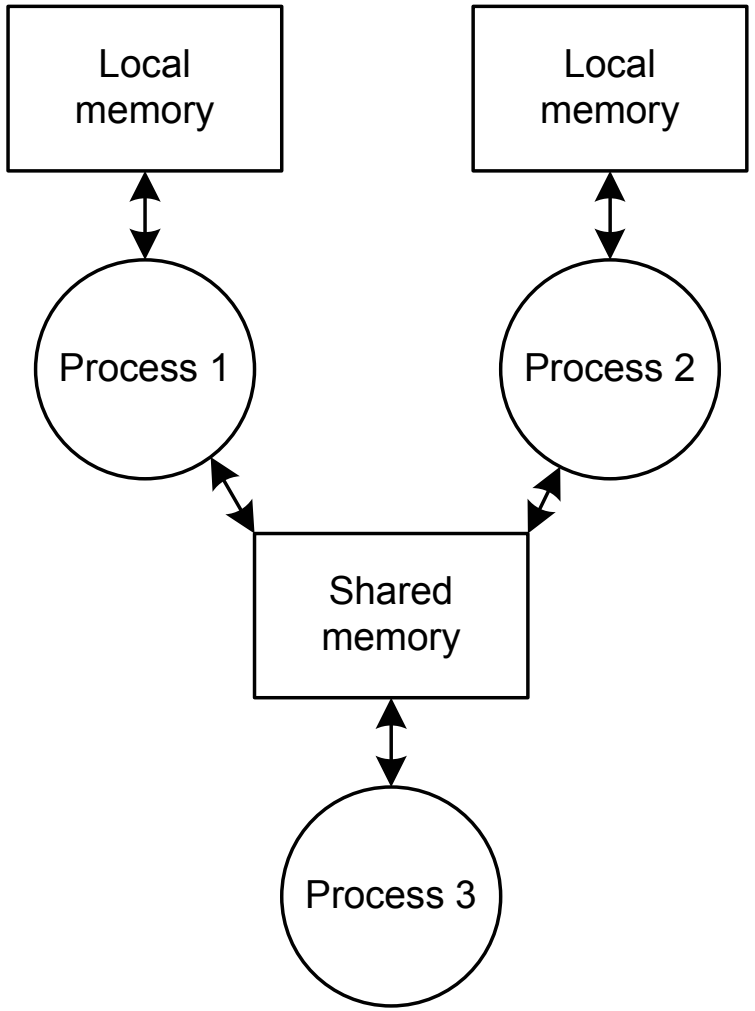
Процессы обменивающиеся сообщениями

- Несколько параллельно выполняющихся процессов, динамическое создание процессов
- Локальные данные процессов
- Передача сообщений из процесса в процесс – адресуется конкретный процесс, а не канал



Процессы/потоки с общей памятью

- Несколько параллельно выполняющихся процессов/потоков, динамическое создание процессов
- Локальные и общие данные процессов
- Использование механизмов синхронизации при работе с общей памятью



Проектирование многопоточных программ

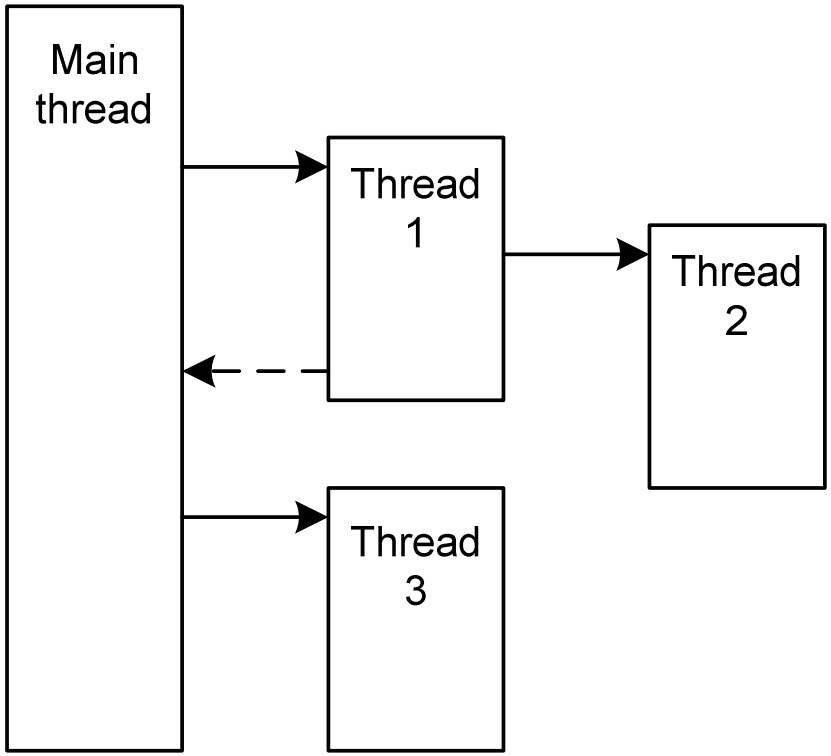
- Модель взаимодействующих потоков с общей памятью
  - Динамическое создание потоков
  - Использование различных объектов синхронизации
- Стандарты и библиотеки
  - POSIX Threads
  - OpenMP
  - Windows Threads
  - Boost Threads
  - Intel TBB

Создание многопоточных программ на основе POSIX Threads

- POSIX Threads – стандарт IEEE 1003.1, ISO/IEC 9945 (http://pubs.opengroup.org/onlinepubs/009695399/mindex.html)
- Pthreads определяет интерфейсы для языков C/C++
- Имеются реализации для ОС Linux, FreeBSD, Solaris, QNX, MS Windows

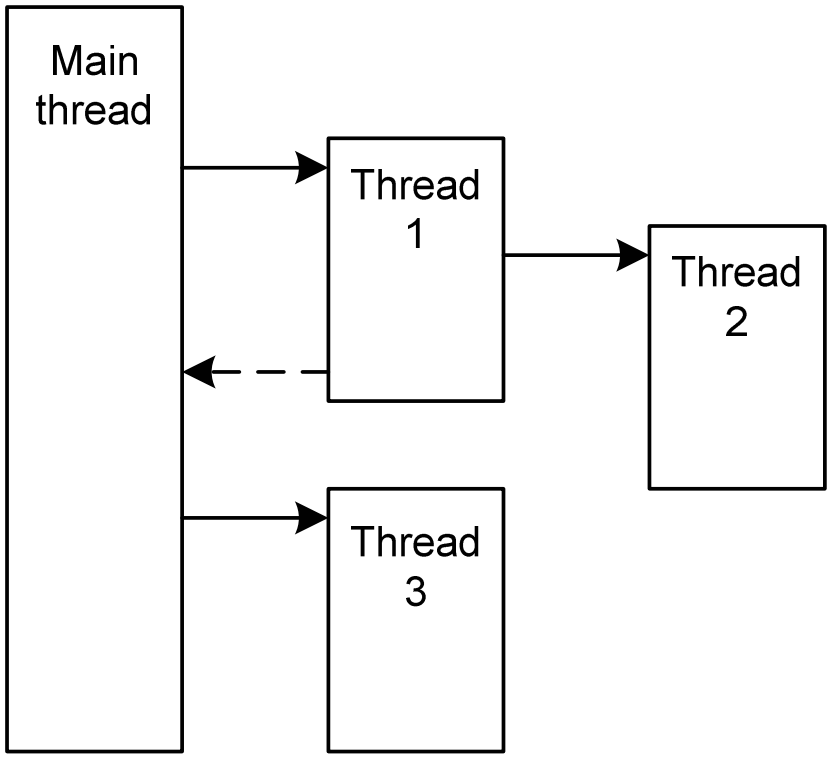
Модель параллельных вычислений PThreads

- Поток программы является основным объектом для организации параллельного выполнения функций в многопоточной программе
- При запуске программы начинает выполняться один главный поток – остальные потоки создаются динамически
- В созданном потоке запускается на выполнение функция программы
- Потоки могут создаваться в любых потоках



Модель параллельных вычислений PThreads

- Завершение потоков
  - завершение функции потока
  - завершение программы
  - вызовы специальных функций из текущего или других потоков
- Два типа потоков
  - detached-поток
  - обычный (не-detached) поток
- Ожидание завершения дочернего потока не-detached в родительском потоке



Модель памяти и синхронизация в PThreads

- Потоки программы могут обращаться ко всем видимым объектам программы
  - Локальные переменные потока – определяются только правилами языка программирования
  - Разделяемые переменные
- Для разграничения доступа к разделяемым объектам, а также для обеспечения определенных последовательностей выполнения конструкций в потоках используются объекты синхронизации

Потоки и объекты синхронизации PThreads

- pthread_t – поток программы
- pthread_mutex_t – объект синхронизации типа мьютекс
- sem_t – объект синхронизации типа семафор
- pthread_rwlock_t – объект синхронизации типа RWLock
- pthread_spinlock_t – объект синхронизации типа Spin
- pthread_cond_t – переменная состояния
- pthread_barrier_t – барьер
- pthread_key_t – ключ идентификации данных отдельных потоков
- pthread_once_t – переменная однократного выполнения

Функции управления потоками программы

- `int` **`pthread_create`** `(pthread_t *, const pthread_attr_t *, void *(*start_routine) (void *), void*)` – создание потока и запуск в нем указанной функции
- `int` **`pthread_detach`** `(pthread_t)` – перевод обычного потока в состояние detached
- `void` **`pthread_exit`** `(void *)` – завершение выполнения текущего потока
- `int` **`pthread_join`** `(pthread_t, void **)` – ожидание завершения обычного (не-detached) потока
- `int` **`pthread_kill`** `(pthread_t, int)` – отправка сигнала указанному потоку программы
- `int` **`pthread_cancel`** `(pthread_t)` – прекращение выполнения указанного потока

Пример работы с потоком программы

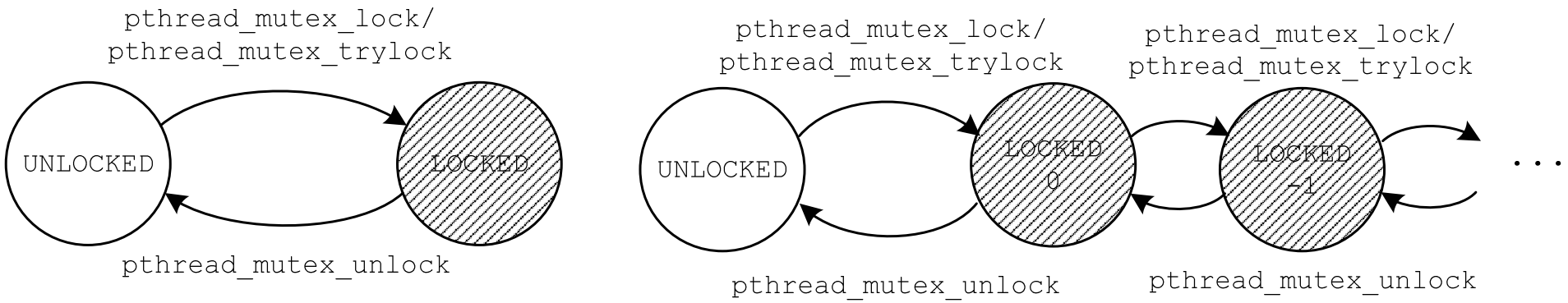
```
#include <pthread.h>
int a = 0;
void* f(void* b){
    a = a + 1;
}

int main() {
    pthread_t t;
    pthread_create(&t, NULL, f, NULL);
    pthread_join(t, NULL);
    printf("%d", a);
}
```

Функции для работы с мьютексом

- `int` **`pthread_mutex_init`** `(pthread_mutex_t *, const pthread_mutexattr_t *)` – инициализация мьютекса
- `int` **`pthread_mutex_destroy`** `(pthread_mutex_t *)` – уничтожение мьютекса
- `int` **`pthread_mutex_lock`** `(pthread_mutex_t *)` – блокировка мьютекса;
- `int` **`pthread_mutex_trylock`** `(pthread_mutex_t *)` – попытка блокировки мьютекса
- `int` **`pthread_mutex_unlock`** `(pthread_mutex_t *)` – освобождение мьютекса
- Функции захвата и освобождения мьютекса должны выполняться строго из одного потока, освобождение не захваченного мьютекса является ошибкой

Типы мьютексов

- Тип мьютекса определяет атрибутом инициализации
  - PTHREAD_MUTEX_NORMAL – обычный мьютекс, допускает операции захвата и освобождения объекта в одном и том же потоке
  - PTHREAD_MUTEX_ERRORCHECK – мьютексы этого типа не допускают повторный захват, освобождения незахваченного объекта или объекта захваченного в другом потоке, в таких случаях выдается ошибка
  - PTHREAD_MUTEX_RECURSIVE – рекурсивный мьютекс допускает многократный захват объекта в одном потоке



Пример программы с мьютексом 1

```
pthread_mutex_t mutex;
int x; // Shared variable

int main(){

    pthread_mutex_init(&mutex, NULL);
    ...
    pthread_mutex_lock(&mutex);
    doSomething(x);
    pthread_mutex_unlock(&mutex);
    ...
    pthread_mutex_destroy(&mutex);

}
```

Пример программы с мьютексом 2

```
int a = 0;
pthread_mutex_t mutex;
void* f(void* b) {
    pthread_mutex_lock(&mutex);
    a = a + 1;
    pthread_mutex_unlock(&mutex);
}
int main(){
    pthread_create(&t, NULL, f, NULL);
    pthread_mutex_lock(&mutex);
    int b = a;
    pthread_mutex_unlock(&mutex);
    pthread_join(t, NULL);
    printf("%d %d", a, b);  // 1,0 или 1,1
}
```

Пример программы с мьютексом 3

```
int a = 0;
pthread_mutex_t mutex;
void* f(void* b) {
    pthread_mutex_lock(&mutex);
    a = a + 1;
    pthread_mutex_unlock(&mutex);
}
int main(){
    pthread_mutex_lock(&mutex);
    pthread_create(&t, NULL, f, NULL);
    int b = a;
    pthread_mutex_unlock(&mutex);
    pthread_join(t, NULL);
    printf("%d %d", a, b);  // ?
}
```

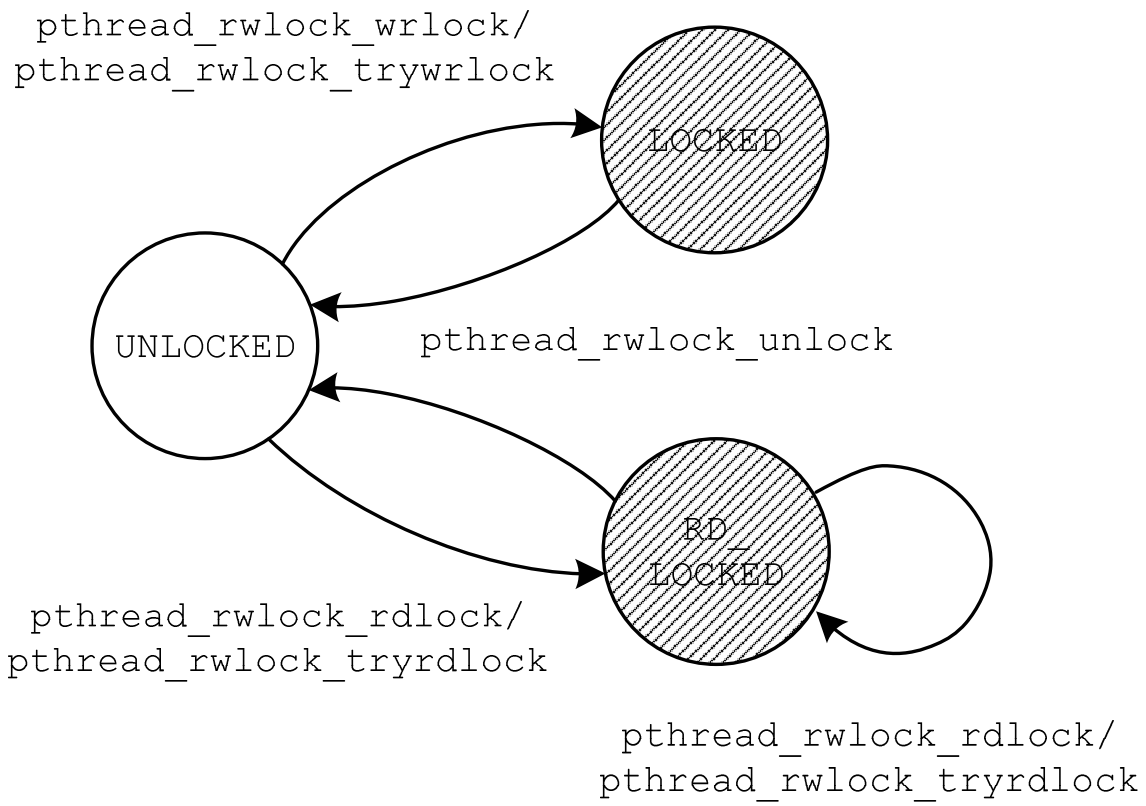
Пример программы с мьютексом 4

```
int a = 0;
pthread_mutex_t mutex;
void* f(void* b) {
    if (!pthread_mutex_trylock(&mutex)){
       a = a + 1;
       pthread_mutex_unlock(&mutex);
    }
}
int main(){
    pthread_create(&t, NULL, f, NULL);
    pthread_mutex_lock(&mutex);
    int b = a;
    pthread_mutex_unlock(&mutex);
    printf("%d %d", a, b);  // ?
}
```

Функции для работы с RWLock

- `int` **`pthread_rwlock_init`** `(pthread_rwlock_t *,const pthread_rwlockattr_t *)` – инициализация объекта синхронизации
- `int` **`pthread_rwlock_destroy`** `(pthread_rwlock_t *)` – уничтожение объекта синхронизации;
- `int` **`pthread_rwlock_rdlock`** `(pthread_rwlock_t *)` – блокировка потока для выполнения операции чтения;
- `int` **`pthread_rwlock_wrlock`** `(pthread_rwlock_t *)` – блокировка потока для выполнения операции записи;
- `int` **`pthread_rwlock_tryrdlock`** `(pthread_rwlock_t *)` – попытка блокировки потока для выполнения операции чтения;
- `int` **`pthread_rwlock_trywrlock`** `(pthread_rwlock_t *)` – попытка блокировки потока для выполнения операции записи;
- `int` **`pthread_rwlock_unlock`** `(pthread_rwlock_t *)` – освобождение объекта синхронизации

Состояния объекта типа RWLock

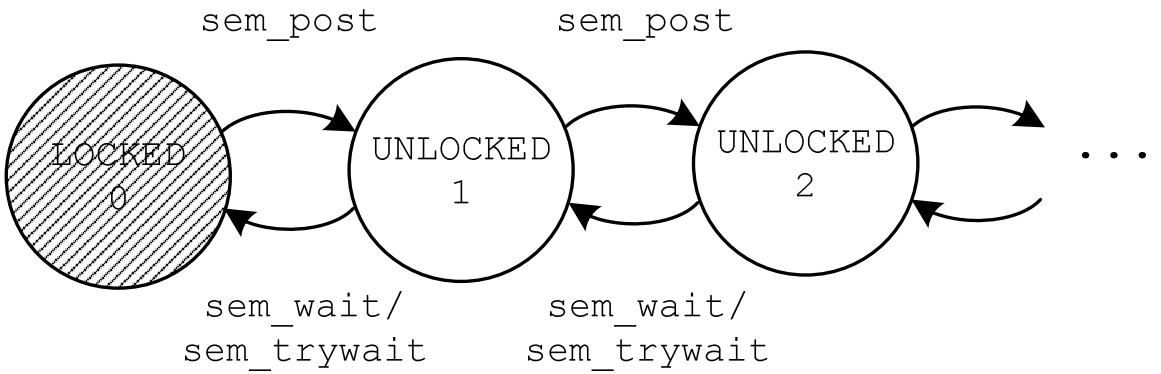


Функции для работы с семафором

- `int` **`sem_init`** `(sem_t *, int, unsigned)` – инициализация семафора, устанавливается начальное значение семафора
- `int` **`sem_destroy`** `(sem_t *)` – уничтожение семафора
- `int` **`sem_post`** `(sem_t *)` – увеличение значения семафора. В случае если значение семафора равно нулю и имеются потоки, ожидающие открытия семафора на конструкции sem_wait, один из таких потоков разблокируется и продолжает свое выполнение. В случае если семафор имеет положительное значение или нет ожидающих потоков, то значение семафора просто увеличивается на единицу
- `int` **`sem_wait`**`(sem_t *)` – уменьшение значения семафора. В случае если значение семафора равно нулю вызывающий поток блокируется до тех пор, пока не будет разблокирован вызовом sem_post. В случае если семафор имеет положительное значение, это значение уменьшается на единицу
- `int` **`sem_trywait`**`(sem_t *)` – попытка уменьшения значения семафора
- `int` **`sem_getvalue`**`(sem_t *restrict, int *restrict)` – получение текущего состояния семафора

Состояния семафора

- Функции для семафора могут вызываться из произвольных потоков



Пример программы с семафором 1

```
#include <semaphore.h>
sem_t sem;

int main(){
  sem_init(&sem, 0, 0);
  pthread_create(&t, NULL, f, NULL);
  ...
  sem_wait(&sem);
  sem_post(&sem);
  ...
  sem_destroy(&sem);
}
```

Пример программы с семафором 2

```
int a = 0;
sem_t sem;
void* f(void* b) {
   a = a + 1;
   sem_post(&sem);
}
int main(){
   sem_init(&sem, 0, 0);
   pthread_create(&t, NULL, f, NULL);
   sem_wait(&sem);
   int b = a;
   pthread_join(t, NULL);
   printf("%d %d", a, b);  // ?
}
```

Пример программы с семафором 3

```
int a = 0;
sem_t sem;
void* f(void* b) {
    a = a + 1;
    sem_post(&sem);
}
int main() {
    sem_init(&sem, 0, 0);
    pthread_create(&t, NULL, f, NULL);
    if (!sem_trywait(&sem)) {
       a = a + 1;
    }
    printf("%d", a); // ?
}
```

Функции для работы с condition variables

- `int` **`pthread_cond_init`** `(pthread_cond_t *, const pthread_condattr_t *)` – инициализация переменной условия
- `int` **`pthread_cond_destroy`** `(pthread_cond_t *)` – уничтожение переменной условия
- `int` **`pthread_cond_wait`** `(pthread_cond_t*, pthread_mutex_t*)` – атомарное освобождение мьютекса и переход к ожиданию события изменения состояния переменной условия, блокировка мьютекса и завершение
- `int` **`pthread_cond_signal`** `(pthread_cond_t *cond)` – генерация события изменения состояния переменной условия для одного потока, ожидающего на вызове функции pthread_cond_wait
- `int` **`pthread_cond_broadcast`** `(pthread_cond_t *cond)` – генерация события изменения состояния переменной условия для всех потоков, ожидающих на вызове функции pthread_cond_wait

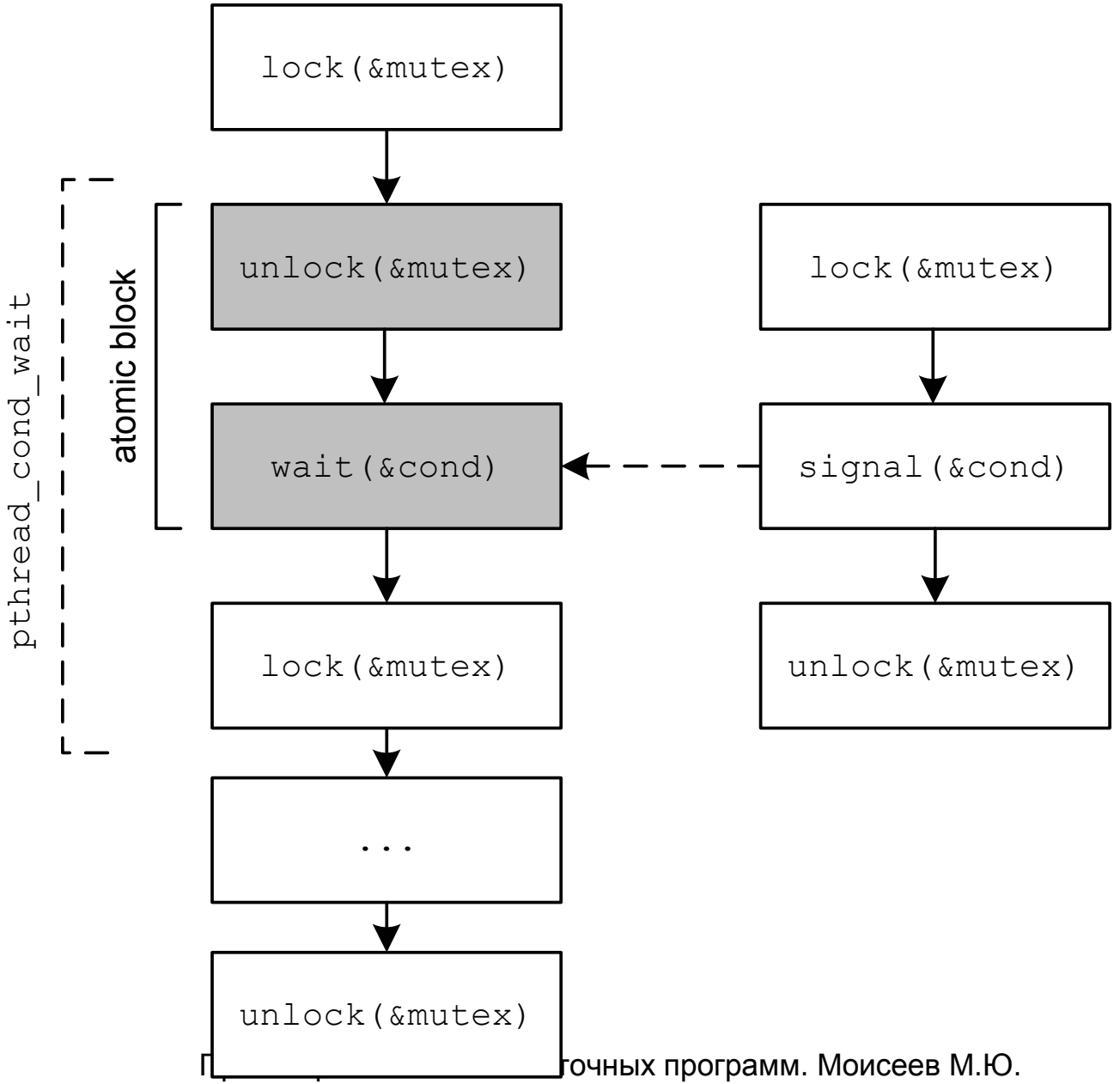
Использование condition variables

- Используются два объекта синхронизации
  - мьютекс – обеспечивает атомарность обработки сигналов
  - переменная условия – реагирует на полученный сигнал
- Вызов функции `pthread_cond_wait` обязательно выполняется под защитой мьютекса
- Вызов функций `pthread_cond_signal` и `pthread_cond_broadcast` также обычно выполняется под защитой того же мьютекса

```
pthread_mutex_lock(&mutex);
pthread_cond_wait(&cond, &mutex);
... //
pthread_mutex_unlock(&mutex);
```

```
pthread_mutex_lock(&mutex);
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mutex);
```

Использование condition variables



Пример программы с condition variable 1

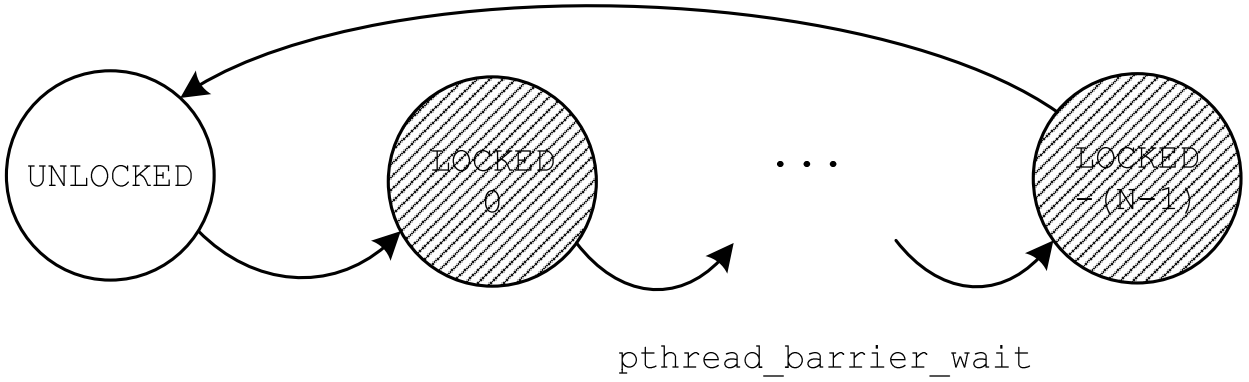
```
int a = 0;
pthread_cond_t cond;
pthread_mutex_t mutex;

int f() {
   pthread_mutex_lock(&mutex);
   a++;
   if (a < SOME_VALUE) {  // Проверка некоторого условия
      pthread_cond_wait(&cond, &mutex);
   } else {
      pthread_cond_broadcast(&cond); // Разбудить всех
   }
   pthread_mutex_unlock(&mutex);
}
```

Пример программы с condition variable 2

```
int a = 0;
int producer() {
   pthread_mutex_lock(&mutex);
   a++;
   if (a => 2) {
         pthread_cond_signal(&cond);
   }
   pthread_mutex_unlock(&mutex);
}
void consumer() {
   pthread_mutex_lock(&mutex);
   pthread_cond_wait(&cond, &mutex);
   a = a – 2;
   pthread_mutex_unlock(&mutex);
}
```

Функции для работы с барьером

- `int` **`pthread_barrier_init`**`(pthread_barrier_t *, const pthread_barrierattr_t *, unsigned)` – инициализация барьера
- `int` **`pthread_barrier_destroy`**`(pthread_barrier_t *)` – уничтожение барьера
- `int` **`pthread_barrier_wait`**`(pthread_barrier_t *)` – ожидание открытия барьера, барьер открывается когда заданное число потоков вызовет функцию pthread_barrier_wait



Функции для работы с переменными однократного выполнения

- `int` **`pthread_once`** `(pthread_once_t *, void (*init_routine)(void))`
  – производит вызов функции в текущем потоке программы, если это первый вызов функции с данным объектом pthread_once_t, в противном случае вызов функции игнорируется

Функции для работы с ключами данных

- `int` **`pthread_key_create`**`(pthread_key_t* [, void(*destructor)(void*)])` – создание ключа
- `int` **`pthread_key_delete`**`(pthread_key_t)` – удаление ключа
- `int` **`pthread_setspecific`**`(pthread_key_t, const void *)` – связывание ключа с локальными данными текущего потока
- `void*` **`pthread_getspecific`**`(pthread_key_t)` – получение данных текущего потока по ключу

Создание параллельных программ на основе OpenMP

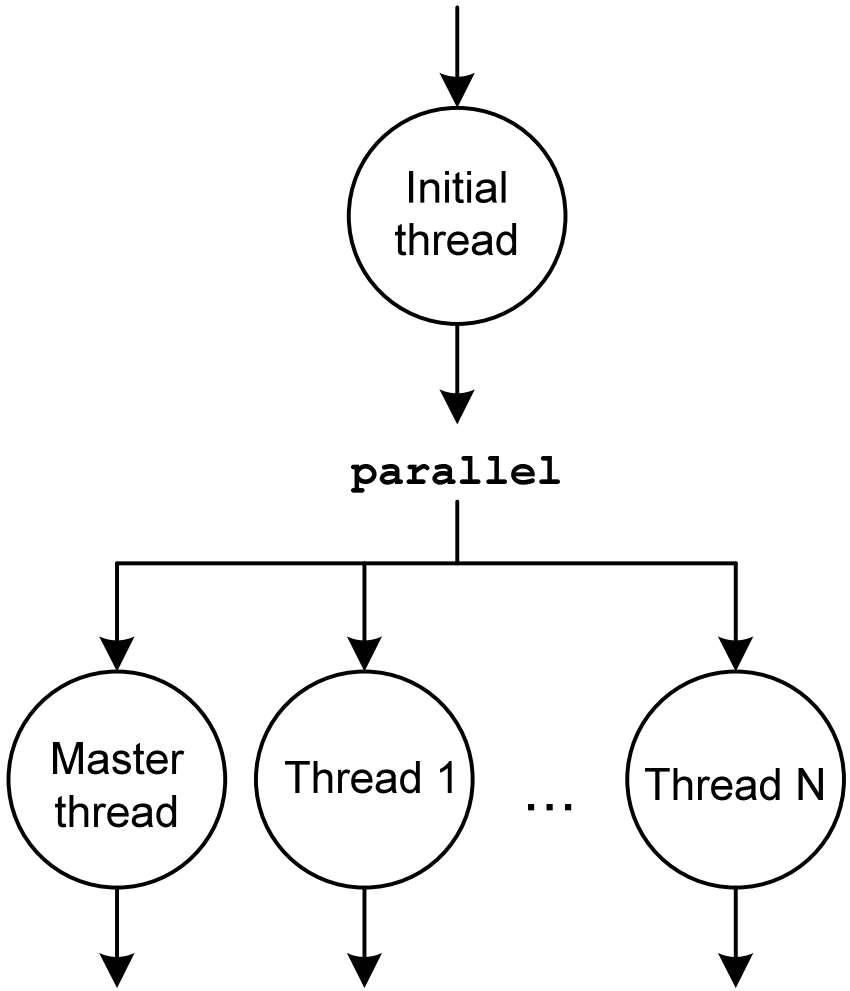
- OpenMP (Open Multi-Processing) – технология создания параллельных программ с общей памятью (http://www.openmp.org)
- OpenMP позволяет создавать программы, параллельно выполняющие одинаковые операции для разных данных (Single Program Multiple Data)
- OpenMP разработан для языков C, C++, Fortran
- Поддержка компиляторами
  - gcc, Oracle Solaris Studio, MS Visual Studio, Intel compilers…

Создание параллельных программ на основе OpenMP

- OpenMP представляет набор директив компилятора, библиотечных функций и переменных окружения
- Директивы OpenMP применяются к структурированным блокам
  - Структурированный блок – простой или составной оператор программы (`{ . . . }`), который имеет единственный вход и единственный выход
- Директивы OpenMP
  - для языков C/C++ – **`#pragma omp …`**
  - для языка Fortran – комментарий

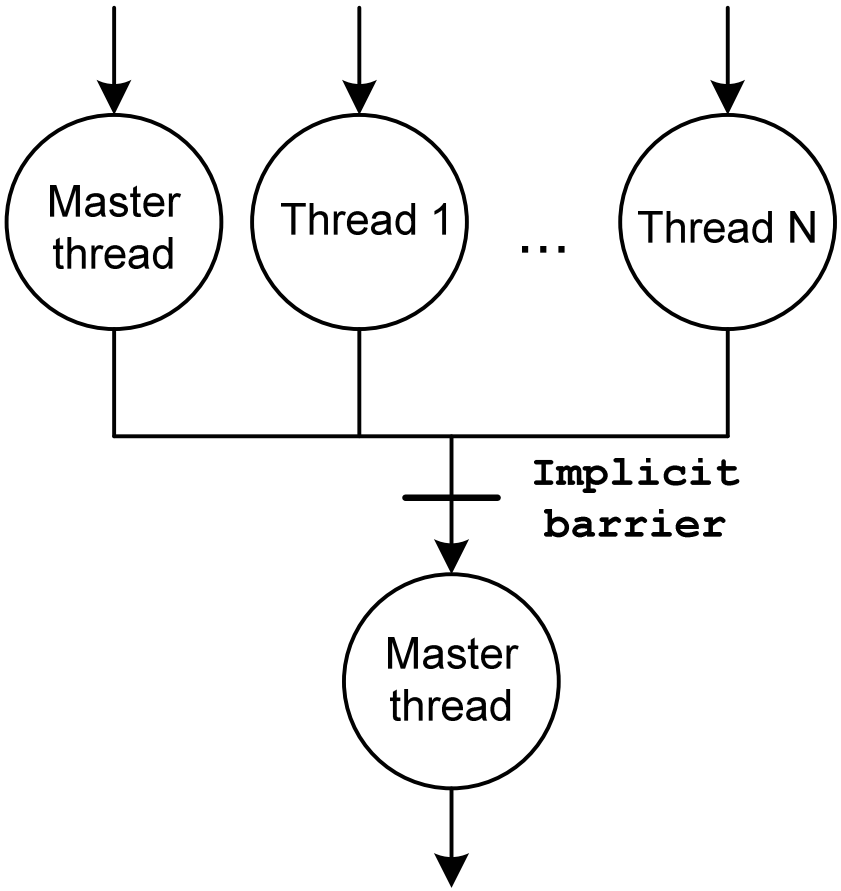
Модель параллельных вычислений OpenMP

- При старте программы запускает один начальный поток (Initial thread)
- При достижении директивы `parallel` порождается группа потоков, родительский поток также входит в эту группу как master thread
- С каждым потоком связывается задача, в которой выполняется код внутри директивы `parallel`
- Задачи в разных потоках параллельно обрабатывают разные данные



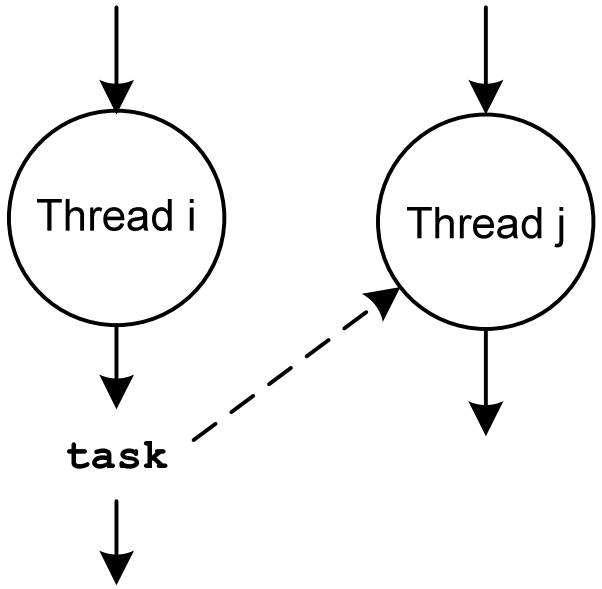
Модель параллельных вычислений OpenMP

- После завершения всех потоков продолжает выполнение master thread, для ожидания потоков используется барьер (неявно)
- Для того чтобы не ожидать завершение всех потоков можно использовать директиву `nowait`
- Программа может содержать произвольное число директив parallel
- Поддерживаются вложенные директивы `parallel` – внутри потока может создаваться своя группы потоков



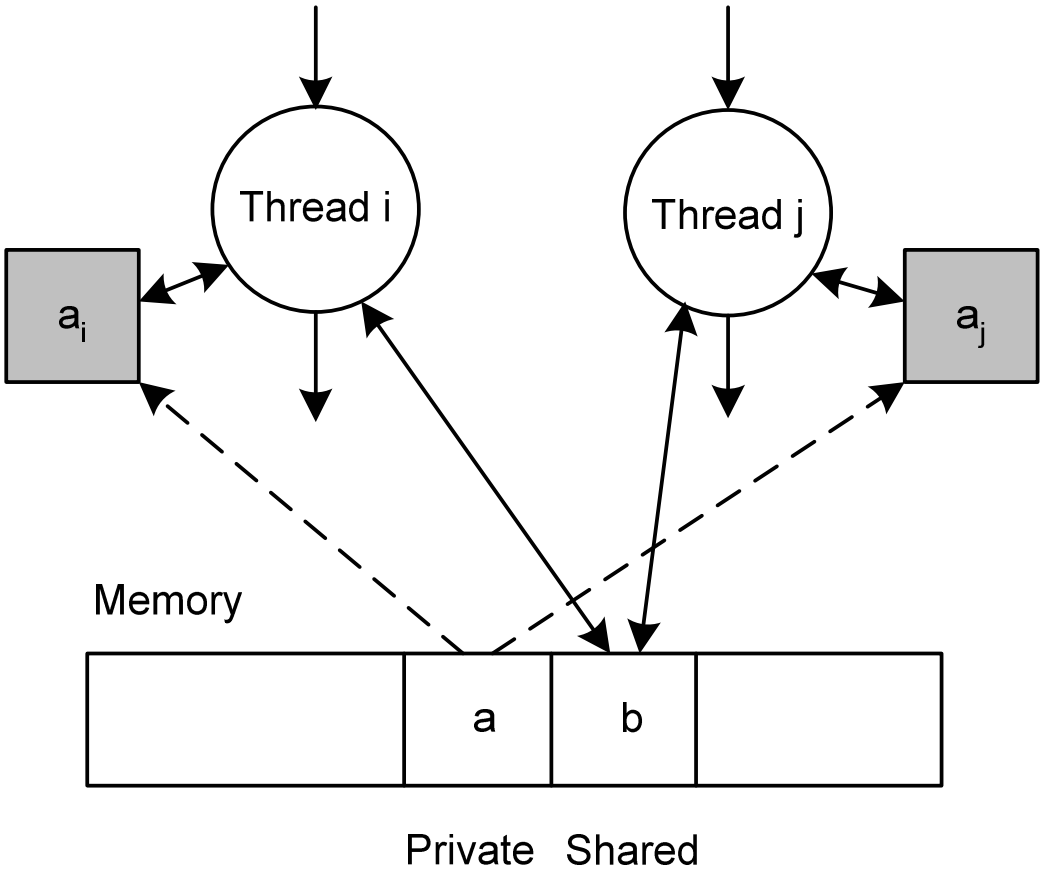
Модель параллельных вычислений OpenMP

- Директива `task` позволяет создавать явные задачи, которые будут выполняться в одном из потоков текущей группы
- Все задачи, созданные с помощью `task` завершаются до завершения потоков данной группы



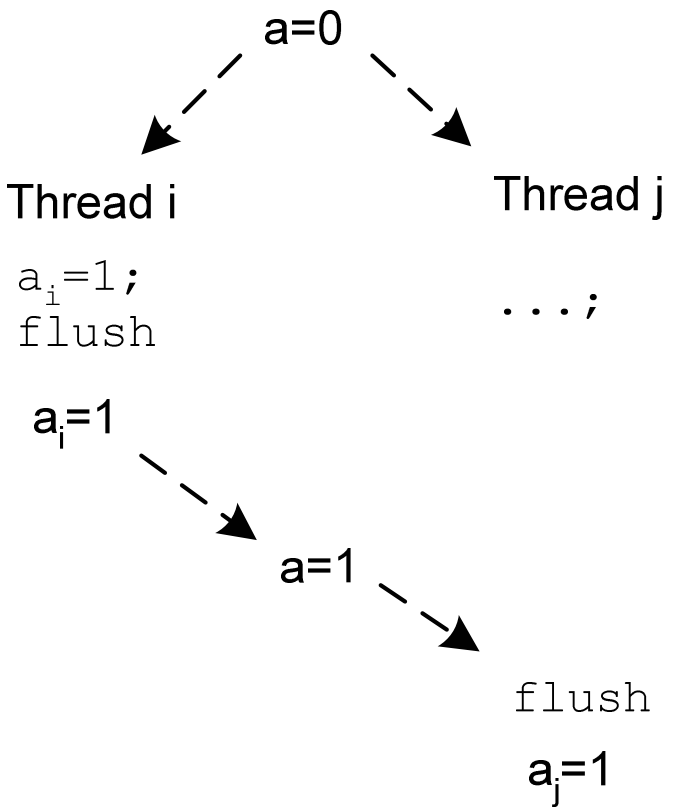
Модель памяти в OpenMP

- При создании потоков для переменной можно создать локальную копию для каждого потока (Private)
- Private переменные используются только в текущем потоке, могут инициализироваться значением оригинальной переменной, результат может быть сохранен в оригинальной переменной
- Shared переменные должны использоваться либо только на чтение, либо защищаться с помощью директив синхронизации



Модель памяти в OpenMP

- Relaxed-consistency модель памяти – значение локальных копий могут различаться в разных потоках
- Для синхронизации значения локальной копии переменной и переменной в памяти используется директива `flush`
- Директива `flush` определяет место последнего изменения переменной и выполняет запись локального значения в память или чтение значения из памяти в локальную копию (при изменении значения в другом потоке)
- Программист должен обеспечить отсутствие конкурентной модификации переменной для которой выполняется `flush`



Основные директивы OpenMP

- Создание и запуск группы параллельных потоков

```
#pragma omp parallel [clause[ [, ]clause] ...]
structured-block
```

```
clause: if(scalar-expression), num_threads(integer-expression),
default(shared | none), private(list), firstprivate(list),
shared(list), copyin(list), reduction(operator: list)
```

Пример создания параллельных потоков

```
#include <stdio.h>
#include <omp.h>
int main(){
   #pragma omp parallel num_threads(2)
   {
      if (omp_get_thread_num() == 0) {
         doSomething(); // In master thread
      } else {
         doSomethingElse(); // In other parallel thread
      }
   }
```

Пример разделяемых и локальных переменных

```
int main(){
   int x = 0; int y = 0;
   #pragma omp parallel num_threads(2) shared(x) private(y)
   {
      if (omp_get_thread_num() == 0) {
         x = 1;
         y = 1;
      } else {
          y = 2;
      }
   }
   printf("%d %d", x, y); // 1, 0
```

Директивы для циклов

- Распараллеливание итераций цикла в потоках текущей группы

```
#pragma omp for [clause[ [, ]clause] ...]
for-loops
clause:private(list), firstprivate(list), lastprivate(list),
reduction(operator: list), schedule(kind[, chunk_size]),
collapse(n), ordered, nowait
```

- Создание группы параллельных потоков для распараллеливания итераций цикла

```
#pragma omp parallel for [clause[ [, ]clause] ...]
for-loops
```

- `for(init-expr; test-expr; incr-expr) structured-block`
  - `for(i = 0; i < 10; i++) {...}`

Пример распараллеливания итераций цикла

```
void simple(int n, float *a, float *b)
{
   int i;
   #pragma omp parallel for
   for (i=1; i<n; i++){  /* i is private by default */
      b[i] = (a[i] + a[i-1]) / 2.0;
   }
}
```

Основные директивы OpenMP

- Выполнение конструкций указанного блока только в одном из потоков текущей группы

```
#pragma omp single [clause[ [, ]clause] ...]
structured-block
clause:private(list), firstprivate(list), copyprivate(list),
nowait
```

- Явное создание параллельной задачи

```
#pragma omp task [clause[ [, ]clause] ...]
structured-block
clause: if(scalar-expression), final(scalar-expression),
untied, default(shared | none), mergeable, private(list),
firstprivate(list), shared(list)
```

Пример работы директивы task

```
struct node {
   struct node *left;
   struct node *right;
}
void traverse( struct node *p ) {
   if (p->left)
   #pragma omp task // p is firstprivate by default
      traverse(p->left);
   if (p->right)
   #pragma omp task // p is firstprivate by default
      traverse(p->right);
   doSomething(p);
}
```

Пример работы директивы single

```
void single_example()
{
   #pragma omp parallel
   {
       #pragma omp single
          printf("Beginning doSomething1");
       doSomething1();
      #pragma omp single nowait
          printf("Finished doSomething1");
       doSomething2();
    }
}
```

Директивы синхронизации

- Ограничение на выполнение следующего блока одновременно только одним потоком

```
#pragma omp critical [(name)]
 structured-block
```

- Создание барьера `#pragma omp barrier`
- Ожидание завершения всех порожденных задач

```
#pragma omp taskwait
```

- Создание атомарного блока конструкций

```
#pragma omp atomic [read | write | update | capture]
expression-stmt | structured-block
```

- Синхронизация локальной копии переменной и переменной в памяти

```
#pragma omp flush [(list)]
```

Пример работы директивы critical

```
void critical_example(float *x)
{
   int ix_next;
   #pragma omp parallel shared(x) private(ix_next)
   {
       #pragma omp critical (xaxis)
          ix_next = dequeue(x);
       doSomething(ix_next);
    }
}
```

Пример работы директивы flush

```
int main(){
   int x = 0;
   #pragma omp parallel num_threads(2)
   {
      if(omp_get_thread_num()==0) {
          #pragma omp atomic
          x++;
       } else {
          while(x == 0){
              #pragma omp flush(x)
          }
          ...
       }
```

Пример работы директивы reduction

```
void reduction1(float *x, int *y, int n)
{
  int i, c; float a, d;
  a = 0.0; c = y[0]; d = x[0];
  #pragma omp parallel for private(i) shared(x, y, n) \
  reduction(+:a) reduction(min:c) reduction(max:d)
  for (i=0; i<n; i++) {
     a += x[i];
     if (c > y[i]) c = y[i];
     d = fmaxf(d,x[i]);
  }
}
```

Runtime Library Routines

- `void omp_set_num_threads(int num_threads)`
- `int omp_get_num_threads(void)`
- `int omp_in_parallel(void)`
- `int omp_get_thread_num(void)`