
COMP 322: Principles of Parallel Programming

Lecture 10: POSIX Threads (Chapter 6)

Fall 2009

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu



Summary of Previous Lecture

- Case study: Successive Over-Relaxation
- HJ Phasers
 - SIG, WAIT, SIG_WAIT registration modes
 - next statement (barrier)
 - signal statement (split-phase barrier)
 - single statement (computation during phase transition)

Acknowledgments for Today's Lecture

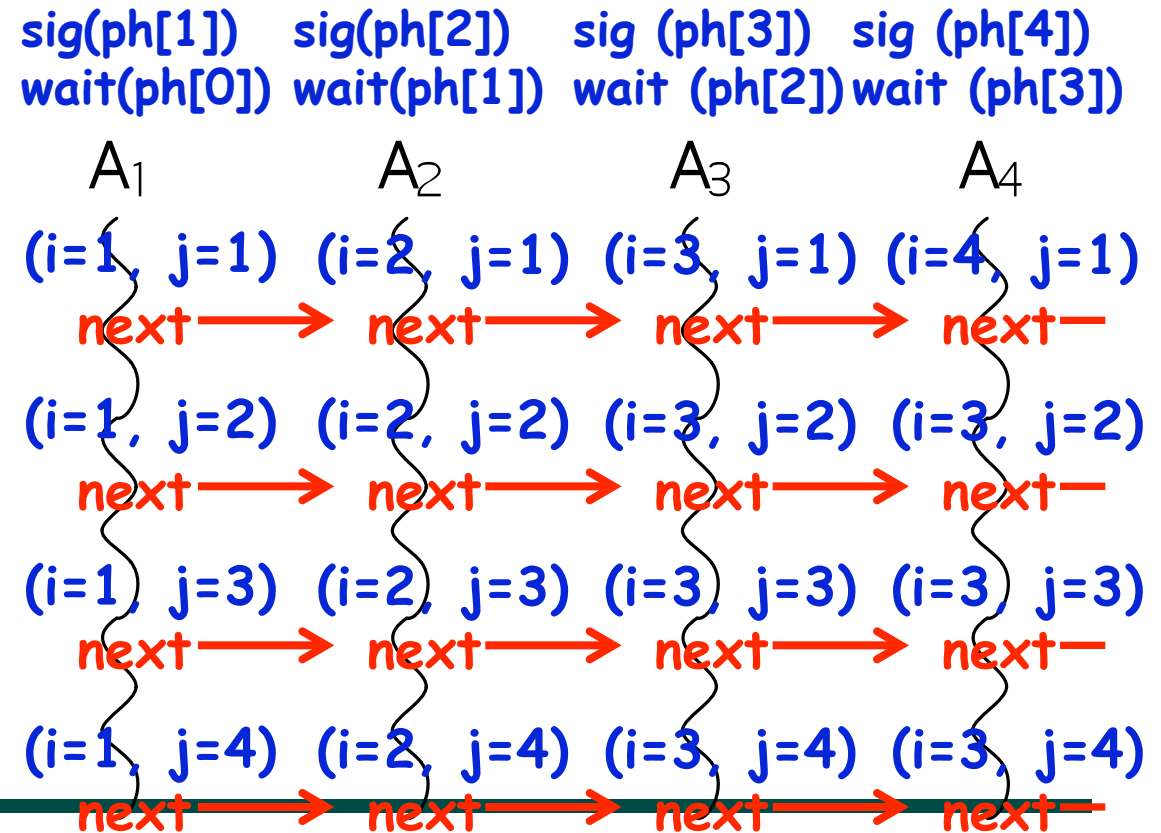
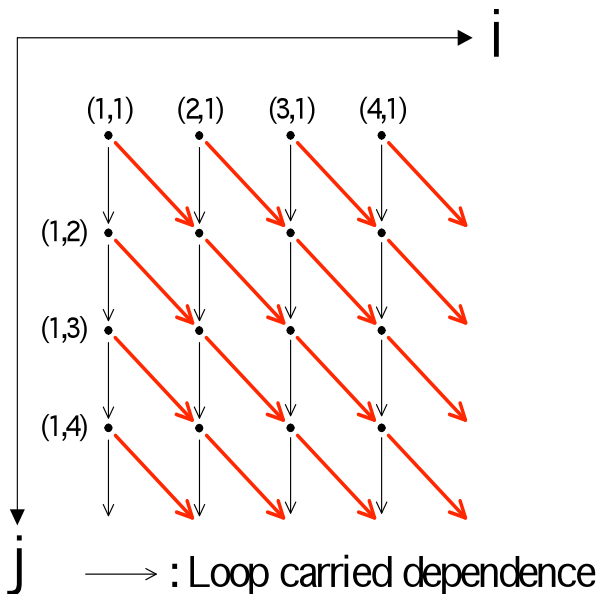
- Course text: "Principles of Parallel Programming", Calvin Lin & Lawrence Snyder
 - Includes resources available at <http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html>
- "Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-point Synchronization", Jun Shirako, David M. Peixotto, Vivek Sarkar, William N. Scherer III
 - <http://www.cs.rice.edu/~vs3/PDF/SPSS08-phasers.pdf>
- "Programming Shared-memory Platforms with Pthreads", John Mellor-Crummey, COMP 422 Lecture 6, January 2009
 - <http://www.owlnet.rice.edu/~comp422/lecture-notes/comp422-Lecture6-Pthreads.pdf>

Example of Pipeline Parallelism with Phasers

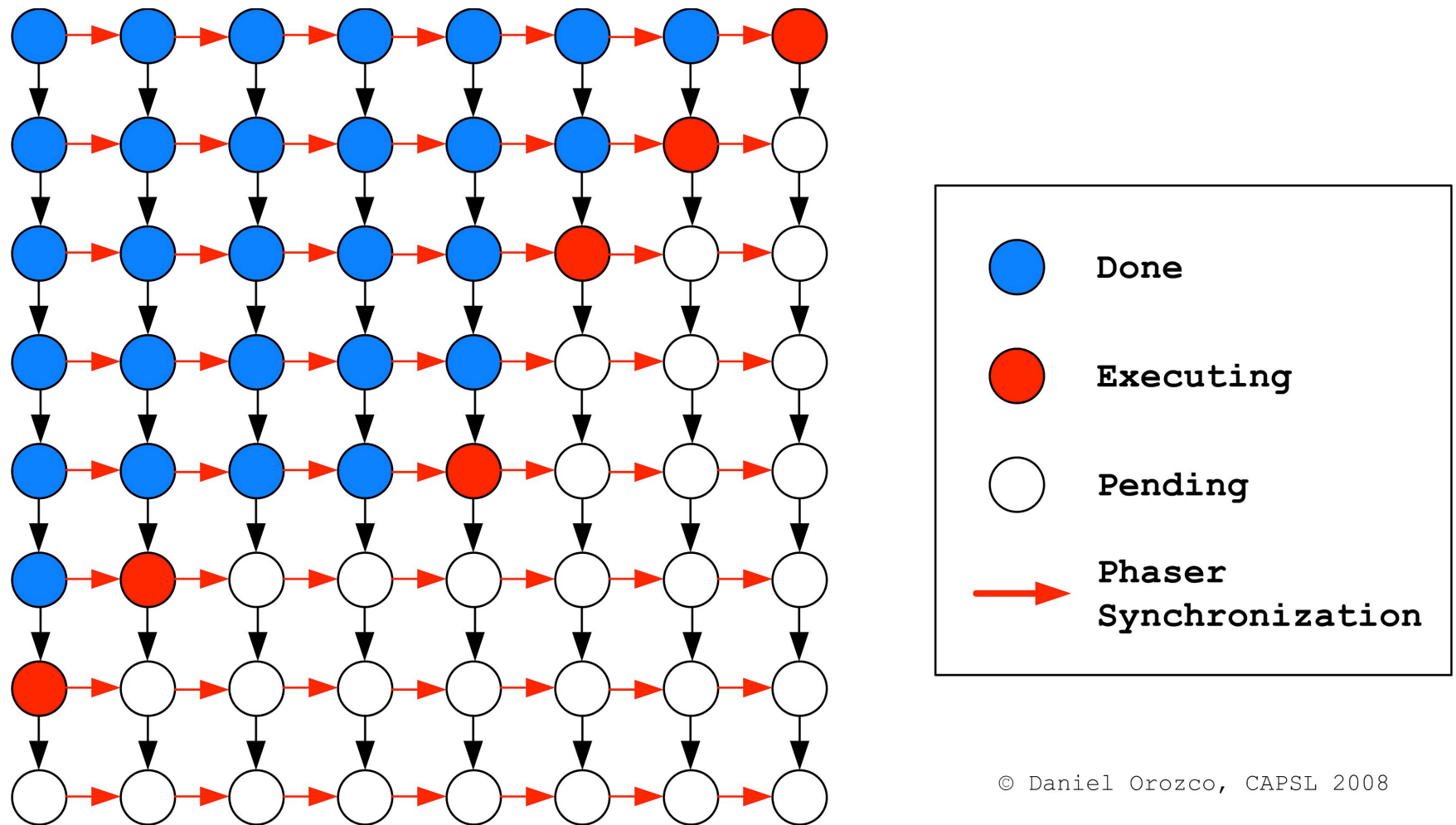
```

finish {
  phaser [] ph = new phaser[m+1];
  for (int i = 1; i < m; i++)
    async phased (ph[i]<SIG>, ph[i-1]<WAIT>) {
      for (int j = 1; j < n; j++) {
        a[i][j] = foo(a[i][j], a[i][j-1], a[i-1][j-1]);
        next;
      } // for
    } // finish
}

```



Example of Pipeline Parallelism with Phasers (contd)



© Daniel Orozco, CAPSL 2008

POSIX Threads

- Standard user threads API supported by most vendors
- Concepts behind Pthreads interface are broadly applicable
 - largely independent of the API
 - useful for programming with other thread APIs as well
 - NT threads
 - Solaris threads
 - Java threads
 - ...
- Threads are peers, unlike Linux/Unix processes
 - no parent/child relationship
- *You're now leaving HJ-land ... Here be dragons!*

PThread Creation (async)

Asynchronously invoke **thread_function** in a new thread

```
#include <pthread.h>
int pthread_create(
    pthread_t *thread_handle, /* returns handle here */
    const pthread_attr_t *attribute,
    void * (*thread_function)(void *),
    void *arg); /* single argument; perhaps a structure */
```

attribute created by **pthread_attr_init**

contains details about

- whether scheduling policy is inherited or explicit
- scheduling policy, scheduling priority
- stack size, stack guard region size

Can use NULL for **pthread_attr_init** for default values

Code Spec 6.1 `pthread_create()`. The POSIX Threads thread creation function.

`pthread_create()`

```
int pthread_create(                // create a new thread
    pthread_t *tid,                // thread ID
    const pthread_attr_t *attr,     // thread attributes
    void *(*start_routine)(void *), // pointer to function to execute
    void *arg                       // argument to function
);
```

Arguments:

- The thread ID of the successfully created thread.
- The thread's attributes, explained below; the NULL value specifies default attributes.
- The function that the new thread will execute once it is created.
- An argument passed to the `start_routine()`.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Notes:

Use a structure to pass multiple arguments to the start routine.

Code Spec 6.6 pthread attributes. An example of how thread attributes are set in the POSIX Threads interface.

Thread Attributes

```
pthread_attr_t attr;           // Declare a thread attribute
pthread_t tid;
pthread_attr_init(&attr);      // Initialize a thread attribute
pthread_attr_setdetachstate(&attr, // Set the thread attribute
    PTHREAD_CREATE_UNDETACHED);
pthread_create(&tid, &attr, start_func, NULL); // Use the attribute
                                                // to create a thread
pthread_join(tid, NULL);
pthread_attr_destroy(&attr);    // Destroy the thread attribute
```

Notes:

There are many other thread attributes. See a POSIX Threads manual for details.

Pthread Termination (force)

- A thread terminates by calling the function `pthread_exit()`. A single argument, a pointer to a `void*` object, is supplied as the argument to `pthread_exit`. This value is returned to any thread that has blocked while waiting for this thread to exit.
- Suspend parent thread until child thread terminates

```
#include <pthread.h>
int pthread_join (
    pthread_t thread, /* thread id */
    void **ptr); /* ptr to location for return code a terminating
                  thread passes to pthread_exit */
```

Code Spec 6.2 `pthread_join()`. The POSIX Threads rendezvous function.

`pthread_join()`

```
int pthread_join(                // wait for a thread to terminate
    pthread_t tid,              // thread ID to wait for
    void **status                // exit status
);
```

Arguments:

- The ID of the thread to wait for.
- The completion status of the exiting thread will be copied into `*status` unless `status` is `NULL`, in which case the completion status is not copied.

Return value:

0 for success. Error code from `<errno.h>` otherwise.

Notes:

Once a thread is joined, the thread no longer exists, its thread ID is no longer valid, and it cannot be joined with any other thread.

Example: Creation and Termination (main)

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 32
void *compute_pi (void *);
...
int main(...) {
    ...
    pthread_t p_threads[NUM_THREADS];
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    for (i=0; i< NUM_THREADS; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void*) &hits[i]);
    }
    for (i=0; i< NUM_THREADS; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    ...
}
```

default attributes

thread function

thread argument

Example: Thread Function (compute_pi)

```
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double x_coord, y_coord;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        x_coord = (double)(rand_r(&seed))/((1<<15)-1) - 0.5;
        y_coord = (double)(rand_r(&seed))/((1<<15)-1) - 0.5;
        if ((x_coord * x_coord + y_coord * y_coord) < 0.25)
            local_hits++;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

tally how many random
points fall in a unit circle
centered at the origin

rand_r: reentrant random
number generation in
[0,2¹⁵-1]

Race Conditions in a Pthreads Program

Consider

```
/* threads compete to update global variable best_cost */  
if (my_cost < best_cost)  
    best_cost = my_cost;
```

—two threads

—initial value of best_cost is 100

—values of my_cost are 50 and 75 for threads t1 and t2

- After execution, best_cost could be 50 or 75
- 75 does not correspond to any serialization of the threads

Critical Sections and Mutual Exclusion (isolated)

- Critical section = must execute code by only one thread at a time

```
/* threads compete to update global variable best_cost */
```

```
if (my_cost < best_cost)
```

```
    best_cost = my_cost;
```

- Mutex locks enforce critical sections in Pthreads
 - mutex lock states: locked and unlocked
 - only one thread can lock a mutex lock at any particular time
- Using mutex locks
 - request lock before executing critical section
 - enter critical section when lock granted
 - release lock when leaving critical section

created by
`pthread_mutex_attr_init`
specify type:
normal, recursive, errorcheck

- Operations

```
int pthread_mutex_init (pthread_mutex_t *mutex_lock,  
                        const pthread_mutexattr_t *lock_attr)
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex_lock)
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex_lock)
```

Code Spec 6.7 The POSIX Threads routines for acquiring and releasing mutexes.

Acquiring and Releasing Mutexes

```
int pthread_mutex_lock(           // Lock a mutex
    pthread_mutex_t *mutex);
int pthread_mutex_unlock(         // Unlock a mutex
    pthread_mutex_t *mutex);
int pthread_mutex_trylock(        // Nonblocking lock
    pthread_mutex_t *mutex);
```

Arguments:

Each function takes the address of a mutex variable.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Notes:

The `pthread_mutex_trylock()` routine attempts to acquire a mutex but will not block. This routine returns the POSIX Threads constant `EBUSY` if the mutex is locked.

Code Spec 6.8 The POSIX Threads routines for dynamically creating and destroying mutexes.

Mutex Creation and Destruction

```
int pthread_mutex_init(           // Initialize a mutex
    pthread_mutex_t *mutex,
    pthread_mutexattr_t *attr);
int pthread_mutex_destroy(        // Destroy a mutex
    pthread_mutex_t *mutex);
int pthread_mutexattr_init(       // Initialize a mutex attribute
    pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(    // Destroy a mutex attribute
    pthread_mutexattr_t *attr);
```

Arguments:

- The `pthread_mutex_init()` routine takes two arguments, a pointer to a mutex and a pointer to a mutex attribute. The latter is presumed to have already been initialized.
- The `pthread_mutexattr_init()` and `pthread_mutexattr_destroy()` routines take a pointer to a mutex attribute as arguments.

Notes:

If the second argument to `pthread_mutex_init()` is `NULL`, default attributes will be used.

Code Spec 6.9 An example of how dynamically allocated mutexes are used in the POSIX Threads interface.

Dynamically Allocated Mutexes

```
pthread_mutex_t *lock;                // Declare a pointer to a lock
lock=(pthread_mutex_lock_t *) malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL);
/*
 * Code that uses this lock.
 */
pthread_mutex_destroy(lock);
free(lock);
```

Mutex Types

- Normal
 - thread deadlocks if tries to lock a mutex it already has locked
- Recursive
 - single thread may lock a mutex as many times as it wants
 - increments a count on the number of locks
 - thread relinquishes lock when mutex count becomes zero
- Errorcheck
 - report error when a thread tries to lock a mutex it already locked
 - report error if a thread unlocks a mutex locked by another thread


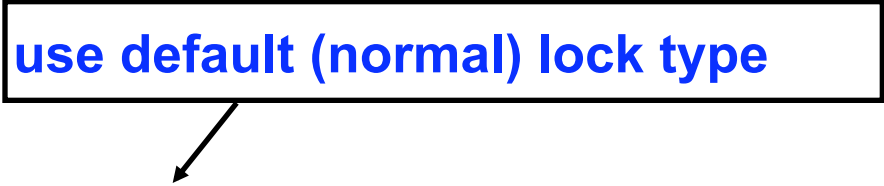
Example: Reduction Using Mutex Locks

```
pthread_mutex_t cost_lock;
...
int main() {
    ...
    pthread_mutex_init(&cost_lock, NULL);
    ...
}

void *find_best(void *list_ptr) {
    ...
    pthread_mutex_lock(&cost_lock);    /* lock the mutex */
    if (my_cost < best_cost)
        best_cost = my_cost;
    pthread_mutex_unlock(&cost_lock); /* unlock the mutex */
}
```

use default (normal) lock type

critical section



Producer-Consumer Using Mutex Locks


Constraints

- **Producer thread**
 - must not overwrite the shared buffer until previous task has picked up by a consumer
- **Consumer thread**
 - must not pick up a task until one is available in the queue
 - must pick up tasks one at a time

Producer-Consumer Using Mutex Locks

```
pthread_mutex_t task_queue_lock;
int task_available;
...
main() {
    ...
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    ...
}
void *producer(void *producer_thread_data) {
    ...
    while (!done()) {
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task); task_available = 1;
                inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
```

critical section



Producer-Consumer Using Locks

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    /* local data structure declarations */
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0;
                extracted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
        process_task(my_task);
    }
}
```

critical section



Overheads of Locking

- Locks enforce serialization
 - threads must execute critical sections one at a time
- Large critical sections can seriously degrade performance
- Reduce overhead by overlapping computation with waiting

```
int pthread_mutex_trylock(pthread_mutex_t *mutex_lock)
```

- acquire lock if available
- return EBUSY if not available
- enables a thread to do something else if lock unavailable

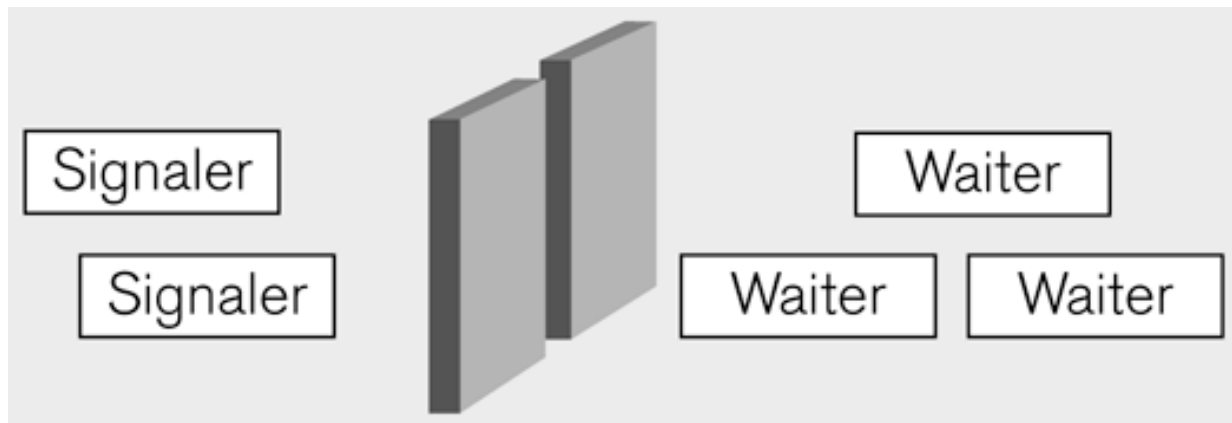
Condition Variables for Synchronization

Condition variable: associated with a **predicate** and a **mutex**

- Using a condition variable
 - thread can block itself until a condition becomes true
 - thread locks a mutex
 - tests a predicate defined on a shared variable
 - if predicate is false, then wait on the condition variable
 - waiting on condition variable unlocks associated mutex
 - when some thread makes a predicate true
 - that thread can signal the condition variable to either
 - wake one waiting thread
 - wake all waiting threads
 - when thread releases the mutex, it is passed to first waiter

Condition Variables

- Figure 6.2. Condition variables act like a gate. Threads wait outside the gate by calling `pthread_cond_wait()`, and threads open the gate by calling `pthread_cond_signal()`.



Pthread Condition Variable API

/ initialize or destroy a condition variable */*

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

/ block until a condition is true */*

```
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 *wtime);
```

abort wait if time exceeded

/ signal one or all waiting threads that condition is true */*

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

wake one

wake all

Condition Variable Producer-Consumer (main)

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */
main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);
    /* create and join producer and consumer threads */
}
```

default
initializations



Producer Using Condition Variables

```
void *producer(void *producer_thread_data) {  
    int inserted;  
    while (!done()) {  
        create_task();  
        pthread_mutex_lock(&task_queue_cond_lock);  
        while (task_available == 1)  
            pthread_cond_wait(&cond_queue_empty,  
                             &task_queue_cond_lock);  
        insert_into_queue();  
        task_available = 1;  
        pthread_cond_signal(&cond_queue_full);  
        pthread_mutex_unlock(&task_queue_cond_lock);  
    }  
}
```

releases mutex on wait



note
loop



reacquires mutex when woken



Consumer Using Condition Variables

```
void *consumer(void *consumer_thread_data) {  
    while (!done()) {  
        pthread_mutex_lock(&task_queue_cond_lock);  
        while (task_available == 0)  
            pthread_cond_wait(&cond_queue_full,  
                             &task_queue_cond_lock);  
        my_task = extract_from_queue();  
        task_available = 0;  
        pthread_cond_signal(&cond_queue_empty);  
        pthread_mutex_unlock(&task_queue_cond_lock);  
        process_task(my_task);  
    }  
}
```

note loop {

releases mutex on wait

reacquires mutex when woken

Code Spec 6.10 `pthread_cond_wait()`. The POSIX Thread routines for waiting on condition variables.

```
pthread_cond_wait()  
  
int pthread_cond_wait(  
    pthread_cond_t *cond,                // Condition to wait on  
    pthread_mutex_t *mutex);             // Protecting mutex  
  
int pthread_cond_timedwait(  
    pthread_cond_t *cond,  
    pthread_mutex_t *mutex,  
    const struct timespec *abstime);      // Time-out value
```

Arguments:

- A condition variable to wait on.
- A mutex that protects access to the condition variable. The mutex is released before the thread blocks, and these two actions occur atomically. When this thread is later unblocked, the mutex is reacquired on behalf of this thread.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Code Spec 6.11 `pthread_cond_signal()`. The POSIX Threads routines for signaling a condition variable.

```
pthread_cond_signal()  
  
int pthread_cond_signal(  
    pthread_cond_t *cond);           // Condition to signal  
int pthread_cond_broadcast(  
    pthread_cond_t *cond);          // Condition to signal
```

Arguments:

A condition variable to signal.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Notes:

- These routines have no effect if there are no threads waiting on `cond`. In particular, there is no memory of the signal when a later call is made to `pthread_cond_wait()`.
- The `pthread_cond_signal()` routine may wake up more than one thread, but only one of these threads will hold the protecting mutex.
- The `pthread_cond_broadcast()` routine wakes up all waiting threads. Only one awakened thread will hold the protecting mutex.

Barriers (Phasers)

- Pthreads provides only basic synchronization constructs
- Build higher-level constructs from basic ones e.g., *barriers*
- Pthreads extension includes barriers as synchronization objects (available in Single UNIX Specification)
 - Enable by `#define _XOPEN_SOURCE 600` at start of file
- Initialize a barrier for count threads
 - `int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrier_attr_t *attr, int count);`
- Each thread waits on a barrier by calling
 - `int pthread_barrier_wait(pthread_barrier_t *barrier);`
- Destroy a barrier
 - `int pthread_barrier_destroy(pthread_barrier_t *barrier);`

Summary of Today's Lecture

- Example of Pipeline Parallelism with HJ Phasers
- Overview of POSIX Threads (Chapter 6)
- Question for you to think about:
 - Which POSIX Threads examples in the lecture or book cannot be expressed using HJ constructs?