# Chapter 11
# Threads

**Cheng-Hsin Hsu**
*National Tsing Hua University*
*Department of Computer Science*

Parts of the course materials are courtesy of Prof. Chun-Ying Huang

# Outline

- Overview and introduction
- Thread creation
- Thread termination
- Thread synchronization

# Introduction

- We have introduced the relationships between processes
  - There is only a limited amount shares can occur between related processes
- Here we are going to introduce threads
  - It is able to perform multiple tasks within a single process
  - All threads within a single process have access to the same process components, e.g,. file descriptors and memory
- If a single resource is shared among multiple threads
  - We need synchronization mechanisms to prevent multiple threads from viewing inconsistencies in their shared resources

# Thread Concepts

- A typical UNIX process can be thought of as having a single thread of control
  - Each process is doing only one thing at a time
- With multiple threads of control
  - We can design our programs to do more than one thing at a time within a single process
  - Each thread handles a task
- Benefits of using multiple threads
  - Simplify code that deals with asynchronous events
  - Shares the same memory spaces and file descriptors
  - Problems can be partitioned to improve overall program throughput
  - Interactive programs can realize improved response time

# Thread Concepts (Cont'd)

- Multithread programming can be realized also on a single processor
  - The performance still gets improved as there is always I/O operations that block the execution of a process
  - However, if you have a multiprocessor system, threaded program may run faster
- A thread consists of the information necessary to represent an execution context within a process
  - Thread ID
  - Register values, stack content, a signal mask, an errno variable
  - Scheduling priority and policy, and
  - Thread specific data

# The UNIX Thread Standard

- Defined by POSIX
  - Portable Operating System Interface for UNIX
  - POSIX.1-2001
  - Also known as “pthreads” or “POSIX threads”
- Build programs with thread supports
  - Your program has to include <pthread.h>
  - To compile a C/C++ program with thread support, you have to add “-pthread” or “-lpthread” argument when compiling with gcc

# Linux Implementation of POSIX Threads

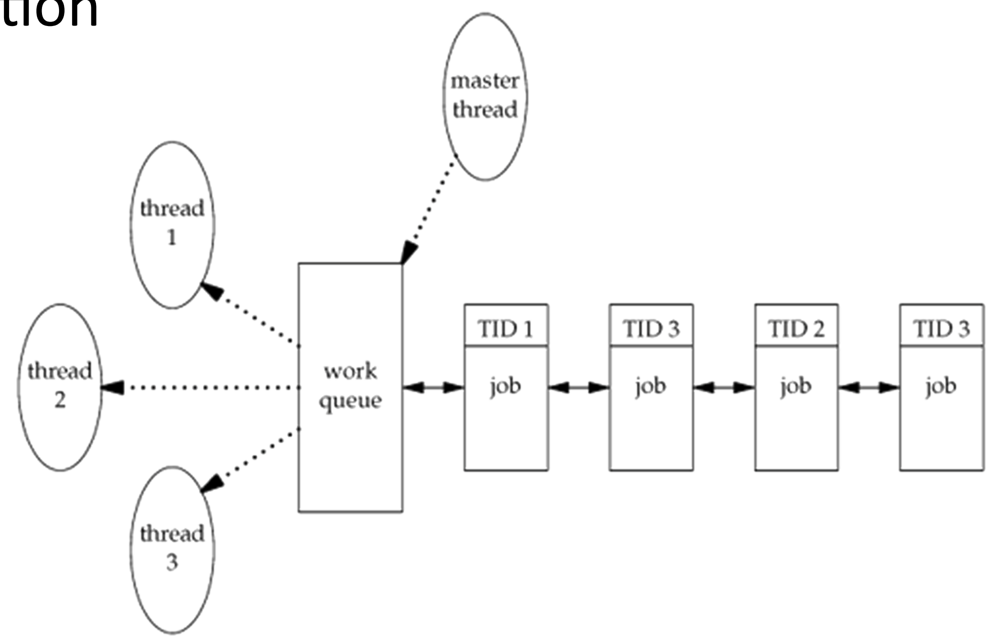
- Threads are implemented via the clone system call
  - Basically, they are processes sharing more information
- Two flavors: LinuxThreads and NPTL
- LinuxThreads: The original thread implementation
- NPTL: Native POSIX Thread Library
  - Better conformance to POSIX.1
    - For example, POSIX.1 requires threads of a process obtaining the same PID value when calling getpid(), but LinuxThreads does not follow it
  - Better performance
  - Require supports from the C library and the kernel
- Both are 1:1 thread model
- That is, each thread maps to a kernel scheduling entity

# Thread Identification

- Every thread has a thread ID
  - It may be not unique in the system
  - The thread ID has significance only within the context of the process to which it belongs
- The pthread_t data type
  - Similar to pid_t, pthread_t is used to identify a thread
  - It can be a structure (not forced to be an integer)
- Test the equivalence of thread IDs
  - int pthread_equal(pthread_t tid1, pthread_t tid2);
  - Returns: nonzero if equal, zero otherwise
- Get the current thread ID
  - pthread_t pthread_self(void);

# Thread ID: A Job Queue Example

- A master thread assign jobs to worker threads by their IDs
- A worker thread only removes the job tagged with its own thread ID
- This can be done by examining the thread ID using the pthread_equal function



# Thread Creation

- With pthreads, when a program runs, it also starts out as a single process with a single thread of control
- Create additional threads
  - int pthread_create(pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void*), void *restrict arg);
    - "thread" should be the address of a previous declared pthread_t variable
    - "attr" is used to customize thread attributes
    - The newly created thread starts running the "start_routine" function
    - The "arg" is then passed to the "start_routine" function
  - Returns: 0 if OK, error number on failure

# Thread Creation (Cont'd)

- When a thread is created …
  - There is no guarantee which thread runs first
    - The newly created thread or the calling thread?
  - The newly created thread
    - Has access to the process address space, and
    - Inherit floating-point environment and signal mask from the calling process
- pthread functions usually return an error code when they fail
  - They do not use the errno variable
  - It is not recommended to use global variables for status checks
  - However, the per thread copy of errno is still provided for compatibility

# Thread Creation, an Example

```
pthread_t ntid;
void printids(const char *s) {
    pid_t pid = getpid();
    pthread_t tid = pthread_self();
    printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid,
        (unsigned int) tid, (unsigned int) tid);
}
void * thr_fn(void *arg) {
    printids("new thread: ");  return((void *)0);
}
int main(void) {
    int err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_quit("can't create thread: %s\n", strerror(err));
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

# Thread Creation, an Example (Cont'd)

- The result can be different on vaious platforms
  - The pthread_t may be not an integer
  - The getpid() function may return different values for the two thread (although it is expected to return the same value)

```
$ ./fig11.2-threadid
new thread:  pid 3207 tid 3084950416 (0xb7e09b90)
main thread: pid 3207 tid 3084953264 (0xb7e0a6b0)
```

# Thread Termination

- Terminate the entire process
  - If any thread within a process calls exit, _Exit, or _exit
  - If received a signal with the default action of terminating the process
- Terminate a single thread
  - Return from the start routine.
    - The return value is the thread's exit code
  - Cancelled by another thread in the same process
  - The thread calls pthread_exit

# Thread Termination Status

- The exit status of a process can be retrieved using wait functions – wait, waitpid, …, etc
- The exit status of a thread can also be retrieved
- The pthread_join function
  - int pthread_join(pthread_t thread, void **value_ptr);
  - Returns: 0 if OK, error number on failure
  - This function suspends the calling thread
    - Unless the target thread has already terminated
  - The retrieved exit status is stored in value_ptr, if it is not NULL
  - The target thread is then placed in a "detached" state
    - The storage for that thread is released

# Thread Termination Status (Cont'd)

- The storage of a thread can be released immediately right on its termination
- The pthread_detach function
  - Set the state of a thread to be "detached"
  - int pthread_detach(pthread_t thread);
  - Returns: 0 if OK, error number on failure
- A detached thread can not be joined
  - A call to pthread_join for a detached thread will return EINVL

# Thread Termination, an Example

```
void * tfn1(void *a) { printf("thread 1 returning\n"); return((void *)1); }
void * tfn2(void *a) { printf("thread 2 exiting\n"); pthread_exit((void *)2); }
int main(void) {
     int  err;
     pthread_t tid1, tid2;
     void *tret;
     err = pthread_create(&tid1, NULL, tfn1, NULL);
     if (err != 0)err_quit("can't create thread 1: %s\n", strerror(err));
     err = pthread_create(&tid2, NULL, tfn2, NULL);
     if (err != 0)err_quit("can't create thread 2: %s\n", strerror(err));
     err = pthread_join(tid1, &tret);
     if (err != 0)err_quit("can't join with thread 1: %s\n", strerror(err));
     printf("thread 1 exit code %d\n", (int)tret);
     err = pthread_join(tid2, &tret);
     if (err != 0)err_quit("can't join with thread 2: %s\n", strerror(err));
     printf("thread 2 exit code %d\n", (int)tret);
     exit(0);
}
```

```
$ ./fig11.3-exitstatus
thread 1 returning
thread 2 exiting
thread 1 exit code 1
thread 2 exit code 2
```

# void * and pthread Functions

- In pthread_create and pthread_exit function, we pass arguments using the "void *" type
  - The typeless pointer
- The pointer can be used to pass more than a single value
  - Values can be store in a data structure
  - The pointer of the data structure is then passed to pthread_create or pthread_exit
- However, the data structure should not be placed on the stack
  - When a thread is terminated, the memory of its stack is released
  - It might be reused by other threads

# Cancelling a Thread

- The pthread_cancel function
  - int pthread_cancel(pthread_t tid);
  - Returns: 0 if OK, error number on failure
- It just like the thread *tid* calls pthread_exit(PTHREAD_CANCELED)
- The thread tid can select to ignore or control how it is canceled
- pthread_cancel does not wait for the thread to terminate
  - It simply makes the request

# Cleanup Functions

- Recall the atexit function
  - Register functions that execute when a process terminates
- Similar works can be done for threads
  - void pthread_cleanup_push(void (*rtn)(void *), void *arg);
  - void pthread_cleanup_pop(int execute);
- The registered routines is executed when …
  - Making a call to pthread_exit
  - Responding to a cancellation request
  - Making a call to pthread_cleanup_pop with a nonzero execute argument
    - If the argument is zero, it just remove the routine on stack top

# Comparison of Process and Thread Primitives

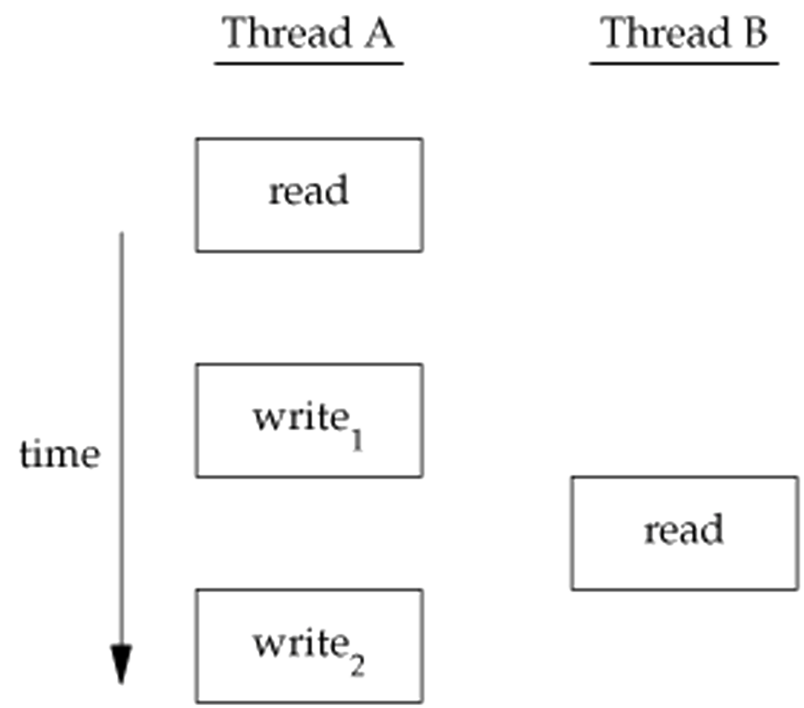
| Process Primitive | Thread Primitive | Description |
|---|---|---|
| fork | pthread_create | create a new flow of control |
| exit | pthread_exit | exit from an existing flow of control |
| waitpid | pthread_join | get exit status from flow of control |
| atexit | pthread_cleanup_push | register function to be called at exit from flow of control |
| getpid | pthread_self | get ID for flow of control |
| abort | pthread_cancel | request abnormal termination of flow of control |

# Thread Synchronization

- Threads of a process share the same memory
- Each thread must see a consistent view of data
- The data is always consistent if …
  - Each thread uses variables that other threads do not read or modify
  - Variables are read-only
- However, if a thread modifies a shared data
  - We need to synchronize the threads to ensure that they do not use an invalid value

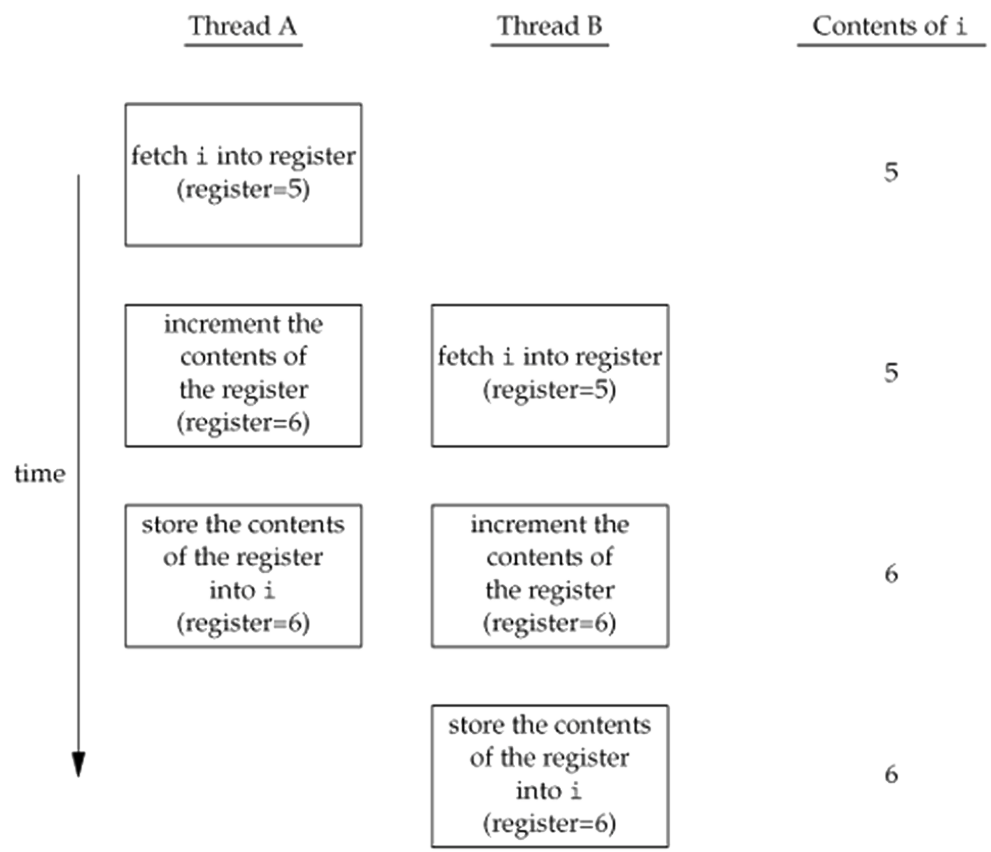
# Unsafe Access of Shared Variables, Example #1

- Two threads, one for updating and one for reading
  - Suppose a write operation needs two cycles
  - The read operation occurs during the write operations



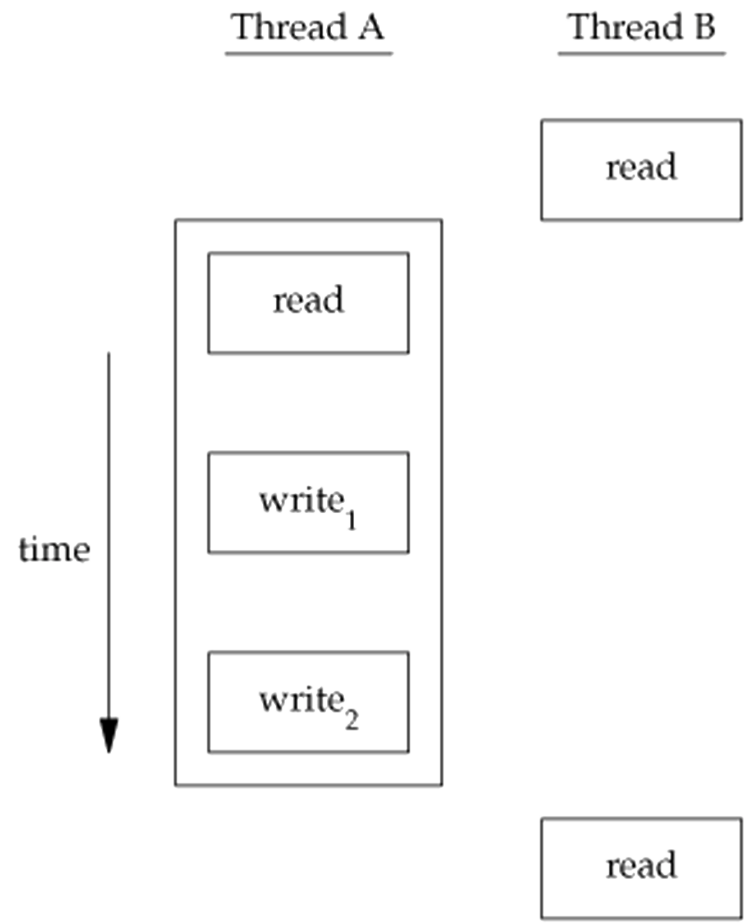
# Unsafe Access of Shared Variables, Example #2

- Two threads, both increasing a variable
  - Read the memory location into a register
  - Increment the value in the register
  - Write the new value back to the memory location



# Synchronized Memory Access

- To solve the previous problem, we have to use a lock that allows only one thread to access the variable at a time
- If thread B wants to read the variable, it acquires a lock
- If thread A updates the variable, it acquires the same lock
  - Thread B will be unable to read the variable until thread A releases the lock



# Mutexes

- Mutual exclusives
- A mutex is basically a lock
  - We set (lock) it before accessing a shared resource
  - It is released (unlocked) when we're done
- When a mutex is set …
  - Any other thread that tries to set it will be blocked until the lock holder releases it
  - If more than one thread is blocked when a mutex is unlocked
    - All threads blocked on the lock will be made runnable
    - The first one to run will be able to set the lock
    - The others will see that the mutex is still locked and go back to wait
  - Only one thread will proceed at a time

# pthread Mutexes

- Data type: pthread_mutex_t
- Initialize and destroy
  - int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
  - int pthread_mutex_destroy(pthread_mutex_t *mutex);
  - Returns: 0 if OK, error number on failure
- Alternatively
  - pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

# pthread Mutexes (Cont'd)

- Lock and unlock
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - int pthread_mutex_trylock(pthread_mutex_t *mutex);
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
  - Returns: 0 if OK, error number on failure

# A Mutex Example – Protect a Data Structure

```
struct foo {
    int     f_count;
    pthread_mutex_t f_lock;
    /* ... more stuff here ... */
};

struct foo * foo_alloc(void) { /* allocate the object */
    struct foo *fp;
    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... continue initialization ... */
    }
    return(fp);
}
```

# A Mutex Example – Protect a Data Structure (Cont'd)

```
void foo_hold(struct foo *fp) { /* add a reference to the object */
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

void foo_rele(struct foo *fp) { /* release a reference to the object */
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

# Deadlock Avoidance

- How deadlock happens?
  - Case #1: A thread lock the same mutex twice
  - Case #2: Two threads (T1/T2) and two mutexes (MA/MB)
    - T1 locks MA and then locks MB
    - T2 locks MB and then locks MA
    - T1 and T2 may block each other
- Avoidance
  - Case #1 is easier to avoid
  - Case #2: Mutexes has to be locked in the same order
    - Every thread locks MA first and then locks MB
    - However, it is sometimes difficult to apply an ordered lock
  - The pthread_mutex_trylock function
    - Make sure that we can lock all required mutexes at one time

# Reader-Writer Lock

- Similar to mutexes, but allow higher degree of parallelism
- With a mutex, it can be only
  - Locked, or
  - Unlocked
- With a reader-write lock, it can be
  - Locked in read mode
  - Locked in write mode, or
  - Unlocked
- Reader-Write Lock
  - Multiple reader locks can be acquired simultaneously
  - Only one can lock in write mode
  - If a reader/writer already locks, the coming writer/reader must wait until it unlocks

# pthread Reader-Writer Lock

- Data type: pthread_rwlock_t
- Initialize and destroy
  - int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);
  - int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
  - Returns: 0 if OK, error number on failure
- Lock and unlock
  - int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
  - int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
  - int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
  - int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
  - int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
  - Returns: 0 if OK, error number on failure

# Condition Variable

- Condition variable is another synchronization mechanism available to threads
- It has to be used with mutexes
  - The condition itself is protected by a mutex
  - A thread must first lock the mutex to change the condition state
- Condition variable allows a thread to wait in a race-free way for arbitrary conditions to occur

# pthread Condition Variables: Initialize and Destroy

- Data type: pthread_cond_t
- Initialize and destroy
  - int pthread_cond_init(pthread_cond_t *restrict cond, pthread_condattr_t *restrict attr);
  - int pthread_cond_destroy(pthread_cond_t *cond);
  - Returns: 0 if OK, error number on failure
- Alternatively
  - pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

# pthread Condition Variables: Wait for the Condition

- Synopsis
  - int pthread_cond_wait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex);
  - int pthread_cond_timedwait(pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
- The condition wait function unlocks the mutex first
- It then waits for the condition to occur
  - The running state of the current thread is set to sleeping

# pthread Condition Variables: Timed Wait

- The timespec data structure
- It is the absolute time that the wait gives up

```
struct timespec {
    __time_t tv_sec;    /* Seconds.  */
    long int tv_nsec;   /* Nanoseconds.  */
};
```

- An example of setting the absolute expire time

```
void maketimeout(struct timespec *tsp, long minutes) {
    struct timeval now; /* get the current time */
    gettimeofday(&now);
    tsp->tv_sec = now.tv_sec;
    tsp->tv_nsec = now.tv_usec * 1000;    /* usec to nsec */
    /* add the offset to get timeout value */
    tsp->tv_sec += minutes * 60;
}
```

# pthread Condition Variables: Notifications

- Notify threads that a condition has been satisfied
  - int pthread_cond_broadcast(pthread_cond_t *cond);
  - int pthread_cond_signal(pthread_cond_t *cond);
  - Returns: 0 if OK, error number on failure
- pthread_cond_broadcast
  - Wake up all waiting threads
- pthread_cond_signal
  - Wake up one waiting threads
  - POSIX.1 allows the implementation wakes up more than one threads
  - Waked up threads have to contend for the mutex

# pthread Condition Variables: An Example

```
struct msg {  struct msg *m_next; /* ... more stuff here ... */ };
struct msg *workq;
pthread_cond_t qready = PTHREAD_COND_INITIALIZER;
pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;
void process_msg(void) {
      struct msg *mp;
      for (;;) {
            pthread_mutex_lock(&qlock);
            while (workq == NULL) pthread_cond_wait(&qready, &qlock);
            mp = workq;
            workq = mp->m_next;
            pthread_mutex_unlock(&qlock);
            /* now process the message mp */
      }
}
void enqueue_msg(struct msg *mp) {
      pthread_mutex_lock(&qlock);
      mp->m_next = workq;
      workq = mp;
      pthread_mutex_unlock(&qlock);
      pthread_cond_signal(&qready);
}
```

# Example: An Implementation of a Worker Queue – Jobs

- Job header

```
class Job {
private:
  pthread_t tid;
  int ch;
public:
  Job(int ch = 0, pthread_t tid = 0);
  pthread_t getId();
  void setId(pthread_t tid);
  int getChar();
  void setChar(int ch);
};
```

- Job Implementation

```
Job::Job(int ch, pthread_t tid) {
  this->ch = ch;
  this->tid = tid;
}
pthread_t Job::getId() {
  return tid;
}
void Job::setId(pthread_t tid) {
  this->tid = tid;
}
int  Job::getChar() {
  return ch;
}
void Job::setChar(int ch) {
  this->ch = ch;
}
```

# Example: An Implementation of a Worker Queue – Global Definition

```
#define ASSIGNID    /* Assign thread id to jobs */
#define ORDERED     /* Ensure that jobs are processed in the order */
#define N_WORKERS     3

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
std::list<Job> jobqueue;

int do_the_job(long id, int ch) {
  if(ch == -1)  /* terminate the worker */
    return -1;
  printf("worker-%ld: %c\n", id, ch);
  return 0;
}
```

# Example: An Implementation of a Worker Queue – main Func (1/4)

```
int main(int argc, char *argv[]) {
    pthread_t workers[N_WORKERS];
    // check args
    if(argc < 2) {
        fprintf(stderr, "usage: %s input-string\n", argv[0]);
        return -1;
    }
    // create workers
    for(int i = 0; i < N_WORKERS; i++) {
        if(pthread_create(&workers[i], NULL,
                          worker_main, (void *) (long) i) != 0) {
            fprintf(stderr, "create worker[%d] failed\n", i);
            exit(-1);
        }
    }
```

# Example: An Implementation of a Worker Queue – main Func (2/4)

```
    // create jobs
    for(char *ptr = argv[1]; *ptr; ptr++) {
#ifdef  ASSIGNID
        Job j(*ptr, workers[(ptr - argv[1]) % N_WORKERS]);
#else
        Job j(*ptr);
#endif
        pthread_mutex_lock(&mutex);
        jobqueue.push_back(j);
        pthread_mutex_unlock(&mutex);
        pthread_cond_signal(&cond);
    }
```

# Example: An Implementation of a Worker Queue – main Func (3/4)

```
    // terminate workers
    for(int i = 0; i < N_WORKERS; i++) {
#ifdef  ASSIGNID
        Job j(-1, workers[i]);
#else
        Job j(-1);
#endif
        pthread_mutex_lock(&mutex);
        jobqueue.push_back(j);
        pthread_mutex_unlock(&mutex);
        pthread_cond_signal(&cond);
    }
```

# Example: An Implementation of a Worker Queue – main Func (4/4)

```
    // process all jobs
    size_t jobs;
    do {
        pthread_mutex_lock(&mutex);
        jobs = jobqueue.size();
        pthread_mutex_unlock(&mutex);
        pthread_cond_signal(&cond);
    } while(jobs > 0);
    // wait for all workers
    for(int i = 0; i < N_WORKERS; i++) {
        void *ret;
        pthread_join(workers[i], &ret);
    }
    //
    return 0;
}   /* end of main() */
```

# Example: An Implementation of a Worker Queue – The Worker

```
void* worker_main(void *arg) {
  long id = (long) arg;
  printf("# worker-%ld created\n", id);
  while(1) {
    Job j;
    pthread_mutex_lock(&mutex);
    pthread_cond_wait(&cond, &mutex);
    // has at least one job
    j = jobqueue.front();
    if(j.getId() == 0
    || pthread_equal(pthread_self(), j.getId())) {
      jobqueue.pop_front();
#ifdef  ORDERED
      // follow the order: unlock after job is done
      if(do_the_job(id, j.getChar()) < 0) {
        pthread_mutex_unlock(&mutex);
        break;
      }
#endif
    } else {
      } else {
      pthread_mutex_unlock(&mutex);
      continue;
    }
    /* unlock before doing the job */
    pthread_mutex_unlock(&mutex);
#ifndef ORDERED
    // could be out of order
    if(do_the_job(id, j.getChar()) < 0)
      break;
#endif
  }
  printf("# worker-%ld terminated\n",
         id);
  return NULL;
}
```

# Spin Lock

- Mutex  blocks a process by sleeping
- Spin lock blocks by busy-waiting, or spinning, until the local is acquired
  - More responsive: never being rescheduled
  - Consumes more CPU cycles due to spinning
  - Useful for non-preemptive schedulers
- Spin lock is less crucial for preemptive schedulers
  - Modern mutex may be implemented with a combination of spinning and sleeping

# Barriers

- Barriers are used to coordinate multiple threads working in parallel
- Allow each thread to wait until all cooperating threads have reached the same point
- Create and destroy a barrier

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
                         const pthread_barrierattr_t *restrict attr,
                         unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- Wait for a barrier

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

# Barrier Example (1/3)

```
#define HAS_BARRIER
#define N 5

static pthread_barrier_t barrier;

void *worker(void *arg) {
        long i, id = (long) arg;
        for(i = 0; i < id+1; i++) {
                printf("%ld", id+1);
        }
        printf("[%ld/done]\n", id+1);
#ifdef HAS_BARRIER
        pthread_barrier_wait(&barrier);
#endif
        return NULL;
}
```

# Barrier Example (2/3)

```
int main() {
        long i;
        pthread_t tid;
#ifdef HAS_BARRIER
        pthread_barrier_init(&barrier, NULL, N+1);
#endif
        for(i = 0; i < N; i++) {
                if(pthread_create(&tid, NULL, worker, (void*) i) != 0) {
                        fprintf(stderr, "pthread_create failed.\n");
                        return -1;
                }
        }
#ifdef HAS_BARRIER
        pthread_barrier_wait(&barrier);
        pthread_barrier_destroy(&barrier);
#endif
        printf("all done.\n");
        return 0;
}
```

# Barrier Example (3/3)

- Without HAS_BARRIER

```
$ ./barrier
3314444[4/done]
all done.

$ ./barrier
all done.

$ ./barrier
333[3/done]
1[1/done]
22[2/done]
4444[4/done]
all done.
```

- With HAS_BARRIER

```
$ ./barrier
22[2/done]
13[1/done]
4444[4/done]
55555[5/done]
33[3/done]
all done.

$ ./barrier
43233[3/done]
2[2/done]
555551[1/done]
[5/done]
444[4/done]
all done.
```

# Assignment #8 (5%)

- Exercise questions 11.1 – 11.5, each question is worth 1%
- Due date: Dec 12$^{th}$, 2016