TEXAS TECH UNIVERSITY

CS4379: Parallel and Concurrent Programming
CS5379: Parallel Processing

Lecture 15

Guest Lecture by Wei Zhang

X-Spirit.Zhang@ttu.edu

Dr. Yong Chen

Associate Professor

Computer Science Department

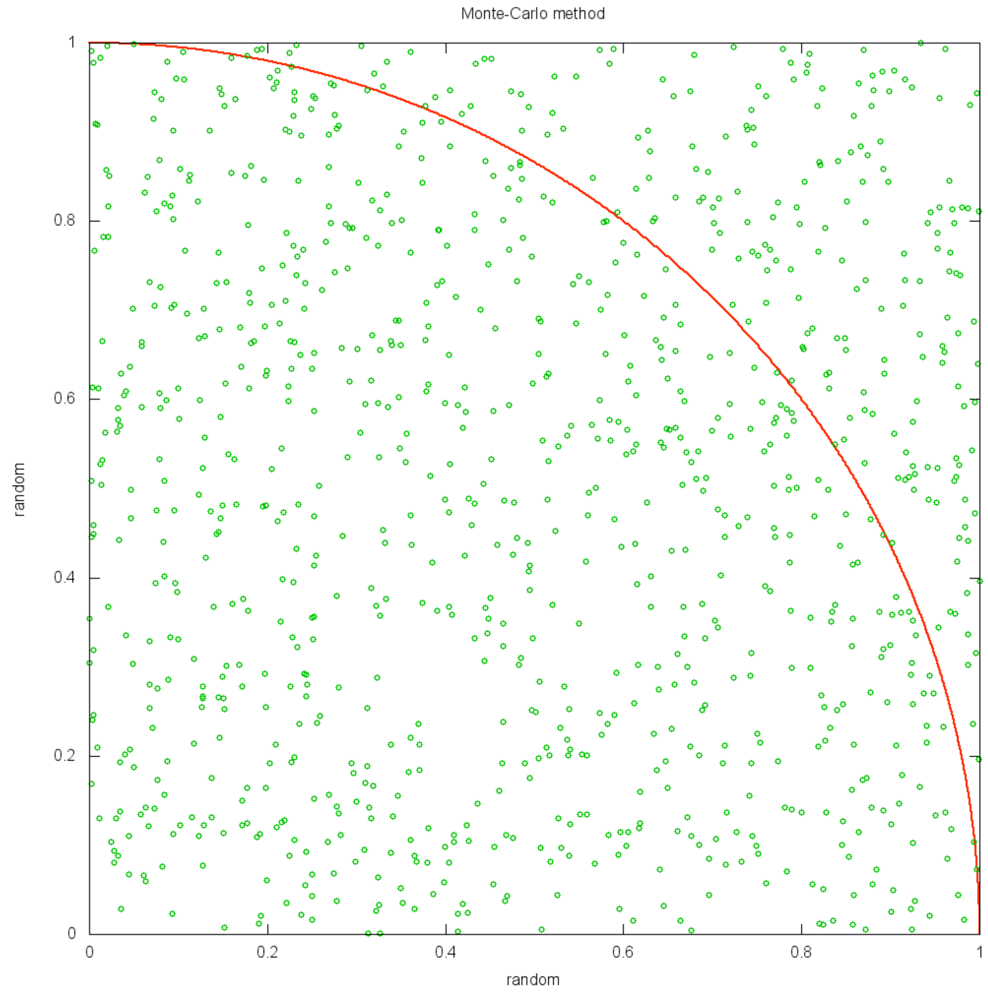
Texas Tech University

Course Info

- **Lecture Time**: TR, 12:30-1:50
- **Lecture Location**: ECE 217
- **Sessions**: CS4379-001, CS4379-002, CS5379-001, CS5379-D01
- **Instructor:** Yong Chen, Ph.D., Associate Professor
- **Email:** yong.chen@ttu.edu
- **Phone:** 806-834-0284
- **Office**: Engineering Center 315
- **Office Hours**: 2-4 p.m. on Wed., or by appointment
- **TA:** Mr. Ghazanfar Ali, Ghazanfar.Ali@ttu.edu
- **TA Office hours:** Tue. and Fri., 2-3 p.m., or by appointment
- **TA Office:** EC 201 A
- **More info:**
  - http://www.myweb.ttu.edu/yonchen
  - http://discl.cs.ttu.edu; http://cac.ttu.edu/; http://nsfcac.org

Pthreads Example Code: Compute pi

- Generating random points in a unit length square
- Counting the number of points fall within circle
- The faction of random points: pi/4



Pthreads Example Code: Compute pi

- Assigns a fixed number of points to each thread
- Each thread generates these random points
- Collecting the number of points land in the circle
- After all threads finish, all threads counts are combined to compute pi
- Calculate fraction over all threads and multiply by 4

Pthreads Example Code: Compute pi

```
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main() {
    ...
    pthread_t p_threads[MAX_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];
    }
    ...
}
```

Pthreads Example Code: Compute pi

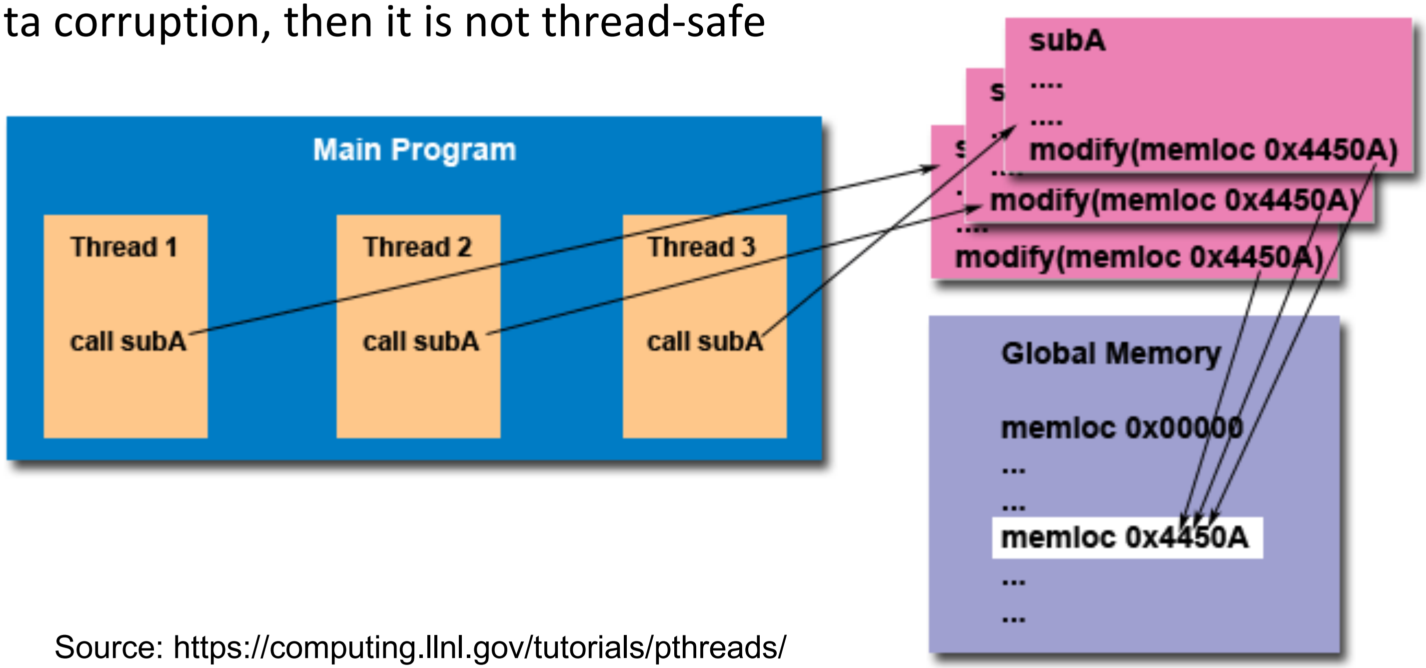
```
void *compute_pi (void *s) {
   int seed, i, *hit_pointer;
   double rand_no_x, rand_no_y;
   int local_hits;
   hit_pointer = (int *) s;
   seed = *hit_pointer;
   local_hits = 0;
   for (i = 0; i < sample_points_per_thread; i++) {
      rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
      rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
      if (rand_no_x * rand_no_x +
         rand_no_y * rand_no_y <= 1.0)
         local_hits ++;
      seed *= i;
   }
   *hit_pointer = local_hits;
   pthread_exit(0);
}
```

Programming Notes

- Note the use of the function *rand_r* (instead of superior random number generators such as drand48)
- Reentrant functions: thread safety (thread-safe code)
- Can be safely called when another instance has been suspended in the middle of its invocation
- Thread can be preempted
- Non-reentrant function does not work as desired if another thread executes the same function
- Reading: https://en.wikipedia.org/wiki/Reentrancy_(computing)

Programming Notes

- Thread-safety:
  - E.g. an app creates several threads, each making a call to the same library routine
  - This library routine accesses/modifies a global structure or location in memory
  - As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time
  - If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe



Source: https://computing.llnl.gov/tutorials/pthreads/

Outline

- Questions?

- Overview of programming models and thread basics
- The POSIX Threads
- Synchronization Primitives in Pthreads
  - Mutual Exclusion for Shared Variables
  - Producer-Consumer problem
  - Condition Variables for Synchronization
  - Controlling Thread and Synchronization Attributes
  - Composite Synchronization Constructs
- Compiling and running Pthreads program

Synchronization Primitives in Pthreads

- When multiple threads attempt to manipulate a shared data item, the results can often be incoherent if proper care is not taken to synchronize them
- Consider:

```
/* each thread tries to update a shared variable
   best_cost as follows */
if (my_cost < best_cost)
   best_cost = my_cost;
```

- Assume that there are two threads, the initial value of best_cost is 100, and the values of my_cost are 50 and 75 at threads t1 and t2.
- Depending on the schedule of the threads, the value of best_cost could be 50 or 75!

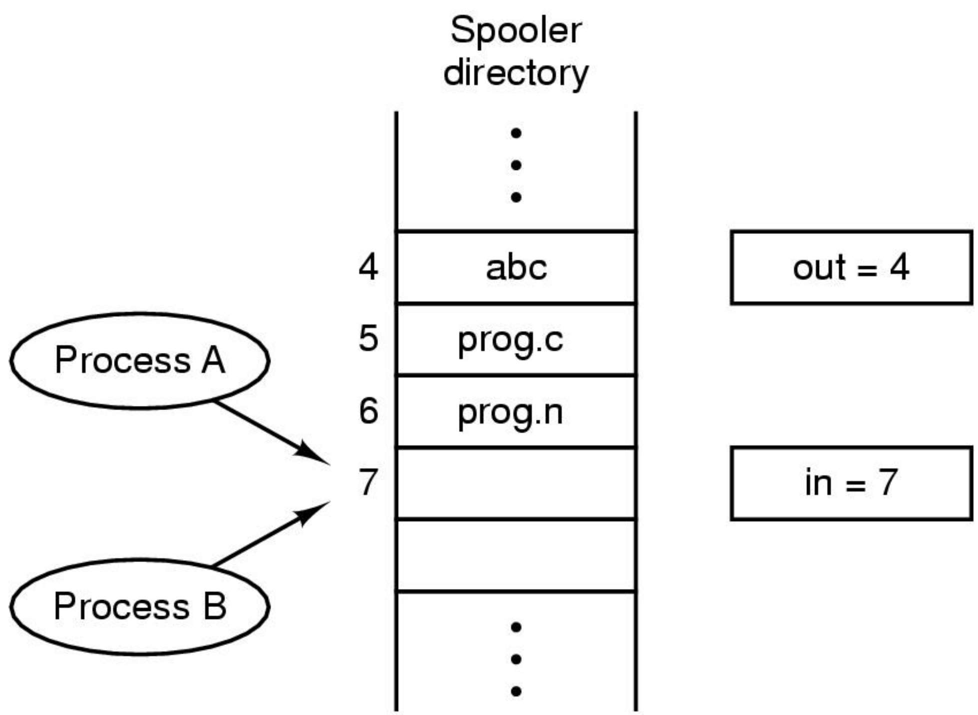
Synchronization Primitives in Pthreads

- Two problems
  - Parallel/concurrent nature (non-deterministic)
  - The possible value of 75 is inconsistent - it does not correspond to any serialization of the threads

- Race condition
  - The result of the computation depends on the race between competing threads

Race Condition

- Two or more processes read/write some shared data, if the sequence of execution of the two processes makes a difference in the result of their concurrent execution, such situation is called race conditions



Without proper coordination, the result can be incorrect (and the result varies depending on which process runs when)!

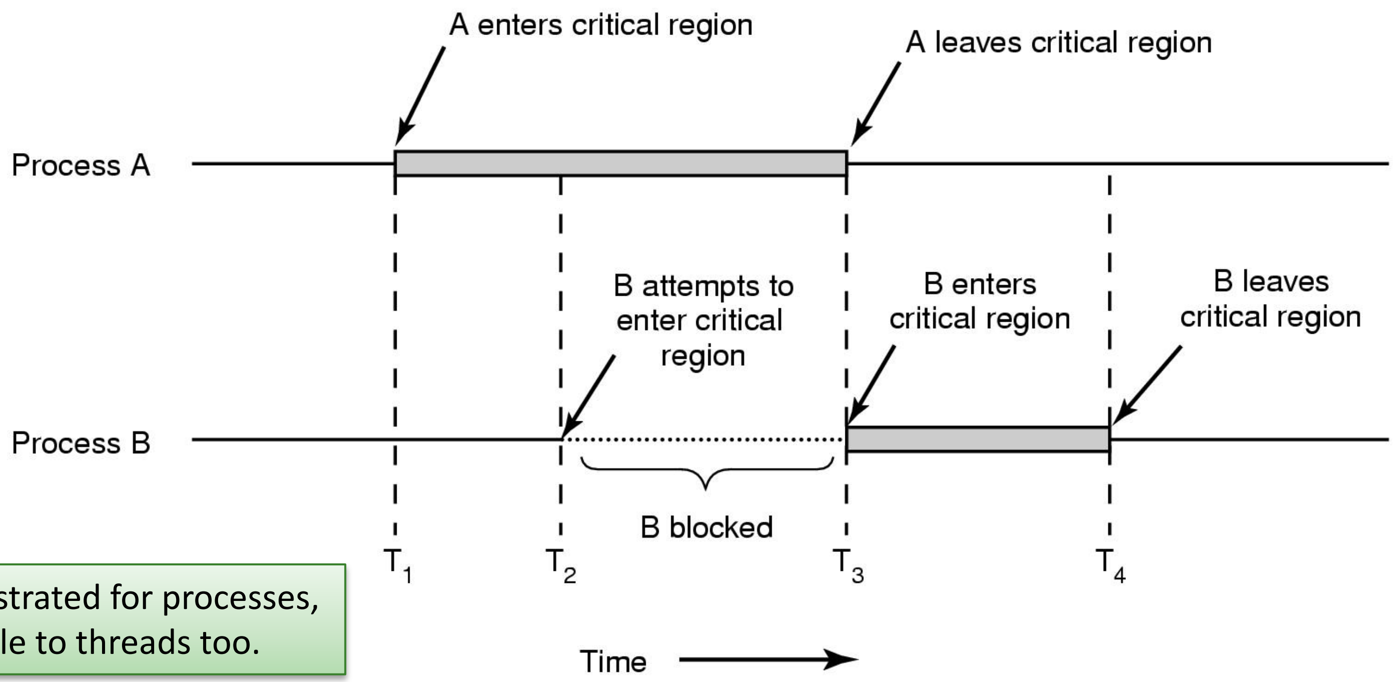
Two processes want to access the spooler directory of a printer at the same time.

Thread-Safety(Cont.)

- Three levels of thread-safety: https://en.wikipedia.org/wiki/Thread_safety
  - **Thread safe**: No race condition
  - **Conditionally safe**: shared resources are protected from race condition
  - **Not thread safe**: simultaneous accesses not supported
- Avoid Race Conditions (How to implement in Pthreads API):
  - Mutual exclusion
  - Thread-local storage: https://en.wikipedia.org/wiki/Thread-local_storage
  - Immutable Storage: https://en.wikipedia.org/wiki/Immutable_object
  - Atomic operation https://en.wikipedia.org/wiki/Linearizability
  - Re-entrancy https://en.wikipedia.org/wiki/Reentrancy_(computing)

Solution to Race Conditions

- Mutual exclusion is needed to avoid race conditions and achieve correct results
  - Applies to any shared resources, e.g. memory, files, database, etc.



Example illustrated for processes, but applicable to threads too.

Mutual exclusion using critical regions.

Mutual Exclusion

- The code in the example corresponds to a critical section
  - A segment that must be executed by only one thread at any time
  - Test-and-update needs to be atomic
- Critical sections in Pthreads are implemented using mutex locks (mutual exclusion locks)
- Mutex-locks have two states: locked and unlocked
  - At any point of time, only one thread can lock a mutex lock
  - A lock is an atomic operation.
- A thread entering a critical section first tries to get a lock. It goes ahead when the lock is granted.

Mutual Exclusion (cont.)

- The Pthreads API provides the following functions for handling mutex-locks:

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

Mutual Exclusion (cont.)

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered
```

Not busy waiting!

```
mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Possible implementation of mutex lock and mutex unlock.

Mutual Exclusion

- We can now write our previously incorrect code segment as:

```
pthread_mutex_t minimum_value_lock;
...
main() {
   ....
   pthread_mutex_init(&minimum_value_lock, NULL);
   ....
}
void *find_min(void *list_ptr) {
   ....
   pthread_mutex_lock(&minimum_value_lock);
   if (my_min < minimum_value)
   minimum_value = my_min;
   /* and unlock the mutex */
   pthread_mutex_unlock(&minimum_value_lock);
}
```

Critical section

Outline

- Questions?

- Overview of programming models and thread basics
- The POSIX Threads
- Synchronization Primitives in Pthreads
  - Mutual Exclusion for Shared Variables
  - Producer-Consumer problem
  - Condition Variables for Synchronization
  - Controlling Thread and Synchronization Attributes
  - Composite Synchronization Constructs
- Compiling and running Pthreads program

Producer-Consumer Problem

- The producer-consumer relationship models a class of problems, with the following constraints
- The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
- The consumer threads must not pick up tasks until there is something present in the shared data structure.
- Producer-consumer relations are ubiquitous
- Needs synchronization between producer and consumer

The Producer-Consumer Problem

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count − 1;                    /* decrement count of items in buffer */
          if (count == N − 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Code NOT protected by mutual exclusion! Can have incorrect result!

e.g. a context switch can occur here right after testing the count has a value of 0

The producer-consumer problem with a fatal race condition.

Producer-Consumer Using 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);
        }
    }
}
```

Assumes task queue holds only one task

Signal the task availability

Release the lock

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);
   }
}
```

Signal the queue availability

Release the lock

Types of Mutexes

- Pthreads supports three types of mutexes - normal, recursive, and error-check.
- A normal mutex deadlocks if a thread that already has a lock tries a second lock on it.
- A recursive mutex allows a single thread to lock a mutex as many times as it wants. It simply increments a count on the number of locks. A lock is relinquished by a thread when the count becomes zero.
- An error check mutex reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case).
- The type of the mutex can be set in the attributes object before it is passed at time of initialization.

Overheads of Locking

- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can lead to significant performance degradation.
- It is often possible to reduce the idling overhead associated with locks using an alternate nonblocking function, `pthread_mutex_trylock`.

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

- Returns EBUSY when failed, allow the thread to do other work and poll the mutex for a lock later
- Example in the textbook

Avoiding deadlocks

- Deadlocks can occur when using more than one mutex and to lock multiple mutex from multiple threads

Thread 0	Thread 1
`pthread_mutex_lock(&mutex_a);`	`pthread_mutex_lock(&mutex_b);`
`pthread_mutex_lock(&mutex_b);`	`pthread_mutex_lock(&mutex_a);`

- Avoiding deadlock with hierarchical locking
  - Order locks
  - All threads follow the same order to lock multiple locks

- Avoiding deadlock with nonblocking locking
  - After locking the first mutex, use `pthread_mutex_trylock` to lock additional mutexes
  - If an attempt fails, release all mutexes and start again

Conditions for Deadlocks

- Mutual exclusion condition: each lock (or resource in general) is either currently assigned to exactly one process/thread or is available
- Hold and wait condition: processes/threads currently holding resources that were granted earlier can request new resources
- No preemption condition: resources previously granted cannot be forcibly taken away from a process/thread (must be voluntarily released)
- Circular wait condition: there is a circular list of two or more processes/threads, each of which is waiting for a resource held by another process/thread

All four conditions must be present for a deadlock to occur. If one of them is absent, no deadlock is possible. Hierarchical locking breaks the “Circular wait condition”.

Outline

- Questions?

- Overview of programming models and thread basics
- The POSIX Threads
- Synchronization Primitives in Pthreads
  - Mutual Exclusion for Shared Variables
  - Producer-Consumer problem
  - Condition Variables for Synchronization
  - Controlling Thread and Synchronization Attributes
  - Composite Synchronization Constructs
- Compiling and running Pthreads program

Condition Variables for Synchronization

- Idling overhead for `pthread_mutex_lock()`
- Polling overhead for `pthread_mutex_trylock()`
- A condition variable allows a thread to suspend its execution until specified data reaches a predefined state
  - Interrupt driven mechanism instead of a polled mechanism

Condition Variables for Synchronization

- A condition variable is associated with a predicate. When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition.

- A condition variable always has a mutex associated with it. A thread locks this mutex and tests the predicate defined on the shared variable.
- A single condition variable may be associated with more than one predicate (difficult to debug, discouraged)

Condition Variables for Synchronization

- Pthreads provides the following functions for condition variables:

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

Producer-Consumer Using Condition Variables

```
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 */
}
```

Producer-Consumer 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);
   }
}
```

Producer-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);
  }
}
```

Readings

- Reference book ITPC – Chapter 7
- POSIX Threads Programming, by Blaise Barney, Lawrence Livermore National Laboratory: https://computing.llnl.gov/tutorials/pthreads/

Questions?

Questions/Suggestions/Comments are always welcome!

Write me: yong.chen@ttu.edu
Call me: 806-834-0284
See me: ENGCTR 315

If you write me an email for this class, please start the email subject with [CS4379] or [CS5379].