POSIX Thread Programming

- Standard Thread Library for POSIX-compliant systems
- Supports thread creation and management
- Synchronization using
  - mutex variables
  - condition variables
- At the time of creation, different attributes can be assigned to
  - threads
  - mutex/condition variables

Using Posix Thread Library

- To use this library, #include <pthread.h> in your program.

- In some systems, you may need to link with the pthread library explicitly:

  gcc hello.c -o hello –pthread

Data Types in POSIX

- special data type for threads

- mutex variables for mutual exclusion
  - mutex variables are like *binary semaphores*
  - a mutex variable can be in either *locked* or *unlocked* state

- condition variables using which a thread can sleep until some other thread signals the condition

- various kind of attribute types used when initializing:
  - threads
  - mutex variables
  - condition variables

Functions and Data Types

- All POSIX thread functions have the form:

  pthread[_object] _operation

- Most of the POSIX thread library functions return 0 in case of success and some non-zero error-number in case of a failure.

Data Types (Cont.)

- Following are the important data types in POSIX library

• *pthread_t*	Thread ID for a thread object
• *pthread_mutex_t*	Mutual exclusion lock variable
• *pthread_cond_t*	Condition variable
• *pthread_attr_t*	Thread attribute variable
• *pthread_mutexattr_t*	Mutex lock variable attribute
• *pthread_condattr_t*	Condition variable attribute

Setting Thread Attributes

- **Define and initialize attribute object:**

  pthread_attr_t attr;

  pthread_attr_init (&attr);

- **For example, a thread may be created with specification of certain attributes such stack address and stack size**

- **Programmers new to multi-threading can simply use “default attributes” when creating the thread, mutex locks and condition variables.**
  - **In that case, simply indicate NULL as the pointer to the attribute variable.**

Thread Creation

- *pthread_create* function is used to create a new thread.

An example:

pthread_t producerID;

pthread_create (&producerID, NULL, producer, NULL);

- First argument is the ID of the new thread
- Second argument is a pointer to pthread_attr_t
- Third argument is thread (function) name
- Fourth argument is a pointer to the argument of the thread

Thread Creation

```
pthread_create (
pthread_t * threadID, // thread
const pthread_attr_t *attr,
// attribute object
void * ( * FunctionName ) ( void * ),
// Function pointer with one pointer argument
void * arg )
// Pointer to the argument of the thread
```

Thread Exit and Join

- If any thread executes the system call *exit( ),* the process terminates.
- If the main thread completes its execution, it implicitly calls *exit( ),* and this again terminates the process.
- A thread (the main, or another thread) can exit by calling *pthread_exit( ),* this does not terminate the process.

- A thread can wait for the completion of another thread by using

  *pthread_join (pthread_t thread, void **status)*

Mutex Variables

- Used for mutual exclusion locks.
- A mutex variable can be either *locked* or *unlocked*

  pthread_mutex_t lock; // lock is a mutex variable

- Lock operation

  pthread_mutex_lock(&lock);

- Unlock operation

  pthread_mutex_unlock(&lock);

- Initialization of a mutex variable by default attributes

  pthread_mutex_init(&lock, NULL);

Mutex Variables

```
pthread_mutex_t  mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock ( &mutex );
// Blocks to acquire the lock
......
   critical section
.....
pthread_mutex_unlock ( &mutex );
```

- There is also *pthread_mutex_trylock*: If the mutex is currently locked, returns immediately EBUSY. Otherwise, the calling thread becomes owner until it unlocks.

Condition Variables

- In a critical section, a thread can suspend itself on a *condition variable* if the state of the computation is not right for it to proceed.
  - It will suspend itself by *waiting* on a condition variable.
  - It will, however, release the critical section (mutex) lock at the same time.
  - When that condition variable is *signaled,* it will no longer be blocked because of the "condition": but it will still need to attempt to reacquire that critical section lock and only then will be able to proceed.

- With Posix threads, a condition variable can be associated with only one mutex variable!

Condition Variables

- *pthread_cond_t SpaceAvailable;*
- *pthread_cond_init (&SpaceAvailable, NULL);*

- *pthread_cond_wait*
- *pthread_cond_signal*

  unblock one waiting thread on that condition variable (that thread should still get the "lock" before proceeding)

- *pthread_cond_broadcast*

  unblock all waiting threads on that condition variable

  - **Now all of them will compete to get the "lock"**
  - **Only one at a time can succeed; others must wait for a later opportunity**

Condition Variables

Example:

```
pthread_mutex_lock ( &mutex );
. . . . . .
pthread_cond_wait ( &SpaceAvailable, &mutex);
// now proceed again
. . .
pthread_mutex_unlock( &mutex );
```

- **Some other thread will execute:**

```
pthread_cond_signal ( &SpaceAvailable );
```

- **The signaling thread has priority over any thread that may be awakened**
  - **– "Signal-and-continue" semantics**

Producer-Consumer Problem

- Producer will produce a sequence of integers, and deposit each integer in a bounded buffer (implemented as an array).
- All integers are positive, 1…n
- Producer can deposit -1 when finished, and then terminate.
- Buffer is of finite size: 5 in this example.
- Consumer will remove integers, one at a time, and print them.
- It will terminate when it receives -1.

Definitions and Globals

```
#include<pthread.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
const int N = 5;
int Buffer[5];
int in = 0;
int out = 0;
int count = 0;
pthread_mutex_t lock;
pthread_cond_t SpaceAvailable, ItemAvailable;
```

Producer Thread

```
void * producer (void *arg)
{ int i;
    for ( i = 0; i< 1000; i++) {
      pthread_mutex_lock ( &lock); /* Enter critical section */
      while ( count == N ) /* Make sure that buffer is NOT full */
        pthread_cond_wait ( &SpaceAvailable, &lock) ;
/* Sleep using a condition variable */
/* now  count must be less than N */
   Buffer[in] = i; /* Put item in the buffer using "in"  */
   in = (in + 1) % N;
   count++; /* Increment the count of items in the buffer */
```

Producer Thread (Cont.)

```
pthread_mutex_unlock ( &lock);
pthread_cond_signal( &ItemAvailable );
/* Wakeup consumer, if waiting */
}  /* End of For loop */
/* Put -1 in the buffer to indicate completion to the consumer */
pthread_mutex_lock ( &lock);
while ( count == N )
  pthread_cond_wait( &SpaceAvailable, &lock) ;
Buffer[in] = -1; in = (in + 1) % N; count++;
pthread_mutex_unlock ( &lock );
pthread_cond_signal( &ItemAvailable );
/* Wakeup consumer, if waiting */
} // End of producer
```

Consumer Thread

```
void * consumer (void *arg)
{ int i = 0;
do {
pthread_mutex_lock ( &lock); /* Enter critical section */
while ( count == 0 ) /* Make sure that buffer is NOT empty */
    pthread_cond_wait( &ItemAvailable, &lock) ;
/* Sleep using a condition variable */
/* count must be > 0 */
i = Buffer[out] ; /* Remove item from the buffer using "out"  */
out = (out + 1) % N;
count--; /* Decrement the count of items in the buffer */
```

Consumer Thread (Cont.)

```
printf( "Removed %d \n", i);
pthread_mutex_unlock ( &lock); /* exit critical section */
pthread_cond_signal( &SpaceAvailable);
/* Wakeup producer, if waiting */
} while ( i != -1 );  /* End of Do loop */
} // End of consumer
```

Main program

```
main( )
{
pthread_t    prod, cons; /* thread variables */
int   n;
pthread_mutex_init( &lock, NULL);
pthread_cond_init (&SpaceAvailable, NULL);
pthread_cond_init (&ItemAvailable, NULL);
/* Create producer thread */
if ( n = pthread_create(&prod, NULL, producer ,NULL)) {
fprintf(stderr,"pthread_create :%s\n",strerror(n));
exit(1);
 }
```

Main Program (Cont.)

```
/* Create consumer thread */
if ( n = pthread_create(&cons, NULL, consumer, NULL) )
   {
fprintf(stderr,"pthread_create :%s\n",strerror(n));
exit(1);
  }
/* Wait for the consumer thread to finish. */
if ( n = pthread_join(cons, NULL) ) {
fprintf(stderr,"pthread_join:%s\n",strerror(n));
exit(1);
     }
printf("Finished execution \n" );
} // End of main
```

Working on Your Program

- First solve the problem with pen and paper before starting to code
- Writing multithreaded programs is tricky, be careful with the use of pointers and thread functions.
- Refer to multithreaded programming guides and references when in doubt
  Resources link at
  http://cs.gmu.edu/~aydin/cs571/resources.html