COSC 4397
Parallel Computation

Introduction to POSIX Threads

Edgar Gabriel

with some examples by Kshitij Mehta

Spring 2010

References

Some of the slides in this lecture is based on the following references:

- [http://www.cobweb.ecn.purdue.edu/~eigenman/ECE563/Handouts/pthreads.ppt](http://www.cobweb.ecn.purdue.edu/~eigenman/ECE563/Handouts/pthreads.ppt)
- Rolf Rabenseifner, Georg Hager, Gabriele Jost, Rainer Keller, ‘Hybrid MPI and OpenMP Parallel Programming’, Tutorial S10 at Supercomputing 2007, Reno, Nevada, USA.
- [http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html](http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html)
POSIX Threads Overview

- Shared memory programming model
- POSIX threads (pthreads) programming model:
  - creation of threads
  - managing thread execution
  - managing the shared resources of the process
- IEEE's POSIX Threads Model:
  - programming models for threads in a UNIX platform
  - pthreads are included in the international standards ISO/IEC9945-1

Processes vs. Threads

- Process:
  - an address space with 1 or more threads executing within that address space, and the required system resources for those threads
  - a program that is running

- Thread:
  - a sequence of control within a process
  - shares the resources in that process
Processes vs. Threads (II)

- **Advantages:**
  - significantly lower overhead for creating a thread compared to creating a process
  - significantly lower overhead for switching between threads required the OS compared to switching between processes

- **Drawbacks:**
  - more difficult to debug than single threaded programs
  - for single processor machines, creating several threads in a program may not necessarily produce an increase in performance

Processes vs. Threads (III)

<table>
<thead>
<tr>
<th>Data per process</th>
<th>Data per thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Open files</td>
<td>Processor register</td>
</tr>
<tr>
<td>Child processes</td>
<td>Processor status</td>
</tr>
<tr>
<td>Signal handler</td>
<td>Signal mask</td>
</tr>
<tr>
<td>Timer</td>
<td>Stack</td>
</tr>
<tr>
<td>Accounting</td>
<td></td>
</tr>
</tbody>
</table>
Execution model

- **Main thread:**
  - initial thread created when `main()` (in C) are invoked by the process loader
  - once in the `main()`, the application can create additional threads
  - if the main thread returns, the process terminates even if there are running threads in that process, unless special precautions are taken
  - (to explicitly avoid terminating the entire process, use `pthread_exit()`)
**pthread_create**

```c
pthread_create ( pthread_t * tidp,
               const pthread_attr_t *attr
               void* (*start_rtn) func,
               void *arg);
```

- **tidp**: upon completion of the function set to the thread id of the new thread
- **attr**: argument used to customize the new thread
- **func**: function pointer to be executed by the new thread. The prototype of the function has to be
  ```c
  void * func ( void *arg)
  ```
- **arg**: argument to be passed to the function `func`

**pthread_self and getpid**

```c
pid_t getpid(void);
```

- returns the process id of the calling process.
- all threads that have been spawned from a process have the same process id

```c
pthread_t pthread_self(void);
```

- returns the thread id of the calling thread
- not necessarily unique on a system, only within the calling process
Thread termination

- If any thread within a process calls exit() the entire process is terminated.
- To terminate a thread without effecting the other threads
  - return from the start return. Return value is the exit code of the thread
  - a thread can request that in the same process another thread is cancelled using

  ```c
  int pthread_cancel(pthread_t *threadid);
  ```

- typically done by the main thread
- return 0 if ok, error number on failure.

Thread termination (II)

- A thread can call

  ```c
  int pthread_exit(void *rval_ptr);
  ```

  - `rval_ptr`: single argument to pass to the counter part
  - can also be a pointer to a structure

  ```c
  int pthread_join(pthread_t threadid, void **rval_ptr);
  ```

  - typically called by the master thread
  - `rval_ptr`: will contain the value passed by the terminating thread to pthread_exit()
```c
#include <pthread.h>

int main ( int argc, char **argv )
{
    int threadid, ret;
    int *val;
    // main thread spawns another thread
    ret = pthread_create (&threadid, NULL, tfunc, NULL);
    if ( ret != 0 )
        printf("Error creating a new thread\n");
    // do something
    pthread_join( threadid, &val);
    return (0);
}

void *tfunc (void *arg){
    // do something useful
    pthread_exit ((void *) 1);
    return NULL;
}
```

**Thread synchronization**

- Reading and writing a shared variable between two threads
- Timing between two threads will differ in every iteration
- If you need a specific value for thread B of the variable you need to synchronize access to the shared variable
Thread synchronization

- Three methods discussed here
  - Mutex locks
  - Condition variables
  - Reader-Writer locks

Mutex locks

- Mutex: Mutual exclusion
  - a lock is used before accessing a shared resource and released after the access
  - mutex lock represented by a mutex variable
  - while mutex lock is set, other threads that try to access the lock will be denied
  - if more than one thread wait for the lock, all of them will be made runnable, but only one thread will get the lock
- All threads have to use mutex locks for accessing the variable, else no guarantee on correctness
Mutex locks (II)

```c
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

- **mutex**: mutex variable to be initialized/destroyed
  counter part
  - once initialized, a mutex variable can be used for an
  unlimited number of lock/unlock operations
- **attr**: attributes for the mutex

Mutex locks (III)

```c
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

- **pthread_mutex_lock**: acquire lock for the mutex.
  - If mutex is already blocked by another thread, wait until
    the mutex is unlocked
- **pthread_mutex_trylock**: acquire lock for the mutex.
  - If mutex is already blocked by another thread, do not
    wait but return EBUSY to indicated failure
Thread synchronization revisited

- Example: Force thread B to read value of shared variable after write₂

Simple Example (IIIa)

```c
#include <pthread.h>
int value=0;       // shared variable
pthread_mutex_t mymutex; // mutex variable

int main ( int argc, char **argv )
{
  int threadid, ret;

  // main thread spawns another thread
  ret = pthread_create (&threadid, NULL, tfunc, NULL);
  if ( ret != 0 ) printf("Error creating a thread\n");

  pthread_mutex_init (&mymutex,NULL); //Initialize mutex
  pthread_mutex_lock (&mymutex);      // Acquire mutex lock
  value = 1;                         // write 1
  value ++;                          // write 2
  pthread_mutex_unlock (&mymutex);   // Release lock

  pthread_join( threadid, &val);     // wait for other thread
  pthread_mutex_destroy (&mymutex); // destroy mutex
  return (0);
}
```
Simple Example (IIIb)

```c
void *tfunc (void *arg){
    int localvalue;

    pthread_mutex_lock (&mymutex);  // wait for lock
    localvalue = value;              // read shared variable
    pthread_mutex_unlock (&mymutex);

    pthread_exit ((void *) 1);
    return NULL;
}
```

Mutex locks (IV)

- A thread will deadlock itself if it tries to lock the same mutex twice
- If more than one mutex is used a deadlock can occur if one thread holds lock1 and waits for lock2 and the other thread holds lock2 and waits for lock1
  - Order for accessing mutexes has to be identical in all code paths
  - e.g. need to hold lock1 in order to be allows to hold lock2
Conditions Variables

- The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true.
- Must always be associated with a mutex

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

- Condition variable must be initialized before usage

Condition variables (II)

```c
int pthread_cond_wait (pthread_cond_t *cond,
                       pthread_mutex_t *mutex );
```

- Wait for a condition to be true
- Must be called with mutex locked by the calling thread
  - Will be released internally by the `pthread_cond_wait()` function
  - All threads calling `pthread_cond_wait()` for the same condition variable will be added to a queue
  - New thread acquiring the mutex lock can wake up the thread that called `pthread_cond_wait()`
Condition Variables (III)

```c
int pthread_cond_signal (pthreadCond_t *cond);
int pthread_cond_broadcast (pthreadCond_t *cond);
```

- Wake up the next (`pthread_cond_signal`) or all (`pthread_cond_broadcast`) threads who are waiting on the condition
- only signal threads after changing the state of the condition

Task Parallelism using Master-Worker framework

- Master thread creates a work queue and assigns work for each thread
- threads are sleeping until work is
#include <pthread.h>

#define NUM_THREADS 5;

pthread_mutex_t mutex[NUM_THREADS];  // mutex variable
pthread_cond_t cond[NUM_THREADS];  // condition variables

int main ( int argc, char **argv )
{
    int threadids[NUM_THREADS], ret;

    for ( int i=0; i<NUM_THREADS; i++ )
    {
        ret = pthread_create (&threadids[i], NULL,
                               wake_one_by_one, NULL);
        if ( ret != 0 )  printf("Error creating a thread\n");
        pthread_mutex_init (&mutex[i],NULL); //Initialize mutex
        pthread_cond_init (&cond[i],NULL); //Initialize condition
    }

    wake_one_by_one(); // Master thread executes the same code

    for ( int i=0; i<NUM_THREADS; i++ )
    {
        pthread_join (&threadids[i]);
        pthread_mutex_destroy (&mutex[i]);
        pthread_cond_destroy (&cond[i]);
    }
}

#include <pthread.h>

void wake_one_by_one (void *arg )
{
    int threadid= pthread_self();
    if ( threadid == MASTER ) {
        int i = 1;  // 0 being MASTER
        while(!DONE) {
            do_some_work();
            pthread_mutex_lock (&mutex[i]);
            work_assigned[i] = YES;  //assign work to thread i
            pthread_mutex_unlock(&mutex[i]);
            pthread_cond_signal (&cond_var[i]);
            i++;
        }

        //wakeup remaining threads
        while(i < num_threads) {
            pthread_mutex_lock (&mutex[i]);
            work_assigned[i] = NO_WORK_FOR_YOU;
            pthread_mutex_unlock(&mutex[i]);
            pthread_cond_signal (&cond_var[i]);
            i++;
        }
    }
}
else {
    // for the worker threads
    pthread_mutex_lock(&mutex[tid]);
    if (work_assigned[tid] == YES) {
        // unlock and proceed
        pthread_mutex_unlock(&mutex[tid]);
        proceed_to_work_assignment();
    }
    else if(work_assigned[tid] == NO_WORK_FOR_YOU)
        clean_and_quit();
    else {
        // work_assigned[tid] == NOTHING_YET
        pthread_cond_wait(&cond_var[tid], &mutex[tid]);
        // after getting wakeup signal
        pthread_mutex_unlock(&mutex[tid]);
        if(work_assigned[tid] == YES)
            proceed_to_work_assignment();
        else
            clean_and_quit();
    }
}

Reader-Writer Lock

- Similar to mutexes but with three states
  - locked in read mode
  - locked in write mode
  - unlocked
- Only one thread can hold a reader-writer lock in write mode
  - if a write-mode lock is hold, all lock attempts (even read) are denied
- Multiple threads can hold a reader-writer in read mode
- allows for higher level of parallelism
Reader-writer locks(II)

```c
int pthread_rwlock_init (pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);
int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

Reentrant function

- Functions executed in a multi-threaded environment need to be re-entrant
  - it can be safely called again before its previous invocation has been completed
- Requirements:
  - Must not hold static/global non-constant data.
  - Must not return the address to global, non-constant data.
  - Must work only on the data provided to it by the caller.
  - Must not rely on locks to singleton resources.
  - Must not call non-reentrant functions
- Note: some POSIX functions are not reentrant and have reentrant counterparts (e.g. `strtok` vs. `strtok_r`)