COSC 6374
Parallel Computation

Shared memory programming with POSIX Threads

Edgar Gabriel
Fall 2011

References

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

• http://www.cobweb.ecn.purdue.edu/~eigenman/ECE563/H andouts/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/LinuxTutorialPosixThre ads.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

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() )
Simple Example (I)

```c
#include <pthread.h>

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 new thread\n");
    return (0);
}

void *tfunc (void *arg){
    pid_t pid;
    pthread_t tid;
    pid = getpid(); // determine the process id
    tid = pthread_self(); // determine the thread id
    return NULL;
}
```

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 function. 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()`

Simple Example (II)

```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 A
- read
- write₁
- write₂

Thread B
- read

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_r *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)

- **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

```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);
```

Thread synchronization revisited

- Example: Force thread B to read value of shared variable after write\(_2\)
Simple Example (Illia)

```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 (Illib)

```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 lock\textsubscript{1} and waits for lock\textsubscript{2} and the other thread holds lock\textsubscript{2} and waits for lock\textsubscript{1}
  - Order for accessing mutexes has to be identical in all code paths
  - e.g. need to hold lock\textsubscript{1} in order to be allowed to hold lock\textsubscript{2}

Condition 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 (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_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 available

```c
#include <pthread.h>

pthread_mutex_t mutex;  // mutex variable
pthread_cond_t cond;    // condition variables
work_struct work_item;  // structure for work assignment

int main ( int argc, char **argv )
{
    int threadids[NUM_THREADS], ret;
    pthread_mutex_init (&mutex,NULL); //Initialize mutex
    pthread_cond_init (&cond, NULL);  //Initialize condition

    for ( int i=0; i<NUM_THREADS; i++ ) {
        ret = pthread_create (&threadids[i], NULL, fwake, NULL);
        if ( ret != 0 ) printf("Error creating a thread\n");
    }

    while (!done ){
        // when next piece of work is available assign it to a thread
        pthread_mutex_lock (&mutex );
        work_assigned=YES;
        work_item = …;
        pthread_cond_signal (&cond);
        pthread_mutex_unlock(&mutex);
    }
}
void fmake (void *arg)
{
    work_t local_work;

    while (1) {
        pthread_mutex_lock(&mutex);
        if (work_assigned == NOTHING_YET) {
            // wait for work
            pthread_cond_wait(&cond, &mutex);
        }
        // Thread awake, perform work

        // e.g. remove the work from the work_queue
        // under protection of the lock
        if (work_assigned == DONE) clean_and_quit();

        local_work = work_item;
        pthread_mutex_unlock(&mutex);
        // after work has been removed from queue,
        // release lock such that master can add
        // the next work piece and wake up another thread
        perform_work(local_work);
    }
}

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
Edgar Gabriel

Reader-writer locks (II)

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-rentrant
  - 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)

Edgar Gabriel