POSIX Thread Programming

CS 552 – Spring 2007

Binghamton University

Basics

- Process v/s threads
  - Process – Unit of resource ownership (E.g.: Data segment, Text segment)
  - Threads – Unit of dispatching (E.g.: Program Counter, execution state)
- User level threads v/s Kernel level threads
- Advantages of multi-threaded programming.
- In this lecture, we will discuss POSIX thread programming

Overview

- Threads and POSIX threads
- POSIX Thread API
  - Thread creation and destruction
  - Thread Management - Joining and detaching
- Mutex
- Condition Variables

Overview

- Threads and POSIX threads
- POSIX Thread API
  - Thread creation and destruction
  - Thread Management - Joining and detaching
- Mutex
- Condition Variables

Threads and POSIX threads

- Different hardware specific threads (e.g.: Solaris 2 threads)
- Standardization of thread programming API was critically needed for portability
- IEEE POSIX 1003.1c standard came up for UNIX based systems
  - API for ’C’ language
  - All implementations following this standard are called as POSIX threads or Pthreads
- A common header file – `pthread.h`
- A library for thread (In some implementations it is included in `libc`)
- The API should be generic but at the same time has to be aware of primary advantages of threads over process

Compilation

- Include `pthread.h`
- Solaris

  ```
  cc -mt -lpthread
  ```

- Linux

  ```
  gcc -pthread ...
  ```

- Do not manually define `_REENTRANT`, or link against any libraries.

Overview

- Threads and POSIX threads
- POSIX Thread API
  - Thread creation and destruction
  - Thread Management - Joining and detaching
- Mutex
- Condition Variables

Thread Creation

- The process has an inherent *main* thread
- What we want is to create a thread that starts executing at a given point of entry
  - Think of function as an entry point to execute thread
  - We should be able to pass an argument to this function as an input to thread
- `pthread_t` is the data structure to represent a thread. (E.g: `pthread_t myThread;`)
- Function `pthread_create` is used to create a new thread
- Signature is `int pthread_create(pthread_t *thread, const pthread_attr_t attr, void *(*start_routine)(void*), void *arg);`
  - `thread` – A pointer to `pthread_t`.
  - `attr` – The thread attribute. Passing `NULL` will initialize it by default attributes
  - `start_routine` – The point of entry for the new thread that will be created
  - `arg` – The argument for this thread

Thread Exit

- Use `pthread_exit(void* retVal)` – `man pthread_exit` for more information
- The `retVal` is returned to the thread that may *join* this thread
- `pthread_exit` does NOT close files or free memory. This things has to be done by user at appropriate places

Sample Pseudocode

```
#include<pthread.h>
void* myThreadStart(void* myarg){
 ...
 pthread_exit(NULL);
}
int main(...){
 pthread_t myThread;
 char* myarg=(char*)malloc(sizeof(char)*5);
 myarg=strcpy(myarg,“1234”);
 pthread_create(&myThread, NULL, myThreadStart, (void*)myarg);
 ...
}
```

Termination of process vs. thread

```
#include<pthread.h>
void* myThreadStart(void* myarg){
 sleep(5);
 printf(“Printing inside myThreadStart\n”);
 pthread_exit(NULL);
}
int main(...){
 pthread_t myThread;
 char* myarg=(char*)malloc(sizeof(char)*5);
 myarg=strcpy(myarg,“1234”);
 pthread_create(&myThread, NULL, myThreadStart, (void*)myarg);
 return 0;
}
```

- What happens when `main()` calls `return`?
- How to avoid the above case?

Argument Passing

- All the parameters are passed as reference and should be casted to `(void*)`
- Note that the thread can be scheduled at any time after the `pthread_create` call. So . . .
  - Make sure you do not pass stuff on the stack unless you are sure it will persist through the lifetime of the thread.
  - Use `malloc`. Don't forget `free`'ing at appropriate places
  - Should be thread-safe
- You can pass only one argument. If you have to pass multiple arguments, think of using an appropriate structure

Thread Attributes

How to specify various attributes of the thread (We will visit some of the attributes in a next few slides. E.g: detached or joinable)

- Type: `pthread_attr_t`
- Before using initialize the thread attribute by using `pthread_attr_init(pthread_attr_t *attr)`
- Set various attributes by using `pthread_attr_setXXX()` function
- Destroy the attribute by using `pthread_attr_destroy(pthread_attr_t *attr)`

Overview

- Threads and POSIX threads
- POSIX Thread API
  - Thread creation and destruction
  - Thread Management - Joining and detaching
- Mutex
- Condition Variables

Thread Management

- Thread joining
  - Wait for another thread to complete before continuing
  - When will this be helpful? – Barriers?
  - Get the return status of the completed thread – Remember `pthread_exit` call?
  - Signature: `pthread_join(pthread_t thread, void **status)`
- Thread Detaching
  - A rule to disallow other threads using “join” on a given thread.
  - Tells the thread library to reclaim the storage for the thread after it is done.

    No one will be waiting to *join* the thread
- Consider the two alternatives during designing the system.
  - Explicitly make the thread “Joinable” (different version may have different default values)
  - Free more resources when you know that no one is going to wait for the thread

Thread Management

- Join and Detach are set by using thread attributes `pthread_attr_t`
- *Generally*, the default thread attribute specifies that the thread is ’joinable’
- Creating a joinable or detached thread during `pthread_create`
  - Using `pthread_attr` – Function `pthread_attr_setdetachstate()`
- Detaching a thread explicitly
  - `pthread_detach(pthread_t thread)`
- POSIX standard specifies that threads should be created as joinable. But many implementations allow detachment

Join example

```
#include<pthread.h>
void* myThreadStart(void* myarg){
 sleep(5);
 printf(“Printing inside myThreadStart\n”);
 pthread_exit(NULL);
}
int main(...){
 pthread_t myThread; int status, ret; pthread_attr_t attr;
 char* myarg=(char*)malloc(sizeof(char)*5);
 pthread_attr_init(&attr);
 pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
 myarg=strcpy(myarg,“1234”);
 pthread_create(&myThread, NULL, myThreadStart, (void*)myarg);
 ret = pthread_join(myThread, (void **)&status); return 0;
}
```

- Wait for all threads before “return”ing from main thread

Thread management - Handy routines

- Getting the thread identifier

  `pthread_t pthread_self()`

- Comparing the threads

  `int pthread_equal(pthread_t t1, pthread_t t2)`

  Do not use == operator on `pthread_t`

- A thread can yield the processor, go back and wait in its run queue
  - `void sched_yield()`
  - `#include sched.h` and compile with `-lrt` if using `sched_yield()`
  - **NOTE:** `void pthread_yield ()` was in prior POSIX standards and not in final version. It was found that `sched_yield()` of the realtime library routine does the same job as that of `void pthread_yield ()`. So `pthread_yield` was removed. Hence, the discrepancies in compile time options and header files.

Overview

- Threads and POSIX threads
- POSIX Thread API
  - Thread creation and destruction
  - Thread Management - Joining and detaching
- Mutex
- Condition Variables

Mutex

- Many threads may read/write the same shared data and can be scheduled independently:
  - Keep the data consistent. Avoid half-done writes and do not read a data that is not completely written
- Protect *critical sections* by using locks – Mutex (**Mut**ual **ex**clusion)
- Only one thread can own a mutex (lock)
- If a thread already owns a mutex, other threads wait till it is released
- Main Idea:

  Use a mutex variable

  Acquire the mutex before entering critical section

Mutex – API

- Declare using `pthread_mutex_t`
- Initializing:
  - Static initialization : `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER`
  - Dynamic initialization : `pthread_mutex_t *mutex; pthread_mutex_init(pthread_mutex_t *, pthread_mutexattr_t* )`
- Mutex attributes: Of type `pthread_mutexattr_t`
  - Check man page for supported attributes (may vary across OS)
- Destroying the attributes
  - `pthread_mutexattr_destroy(pthread_mutexattr_t *attr)`
- Destroying the mutex
  - `pthread_mutex_destroy()` – Use it when you no longer need the mutex.

Using mutex

- Locking the mutex
  - `pthread_mutex_lock()` – If mutex is free then it locks the mutex. If mutex is owned by another thread then it *blocks* till the mutex is released
- Lock a mutex only if it is free. Otherwise return
  - `pthread_mutex_trylock()`
  - If the mutex was acquired, then returns a value 0. Else will return appropriate busy or error code
  - When is it useful?

    Deadlock prevention. Recall “Dining Philosopher’s problem”.
- Unlocking a mutex
  - `pthread_mutex_unlock()`

Using mutex – Guidelines

- Mutex does not inherently do too much of concurrency control / deadlock prevention

  Programmers have to explicitly use it to avoid deadlocks

- Think before locking two or more mutexes – Remember the dining philosophers

- Performance perspective: Acquire the mutex only when needed and release it as soon as it done

- More ? . . . Share with the class using listserv

Mutex Example

Write a RecursiveMutex class with locking and unlocking facility.

- RecursiveMutex.lock() and RecursiveMutex.unlock()
- As in pthread mutex, only one thread can hold `RecursiveMutex`
- But once the lock is held, the thread can call the `RecursiveMutex.lock()` any number of times
- The `RecursiveMutex` will not be released to other threads, unless the owner thread calls equal number of unlocks.

RecursiveMutex - Hints

- Keep track of which thread has the `RecursiveMutex` object.
- You need to keep a count of how many times a thread has called `RecursiveMutex.lock()`.
- Many different Recursive mutex objects can be created. Each one serves as a different mutex.
- Sample code available on

`http://cs.binghamton.edu/~vinkolar/cs552/sampleCode/RecursiveMutex/`

Overview

- Threads and POSIX threads
- POSIX Thread API
    - Thread creation and destruction
    - Thread Management - Joining and detaching
- Mutex
- Condition Variables

Condition variables

- Is mutex enough/efficient/easy for thread programming ? (Assume an average programmer)
  - Acquiring lock needs continuous “polling” – Overhead
  - Most of the times, programmers want to “wait” for a desired data state before acquiring locks
    - Standard example: A *Consumer* wants to wait till the *Producer* produces the data
- “Condition variables” help to access data at a given *data state* efficiently
- Two generic operations:
  - Wait: Wait till the data is in a given state
  - Signal: Wake up the waiting threads
- Relate the above operations to producer-consumer problem.

Condition variables

- Declaring a condition variable: `pthread_cond_t`
- Initializing and destroying a condition variable:
  - Statically: `pthread_cond_t cond= PTHREAD_COND_INITIALIZER;`
  - Dynamically: `pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)`
  - `pthread_cond_destroy(pthread_cond_t *cond);`
- Initializing and destroying the condition variable attributes:
  - `pthread_condattr_init()`
  - `pthread_condattr_destroy()`
- Wait and signal

```
pthread_cond_wait()
pthread_cond_signal()
pthread_cond_broadcast()
```

Condition variables

- Is condition variable enough? Does it help maintaining critical sections in a uncorrupted state?

  Use mutex in conjunction with condition variables
- Steps:
  - Recognize the shared data (e.g.: int myData)
  - Associate a condition with the data (e.g.: pthread_cond_t myDataCond) and initialize it
  - Associate a mutex (e.g.: pthread_mutex_t myDataMutex)
  - Create threads A and B to do work

Condition variables

- Thread - A
  - Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
  - Lock associated mutex and check value of a global variable
  - Call `pthread_cond_wait()` to perform a blocking wait for signal from Thread-B. Note that a call to `pthread_cond_wait()` automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
  - When signalled, wake up. Mutex is automatically and atomically locked.
  - Explicitly unlock mutex
  - Continue

Condition variables

- Thread - B
    - Do work
    - Lock associated mutex
    - Change the value of the global variable that Thread-A is waiting upon.
    - Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
    - Unlock mutex.
    - Continue

Condition Variable – Waiting

- `pthread_cond_wait()` blocks the calling thread until the specified condition is signalled.
- This routine should be called while mutex is locked, and it will automatically release the mutex while it waits.
- After signal is received and thread is awakened, mutex will be automatically locked for use by the thread. The programmer is then responsible for unlocking mutex when the thread is finished with it.

Condition Variable – Signalling

- The pthread_cond_signal() routine is used to signal (or wake up) another thread which is waiting on the condition variable.
- It should be called after mutex is locked, and must unlock mutex in order for pthread_cond_wait() routine to complete.
- The pthread_cond_broadcast() routine should be used instead of pthread_cond_signal() if more than one thread is in a blocking wait state.

Condition Variable – Usage

- It is a logical error to call pthread_cond_signal() before calling pthread_cond_wait().
- Proper locking and unlocking of the associated mutex variable is essential when using these routines. For example:
  - Failing to lock the mutex before calling pthread_cond_wait() may cause it NOT to block.
  - Failing to unlock the mutex after calling pthread_cond_signal() may not allow a matching pthread_cond_wait() routine to complete (it will remain blocked).

Condition Variable example

Write a Semaphore class with up() and down() member functions.

- Maintain a counter.
- up() function called from a thread will increment the counter
- down() function will decrement the counter only if the counter 1
  Else the thread will wait till the counter 1
- The thread calling up() function will “signal” the thread “wait”ing in the down() function if counter was zero
- Similar to Producer-Consumer problem
- Sample code available on

`http://cs.binghamton.edu/~vinkolar/cs552/sampleCode/Semaphore/`

Example: Reader-writer locks

- Consider “Reader” and “Writer” threads that will read/write a file, respectively.
- Objective: Maintain the file in a consistent state.
- There can be atmost one writer active at any given point of time
- Any number of readers can be active at the same time.

Example: Reader-writer locks

Few Design decisions:

- Suppose a writer has the lock and there is both a reader and a writer waiting. Who should run when the lock is released?
- Suppose there is a writer waiting and reader has the lock. Should we let in another reader?
  - Starvation to writers. How do you prevent it?
- Sample code available on

`http://cs.binghamton.edu/~vinkolar/cs552/sampleCode/RwMutex/`

Miscellaneous notes

- Your program may run perfectly on a single processor machine. It can no way ensure correctness on multi-processor machines
- Work on bingsuns. Submission will be tested on bingsuns for thread based assignment

Appendix – PThread function groups

Routine Prefix	Functional Group
`pthread_`	Threads themselves and miscellaneous subroutines
`pthread_attr_`	Thread attributes objects
`pthread_mutex_`	Mutexes
`pthread_mutexattr_`	Mutex attributes objects
`pthread_cond_`	Condition variables
`pthread_condattr_`	Condition attributes objects
`pthread_key_`	Thread-specific data keys

Table 1: Summary of thread functions

References

- POSIX Threads Programming.
  `http://www.llnl.gov/computing/tutorials/pthreads/`
- Prof. Ken's OS course material
  `http://grid.cs.binghamton.edu/cs552/`
- Thanks to Mr. Avadh Patel for providing sample code for RecursiveMutex, Semaphore and RwMutex examples.