Pre-lab #2 tutorial

ECE 254
Operating Systems and Systems Programming

May 24, 2012

Content

- Concurrency
- Concurrent Programming
- Thread vs. Process
- POSIX Threads
- Synchronization and Critical Sections
- Mutexes
- Semaphores

Concurrency

- Concurrency is a property where multiple paths of execution are running simultaneously
- Multiple processes/threads running on a single processor
- Multiple processes/threads running on multiple processors/machines (true concurrency)
- Requires a different programming paradigm: Concurrent programming

Concurrent Programming

- A programming paradigm that involves designing programs so that they can run in parallel
- Concurrent programs can share data and resources
- Two mechanisms of communication among concurrent components:
  - Message passing
  - *Shared memory*

Concurrent Programming (Cont.)

- Main challenges of concurrent programming:
    - Ensuring correct and meaningful order of execution
    - Coordinating access to shared data and resources to ensure their consistency throughout execution

Thread vs. Process

- A thread is a mechanim to allow a program to run things in parallel (like processes)
- A thread is a finer-grained unit of execution than a process. Each running process has at least one thread
- Each process has its own memory space, variables etc.
- Threads in a process share the same memory space, variables etc.

POSIX Threads

- POSIX (**P**ortable **O**perating **S**ystem **I**nterface) is a group of standards for compatability among different operating systems
- POSIX Threads (*pthreads*) is a standard for creating and using threads
- We will use the GNU/Linux implementation of the *pthreads*

Thread Creation

- We will mainly use three functions: *pthread_create* , *pthread_join* and *pthread_cancel*
- We use the *pthread_create* to create a new thread, it takes four parameters as follows:
  - A pointer to *pthread_t* variable which stores the thread ID
  - A pointer to a thread attribute object which specifies how the thread interacts with the program, or a *NULL* value
  - Pointer to the function the thread will execute. The function must have a return type *void** and takes *void** as a parameter
  - A thread argument value of type *void** or *NULL* value

Thread Creation (Cont.)

```
#include <pthread.h>
#include <stdio.h>

void*
print_func (void* args)
{
  int count;
  int i;

  count = *(int *) args;
  for (i = 0; i < count; ++i)
    fprintf (stdout, "Hello world!\n");

  return NULL;
}

int
main (int argc, char **argv)
{
  pthread_t thread_id;
  int count;
  int i;

  count = 10000;

  pthread_create (&thread_id, NULL, &print_func, (void*) &count);
  pthread_join (thread_id, NULL);

  return 0;
}
```

Thread Creation (Cont.)

- We use the *pthread_join* to ensure that the thread executing *main* will wait for the other thread to finish
- This ensures that the created thread will no be using deallocated variables

Thread Cancellation

- One thread can request the termination of another thread by calling *pthread_cancel*, passing the thread ID as an argument
- A thread can have one of three different "cancellation" states:
  - Asynchronously cancelable
  - Synchronously cancelable
  - Uncancelable
- A thread can alter its "cancellation" at any time using the *pthread_setcanceltype* function

Synchronization and Critical Sections

- There is no way to tell when a specific thread will be scheduled or at which line of code the OS will suspend one thread and re-schedule another
- If there is a concurrency bug, it is very hard to fix since it won't be deterministic
- The main cause of concurrency bugs is threads attempting to access the same resource/variable at the same time. This is called a *race condition*
- Race conditions can crash a program, or even worse leave variables in a corrupted state which could radically alter the behavior of the program

Race Condition

```
#include <malloc.h>

struct job
{
  struct job *next;
};

struct job *job_queue;

void*
thread_function (void *arg)
{
  while (job_queue != NULL)
  {
    struct job *next_job = job_queue;

    job_queue = job_queue->next;
    process_job (next_job);

    free (next_job);
 }

 return NULL;
}
```

Atomicity

- To avoid concurrency bugs and race conditions, you have to make sure that operations on shared variables are *atomic*
- An atomic operation means it is totally indivisible and uninterruptible
- To ensure atomicity we will use:
    - Mutexes
    - Semaphores

Atomicity (Cont.)

- Mutex stands for *MUTual EXclusion locks*
- A special lock that one thread can lock at a time, before enterting a *critical section*
- Any thread attempting to lock an already locked mutex will be blocked
- The underlying OS ensures that no race conditions will occur among threads using the mutex
- To create a mutex on Linux:

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```

- To lock/unlock a mutex:

```
pthread_mutex_lock (&mutex)
pthread_mutex_unlock (&mutex)
```

Mutex in action

```
#include <malloc.h>
#include <pthread.h>

    ...................
pthread_mutex_t job_queue_mutex;
pthread_mutex_init (&job_queue_mutex, NULL);

void*
thread_function (void *arg)
{
  while (1)
  {
    struct job* next_job;

    /** Start of critical section **/
    pthread_mutex_lock (&job_queue_mutex);
    if (job_queue == NULL)
    {
      next_job = NULL;
    }
    else
    {
      next_job = job_queue;
      job_queue = job_queue->next;
    }
    pthread_mutex_unlock (&job_queue_mutex);
    /** End of critical section **/

    process_job (next_job);
    free (next_job);
  }
  return NULL;
}
```

Deadlocks

- Mutexes introduce another type of concurrency bugs: *deadlocks*
- One or more threads can block forever waiting for a mutex to be unlocked
- A deadlock can easily take place if a thread who originally locked a mutex tries to lock it again
- In Linux, this can happen with the default type of mutexes, the *fast mutex*
- The *recursive mutex*, on the other hand, can be locked any number of times, but it must also be unlocked the same number of times

Semaphores

- In the previous queue example, what if all the jobs are not queued in advance?
- There is a possibility that threads can finish processing before all jobs are pushed onto the queue
- We need a *semaphore* to synchronize the threads, so that they can "wait" if there are no jobs on the queue at the moment

Semaphores (Cont.)

- A semaphore stores a non-negative counter which can be incremented/decremented. It supports two main operations:
  - *wait*: decrements the value of a semaphore by one. If the value is already zero, the function blocks and hence the calling thread
  - *post*: increments the value of a semaphore by one. If the value was originally zero, one of the blocked threads (if any) will be allowed to run
- We will use the POSIX standard semaphore for thread synchronization on Linux. There is another type of Linux semaphore used with processes (not covered)
- Semaphore common operations:

```
sem_wait
sem_post
sem_trywait   // Non blocking wait
sem_getvalue  // Get value of the semaphore
```

Semaphores Example

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>

struct job
{
  struct job *next;
};

struct job *job_queue;

pthread_mutex_t job_queue_mutex;
pthread_mutex_init (&job_queue_mutex, NULL);

sem_t job_queue_count;

void
initialize_job_queue ()
{
  job_queue = NULL;
  sem_init (&job_queue_count, 0, 0);
}

  ..........................
```

Semaphores Example (cont.)

```
  .........................

void*
thread_function (void *arg)
{
  while (1)
  {
    struct job *next_job;

    sem_wait (&job_queue_count);

    /** Start of critical section **/
    pthread_mutex_lock (&job_queue_mutex);
    next_job = job_queue;
    job_queue = job_queue->next;
    pthread_mutex_unlock (&job_queue_mutex);
    /** End of critical section **/

    process_job (next_job);
    free (next_job);
  }
  return NULL;
}

  .........................
```

Semaphores Example (cont.)

```
    ..........................

void
enqueue_job (/* Pass job-specific data here */)
{
  struct job *new_job;

  new_job = (struct job*) malloc (sizeof (struct job));

  /** Start of critical section **/
  pthread_mutex_lock (&job_queue_mutex);
  new_job->next = job_queue;
  job_queue = new_job;

  sem_post (&job_queue_count);

  pthread_mutex_unlock (&job_queue_mutex);
  /** End of critical section **/
}
```

Questions

Questions ?

References

- M. Mitchell, J. Oldham, and A. Samuel (2001). Threads , *Advanced Linux Programming*. Indianapolis:New Riders Publishing