Project 2 : Concurrency I

15-412 Operating Systems

Due: Wed Feb 19 23:59:59 EST 2003

1 Overview

An important aspect of operating system design is organizing tasks that run concurrently and share memory. Concurrency concerns are paramount when designing multi-threaded programs that share some critical resource, be it some device or piece of memory. In the second project you will write a thread library, concurrency primitives, and a simple game that uses the libraries you write. This document provides the background information and specification for writing the thread library, and concurrency primitives.

The thread library will be based on the provided sys_minclone system call, which will be described later. The basic features of a user level thread library will be will be implemented, including the ability to join threads.

The concurrency primitives will be based on the XCHG instruction for atomically exchanging registers and memory or registers and registers. With this instruction you will implement mutexes, and condition variables.

2 Goals

- Becoming familiar with the ways in which operating systems support user libraries by providing system calls to create processes, affect scheduling, etc.
- Becoming familiar with programs that involve a high level of concurrency, and the sharing of critical resources, including the tools that are used to deal with these issues.
- This project will involve a substantial amount of code, so it will be critical to organize your work in an effective and straightforward way.
- Working with a partner is also an important aspect of this project. You will be working with a partner on your kernel and file system, so it is important to be familiar with scheduling time to work, a preferred working environment, and developing a good group dynamic before beginning those larger projects.

3 Important Dates

- Wednesday, February 5th: Project Assigned.
- Thursday, February 6th: Read the handout many times until you have your brain wrapped around it. Then read it again. Send mail to staff-412@cs with questions.
- Wednesday, February 12th: A majority of your code for the project should be designed and written, so that there is plenty of time to debug and discover concurrency issues.
- Wednesday, February 19th: Project Due at 2359 EST.

4 The System Call Interface

The kernel provides system calls for your use. It however does not provide a C library for accessing those calls. For this project you will write an assembly code wrapper around the system calls, and use that to develop a C library.

To ask the kernel to do a system call, the following protocol is followed. The argument is placed in %esi, the system call number(defined in user_syscall.h) is placed in %edi, and then interrupt number 0x40 is raised by using the INT instruction. As usual, the return value of the system call is stored in %eax on return from the call. Please remember that %esi and %edi are callee saved registers. For example, to write a wrapper around the get_pid() system call, the following assembly code could be written.

```
get_pid:
 push %ebp        /* push the base pointer on to the stack */
 mov  %esp, %ebp  /* move the stack pointer into the base pointer */
 push %edi        /* save %edi on the stack */
 mov  $0x16, %edi /* move 22d = 16h into %edi */
 int  $0x40       /* the syscall */
 pop  %edi        /* at this point the requested pid is in %eax,
                     we restore the previous value of %edi */
 mov  %ebp, %esp  /* restore the stack pointer */
 pop  %ebp        /* restore the base pointer */
 ret              /* return from the system call wrapper */
```

You may find it convenient to write some more general assembly code wrapper for the system call interface, rather than writing code like this for each system call.

4.1 Provided System Calls

The kernel provides the following system calls to support your library. Please use the following naming convention in writing your system call stubs. Unless otherwise noted, system calls return zero on success and a negative number on an error.

- int sys_minclone(void *), syscall number: 13. A call to this function creates a child process that is an exact duplicate of the calling process with the exception of a single register. The parent and child processes share the same virtual memory space, and differ only in the contents of one register when the function returns. You may pass an argument to your sys_minclone wrapper, however, the system call takes no arguments. On success, zero is returned to the child process, and the pid of the child process is returned to the parent. Minus one is returned on error.

- int sys_yield(int pid), system call number: 14. The yield system call will cause the calling process to stop running, and allow another process to run. The kernel will schedule the yielded process to run some finite amount of time in the future. If pid is negative, the kernel's scheduler picks which process runs next, if pid is the process ID of some runnable process, then that process is run next.

- int sys_stopme(int *lock), system call number: 18. The stopme system call will place the calling process in a blocked queue if the integer pointed to by lock is zero. The process will continue to run normally otherwise. This system call is atomic.
- int sys_markrunnable(int pid), system call number: 17. If process pid is blocked, this system call will allow it to be scheduled to run by the kernel again. If process pid is not blocked or does not exist, minus one is retunred.
- void *sys_sbrk(void *addr), system call number: 12. The sbrk system call will move the top of the calling process' heap to the address specified in the argument. If zero is passed as an argument, the current address of the top of the heap will be returned.
- int sys_get_pid(void), system call number: 22. Returns the pid of the currently running process.
- void sys_exit(int status), system call number: 5. This function will end execution of the calling process. If the calling process is not sharing resources with any other processes, those resources are deallocated. Your user programs must call sys_exit when they are finished.
- void sys_readline(char *buf), system call number: 8. This function takes a pointer to a buffer of at least eighty bytes and inserts characters from the console into the buffer until in encounters a newline or fills the buffer.
- void sys_set_term_color(int color), system call number: 10. Changes the console to color 'color.' The colors are defined in user_syscall.h.
- void sys_set_cursor_pos(pos_t *), system call number: 11. Sets the cursor position. Takes a pointer to a pos_t structure as defined in user_syscall.h.
- int sys_rand(void), system call number: 20. Returns a random integer.
- int sys_sleep(int time), system call number: 19. Deschedules the calling process for 'time' milliseconds.
- int sys_paint_sprite(sprite_t *), system call number: 15. Takes a pointer to a sprite_t structure as defined in user_syscall.h. Paints sprite at location of cursor. Hides the cursor automatically. The cursor is shown on subsequent calls to sys_print. Returns zero on success, and minus one on an error.
- char sys_get_single_char(void), system call number: 16. Takes nothing and returns a single key press. The calling process will be blocked until there is a character from the console to give to it.

5 Writing a User Level Thread Library

For this project you will implement a user level thread library based on sys_minclone, a modified version of the fork system call. As part of the library you will also implement user level mutexes and condition variables.

5.1 The Thread Library API

- int thr_init(unsigned int size) - This function is responsible for initializing the thread library. The argument 'size' specifies the size of the stack(plus thread private memory) that each thread will have. This function must be called before any call to any of the other thread library functions, and it must only be called once. Calling other thread functions before calling thr_init may have an undefined effect. This function returns zero on success, and a negetive number on an error.

- int thr_create(void *(*func)(void *), void *arg) - This function creates a new thread to run func(arg). This function should allocate a stack and private memory area for the new thread and then call sys_minclone. A stack frame should be created for the child, and the child should be provided with some way of accessing its thread identifier(tid). On success the thread id of the new thread is returned, on error a negative number is returned.
  Great care must be taken by the new child process such that the parent's stack remains consistent. For this reason it is suggested that in the assembly wrapper for sys_minclone do something like the following be done:

  ```
  sys_minclone:
          ...
          int     $0x40          /* the syscall */
          test   %eax, %eax      /* check if it returned 0 */
          je     newproc         /* if it did jump to newproc */
          ...
  newproc:
          /* point the stack pointer at the new stack */
          /* jump to func */
  ```

- int thr_join(int tid, int *departed, void **status) - This function suspends execution of the calling thread, and waits for thread 'tid' to thr_exit if it exists. If 'tid' is zero any thread associated with the process is joined on. If departed is not NULL, it points to a location where the tid of the departing thread should be stored. If status is not NULL, the value passed to thr_exit by the terminating thread will be placed in the location referenced by status. Only one thread may join on any given thread. Others will return an error. If thread 'tid' does not exist, an error will be returned. This function returns zero on success, and a negetive number on an error.

- void thr_exit(void *status) - This function exits the thread with exit status 'status.' If a thread does not call thr_exit, the behavior should be the same as if the function did call thr_exit and passed in the return value from the thread's body function.

- int thr_getid(void) - Returns the tid of the currently running thread.

You may also need to write some support functions. The one below is optional and suggested, but not a official member of the API.

- void *thr_private() - Returns a pointer to the private area of the currently running thread.

5.2 Mutexes

Mutual exclusion locks prevent multiple threads from simultaneously executing critical sections of code. To implement mutexes you may use the XCHG instruction documented on page 3-714 of the Intel instruction set reference, and the yield() system call. For more information on the behavior of mutexes, feel free to refer to the text, or to the Solaris or Linux man pages for the functions of names pthread_mutex_init, etc..

- int mutex_init(mutex_t *mp) - This function should initialize the mutex pointed to by mp. Effects of the use of a mutex before the mutex has been initialized may be undefined. This function returns zero on success, and a negetive number on an error.
- int mutex_destroy(mutex_t *mp) - This function should destroy the mutex pointed to by mp. The effects of using a mutex after it has been destroyed may be undefined. If this function is called while the mutex is locked, it should immediately return an error. This function returns zero on success, and a negative number on an error.
- int mutex_lock(mutex_t *mp) - A call to this function ensures mutual exclusion in the region between itself and a call to mutex_unlock. A thread calling this function while another thread is in the critical section will yield until it is able to claim the lock. This function returns zero on success, and a negetive number on an error.
- int mutex_unlock(mutex_t *mp) - Signals the end of a region of mutual exclusion. The calling thread gives up its claim to the lock. This function returns zero on success, and a negetive number on an error.

5.3 Condition Variables

Condition variables are used for waiting, for a while, for mutex-protected state to be modified by some other thread. A condition variable allows a thread to voluntarily relinquish the CPU so that other threads may make changes to the shared state, and then tell the waiting thread that they have done so. If there is some shared resource, threads may de-schedule themselves and be woken up by whichever thread was using that resource when that thread is finished with it. In implementing condition variables, you may use your mutexes, and the system calls sys_stopme and markrunnable. For more information on the behaviour of condition variables, please the man pages on either Solaris or Linux for the functions pthread_cond_wait, etc.

- int cond_init(cond_t *cv) - This function should initialize the condition variable pointed to by cv. Effects of using a condition variable before it has been initialized may be undefined. This function returns zero on success and a number less than zero on error.
- int cond_destroy(cond_t *cv) - This function should destroy the condition variable pointed to by cv. Effects of using a condition variable after it has been destroyed may be undefined. If cond_destroy is called while threads are still blocked waiting on the condition variable, then the function should return an error immediately. This function returns zero on success and a number less than zero on an error.
- int cond_wait(cond_t *cv, mutex_t *mp) - The condition wait function allows a thread to wait for a condition and release the associated mutex that it needs to hold to check that

condition. The calling thread blocks, waiting to be signaled. The blocked thread may be awakened by a cond_signal or a cond_broadcast. This function returns zero on success, and a negetive number on an error.

- int cond_signal(cond_t *cv) - This function should wake up a thread waiting on the condition variable pointed to by cv, if one exists. This function returns zero on success, and a negetive number on an error.
- int cond_broadcast(cond_t *cv) - This function should wake up all threads waiting on the condition variable pointed to by cv. This function returns zero on success, and a negetive number on an error.

5.4 Notes on the Thread Library

Please keep in mind that much of the code for this project needs to be thread safe. In particular the thread library itself should be thread safe.

5.5 Distribution Files

The tarball for this project has been posted on the course webpage. Note that the Makefile included with this project runs a script which will check to see if you are using the latest version of the provided libraries. Please obtain the new libraries if the script notifies you that you are using an out of date version. We will post details on new libraries when they are made available.

Please read the README included with the tarball.

6 Game Details

Details on the game, as well as support libraries for it, will be made available early next week. Until then, work on completing and testing your thread library.

7 The C Library

In general the C Library functions are not provided. You may use:

- mm_malloc()
- mm_free()

to dynamically allocate memory, but any other functions you want such as strlen or memcpy etc, you must write on your own. Further, mm_malloc() and mm_free() are not thread safe. You must call mm_init() once in your thread family before using malloc.

8 Debugging

At the current time, we have a special library call that we will provide that will write strings to a log file called kernel.log and to the simics console.

- void lprintf(char *format, ...). It takes a printf style format string, and a list of things to print.

We are working on giving simics the ability to debug arbitrary threads. In the meantime, you can experiment with debugging the currently running thread. This probably means needing to be creative about setting breakpoints in the simics symbolic debugger.

9 Deliverables

You will implement the functions for the thread library, and concurrency tools conforming to the documented APIs. You will hand in all source files that you generate, as well as a Makefile that builds your code. You will also hand in a DESIGN file described below. It is also suggested that you write a README to document your project, including a description of of your source files, along with any existing issues, or peculiarities with your library and game.

10 Grading Criteria

You will be graded on the completeness and correctness of your project based on these criteria.

- We will read your code carefully, so we need to be able to understand it. You should provide a design document, in plain text, called DESIGN, with a description of reasonable length for each function. For example a long and complicated function should probably have a longer description that a short simple function.
- Correctly conforming to and implementing the provided API.
- Identifying concurrency issues, and writing thread safe code. Your thread library and concurrency tools will be run on many test cases and carefully read.
- Your thread library functions should not become deadlocked or be smacked down by the kernel.
- The quality of your explanation of the workings of your project at a possible demo and code review.

11 Strategy

1. Read the handout.
2. Read it again.
3. Write system call wrappers and get them working with the possible exception of minclone.
4. Design and make a draft version of mutexes and condition variables.
5. Think hard about stacks. What should the child's stack look like before and after a sys_minclone?

6. Get sys_minclone working.
7. Write and test thr_create, and thr_exit. Don't worry about reporting exit status, yet.
8. Test mutexes and condition variables
9. Write and test thr_join.
10. Worry about reporting the exit status.
11. Put the thread library and concurrency tools together;
12. Test. Test. Test. Test. Test. Test.
13. Begin on using your thread library for the game.