Project 2 : User Level Thread Library

15-410 Operating Systems

September 19, 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 this project you will write a thread library and concurrency primitives. This document provides the background information and specification for writing the thread library, and concurrency primitives.

We will provide you with a miniature operating system kernel (called "Pebbles") which implements a minimal set of system calls, and some multi-threaded programs. These programs will be linked against your thread library, stored on a "RAM disk", and then run under the supervision of the Pebbles kernel.

The thread library will be based on the provided `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.
- Developing skills necessary to produce a substantial amount of code, such as organization and project planning.
- Working with a partner is also an important aspect of this project. You will be working with a partner on subsequent projects, so it is important to be familiar with scheduling time to work, a preferred working environment, and developing a good group dynamic before beginning larger projects.
- Coming to understand the dynamics of source control in a group context, e.g., when to branch and merge.

3 Important Dates

Wednesday, September 17th Project 2 begins

Wednesday, September 24th You should have thread creation, mutexes, and condition variables working well.

Wednesday, October 1st Project 2 due at 23:59:59

4 User Execution Environment

Pebbles supports multiple independent processes, which do not share memory. Each process contains several memory regions. From lowest memory address to highest, they are:

- A read-only code region containing machine instructions
- A data region containing some variables. Part of the data region is derived from an executable program file, but the process may ask the operating system to increase the size of the data region. If the operating system is willing to do this, new pages of memory are added to the top of the data region.
- A stack region containing a mixture of variables and procedure call return information. The stack begins at some "large" address and grows downward toward the top of the data region. Of course, if they collide, disaster will result.

The boundary between the top of the data region and the "memory hole" (which reaches to the bottom of the stack region) is called, in homage to the Unix tradition, "the break." The break is a pointer to the smallest memory address in the hole, so growing the data region and increasing the break are by definition equivalent (see `brk()` below).

Pebbles allows one process to create another though the use of the `fork` and `exec` system calls, which you will not use for Project 2 (the shell program which we provide so you can launch your test programs does use them).

In addition, Pebbles allows a process to create "clones," processes which share the entire memory space of their parent (and siblings). Once a process has created one or more clones, the effect is that there are multiple schedulable register sets sharing one set of memory resources. A carefully designed set of cooperating library routines can leverage this feature to provide a simplified version of POSIX "pthreads."

5 The System Call Interface

While the kernel provides system calls for your use, It does not provide a "C library" which accesses those calls. You will need to begin your implementation by writing an assembly code "wrapper" or "stub" for each system call (well, all but one, see below). Stub routines should be one per file and you should arrange that the Makefile infrastructure you are given will build them into `libsyscall.a` (see the `README` file in the tarball).

To invoke a system call, the following protocol is followed. If the system call takes one 32-bit parameter, it is placed in the `%esi` register. Then the appropriate interrupt, as defined in `syscall_nums.h`, is raised via the `INT x` instruction (each system call has been assigned its own `INT` instruction, hence its own value of `x`)). If the system call expects more than one 32-bit parameter, you should construct a "system call packet" containing the parameters and place the *address* of the packet in `%esi`. In C you would create a structure like this:

```
struct read_line_parms {
  int len;
  char *buf;
} rlp;
```

After filling in the struct, you would arrange for `&rlp` to be placed in `%esi`. When the system call completes, the return value, if any, will be available in the `%eax` register.

Please remember that `%esi` and `%edi` are "callee-saved" registers according to the C run-time calling convention in use by gcc. This means that the C function which called your stub routine did so with the expectation that `%esi` and `%edi` would not be modified. If you use either of those registers in your assembly code, you will probably need to save them to and restore them from the stack each time.

5.1 Provided System Calls

The kernel provides the following system calls to support your library. *You must use the following naming convention, declared for you in `user/lib/inc/syscall.h`, when writing your system call stubs, so our test programs can call your stubs.*

Unless otherwise noted, system calls return zero on success and an error code less than zero if something goes wrong.

- `int yield( int pid )` - Defers execution of the calling process to a time determined by the scheduler, in favor of the process with process ID `pid`. If `pid` is -1, the caller is not expressing a preference for which process should run next. The only processes whose scheduling should be affected by `yield()` are the calling process, and the process that is `yield()`ed to. If the process with process ID `pid` is not runnable, or doesn't exist, then an integer error code less than zero is returned. Zero is returned on success.

- `int deschedule( int *reject )` - Examines the integer pointed to by `reject`. If the integer is non-zero, the call returns immediately with return value zero. If the integer pointed to by `reject` is zero, then the calling process will be suspended (not run by the scheduler) until some other process makes a call to `make_runnable()` on the process that called `deschedule()`. An integer error code less than zero is returned if `reject` is not a valid pointer. This system call is *atomic* with respect to `make_runnable()`: the process of examining `reject` and suspending the process will not be interleaved with any execution of `make_runnable()` by another process.

- `int make_runnable( int pid )` - Makes the `deschedule()`d process with process ID `pid` runnable by the scheduler. On success, zero is returned. If `pid` is not the process ID of a process suspended due to having called `deschedule()`, then an integer error code less than zero is returned.

- `void *brk( void *addr )` - Sets the break value of the calling process. The initial break value of a process is the address immediately above the last address used for program instructions and data. This call has the effect of allocating or deallocating enough memory to cover only up to the specified address, rounded up to an integer multiple of the page size. If `addr` is less than the initial break value, or if `addr` is within four pages of the stack, the break point is not changed. The return value of `brk()` is always the break value after the call has been performed, even if the break value does not change.

- `int get_pid( void )` - Returns the process ID of the calling process.
- `void exit( int status )` - Terminates execution of the calling process immediately. In the case that the process' resources are not being shared, all resources used by the calling process are reclaimed.
- `void read_line( int len, char *buf )` - Reads the next line from the console and copies it into the buffer pointed to by `buf`. If there is no line of input currently available, the calling process is descheduled until one is. The length of the buffer is indicated by `len`. If the length of the line exceeds the length of the buffer, only `len`-1 characters should be copied into `buf`. The `read_line()` call will NULL-terminate `buf` if there is room after the last character placed therein.
- `int print( int size, char *buf )` - Prints the specified bytes to the console. The length of the buffer `buf` is indicated by `len`. Returns an integer error code less than zero if the buffer specification is invalid. Returns zero otherwise.
- `void set_term_color( int color )` - Sets the terminal print color for any future output to the console. If `color` does not specify a valid color, an integer error code less than zero will be returned. Zero is returned on success.
- `void set_cursor_pos( int row, int col )` - Sets the cursor to the location `(row, col)`. If the location is not valid, an integer error code less than zero is returned. Zero is returned on success.
- `int sleep( int time )` - Deschedules the calling process until at least `ticks` timer interrupts (which occur every 10 milliseconds) have occured after the call. Returns immediately if `ticks` is zero. Returns an integer error code less than zero if `ticks` is negative. Returns zero otherwise.
- `char getchar( void )` - Returns a single character from the character input stream. If the input stream is empty the process is descheduled until a character is available.

In addition to the system calls accessed through stub routines, you will use the `minclone()` system call. However, you will *not* create a stub of the form `int minclone(void)`. When you call `minclone()`, the kernel creates a new thread. *Almost* everything about the new thread is identical to the corresponding part of the old thread. For example, they share the existing address space and all memory regions. Likewise, all register values are identical, with the exception of `%eax`, which is set to the process/thread ID of the created thread in the calling thread and zero in the new thread. If something goes wrong, an error code less than zero is returned to the calling thread, and no new thread is created.

6 Thread Library API

The library you will write will contain:

- Thread management calls
- Mutexes and condition variables
- Semaphores
- Readers/writers locks

Please note that all lock-like objects are defined to be "unlocked" when created. Semaphores are defined to contain the value 1 when created.

Unlike system call stubs, thread library routines need not be one-per-source-file, but you should use good judgement in how you partition them. You should arrange that the Makefile infrastructure you are given will build all of them them into `libthread.a` (see the `README` file in the tarball).

6.1 Thread Management 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 negative 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 for the new thread and then invoke the `minclone()` system call. 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.

  You should pay attention to (at least) two stack-related issues. First, the stack pointer should essentially always be aligned on a 32-bit boundary (i.e., %esp mod 4 == 0). Second, you need to think very carefully about the relationship of a new thread to the stack of the parent thread, especially right after the `minclone()` system call has completed.

- `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 is the address 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 immediately. If thread `tid` does not exist, an error will be returned. This function returns zero on success, and a negative 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 thread ID of the currently running thread.

6.2 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. 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 negative 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 should block until it is able to claim the lock. This function returns zero on success, and a negative 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 negative number on an error.

6.3 Condition Variables

Condition variables are used for waiting, for awhile, 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 `deschedule()` and `make_runnable()`. For more information on the behaviour of condition variables, please see 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 negative 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 negative 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 negative number on an error.

6.4 Semaphores

As discussed in class, semaphores are a higher-level construct than mutexes and condition variables. Implementing semaphores on top of mutexes and condition variables should be a straightforward but hopefully illuminating experience.

- `int sem_init( sem_t *sem )` - This function should initialize the semaphore pointed to by `sem`. Effects of using a semaphore before it has been initialized may be undefined. This function returns zero on success and a number less than zero on error.

- `int sem_destroy( sem_t *sem )` - This function should destroy the semaphore pointed to by `sem`. Effects of using a semaphore after it has been destroyed may be undefined. If `sem_destroy()` is called while threads are still blocked waiting on the semaphore, then the function should return an error immediately. This function returns zero on success and a number less than zero on an error.

- `int sem_wait( sem_t *sem )` - The semaphore wait function allows a thread to decrement a semaphore value, and may cause it to block indefinitely until it is legal to perform the decrement. This function returns zero on success, and a negative number on an error.

- `int sem_signal( sem_t *sem )` - This function should wake up a thread waiting on the semaphore pointed to by `sem`, if one exists, and should update the semaphore value regardless. This function returns zero on success, and a negative number on an error.

6.5 Readers/writers locks

Please refer to Section 7.5.2 of the textbook. We expect you to solve at least the "second" readers/writers problem, but we would like to point out that there are other formulations than the "first" and "second." You may choose to implement something "at least as good as" the "second" case. Of course, no matter what you choose to implement you should explain what, how, and why. You may choose which underlying primitives (i.e., mutex/cvar or semaphore) to employ, but once again we are interested in the *reasoning* you employ.

- `int rwlock_init( rwlock_t *rwlock )` - This function should initialize the lock pointed to by `rwlock`. Effects of using a lock before it has been initialized may be undefined. This function returns zero on success and a number less than zero on error.

- `int rwlock_destroy( rwlock_t *rwlock )` - This function should destroy the lock pointed to by `rwlock`. Effects of using a lock after it has been destroyed may be undefined. If `rwlock_destroy()` is called while threads are still blocked waiting on the lock, then the function should return an error immediately. This function returns zero on success and a number less than zero on an error.

- `int rwlock_lock( rwlock_t *rwlock, int type )` - The `type` parameter is required to be either `RWLOCK_READ` (for a shared lock) or `RWLOCK_WRITE` (for an exclusive lock). This function blocks the calling thread until it has been granted the requested form of access. This function returns zero on success, and a negative number on an error.

- `int rwlock_unlock( rwlock_t *rwlock )` - This function indicates that the calling thread is done using the locked state in whichever mode it was granted access for. Whether a call to this function does or does not result in a thread being awakened depends on the policy you chose to implement. This function returns zero on success, and a negative number on an error.

6.6 A Note 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.

6.7 Distribution Files

The tarball for this project has been posted on the course webpage. Please read the README included with the tarball.

7 Documentation

For each project in 15-410, functions and structures should be documented using doxygen. Doxygen uses syntax similar to Javadoc. The Doxygen documentation can be found on the course website. The provided Makefile has a target called `html_doc` that will invoke doxygen on the source files listed in the Makefile.

8 The C Library

This is simply a list of the most common library functions that are provided. For details on using these functions please see the appropriate `man` pages.

Other functions are provided that are not listed here. Please see the appropriate header files for a full listing of the provided functions.

Some functions typically found in a C I/O library are provided by `user/lib/libstdio.a`. The header file for these functions is `user/lib/inc/stdio.h`.

- `int putchar(int c)`

- `int puts(const char *str)`

- `int printf(const char *format, ...)`
- `int sprintf(char *dest, const char *format, ...)`
- `int snprintf(char *dest, int size, const char *formant, ...)`
- `int sscanf(const char *str, const char *format, ...)`
- `void lprintf( const char *format, ...)`

Note that `lprintf()` is the user-space analog of the `lprintf_kern()` you used in Project 1.

Some functions typically found in various places in a standard C library are provided by `user/lib/libstdlib.a`. The header files for these functions, in `user/lib/inc`, are `stdlib.h`, `assert.h`, and `ctype.h`.

- `int atoi(const char *str)`
- `long atol(const char *str)`
- `long strtol(const char *in, const char **out, int base)`
- `unsigned long strtoul(const char *in, const char **out, int base)`
- `void panic(const char *format, ...)`
- `void assert(int expression)`

We are providing you with *non-thread-safe versions* of the standard C library memory allocation routines. You are *required* to provide a thread-safe wrapper routine with the appropriate name (remove the underscore character) for each provided routine. These should be genuine wrappers, i.e., you should *not* copy and modify the source code for the provided routines.

- `void *_malloc(size_t size)`
- `void *_calloc(size_t nelt, size_t eltsize)`
- `void *_realloc(void *buf, size_t new_size)`
- `void _free(void *buf)`

Some functions typically found in a C string library are provided by `user/lib/libstring.a`. The header file for these functions is `user/lib/inc/string.h`.

- `int strlen(const char *s)`
- `char *strcpy(char *dest, char *src)`
- `char *strncpy(char *dest, char *src, int n)`
- `char *strdup(const char *s)`
- `char *strcat(char *dest, const char *src)`

- `char *strncat(char *dest, const char *src, int n)`
- `int strcmp(const char *a, const char *b)`
- `int strncmp(const char *a, const char *b, int n)`
- `void *memmove(void *to, const void *from, unsigned int n)`
- `void *memset(void *to, int ch, unsigned int n)`
- `void *memcpy(void *to, const void *from, unsigned int n)`

9 Debugging

The same `MAGIC_BREAK` macro which you used in Project 1 is also available to user processes in Project 2 if you `#include` the `user/inc/magic_break.h` header file.

The function call `lprintf()` may be used to output debugging messages from user programs. Its prototype is in `user/lib/inc/stdio.h`.

Also, user processes can be symbolically debuged using the Simics symbolic debugger.

10 Deliverables

Implement the functions for the thread library, and concurrency tools conforming to the documented APIs. Hand in all source files that you generate. Make sure to provide a design description in README.dox, including an overview of existing issues and any interesting design decisions you made.

11 Grading Criteria

You will be graded on the completeness and correctness of your project. A complete project is composed of a reasonable attempt at each function in the API. Also, a complete project follows the prescribed build process, and is well documented. A correct project implements the provided specification. Also, processes using the API provided by a correct project will not be killed by the kernel, and will not suffer from inconsistencies related to concurrency errors in the library. Please note that there exist concurrency errors that even carefully written test cases may not expose. Read and think through the code carefully. Do not forget to consider pathalogical cases.

The most important parts of the assignment to complete are the thread management, mutex, and condition variable calls. These should be well-designed and solidly implemented. It is probably unwise to devote substantial coding effort to the other parts of the library before the core is reliable.

12 Strategy

1. Read the handout.

2. **Right away** write system call wrappers for one or two system calls and run a small test program using those system calls. This is probably the best way to engage yourself in the project and to get an initial grasp of its scope. A good first choice is `exit()`, since the C run-time start-up code requires an `exit()` stub to exist before you can build any test program. A good second choice would be `print()`.
3. Write the remaining system call wrappers (with the exception of `minclone()`).
4. Design and make a draft version of mutexes and condition variables. In order to do that, you will probably need to perform a hazard analysis of which system calls or system call sequences would harm each other if their execution were interleaved by the scheduler switching from one process to another.
5. What can you test at this point? Be creative.
6. Think hard about stacks. What should the child's stack look like before and after a `minclone()`?
7. Write and test `thr_create()`.
8. Write `thr_exit()`. Don't worry about reporting exit status, yet—it's tricky enough without that!
9. Test mutexes and condition variables.
10. Write and test `thr_join()`.
11. Worry about reporting the exit status.
12. This might be a good point to relax and have fun writing semaphores.
13. Test. Debug. Test. Debug. Test. Sleep once in a while.
14. Design and implement readers/writers locks.
15. Celebrate! You have created a robust and useful kernel supported user level thread library.