Project 2: User Level Thread Library

15-410 Operating Systems

September 26, 2004

1 Overview

An important aspect of operating system design is organizing computations 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. Pebbles is documented by the companion document, "Pebbles Kernel Specification," which should probably be read *before* this one.

The thread library will be based on the `thread_fork` system call provided by Pebbles, which provides a "raw" (unprocessed) interface to kernel-scheduled threads. Your library will provide a basic but usable interface on top of this elemental thread building block, 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 22nd Project 2 begins

Wednesday, September 29th You should have thread creation, mutexes, and condition variables working well.

Friday, October 8th Project 2 due at 23:59:59

4 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.

You need not ensure bounded waiting if your project documentation includes an appropriate explanation of why you believe it is likely to be true (or at least not flagrantly false) without effort on your part.

Unlike system call stubs (see "Pebbles Kernel Specification"), thread library routines need not be one-per-source-file, but we expect you to use good judgement when partitioning them (and this may influence your grade to some extent). You should arrange that the Makefile infrastructure you are given will build your library into `libthread.a` (see the `README` file in the tarball).

User programs will include `thread.h`, but will not directly include other header files you might write.

4.1 Thread Management API

- `int thr_init( unsigned int size )` - This function is responsible for initializing the thread library. The argument `size` specifies the amount of stack space which will be available for use by threads created with `thr_create()`.

  This function returns zero on success, and a negative number on error.

  The thread library can assume that programs using it are well-behaved in the sense that they will call `thr_init()`, exactly once, before calling any other thread library function (including memory allocation functions in the `malloc()` family, described below) or invoking the `thread_fork` system call. Also, you may assume that all threads of a task using your thread library will call `thr_exit()` instead of directly invoking the `exit()` system call.

- `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 `thread_fork` system call in an appropriate way. 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 `thread_fork` 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 belonging to the invoking thread's task 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 departing 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 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.

- `int thr_yield(int tid)` - Defers execution of the invoking thread to a later time in favor of the thread with ID `tid`. If `tid` is -1, yield to some unspecified thread. If the thread with ID `tid` is not runnable, or doesn't exist, then an integer error code less than zero is returned. Zero is returned on success.

Note that the "thread IDs" generated and accepted by your thread library routines (e.g., `thr_getid()`, `thr_join()`) are not required to be the same "thread IDs" which are generated and accepted by the thread-related system calls (e.g., `thread_fork`, `gettid()`, `make_runnable()`). If you think about how you would implement an "M:N" thread library, or a user-space thread library, you will see why these two name spaces cannot always be the same. Whether or not you use kernel-issued thread ID's as your thread library's thread ID's is a design decision you will need to consider.

4.2 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 `pthread_mutex_init()` manual page.

- `int mutex_init( mutex_t *mp )` - This function should initialize the mutex pointed to by mp. The effects of using a mutex before it has been initialized, or of initializing it when it is already initialized and in use, are undefined (and may be startling). This function returns zero on success, and a negative number on error.

- `int mutex_destroy( mutex_t *mp )` - This function should "deactivate" the mutex pointed to by `mp`. The effects of using a mutex from the time of its destruction until the time of a possible later re-initialization are 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 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 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 error.

4.3 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 awakened 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 refer to the Solaris or Linux documentation on `pthread_cond_wait()`.

- `int cond_init( cond_t *cv )` - This function should initialize the condition variable pointed to by `cv`. The effects of using a mutex before it has been initialized, or of initializing it when it is already initialized and in use, are undefined. This function returns zero on success and a number less than zero on error.
- `int cond_destroy( cond_t *cv )` - This function should "deactivate" the condition variable pointed to by `cv`. The effects of using a condition variable from the time of its destruction until the time of a possible later re-initialization are 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 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 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 error. Note that "no threads waiting" is *not* an error condition.
- `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 error.

  Note that `cond_broadcast()` should *not* awaken threads which may invoke `cond_wait(cv)` "after" this call to `cond_broadcast()` has begun execution.[1]

4.4 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.

[1] If that sounds a little fuzzy to you, you're right–but if you think about it a bit longer it should make sense.

- `int sem_init( sem_t *sem, int count )` - This function should initialize the semaphore pointed to by `sem` to the value `count`. 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 "deactivate" 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 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 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 error.

4.5 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 "deactivate" 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 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 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 error.

  Note: We *will not grade* your readers/writers implementation unless your thread library passes a specified series of tests; see Section 9.

4.6 Safety & Concurrency

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. However, by its nature a thread library must also be concurrent. In other words, you may *not* solve the thread-safety problem with a hammer, such as using a global lock to ensure that only one thread at a time can be running thread library code. In general, it should be possible for many threads to be running each library interface function "at the same time."

4.7 Distribution Files

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

5 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.

6 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 `410user/lib/libstdio.a`. The header file for these functions is `410user/lib/inc/stdio.h`, aka `#include<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 `410user/lib/libstdlib.a`. The header files for these functions, in `410user/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., do *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)`

You may assume that no calls to functions in the "malloc() family" will be made before the call to `thr_init()`.

These functions will typically seek to allocate memory regions from the kernel which start at the top of the data segment and proceed to grow upward. You will thus need to plan your use of the available address space with some care.

Some functions typically found in a C string library are provided by `410user/lib/libstring.a`. The header file for these functions is `410user/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)`

7 Debugging

The same `MAGIC_BREAK` macro which you used in Project 1 is also available to user code 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 `410user/lib/inc/stdio.h`.

Also, user code can be symbolically debuged using the Simics symbolic debugger. **If you restrict yourself to debugging with printf() it may cost you significant amounts of time.**

8 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.

9 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, code using the API provided by a correct project will not be killed by the kernel, and will not suffer from inconsistencies due 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 your code carefully. Do not forget to consider pathological cases.

The most important parts of the assignment to complete are the thread management, mutex, and condition variable calls. These should be well-designed, solidly implemented, and thoroughly tested with `misbehave()` (see below). It is probably unwise to devote substantial coding effort to the other parts of the library before the core is reliable. In particular, we **will not grade** readers/writers implementations for Project 2 submissions which do not pass the "hurdle" subset of the test suite (see the project web page for details).

10 Debugging

10.1 Requests for Help

Please do not ask for help from the course staff with a message like this:

> The kernel is killing my threads! Why?

or

> Why is my program stuck in `malloc()`?

An important part of this class is developing your debugging skills. In other words, when you complete this class you should be able to debug problems which you previously would not have been able to handle.

Thus, when faced with a problem, you need to invest some time in figuring out a way to characterize it and close in on it so you can observe it in the actual act of destruction. Your reflex when running into a strange new problem should be to start thinking, not to start off by asking for help.

Having said that, if a reasonable amount of time has been spent trying to solve a problem and no progress has been made, do not hesitate to ask a question. But please be prepared with a list of details and an explanation of what you have tried and ruled out so far.

10.2 Debugging Strategy

In general, when confronted by a mysterious problem, you should begin with a "story" of what you *expect* to be happening and measure the system you're debugging to see where its behavior diverges from your expectations.

To do this your story must be fairly detailed. For example, you should have a fairly good mental model of the assembly code generated from a given line of C code. To understand why "a variable has the wrong value" you need to know how the variable is initialized, where its value is stored at various times, and how it moves from one location to another. If you're confused about this, it is probably good for you to spend some time with `gcc -S`.

Once your "story" is fleshed out, you will need to measure the system at increasing levels of detail to determine the point of divergence. You will find yourself spending some time thinking about how to pin your code down to observe whether or not a particular misbehavior is happening. You may need to write some code to periodically test data-structure consistency, artificially cause a library routine to fail to observe how your main code responds, log actions taken by your code and write a log-analyzer perl script, etc.

Please note that the memory allocator we provide you with is very similar to the allocator written by 15-213 students in the sense that errors reported by the allocator, or program crashes which take place inside the allocator, are likely to mean that the user of some memory overflowed it and corrupted the allocator's meta-data.

11 Strategy

11.1 Suggestions

First, this may be the first time you have written code with this variety and density of concurrency hazards. If so, you will probably find this code much harder to debug than code you've written before, i.e., you should allocate more debugging time than usual. Of course, the silver lining in this cloud is that experience debugging concurrent code will probably be useful to you after you leave this class.

Second, *several* of the thread library functions are *much* harder then they first appear. It is fairly likely that you will write half the code for a thread library function before realizing that you've never written "that kind of code" before. When this happens the best course of action is probably to come to a complete stop, think your way through the problem, and then explain the problem and your proposed solution to your partner. It may also happen that as you write your fifth function you realize your second must be scrapped and re-written.

Third, the Pebbles kernel offers a feature intended to help you increase the solidity of your code. A special system call, `void misbehave( int mode )`, alters the behavior of the kernel in ways which may expose unwarranted assumptions or concurrency bugs in your library code. Values for `mode` range from zero (the default behavior) to thirty-one, or you may select -1 for behavior which may be particularly challenging. As you experiment with `misbehave()`, you may become able to predict or describe the behavior of a particular `mode`. Each group must keep confidential its own understanding of the meanings of particular `mode` values.

Fourth, we recommend *against* splitting the assignment into two parts, working separately until the penultimate day, and then meeting to "put the pieces together." Instead, we recommend the opposite, namely that you make it a habit to read *and talk about* each other's code every few days. **You may encounter an exam question related to code your partner wrote.**

Fifth, instead of typing linked-list traversal code 100 times throughout your library, thus firmly and eternally committing yourselves to a linear-time data structure, give some consideration to encapsulation.

Sixth, we strongly recommend that you use a source-control system to manage the evolution and/or devolution of your code. While the complexity of this project does not outright necessitate the use of source control, this is a good opportunity for you to get used to it and set up a work flow with your partner.

11.2 Steps

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 `thread_fork`).
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 thread 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 `thread_fork`?
7. Write and test `thr_init()` and `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. Try all the `misbehave()` flavors.
11. Write and test `thr_join()`.
12. Worry about reporting the exit status.
13. This might be a good point to relax and have fun writing semaphores.

14. Test. Debug. Test. Debug. Test. Sleep once in a while.
15. Try all the `misbehave()` flavors (again).
16. Design, implement, and test readers/writers locks.
17. Celebrate! You have created a robust and useful kernel-supported user level thread library.

11.3 Questions & Challenges

Below we briefly discuss common questions about this assignment and issue several optional challenges. It is very important that your implementation be solid, and you should not be diverted from this primary goal by attempting to solve these challenges. However, we are providing this challenge list as a way for interested students to deepen their understanding and sharpen their design skills.

11.3.1 Questions

From time to time we are asked how many threads must be supported by a library implementation. In general the answer is that the thread library should not be a limiting factor—it should be possible to use all available memory for threads, and of course it could happen one day that Pebbles would run on a machine with more memory. In the other direction, if you feel you must impose a static limit on the number of threads (or some other run-time feature), you should document your reasoning and we will attempt to take it into account.

It has been pointed out to us that, if a thread is killed by the kernel as a result of some improper behavior, your thread library has no way to find out about this and take remedial action. This is true, and thus the thread library cannot be held responsible for gross misbehavior on the part of threads it hosts. In fact, you can probably think of other sorts of errant behavior which your library can't reasonably protect against.

11.3.2 Challenge: efficient `thr_getid()`

There is an easy way to implement `thr_getid()`, but it is woefully inefficient. Can you do better? We have given you a serious hint.

11.3.3 Challenge: `thr_init()`

Is it really necessary that `thr_init()` be called before `malloc()`? How might you build `malloc()` to make that unnecessary? Is it really necessary to require the root thread of a task to explicitly call `thr_exit()`? Is there a way `thr_init()` can arrange for that call to happen automatically?

11.3.4 Challenge: "reaper thread"

If you feel you need a "reaper thread," consider whether it's *really* necessary.

11.3.5 Challenge: memory-efficient `thr_exit()`

Since there is no bound on how much time can pass between a thread exiting and its "parent" or "manager" thread calling `thr_join()`, it is undesirable for a "zombie thread" to hold onto large amounts of memory. Can you avoid this situation? There are multiple approaches, with different tradeoffs.