Project 2: User Level Thread Library

15-410 Operating Systems

September 21, 2011

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.

You will implement mutexes and condition variables based on your consideration of the options provided by the x86 instruction set—including, but not limited to, the `XCHG` instruction for atomically exchanging registers with memory or registers with registers. The lecture material discusses several atomic operations in more detail.

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.

The partner goal is an important one–this project gives you an opportunity to debug not only your code deliverables but also your relationship with your partner. You may find that some of the same techniques apply.

3 Important Dates

Wednesday, September 21st Project 2 begins

Monday, September 26th You should be able to draw a *very detailed* picture of the parent and child stacks during `thr_create()` at the point when the child does its first `PUSHL` instruction. In your design multiple pictures may be equally plausible, but it is important that you be able to draw at least one case in detail. It is wise for each partner to independently draw this picture before you compare notes and agree on all the details. It is not wise to skip this step (unless you have previously written a thread library).

Wednesday, September 28th You should have thread creation working well enough to pass the STARTLE test we provide. In particular, you should be able to run:

```
[410-shell]$ misbehave_wrap 0 startle
```

Friday, September 30th You should have thread creation, mutexes, and condition variables working well.

Tuesday, October 4th If you haven't least begun debugging CYCLONE and AGILITY_DRILL by this point, you run the risk of turning in a thread library with serious structural flaws and lacking concurrency debugging experience useful for the kernel project.

Friday, October 7th Project 2 due at 23:59:59

4 Thread Library API

The library you will write will contain:

- System-call stub routines (see the "Pebbles Kernel Specification" document)
- A software exception handler which implements automatic stack growth for legacy programs
- 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.

Unlike system call stubs (see the "Pebbles Kernel Specification" document), thread library routines need not be one-per-source-file, but we expect you to use good judgment 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).

You may assume that programs which use condition variables will include `cond.h`, programs which use semaphores will include `sem.h`, etc.

4.1 Return values

You will note that many of the thread-library primitives (e.g., `mutex_unlock()`) are declared as returning void. This is because there are some operations that can't meaningfully "return an error code." Consider what would happen if a program tried to invoke `exit()` and `exit()` "failed." What could the program do? Or consider the `free()` storage-allocator function. If a program called `free()` on an address, and `free()` said "No," what would that mean? Should the program continue using the memory area or not? Could it reasonably expect the next call to `malloc()` to work?

"Returning an error" is sensible when an operation might reasonably fail in plausible, non-emergency circumstances and where higher-level code can do something sensible to recover, or at least has a reasonable chance to explain to a higher authority what went wrong. If an operation *cannot* fail in reasonable circumstances (i.e., a failure means the computational state is irrevocably broken) and there is no reasonable way for higher-level code to do anything reasonable, other approaches are required, and void functions may be reasonable.

Note well that the author of a void function bears the responsibility of designing the implementation in such a way that the code fails *only* in "impossible" situations. This may require the author to design other parts of the code to take on extra responsibilites so the "must work reliably" functions are indeed reliable.

Some of the thread-library interface functions below are declared as void functions. In each case, you will need to list possible failure cases and think through them. The function will need to work in all "might reasonably happen" situations and do something reasonable if it discovers that the computation is irretrievably broken. You will generally need to consider and trade off the cost of checking for a particular bad situation against how bad it would be to leave the situation undetected. For more guidance, refer to the "Errors" lecture.

4.2 Automatic stack growth

If you examine the `_main()` "main wrapper" code in `crt0.c`, you will find that it receives two "extra" parameters from the kernel's program loader, called `stack_high` and `stack_low`, and passes them to a function called `install_autostack()` before `main()` runs. These two parameters are the highest and lowest virtual address of the initial stack space that the kernel created before launching the program (when `install_autostack()` is called, `%ESP` will naturally be some value between `stack_high` and `stack_low`). You are expected to register a `swexn()` exception handler of your own devising so that appropriate page faults result in automatic stack growth according to the venerable Unix tradition (automatic stack growth is discussed further in the "Pebbles Kernel Specification" document). Your handler is not expected to "correct" or otherwise handle any exceptions other than page faults, and is also not expected to "correct" page faults which are unrelated to automatic stack growth. We have provided a simple test program, `stack_test1`, which you can use to exercise your stack-growth handler.

4.3 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 each thread 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 `vanish()` system call (and that the root thread will call `thr_exit()` instead of `return()`'ing from `main()`).

Note that automatic stack growth is not a typical feature in multi-threaded environments. This means that you should give careful consideration to what should happen when various threads receive stack-related page faults after the first call to `thr_init()`.

- `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, void **statusp )` -

  This function "cleans up" after a thread, optionally returning the status information provided by the thread at the time of exit.

  The target thread `tid` may or may not have exited before `thr_join()` is called; if it has not, the calling thread will be suspended until the target thread does exit.

  If `statusp` is not NULL, the value passed to `thr_exit()` by the joined thread will be placed in the location referenced by `statusp`.

  Only one thread may join on any given target thread. Other attempts to join on the same thread should return an error promptly. If thread `tid` was not created before `thr_join(tid)` was called, 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 other than the root thread returns from its body function instead of calling `thr_exit()`, the behavior should be the same as if the function had called `thr_exit()` specifying the return value from the thread's body function.

  Note that `status` **is not a "pointer to a void."** It is frequently not a pointer to anything of any kind. Instead, `status` is a pointer-sized opaque data type which the thread library transports uninterpreted from the caller of `thr_exit()` to the caller of `thr_join()`.

- `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.

However, you **must not** aggressively recycle thread ID's, as this significantly reduces the utility of, e.g., `thr_yield()`.

4.4 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.
- `void mutex_destroy( mutex_t *mp )` - This function should "deactivate" the mutex pointed to by `mp`. It is illegal for an application to use a mutex after it has been destroyed (unless and until it is later re-initialized). It is illegal for an application to attempt to destroy a mutex while it is locked or threads are trying to acquire it.
- `void 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 an interfering critical section must not proceed until it is able to claim the lock.
- `void mutex_unlock( mutex_t *mp )` - Signals the end of a region of mutual exclusion. The calling thread gives up its claim to the lock. It is illegal for an application to unlock a mutex that is not locked.

For the purposes of this assignment, you may assume that a mutex should be unlocked only by the thread that most recently locked it.[1]

4.5 Condition Variables

Condition variables are used for waiting, for a while, for mutex-protected state to be modified by some other thread(s). 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, you may refer to the Solaris or Linux documentation on `pthread_cond_wait()`.

[1]Opinions differ, but you might want to wait until after the scheduling lecture(s) before solidifying yours.

- `int cond_init( cond_t *cv )` - This function should initialize the condition variable pointed to by `cv`. The effects of using a condition variable 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.

- `void cond_destroy( cond_t *cv )` - This function should "deactivate" the condition variable pointed to by `cv`.

  It is illegal for an application to use a condition variable after it has been destroyed (unless and until it is later re-initialized). It is illegal for an application to invoke `cond_destroy()` on a condition variable while threads are blocked waiting on it.

- `void 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()`. Upon return from `cond_wait()`, `*mp` has been re-acquired on behalf of the calling thread.

- `void cond_signal( cond_t *cv )` - This function should wake up a thread waiting on the condition variable pointed to by `cv`, if one exists.

- `void cond_broadcast( cond_t *cv )` - This function should wake up all threads waiting on the condition variable pointed to by `cv`.

  Note that `cond_broadcast()` should *not* awaken threads which may invoke `cond_wait(cv)` "after" this call to `cond_broadcast()` has begun execution.[2]

4.6 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, 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.

- `void 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.

  It is illegal for an application to use a semaphore after it has been destroyed (unless and until it is later re-initialized). It is illegal for an application to invoke `sem_destroy()` on a semaphore while threads are waiting on it.

- `void 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.

- `void 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.

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

4.7 Readers/writers locks

Readers/writers locks allow multiple threads to have “read” access to some object simultaneously. They enforce the requirement that if any thread has “write” access to an object, no other thread may have either kind of access (“read” or “write”) to the object at the same time. These types of locking behaviors are often called “shared” (for readers) and “exclusive” (for writers) locks. Refer to Section 7.5.2 of the textbook for details.

The generic version of this problem is called the “readers/writers problem.” Two standard formulations of the readers/writers problem exist, called unimaginatively the “first” and “second” readers/writers problems. In the “first” readers/writers problem, no reader will be forced to wait unless a writer has already obtained an exclusive lock. In the “second” readers/writers problem, no new reader can acquire a shared lock if a writer is waiting. You should think through the reasons that these formulations allow starvation of different access types; starvation of writers in the case of the “first” readers/writers problem and starvation of readers in the case of the “second” readers/writers problem.

In addition to a correct implementation of shared and exclusive locking, we expect you to implement a solution that is “at least of good as” a solution to the “second” readers/writers problem. That is, your solution should not allow starvation of writers. Your solution need not strictly follow either of the above formulations: it is possible to build a solution which does not starve any client. No matter what you choose to implement, you should explain what, how, and why.

You may choose which underlying primitives (e.g., mutex/cvar or semaphore) you use to implement readers/writers locks. Once again, you should explain the reasoning behind your choice.

- `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.

- `void rwlock_destroy( rwlock_t *rwlock )` - This function should “deactivate” the lock pointed to by `rwlock`.

  It is illegal for an application to use a readers/writers lock after it has been destroyed (unless and until it is later re-initialized). It is illegal for an application to invoke `rwlock_destroy()` on a lock while the lock is held or while threads are waiting on it.

- `void 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.

- `void 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 situation and the policy you chose to implement.

  It is illegal for an application to unlock a readers/writers lock that is not locked.

- `void rwlock_downgrade( rwlock_t *rwlock )` - This function must be called by a thread that already owns the lock in `RWLOCK_WRITE` mode at a time when it no longer requires exclusive access to the protected resource. When the function returns: no threads hold the lock in `RWLOCK_WRITE` mode; the invoking thread, and possibly some other threads, hold the lock in `RWLOCK_READ` mode; previously blocked or newly arriving writers must still wait for the lock to be released entirely.

During the transition from `RWLOCK_WRITE` mode to `RWLOCK_READ` mode the lock should at no time be unlocked. This call should not block indefinitely.[3]

Note: We *will not grade* your readers/writers implementation unless your thread library passes a specified series of tests; see Section 10.

4.8 Safety & Concurrency

Please keep in mind that much of the code for this project must 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.”

As you design your library, your model should be that some system calls “take a while to run.” You should try to avoid situations where “too many” threads are waiting “too long” because of this. This paragraph provides a design hint, not implementation rules: acting on it will require you to think about system calls and the meanings of “too many” and “too long.”

4.9 Distribution Files

To begin working on the project, fetch and unpack the tarball posted on the course web page. Please read the README included therein.

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 Thread Group Library

A commonly used program paradigm involves one or more manager threads overseeing the completion of a large task which has been split into parts assigned to a pool of worker threads. Examples of this model include databases, Apache, and Firefox. Once a worker thread has completed its job, it exits; manager threads dispatch new worker threads based on system load, new requests, and the results obtained by previous worker threads. In this environment it is not convenient for a manager to know which particular worker thread it should next call `thr_join()` on; instead it is convenient to wait until the next thread in the worker pool completes.

We have provided you with a simple library implementing “thread groups.” This library essentially provides an abstraction layer above the thread library you will write–a compliant program will use `thrgrp_create()` and `thrgrp_join()` instead of calling `thr_create()` and `thr_join()` directly.

These functions and their requisite data structures are defined in 410user/libthrgrp/thrgrp.c and 410user/libthrgrp/thrgrp.h.

[3] We do not ask you to implement this function’s partner, “`rwlock_upgrade()`”—and for good reason! See if you can figure out why.

- `thrgrp_group_t`
  A structure representing a thread group.

- `thrgrp_init_group(thrgrp_group_t *tg)`
  This function initializes a thread group. It must be called before the thread group is used for anything. Returns 0 on success, non-zero otherwise.

- `thrgrp_destroy_group(thrgrp_group_t *tg)`
  This function destroys a thread group, cleaning up all of its memory. This should be called if a thread group isn't to be used further. The effects of using a thread group after it has been destroyed are be undefined. Returns 0 on success, non-zero otherwise.

- `thrgrp_create(thrgrp_group_t *tg, void *(*func)(void *), void *arg)`
  This function spawns a new thread (analogous to `thr_create()`) in the threadgroup `tg`. The spawned thread must not call `thr_exit()`. Instead, `func()` should return an exit code (of type `void *`) which will be made available to a manager thread.

  Returns 0 on success, non-zero otherwise.

- `thrgrp_join(thrgrp_group_t *tg, void **statusp)`
  If there are any unreaped threads in the thread group `tg` then it will reap one of them, setting `*statusp` appropriately, and return. If there are no unreaped threads in the group, it will block until one does exit, reap it, and return.

7 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/libstdio.a`. The header file for these functions is `410user/libstdio/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/libstdlib.a`. The header files for these functions 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/libstring.a`. The header file for these functions is `410user/libstring/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)`

8 Debugging Support Code

The same `MAGIC_BREAK` macro which you used in Project 1 is also available to user code in Project 2 if you `#include` the `410user/libsimics/simics.h` header file.

The function call `lprintf()` may be used to output debugging messages from user programs. Its prototype is in `410user/libsimics/simics.h`.

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

9 Deliverables

Implement the functions for automatic stack growth, 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. **Any use of, or reliance on, outside code must be in accordance with course policy as stated in the syllabus.**

10 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).

Because a thread library is designed to support the activities of multiple threads, concurrency (the ability to get more than one thing done at a time) is important. As you do your design, ask yourself which things your thread library can accomplish in parallel. You may tune your code, especially your locking code, based on the assumption that it will run on a uni-processor machine. However, if we were to run your kernel on a multi-processor machine instead, the core of the thread library shouldn’t artificially limit concurrency–if there were four processors, threads running on top of your thread library should be able to use all four productively most of the time. Another way to think about this is that on a uni-processor machine the timer and the kernel scheduler determine when thread execution is interleaved; if the structure of your thread library results in much less concurrency (interleaving) than the kernel provides, something isn’t right.

Finally, code that is robust doesn’t randomly refuse to perform its job. It is not really robust for `mutex_lock()` to refuse to lock something because it can’t allocate memory, and it is *downright* *unreasonable* for `cond_wait()` to refuse to block a thread because of a memory-allocation problem: what's the caller supposed to do—keep running? These and similar operations should do their jobs in a prompt and reliable manner.

11 Debugging

11.1 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.

11.2 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.

Don't forget about the debugger. In particular, any time you find yourself "stuck," please review the course web site's page listing useful debugger commands. If there are many which you aren't using, that probably represents a mistake.

Please note that the user-space 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. In the other direction, complaints by "lmm" are coming from the kernel's memory allocator, and probably indicate kernel bugs (see below).

11.3 Reference Kernel Panics and Crashes

If the Pebbles kernel tells you something went horribly wrong and drops you into the debugger, don't panic. It probably won't happen to most of you, but we are fully aware that we haven't nailed the last bug yet...

It's probably a good idea for you to tar up your working directory and make a brief note of what you were doing when the kernel ran into trouble. For example, what sequence of test programs had you run since boot? If you have a short repeatable way of getting the kernel to die, that's excellent, and we'd appreciate a snapshot that lets us reproduce it, even if you then go on to modify your code to make the crash go away.

To send us a snapshot, tar it up somewhere in your group's scratch directory,

```
tar cfz .../mygroup/scratch/kcrash.somename.tgz .
```

create a brief summary of how to reproduce it,

```
$EDITOR .../mygroup/scratch/kcrash.somename.README
```

and send a brief note to the staff mailing list. While such an event will of course attract our attention, it's not likely that we can provide a fix in a small number of minutes...you may need to try to guess what went wrong and work around it temporarily, or work on some other part of your project for a while.

12 Strategy

12.1 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, we have observed that a particularly bad division of labor is for one person to write system call stubs, linked lists, queues, and maybe semaphores, while the other person writes everything else. This puts the first person at risk of doing poorly on exams.

Sixth, 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.

Seventh, 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.

Eighth, don't forget to do an update when `make` starts beeping at you. If you're in the middle of debugging a problem, you probably don't want to switch kernels, but you generally *do* want to upgrade when we issue new things, because we do so to help. A particularly bad thing to do is to work on your thread library for two weeks using the very oldest kernel and then 15 minutes before the assignment deadline switch to the very newest one and find that one time in a thousand you call `new_pages()` in an improper way which got through before and doesn't any more. So don't do that. The update process gives you the power to decide when to import changes, but that means the responsibility lies with you as well.

Ninth, if you find yourself confused about what is meant when the kernel specification or the thread-library handout says "before," try to imagine what "before" might mean on a multi-processor machine.

Finally, we have observed that the single most effective decision a group can make is to schedule standing "work meetings" of one or two hours duration two or three times per week. It is important that these be at **fixed times** each week agreed upon **in advance**. Groups that do this consistently do better on the thread library and kernel projects than groups who don't.

12.2 Suggested Steps

1. Read the handout.

2. Agree on two to three meeting times per week. An excellent thing to discuss early on is what source control system to use. By the way, make sure you configure it to *not* track changes to large random files such as `bootfd.img`, `bootfd.gz`, `user_apps.S`, the contents of `temp/`, etc., or your disk quota will be consumed very quickly.

3. Be sure to review the syllabus material on collaboration and the use of outside code.

4. **Promptly** 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. Good system calls to begin with are `set_status()` and `vanish()`, since the C run-time start-up code invokes the `exit()` library routine, which depends on them. A good second step would be `print()`.

5. Write the remaining system call wrappers (with the exception of `thread_fork`).

6. If you're using revision control, make sure your repositories are private (readable by only you, your partner, and optionally the members of the course staff). Be careful: some popular project-hosting web sites *require* all repositories to be public, and they will publish your code on various search engines even if you do not.

7. If you're not using revision control, you should be.

8. Read at least half of the test code we have provided. Doing this early can avoid potentially-costly last-minute discoveries of misunderstandings.

9. 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 code sequences in your thread library would suffer if the scheduler switched from executing one of your threads to another.

10. Now would not be a bad time to read the source to the "thread group" library (Section 6). If you read the source code we provide *before* "debugging time," it may help you do a better design, and thus need to do less debugging. Now would also probably be a good time to read the textbook material on the "Producer-Consumer" (aka "Bounded-Buffer") pattern.

11. If you haven't yet, agree on two to three meeting times per week.

12. What can you test at this point? Be creative.

13. Think hard about stacks. What should the child's stack look like before and after a `thread_fork`? In fact, it is probably a good idea for you to draw every detail of the parent's stack and the child's stack before and after `thread_fork`. You should reach this point by Monday, September 26th .

14. Write and test `thr_init()` and `thr_create()`. Run the STARTLE test. You should reach this point by Wednesday, September 28th .

15. Write `thr_exit()`. Don't worry about reporting exit status, yet—it's tricky enough without that!

16. Test mutexes and condition variables. Try to reach this point by Friday, September 30th .

17. Try all the `misbehave()` flavors.

18. Write and test `thr_join()`.

19. Worry about reporting the exit status.

20. Write a basic software exception handler which implements automatic stack growth for legacy single-threaded applications. This should not be a lot of code, and you can revisit your implementation later if you wish.

21. This might be a good point to relax and have fun writing semaphores.

22. Test. Debug. Test. Debug. Test. Sleep once in a while.

23. Try all the `misbehave()` flavors (again). Note that most of the tests provided to you by the course staff (see `410user/progs/README`) are really multiple tests if you think about it... you probably shouldn't declare a test "passed" until *all* versions pass. Remember that you should be running CYCLONE and AGILITY_DRILL by Tuesday, October 4th .

24. Design, implement, and test readers/writers locks.

25. Celebrate! You have assembled a collection of raw system calls into a robust and useful thread library.

12.3 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.

12.3.1 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. If, however, you feel you *must* impose an a-priori static limit on the number of threads (or some other run-time feature), we will grade less harshly if you document your reasoning.

Sometimes we are asked to state a simple requirement about bounded waiting (e.g., “Are we required to implement the bounded waiting algorithm presented in the lecture slides?”). Since this is a design class, you should give serious consideration to the issue of bounded waiting and the interplay between bounded waiting and the system environment you will be using. Then you should be in a position to evaluate the necessity of ensuring or approximating bounded waiting and how you might go about doing that. Whatever you choose to do should sensibly balance cost against utility. Your project documentation should briefly but convincingly explain your reasoning.

What should happen if a thread is killed by an exception? Solving this problem in the general case is much too difficult for this assignment, but it probably would be a good idea to think about whether there is anything reasonable you could do with “a bit” of code and, if so, to try to do it.

12.3.2 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.

12.3.3 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? Hint: not all approaches which appear to be solutions to this challenge actually are.

12.3.4 Challenge: “reaper thread”

If you feel you need a “reaper thread,” consider whether it’s *really* necessary.

12.3.5 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.

12.3.6 Challenge: fully functional autostack

Implementing a fully functional autostack that works even in the face of multithreading is an interesting and potentially challenging problem. Can you achieve this?

12.3.7 List Traversal Macros

You may find yourself wishing for a way for a PCB to be on multiple lists at the same time but not relish the thought of writing several essentially identical list traversal routines. Other languages have generic-package facilities, but C does not. However, it is possible to employ the C preprocessor to automatically generate a family of similar functions. If you wish to pursue this approach, you will find a template available in `vq_challenge/variable_queue.h`.