User-space Synchronization and Thread Library

CMPSC473: Operating Systems

Project Deadline: April 15, 2018 11:59 PM

The primary TA for this assignment is Apurva Bhogale. Contact via Canvas please.

Note: DO NOT SHARE ANY PART OF YOUR SOLUTION. DOING SO IS A VIOLATION OF ACADEMIC INTEGRITY.

Both during and after the course, your code should not leave your possession. This includes, but is not limited to, any online repositories (github, bitbucket, etc.), and emails to friends or future employers. This policy applies to all assignments in this course.

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 be provided a thread library, and write concurrency primitives. This document provides the background information and specification for writing the 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.pdf`, which should be read concurrently with this one. Pebbles is a 32-bit OS, so all parts of this project are to be performed using 32-bit conventions.

See Section 5 for a suggested order of completing the assignment.

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 knowledge of system calls, atomicitiy, threads, and synchronization.
- Developing skills necessary to produce a substantial amount of code, such as organization and project planning.

3 Deliverables

In this project, you will be responsible for implementing the following pieces of code that interact with each other. **Additionally, for this project, you must describe your design and your design decisions involved with your implementation at the top of each .c file. Your grade will partially depend on these comments and whether they indicate a good design and accurately reflect your code.**

- Section 6 (4% of your time): You will be creating syscall wrappers for all the Pebbles system calls. This will be accomplished by writing small amounts of 32-bit assembly. You can place the code in `user/libsyscall/syscall.S`. Any other syscall-related header information can be placed in `user/inc/syscall_ext.h`.

- Section 7 (5% of your time): You will be creating a small library of atomic operations. This will also be accomplished by writing small amounts of 32-bit assembly. You can place the code in `user/libatomic/atomic.S`.

- Section 8 (45% of your time): You will be implementing a set of synchronization primitives including mutexes, condition variables, semaphores, and readers/writers locks. This will be based on a combination of the syscalls and atomic operations. Each of these synchronization primitives will contain some internal data, which will be stored in its corresponding struct. You can fill in the struct definitions in `user/inc/{mutex,cond,sem,rwlock}_type.h`. You can place the code for the synchronization primitives in `user/libthread/{mutex,cond,sem,rwlock}.c`. There are multiple ways to build these synchronization primitives, and it is up to you to think about the various design tradeoffs.

- Section 9 (1% of your time) You will be implementing a thread-safe version of malloc. Fortunately for you, you are provided a simple non-thread-safe version of malloc. So you should only be creating small (few lines of code) wrapper functions that make malloc (and it's associated functions) thread-safe. You can place these wrapper functions in `user/libthread/malloc.c`.

- Section 10 (45% of your time) You will be implementing a thread library that creates and manages threads. This will be based on the `thread_fork` system call provided by Pebbles, which provides a "raw" (unprocessed) interface to kernel-scheduled threads. The library will provide a basic but usable interface on top of this elemental thread building block, including the ability to join threads. You can place the code for the thread functions in `user/libthread/thread.c`. Any information you'd like to store in a header file can be placed in `user/libthread/thr_internals.h`.

As you may have noticed, all of your code changes should reside within the `user` directory. Do not make any changes outside this directory; they will not be submitted and will be ignored when grading. See Section 13 for handin instructions.

4 Directory Structure

The structure of the project handout is as follows:

- `user` contains all the code you will be writing for this assignment. See Section 3 for details.

- `410kern` is used for components internal to the project infrastructure and can be safely ignored.

- `410user` is used for libraries, test cases, and other various components in the project infrastructure. In particular, it may be worthwhile checking out the following two directories.

- `410user/inc` contains the header files for various libraries that you may want to include. For example, you can include atomic.h using `#include <atomic.h>`.
- `410user/progs` contains some test cases that you can use to test your program. You are welcome to write your own test cases in `user/progs` and build them by adding them to `user/config.mk`, but this is not required.
- `spec` contains the header files for the Pebbles kernel specification.

5 Strategy

Here is a suggested order of completing the assignment:

1. Read the handout.
2. Be sure to review the academic integrity policy. In summary, all work (e.g., programming assignments, quizzes, exams, etc.) is to be done individually and independently. No outside help of any form is permitted.
3. **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()`.
4. Write the remaining system call wrappers (with the exception of `thread_fork`, of course).
5. Implement the functions in the atomic library. By implementing these, it may give you ideas for the later parts in the assignment.
6. 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.
7. Now would be a good time to write at least an initial version of your `malloc()` wrappers.
8. Write and test `thr_init()` and `thr_create()`. Run the STARTLE test. You should reach this point by roughly a third of the way through the assignment.
9. Write `thr_exit()`. Don't worry about reporting exit status, yet—it's tricky enough without that!
10. Test mutexes and condition variables. Try to reach this point by roughly halfway through the assignment.
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. Remember that you should be running CYCLONE and AGILITY_DRILL by roughly three quarters of the way through the assignment.

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

16. Go back and think through any parts of the assignment where it may make sense to improve the design.

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

6 Syscall Library

You will be implementing multiple system call wrappers. This will require understanding the 32-bit assembly calling conventions as well as how syscalls work. We will cover this in lecture shortly. For more details on how syscalls work with the Pebbles kernel, please review the `pebbles.pdf` document.

Any system calls you are implementing can be placed in the `user/libsyscall/syscall.S` file. You should implement all of the system calls in the syscall library, though you may not end up using all of them for the other parts of the assignment. In particular, swexn is a confusing syscall, but you won't need it for this project – you'll only be implementing the syscall wrapper for it. You are welcome to add other syscall-related functions as needed, and the corresponding header information can be placed in `user/inc/syscall_ext.h`.

7 Atomic Library

You will be implementing several atomic functions in the `user/libatomic/atomic.S` file. This will require understanding the 32-bit assembly calling conventions as well as a few special assembly instructions. The x86 assembly instruction documentation can be found in the Intel Instruction Set Reference. While you are welcome to peruse the 2,000 page document at your leisure, it may be helpful to focus on the `xchg`, `cmpxchg`, and `xadd` instructions.

You should implement all of the functions in the atomic library (listed below), though you may not end up using all of them for the other parts of the assignment. The purpose of the snippets of C code in the descriptions is to precisely explain the behavior of these functions. Note, however, that you should not be using any synchronization primitives for making these atomic. The purpose of these functions is to use the abilities of the CPU to make them atomic. So if you have any assembly that resembles the snippets of C code, then you're probably doing something wrong.

- `int atomic_exchange(volatile int* obj, int desired)` - This function atomically swaps the contents of a memory location (`obj`) with a given value (`desired`) and returns the old value. This function is equivalent to performing the following operation atomically:

```
old_val = *obj;
*obj = desired;
```

```
return old_val;
```

- `int atomic_compare_swap(volatile int* obj, int expected, int desired)` - This function atomically compare the contents of a memory location (`obj`) with a given value (`expected`) and only if they are the same should it modify the contents of the given address with the new value (`desired`). It always returns the old value. This function is equivalent to performing the following operation atomically:

```
old_val = *obj;
if (old_val == expected) {
  *obj = desired;
}
return old_val;
```

- `int atomic_fetch_add(volatile int* obj, int arg)` - This function atomically increments the contents of a memory location (`obj`) by a specified value (`arg`) and returns the old value. This function is equivalent to performing the following operation atomically:

```
old_val = *obj;
*obj += arg;
return old_val;
```

- `int atomic_fetch_sub(volatile int* obj, int arg)` - This function atomically subtracts the contents of a memory location (`obj`) by a specified value (`arg`) and returns the old value. This function is equivalent to performing the following operation atomically:

```
old_val = *obj;
*obj -= arg;
return old_val;
```

- `int atomic_load(volatile int* obj)` - This function atomically loads and returns the value of a memory location (`obj`).
- `void atomic_store(volatile int* obj, int desired)` - This function atomically stores a value (`desired`) into a memory location (`obj`).

8 Synchronization Primitives

You will be implementing mutexes, condition variables, semaphores, and readers/writers locks for this part of the assignment. There are many ways of implementing these synchronization primitives. You are encouraged to explore various designs and come up with a design that is both correct and reasonable. There are varying degrees of reasonableness, and you need to think about the pros and cons of your approach. You should document your design and design considerations (e.g., pros/cons) at the top of each of your .c files, as you will be graded on your design for this assignment.

8.1 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?

"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 responsibilities so the "must work reliably" functions are indeed reliable.

Note further that a void return type is a contractual specification that when the function returns the documented action will have been completed successfully. Said another way, if some circumstance prevents the function from acting as specified, it cannot return.

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.

8.2 Mutexes

Mutual exclusion locks prevent multiple threads from simultaneously executing critical sections of code. To implement the most basic form of mutexes, you may use the `atomic_exchange` function.

- `int mutex_init( mutex_t* mutex )` - This function should initialize the mutex pointed to by `mutex`. It is illegal for an application to use a mutex before it has been initialized or to initialize one when it is already initialized and in use. Mutexes are assumed to be in an "unlocked" state upon initialization. This function returns zero on success, and a negative number on error.

- `void mutex_destroy( mutex_t* mutex )` - This function should "deactivate" the mutex pointed to by `mutex`. 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* mutex )` - 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* mutex )` - 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.

8.3 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()`.

- `int cond_init( cond_t* cond )` - This function should initialize the condition variable pointed to by `cond`. It is illegal for an application to use a condition variable before it has been initialized or to initialize one when it is already initialized and in use. This function returns zero on success, and a negative number on error.

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

  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* cond, mutex_t* mutex )` - 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()`, `*mutex` has been re-acquired on behalf of the calling thread.

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

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

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

[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.

8.4 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, 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.

8.5 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.

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 as 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* rw )` - This function should initialize the lock pointed to by `rw`. 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* rw )` - This function should "deactivate" the lock pointed to by `rw`.

  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* rw, enum rwlock_type 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* rw )` - 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* rw )` - A thread may call this function only if it already holds 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.[2]

Note: as readers/writers locks are a "classic problem" (and widely used in systems-related code), the Internet is full of solutions (good and bad) to various versions of the problem. Please recall that this is a design class, not a copy-and-paste class. We believe it is feasible and very educational for you to design readers/writers locks yourself, "from scratch." So in accordance with the academic integrity policy for the course, please remember that you are not allowed to look online for any hints or solutions to any parts of the assignment. We will be actively checking for violations to this policy.

9 Thread-safe Malloc

You will be implementing *thread-safe versions* of malloc in `user/libthread/malloc.c` using the following *non-thread-safe versions* of the standard C library memory allocation routines.

- `void *_malloc(size_t size)`

- `void *_calloc(size_t nelt, size_t eltsize)`

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

- `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.

10 Thread Management API

- `int thr_init( unsigned int stack_size )` - This function is responsible for initializing the thread library. The argument `stack_size` specifies the amount of stack space which will be available for each thread using the thread library.

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

  The thread library assumes 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()`).

- `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 so that the indicated thread-body function is run appropriately. 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** status )` -

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

  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.

10.1 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."

11 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 in `410user/*` for a full listing of the provided functions. If a library is not provided, you are not allowed to use it.

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, ...)`

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 assert(int expression)`

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)`

12 Testing Your Code

When you go to test your code incrementally, compile your project using `make` in the top-level directory. We have provided some test cases for you to run your code with in the `410user/progs` directory. Note that these are not your typical executables and will not natively run in Linux (as the project is designed to run on the Pebbles OS). Thus, we have provided a special emulator, `peb_exec`, that will emulate the Pebbles OS in a Linux environment.

To use `peb_exec`, go into the `410user/progs` directory and run:

```
./peb_exec program_name
```

For example, you can run the startle test with:

```
./peb_exec startle
```

Please refer to the test code to see what the expected behavior should be. Note that sometimes the test results will be written to the kernel.log file, so be sure to take a look at this file after running a test. Also, make sure you recompile your code when you want to test your code changes.

To get things working with gdb, you have to run peb_exec in the following special way:

```
./peb_exec --gdb program_name
```

There's some special gdb magic to make it appear as if you're natively running your test program, but note that in reality this is an emulator, so it's doing some unusual things such as catching and handling certain segfaults. Most things should work as if you were running a native Linux program, but just be aware that there may be some peculiarities when using gdb.

12.1 Test Case Descriptions

The test cases are testing a variety of functionalities. Their descriptions are below. When you read through them, you can see that some are much easier to get running (as they test less) than others. Something like racer is evidently quite complicated so if that doesn't work try to get something less complicated working.

`actual_wait`: Exercises wait() and vanish().

`atomic_test`: Simple unit test for the atomic functions.

`agility_drill`: Aggressively tests acquire/release of lots of mutexes with the occasional sleep or yield call in the middle.

`agility_drill_sem`: A stress test for semaphores that is identical to the agility_drill mutex stress test with binary semaphores instead of mutexes.

`beady_test`: Comprehensive, practical test of the core of the thread library package. Creates a game in which you attempt to keep a small cursor on a bead by using the , and . keys to move the cursor left and right. The test spawns threads, then uses condition variables and mutexes to maintain shared world state. Also cleans up all threads when 'q' is pressed to quit the game. If you can run this test without problems, you are well on your way to completing the project.

`bg`: Tests running a program in the background.

`broadcast_test`: Very basic unit test for condition variable broadcast.

`cat`: Simple program to print out files.

`cvar_test`: Test condition variable functionality.

`cyclone`: Spawns a thread and then attempts to join it.

`excellent`: A multi-threaded application that sometimes experiences a thread crash.

`getpid_test1`: Tests getid().

`halt_test`: Tests halt().

`join_specific_test`: A (relatively stressful) test of the thread library's ability to join on specific threads.

`juggle`: "Juggles" threads and keeps adding to the number being juggled in a safe way.

`largetest`: This test will keep spawning threads until something "bad" happens and it exits. Usually, this occurs somewhere in the range of hundreds to thousands of threads depending on the implementation.

`mandelbrot`: Creates a set of threads that wander the console drawing a mandelbrot fractal with colored letters. The letters count the number of times a pixel has been visited by any thread. Some pixels are thread traps where threads wait on a condition variable until a thread count threshold is reached and all threads are released (signaled). Each trap displays the number of waiting threads. Tests both mutexes and condition variables. The number of wanderer threads, number of traps, and the thread wakeup threshold are tunable as command line arguments. This test runs until the user enters "q".

`multitest`: This program keeps one copy of several thread based programs that are running at once and is testing the thread implementation obviously.

`mutex_destroy_test`: Testing mutex_destroy(). Look at the test and think about what the behavior should be in such a scenario.

`mutex_test`: Very basic unit test for mutexes.

`nibbles`: Clone of the snake game that tests various syscalls.

`paradise_lost`: A test to try to expose a specific semaphore bug.

`paraguay`: Tests that condition variables work properly even if signaled without the associated lock being held.

`racer`: A strenuous test of mutexes, condition variables, and semaphores. This creates <threads>number of threads. Each thread gets 1 line of the terminal. Every thread attempts to aquire the semaphore which was initialized with a value of <semaphores>. When a thread aquires the sem it prints out <runlength>number of characters, one at a time. There is an additional <pausetime>that can be used to cause each thread to call sleep of that value. Finally there is a "bad" thread that cycles misbehave states and then sleeps for <misbehavetime>. The expected default result is seeing a group of 12 threads run out for 5 characters. Then the next group of 12 threades will run out for 5 characters. The bottom-right corner shows the current number of threads that are active (should always be $\leq$ 12) and the current misbehave state. threads and semaphores should be set to 24 and 12 for best effect runlength should be set to 5-10 to be able to see how many are active, or set to 1 for the most strenuous. Colors are pseudo-randomly rotated by continuously adding get_ticks() to the color value. No real point other than to be trippy.

`rwlock_downgrade_read_test`: Tests downgrading a writers lock to a readers lock.

`semaphore_test`: A basic unit test for semaphores.

`startle`: Simple test of thread creation.

`syscall_test`: Simple test for each of the syscalls. The first parameter indicates which test to run. See the bottom of the test code to see the test options.

`thr_exit_join`: Tests thr_exit() and thr_join() with the child joining the parent.

`thr_join_exit`: Similar to thr_exit_join, but has the parent joining the child.

13 Handin Instructions

Ensure that all your code compiles and is stored somewhere within the `user` directory hierarchy. Then to handin your code, run the following command:

```
make handin
```

You will see a handin.tar.gz file created. Submit this file in Canvas.

14 Grading Criteria

This assignment is worth 20% of the course grade, with roughly half of the weight for the synchronization primitives and roughly the other half of the weight for the thread creation/management functions. The atomic functions, system calls, and malloc wrapper are also going to be a part of your grade, but these will account for a tiny part as they are significantly easier than the two main parts of the assignment.

You will be graded on the completeness, correctness, and design of your project. A complete project is composed of a reasonable attempt at each function in the API. Also, a complete project will properly build 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. Using the provided tests will help you identify bugs in your code, but it is not an exhaustive test suite, and our final grading will potentially involve additional testing for correctness. 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. **Lastly, you must describe your design and your design decisions involved with your implementation. These descriptions should go in comments at the top of each of your .c files. We will be grading your solution based on these comments and whether they indicate a good design and accurately reflect your code.**

It is probably unwise to devote substantial coding effort to the other parts of the library before the core is reliable (mutex, cond var, thread init/create). In particular, we strongly recommend that you attempt readers/writers locks after gaining confidence with concurrent programming and achieving a stable basic thread library.

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. Consider how someone could build a program using your building blocks and what they would think of how your code handles some of these exceptional cases.

15 Debugging

15.1 Gather Information; Think; Repeat

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. The course staff is here to help you, but the ultimate goal is more than just doing well in a course, but rather to provide the experience necessary to solve challenging problems in your career. You will miss out on this experience with too much hand-holding, so start early and use a debugger. It's hard to survive without a debugger, and by now you should have no excuses for not being "good" at using a debugger. You only become good by using it and gaining the practice.

15.2 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 python script, etc.

Don't forget about the debugger. In particular, any time you find yourself "stuck," please review the gdb cheat sheat on Canvas for useful debugger commands.

Please note that the user-space memory allocator we provide you with is similar to the one you wrote for the malloc assignment 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).