CSCI 432: Operating Systems

Project 1: Threads and Monitors

Overview

This project will give you experience writing simple multi-threaded programs using monitors and will help you understand how threads and monitors are implemented.

The project has two parts. In the first part, you will write a simple concurrent program that schedules disk requests. This concurrent program will use a thread library that I provide. In the second part, you will implement your own version of this thread library.

1. Thread library Interface

This section describes the interface to the thread library for this project. You will write a multi-threaded program that uses this interface, and you will also write your own implementation of this interface.

```
int thread_libinit(thread_startfunc_t func, void *arg)

    thread_libinit initializes the thread library.  A user program should call
    thread_libinit exactly once (before calling any other thread functions).
    thread_libinit creates and runs the first thread.  This first thread is
    initialized to call the function pointed to by func with the single
    argument arg.  Note that a successful call to thread_libinit will not
    return to the calling function.  Instead, control transfers to func, and
    the function that calls thread_libinit will never execute again.

int thread_create(thread_startfunc_t func, void *arg)

    thread_create is used to create a new thread.  When the newly created
    thread starts, it will call the function pointed to by func and pass it the
    single argument arg.

int thread_lock(unsigned int lock)
int thread_unlock(unsigned int lock)
int thread_wait(unsigned int lock, unsigned int cond)
int thread_signal(unsigned int lock, unsigned int cond)
int thread_broadcast(unsigned int lock, unsigned int cond)

    thread_lock, thread_unlock, thread_wait, thread_signal, and
    thread_broadcast implement Mesa monitors in your thread library.  We
    covered the semantics of Mesa monitors in lecture.

    A lock is identified by an unsigned integer (0 - 0xffffffff).  Each lock
    has a set of condition variables associated with it (numbered 0 -
    0xffffffff), so a condition variable is identified uniquely by the tuple
    (lock number, cond number).  Programs can use arbitrary numbers for locks
    and condition variables (i.e., they need not be numbered from 0 - n).
```

```
int thread_yield(void)

    thread_yield causes the current thread to yield the CPU to the next
    runnable thread.  It has no effect if there are no other runnable threads.
    thread_yield is used to test the thread library.  A normal concurrent
    program should not depend on thread_yield; nor should a normal concurrent
    program produce incorrect answers if thread_yield calls are inserted
    arbitrarily.
```

Each of these functions returns -1 on failure. Each of these functions returns 0 on success, except for thread_libinit, which does not return at all on success.

Here is the file "thread.h". DO NOT MODIFY OR RENAME IT. thread.h will be included by programs that use the thread library, and should also be included by your library implementation.

```
/*
 * thread.h -- public interface to thread library
 *
 * This file should be included in both the thread library and application
 * programs that use the thread library.
 */
#ifndef _THREAD_H
#define _THREAD_H

#define STACK_SIZE 262144 /* size of each thread's stack */

typedef void (*thread_startfunc_t) (void *);

extern int thread_libinit(thread_startfunc_t func, void *arg);
extern int thread_create(thread_startfunc_t func, void *arg);
extern int thread_yield(void);
extern int thread_lock(unsigned int lock);
extern int thread_unlock(unsigned int lock);
extern int thread_wait(unsigned int lock, unsigned int cond);
extern int thread_signal(unsigned int lock, unsigned int cond);
extern int thread_broadcast(unsigned int lock, unsigned int cond);

/*
 * start_preemptions() can be used in testing to configure the generation
 * of interrupts (which in turn lead to preemptions).
 *
 * The sync and async parameters allow several styles of preemptions:
 *
 *     1. async = true: generate asynchronous preemptions every 10 ms using
 *        SIGALRM.  These are non-deterministic.
 *
 *     2. sync = true: generate synchronous, pseudo-random preemptions before
 *        interrupt_disable and after interrupt_enable.  You can generate
 *        different (but deterministic) preemption patterns by changing
 *        random_seed.
 *
 * start_preemptions() should be called (at most once) in the application
 * function started by thread_libinit().  Make sure this is after the thread
 * system is done being initialized.
```

```
 *
 * If start_preemptions() is not called, no interrupts will be generated.
 *
 * The code for start_preemptions is in interrupt.cc, but the declaration
 * is in thread.h because it's part of the public thread interface.
 */
extern void start_preemptions(bool async, bool sync, int random_seed);

#endif /* _THREAD_H */
```

start_preemptions() is part of the interrupt library I provide (libinterrupt.a, see Section 3.3), but its declaration is included as part of the interface that application programs include when using the thread library. Application programs can call start_preemptions() to configure whether (and how) interrupts are generated during the program. As discussed in class, these interrupts can preempt a running thread and start the next ready thread (by calling thread_yield()). If you want to test a program in the presence of these preemptions, have the application program call start_preemptions() once in the beginning of the function started by thread_libinit().

2. Disk scheduler (30%)

In this part, you will write a concurrent program to issue and service disk requests. I will provide a working thread library (thread.o) for you to use while testing your disk scheduler. As you write your thread library, the disk scheduler program will also help you test your thread library (and vice versa).

The disk scheduler in an operating system gets and schedules disk I/Os for multiple threads. Threads issue disk requests by queueing them at the disk scheduler. The disk scheduler queue can contain at most a specified number of requests (max_disk_queue); threads must wait if the queue is full.

Your program should start by creating a specified number of requester threads to issue disk requests and one thread to service disk requests. Each requester thread should issue a series of requests for disk tracks (specified in its input file). Each request is synchronous; a requester thread must wait until the servicing thread finishes handling its last request before issuing its next request. A requester thread finishes after all the requests in its input file have been serviced.

Requests in the disk queue are NOT serviced in FIFO order. Instead, the service thread handles disk requests in SSTF order (shortest seek time first). That is, the next request it services is the request that is closest to its current track. The disk is initialized with its current track as 0.

Keep the disk queue as full as possible to minimize average seek distance. That is, your service thread should only handle a request when the disk queue has the largest possible number of requests. This gives the service thread the largest number of requests to choose from. Note that the "largest number of requests" varies depending on how many request threads are still active. When at least max_disk_queue requester threads are alive, the largest possible number of requests in the queue is max_disk_queue. When fewer than max_disk_queue requester threads are alive, the largest number of requests in the queue is equal to the number of living requester threads. You will probably want to maintain the number of living requester threads as shared state.

2.1 Input

Your program will be called with several command-line arguments. The first argument specifies the maximum number of requests that the disk queue can hold. The rest of the arguments specify a list of input files (one input file per requester). I.e. the input file for requester r is $argv[r + 2]$, where

$0 <= r < (number\_of\_requesters)$. The number of threads making disk requests should be deduced from the number of input files specified.

The input file for each requester contains that requester's series of requests. Each line of the input file specifies the track number of the request (0 to 999). You may assume that input files are formatted correctly. Open each input file read-only (use ifstream rather than fstream).

2.2 Output

After issuing a request, a requester thread should call (note the space characters in the strings):

```
cout << "requester " << requester << " track " << track << endl;
```

A request is available to be serviced when the requester thread prints this line.

After servicing a request, the service thread should make the following call (note the space characters in the strings):

```
cout << "service requester " << requester << " track " << track << endl;
```

A request is considered to be off the queue when the service thread prints this line.

Your program should not generate any other output.

Note that the console is shared between the different threads. Hence the couts in your program must be protected by a monitor lock to prevent interleaving output from multiple threads.

2.3 Sample Input/Output

Here is an example set of input files (disk.in0 - disk.in4). These sample input files will be available at `http://www.cs.williams.edu/~jeannie/cs432/assignments/proj1.tar.gz`

```
disk.in0   disk.in1   disk.in2   disk.in3   disk.in4
--------   --------   --------   --------   --------
53         914        827        302        631
785        350        567        230        11
```

Here is one of several possible correct outputs from running the disk scheduler with the following command:

```
disk 3 disk.in0 disk.in1 disk.in2 disk.in3 disk.in4
```

(The final line of the output is produced by the thread library, not the disk scheduler.)

```
requester 0 track 53
requester 1 track 914
requester 2 track 827
service requester 0 track 53
requester 3 track 302
service requester 3 track 302
requester 4 track 631
service requester 4 track 631
requester 0 track 785
service requester 0 track 785
requester 3 track 230
```

```
service requester 2 track 827
requester 4 track 11
service requester 1 track 914
requester 2 track 567
service requester 2 track 567
requester 1 track 350
service requester 1 track 350
service requester 3 track 230
service requester 4 track 11
Thread library exiting.
```

2.4 Tips

I will provide a working thread library (thread.o) for you to use while testing your disk scheduler. You should first get your disk scheduler working without preemption, then test it with preemption enabled (Section 1 explains how to configure preemptions in your testing). Compile your disk scheduler with:

```
g++ -std=c++11 -o disk disk.cc thread.o libinterrupt.a -ldl
```

3. Thread library (70%)

In this part, you will write a library to support multiple threads within a single Linux process. Your library will support all the thread functions described in Section 1.

3.1 Creating and swapping threads

You will be implementing your thread library on x86 PCs running the Linux operating system. Linux provides some library calls (getcontext, makecontext, swapcontext) to help implement user-level thread libraries. You will need to read the manual pages for these calls. As a summary, here's how to use these calls to create a new thread:

```
#include <ucontext.h>

/*
 * Initialize a context structure by copying the current thread's context.
 */
getcontext(ucontext_ptr);    // ucontext_ptr has type (ucontext_t *)

/*
 * Direct the new thread to use a different stack.  Your thread library
 * should allocate STACK_SIZE bytes for each thread's stack.
 */
char *stack = new char [STACK_SIZE];
ucontext_ptr->uc_stack.ss_sp = stack;
ucontext_ptr->uc_stack.ss_size = STACK_SIZE;
ucontext_ptr->uc_stack.ss_flags = 0;
ucontext_ptr->uc_link = NULL;

/*
 * Direct the new thread to start by calling start(arg1, arg2).
 */
makecontext(ucontext_ptr, (void (*)()) start, 2, arg1, arg2);
```

Use swapcontext to save the context of the current thread and switch to the context of another thread. Read the Linux manual pages for more details.

3.2 Deleting a thread and exiting the program

A thread finishes when it returns from the function that was specified in thread_create. Remember to de-allocate the memory used for the thread's stack space and context (do this AFTER the thread is really done using it).

When there are no runnable threads in the system (e.g. all threads have finished, or all threads are deadlocked), your thread library should execute the following code:

```
    cout << "Thread library exiting.\n";
    exit(0);
```

3.3 Ensuring atomicity

To ensure atomicity of multiple operations, your thread library will enable and disable interrupts. Since this is a user-level thread library, it can't manipulate the hardware interrupt mask. Instead, I provide a library (libinterrupt.a) that simulates software interrupts. Here is the file "interrupt.h", which describes the interface to the interrupt library that your thread library will use. DO NOT MODIFY IT OR RENAME IT. interrupt.h will be included by your thread library (#include "interrupt.h"), but will NOT be included in application programs that use the thread library.

```
/*
 * interrupt.h -- interface to manipulate simulated hardware interrupts.
 *
 * This file should be included in the thread library, but NOT in the
 * application program that uses the thread library.
 */
#ifndef _INTERRUPT_H
#define _INTERRUPT_H

/*
 * interrupt_disable() and interrupt_enable() simulate the hardware's interrupt
 * mask.  These functions provide a way to make sections of the thread library
 * code atomic.
 *
 * assert_interrupts_disabled() and assert_interrupts_enabled() can be used
 * as error checks inside the thread library.  They will assert (i.e. abort
 * the program and dump core) if the condition they test for is not met.
 *
 * These functions/macros should only be called in the thread library code.
 * They should NOT be used by the application program that uses the thread
 * library; application code should use locks to make sections of the code
 * atomic.
 */
extern void interrupt_disable(void);
extern void interrupt_enable(void);
extern "C" {extern int test_set_interrupt(void);}

#define assert_interrupts_disabled() \
assert_interrupts_private(__FILE__, __LINE__, true)
```

```
#define assert_interrupts_enabled() \
assert_interrupts_private(__FILE__, __LINE__, false)


/*
 * assert_interrupts_private is a private function for the interrupt library.
 * Your thread library should not call it directly.
 */
extern void assert_interrupts_private(const char *, int, bool);

#endif /* _INTERRUPT_H */
```

Note that interrupts should be disabled only when executing in your thread library's code. The code outside your thread library should never execute with interrupts disabled. E.g. the body of a monitor must run with interrupts enabled and use locks to implement mutual exclusion.

3.4 Scheduling order

This section describe the specific scheduling order that your thread library should follow. Remember that a correct concurrent program must work for all thread interleavings, so your disk scheduler must work independent of this scheduling order.

All scheduling queues should be FIFO. This includes the ready queue, the queue of threads waiting for a monitor lock, and the queue of threads waiting for a signal. Locks should be acquired by threads in the order in which the locks are requested (by thread_lock() or in thread_wait()).

When a thread calls thread_create, the caller does not yield the CPU. The newly created thread is put on the ready queue but is not executed right away.

When a thread calls thread_unlock, the caller does not yield the CPU. The woken thread is put on the ready queue but is not executed right away.

When a thread calls thread_signal or thread_broadcast, the caller does not yield the CPU. The woken thread is put on the ready queue but is not executed right away. The woken thread requests the lock when it next runs.

3.5 Error handling

Operating system code should be robust. There are three sources of errors that OS code should handle. The first and most common source of errors come from misbehaving user programs. Your thread library must detect when a user program misuses thread functions (e.g., calling another thread function before thread_libinit, calling thread_libinit more than once, misusing monitors, a thread that tries to acquire a lock it already has or release a lock it doesn't have, etc.). A second source of error comes from resources that the OS uses, such as hardware devices. Your thread library must detect if one of the lower-level functions it calls returns an error (e.g., C++'s new operator throws an exception because the system is out of memory). For these first two sources of errors, the thread function should detect the error and return -1 to the user program (it should not print any error messages). User programs can then detect the error and retry or exit.

A third source of error is when the OS code itself (in this case, your thread library) has a bug. During development (which includes this entire semester), the best behavior in this case is for the OS to detect the bug quickly and assert (this is called a "panic" in kernel parlance). You should use assertion statements copiously in your thread library to check for bugs in your code. These error checks are essential

in debugging concurrent programs, because they help flag error conditions early.

I will not provide you with an exhaustive list of errors that you should catch. OS programmers must have a healthy (?) sense of paranoia to make their system robust, so part of this assignment is thinking of and handling lots of errors. Unfortunately, there will be some errors that are not possible to handle, because the thread library shares the address space with the user program and can thus be corrupted by the user program.

There are certain behaviors that are arguably errors or not. Here is a list of questionable behaviors that should NOT be considered errors: signaling without holding the lock (this is explicitly NOT an error in Mesa monitors); deadlock (however, trying to acquire a lock by a thread that already has the lock IS an error); a thread that exits while still holding a lock (the thread should keep the lock). Ask me if you're unsure whether you should consider a certain behavior an error.

Hint: Autograder test cases 16 and 17 check how well your thread library handles errors.

3.6 Managing ucontext structs

Do not use ucontext structs that are created by copying another ucontext struct. Instead, create ucontext structs through getcontext/makecontext, and manage them by passing or storing pointers to ucontext structs, or by passing/storing pointers to structs that contain a ucontext struct (or by passing/storing pointers to structs that contain a pointer to a ucontext struct, but this is overkill). That way the original ucontext struct need never be copied.

Why is it a bad idea to copy a ucontext struct? The answer is that you don't know what's in a ucontext struct. Byte-for-byte copying (e.g., using memcpy) can lead to errors unless you know what's in the struct you're copying. In the case of a ucontext struct, it happens to contain a pointer to itself (viz. to one of its data members). If you copy a ucontext using memcpy, you will copy the value of this pointer, and the NEW copy will point to the OLD copy's data member. If you later deallocate the old copy (e.g., if it was a local variable), then the new copy will point to garbage. Copying structs is also a bad idea for performance.

Unfortunately, it is rather easy to accidentally copy ucontext structs. Some of the common ways are:

- passing a ucontext by value into a function
- copying the ucontext struct into an STL queue
- declaring a local ucontext variable is almost always a bad idea, since it practically forces you to copy it

You should probably be using "new" to allocate ucontext structs (or the struct containing a ucontext struct). If you use STL to allocate a ucontext struct, make sure that STL class doesn't move its objects around in memory. E.g., using vector to allocate ucontext structs is a bad idea, because vectors will move memory around when they resize.

3.7 Example program

Here is a short program that uses the above thread library, along with the output generated by the program. Make sure you understand how the CPU is switching between two threads (both in function loop). "i" is on the stack and so is private to each thread. "g" is a global variable and so is shared among the two threads.

```
#include <iostream>
```

```
#include "thread.h"
#include <assert.h>

using namespace std;

int g=0;

void loop(void *a) {
    char *id;
    int i;

    id = (char *) a;
    cout <<"loop called with id " << (char *) id << endl;

    for (i=0; i<5; i++, g++) {
          cout << id << ":\t" << i << "\t" << g << endl;
          if (thread_yield()) {
               cout << "thread_yield failed\n";
               exit(1);
          }
    }
}

void parent(void *a) {
    int arg;
    arg = (intptr_t) a;

    cout << "parent called with arg " << arg << endl;
    if (thread_create((thread_startfunc_t) loop, (void *) "child thread")) {
            cout << "thread_create failed\n";
            exit(1);
    }

    loop( (void *) "parent thread");
}

int main() {
    if (thread_libinit( (thread_startfunc_t) parent, (void *) 100)) {
          cout << "thread_libinit failed\n";
          exit(1);
    }
}

OUTPUT:
parent called with arg 100
loop called with id parent thread
parent thread: 0 0
loop called with id child thread
child thread: 0 0
parent thread: 1 1
child thread: 1 2
parent thread: 2 3
child thread: 2 4
parent thread: 3 5
```

```
child thread: 3 6
parent thread: 4 7
child thread: 4 8
Thread library exiting.
```

3.8 Other tips

Start by implementing thread_libinit, thread_create, and thread_yield. Don't worry at first about disabling and enabling interrupts. After you get that system working, implement the monitor functions. Finally, add calls to interrupt_disable() and interrupt_enable() to ensure your library works with arbitrary yield points. A correct concurrent program must work for any instruction interleaving. In other words, I should be able to insert a call to thread_yield anywhere in your code that interrupts are enabled.

3.9 Test cases

An integral (and graded) part of writing your thread library will be to write a suite of test cases to validate any thread library. This is common practice in that real world–software companies maintain a suite of test cases for their programs and use this suite to check the program's correctness after a change. Writing a comprehensive suite of test cases will deepen your understanding of how to use and implement threads, and it will help you a lot as you debug your thread library.

Each test case for the thread library will be a short C++ program that uses functions in the thread library (e.g. the example program in Section 3.7). Each test case should be run without any arguments and should not use any input files. Test cases should exit(0) when run with a correct thread library (normally this will happen when your test case's last runnable thread ends or blocks). If you submit your disk scheduler as a test case, remember to specify all inputs (number of requesters, buffers, and the list of requests) statically in the program. This shouldn't be too inconvenient because the list of requests should be short to make a good test case (i.e. one that you can trace through what should happen).

Your test cases should NOT call start_preemption(), because I am not evaluating how thoroughly your test suite exercises the interrupt_enable() and interrupt_disable() calls.

Your test suite may contain up to 20 test cases. Each test case may generate at most 10 KB of output and must take less than 60 seconds to run. These limits are much larger than needed for full credit. You will submit your suite of test cases together with your thread library, and we will grade your test suite according to how thoroughly it exercises a thread library. See Section 5 for how your test suite will be graded.

4 Project logistics

Write your thread library and disk scheduler in C++ on Linux. The public functions in thread.h are declared "extern", but all other functions and global variables in your thread library should be declared "static" to prevent naming conflicts with programs that link with your thread library.

Compile an application program (app.cc) with a thread library (thread.cc) as follows:

```
g++ -std=c++11 -o thread thread.cc app.cc libinterrupt.a -ldl
```

Use g++ (/usr/bin/g++) to compile your programs. You may use any functions included in the standard C++ library, including (and especially) the STL. You should not use any libraries other than the standard C++ library.

Your thread library must be in a single file and must be named "thread.cc". Your disk scheduler must also be in a single file (e.g. "disk.cc").

I will place copies of thread.h, thread.o, interrupt.h, and libinterrupt.a in project1.tar.gz as well.

5 Grading, auto-grading, and formatting

To help you validate your programs, your submissions will be graded automatically using the auto-grader, and the result will be mailed back to you. You may then continue to work on the project and re-submit.

The student suite of test cases will be graded according to how thoroughly they test a thread library. We will judge thoroughness of the test suite by how well it exposes potential bugs in a thread library. The auto-grader will first compile a test case with a correct thread library and generate the correct output (on stdout, i.e. the stream used by cout) for this test case. Test cases should not cause any compile or run-time errors when compiled with a correct thread library. The auto-grader will then compile the test case with a set of buggy thread libraries. A test case exposes a buggy thread library by causing it to generate output (on stdout) that differs from the correct output. The test suite is graded based on how many of the buggy thread libraries were exposed by at least one test case. This is known as "mutation testing" in the research literature on automated testing.

Remember that your test cases should NOT call start_preemption(), because we are not evaluating how thoroughly your test suite exercises the interrupt_enable() and interrupt_disable() calls. The buggy thread libraries will not have problems with interrupt_disable/enable.

Each of the two parts of this project (thread library and disk scheduler) is considered separately for the purposes of submission. E.g., you can submit and get feedback on your thread library and your disk scheduler in one day (without using any bonus submission), and your bonus submissions for your thread library are distinct from your bonus submissions for your disk scheduler.

Because you are writing concurrent programs, the auto-grader may return non-deterministic results. In particular, test cases 20-24 for the thread library and test case 3 and 4 for the disk scheduler use asynchronous preemption, which may cause non-deterministic results.

Because your programs will be auto-graded, you must be careful to follow the exact rules in the project description:

1. (disk scheduler) Only print the two items specified in Section 2.2.
2. (disk scheduler) Your program should expect several command-line arguments, with the first being max_disk_queue and the others specifying the list of input files for the requester threads.
3. (thread library) The only output your thread library should print is the final output line "Thread library exiting.". Other than this line, the only output should be that generated by the program using your thread library.
4. (thread library) Your thread library should consist of a single file named "thread.cc".
5. Do not modify source code included in this handout (thread.h, interrupt.h).

Use the submit432 program to submit your files. submit432 submits the set of files associated with a project part (disk scheduler, thread library), and is called as follows:

```
submit432 <project-part> <file1> <file2> <file3> ... <filen>
```

Here are the files you should submit for each project part:

1. disk scheduler (project-part 1d)
   - C++ program for your disk scheduler (name should end in ".cc")
     example: submit432 1d disk.cc
2. thread library (project-part 1t)
   - C++ program for your thread library (name should be "thread.cc")
   - suite of test cases (each test case is an C++ program in a separate file). The name of each test case should end in ".cc".
     example: submit432 1t thread.cc test1.cc test2.cc

The official time of submission for your project will be the time of your last submission (of either project part). If you send in anything after the due date, your project will be considered late (and will use up your late days or will receive a zero).

6 Project Writeup

For your project writeup, I would like you to write a short (2–4 pages) paper (using Latex) that summarizes your project. In particular, you should include i) an introductory section that highlights the purpose of the project, ii) an architectural overview/design section that describes the structure of your code (if a figure makes your discussion more clear, use one!), iii) an evaluation section that discusses your test cases and how you verified the correct behavior of your scheduler/thread library, and iv) a conclusion that draws conclusions and reflects on the assignment in general. The purpose of the writeup is to help you gain experience with technical writing. Send me the writeup as a PDF (via email) no later than 48 hours after the due date for the code.