MP 3 – Implementing a userspace thread library

CS 423 – Spring 2011

Revision 1.0

Assigned March 2, 2011
Due March 16, 11:59 PM
Extension 48 hours (penalty 20% of total points possible)

1 Change Log

1.0 Initial Release.

2 Objectives and Background

The purpose of this MP is to help the students:

- Understand the functioning of a userspace thread library
- Gain a strong understanding of locks and condition variables used in Mesa monitor, by actually implementing them.
- Be able to write test-cases to check a multi-threaded code for correctness.

3 Overview

In this MP, you'll implement (in C++) your own thread.o that was supplied to you for MP2. Essesntially, you have to write a library that supports multiple userspace threads within one single Linux process. Your library has to support all the functions declared in the thread.h header file provided to you for MP2. (Note: 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.)

4 Description

4.1 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-space thread libraries. 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);
```

The context of a thread is saved in a `ucontext_t` type structure. It contains information related to the current state of a thread, such as its register-state, set of blocked signals, stack area etc. `swapcontext()` is used to save the context of the current thread and switch context to another thread, thus switching execution from one thread to another. You are encouraged to read the Linux man-pages of these three commands for more details.

4.2 Ensuring Atomicity

To ensure atomicity of multiple operations (such as switching contexts), your thread library will have to enable and disable interrupts. Since this is a user-space thread library, it can't manipulate the hardware interrupt mask, due to insufficient permissions. Instead, we provide a library (`libinterrupt.a`) that simulates software interrupts. The corresponding header file is "`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.

The thread library needs to disable interrupts for its critical section, as it can't itself use any high level primitives (locks, monitors etc), as it itself is a library to provided these high level primitives. High level primitives are typically built on using low level primitives; disabling interrupts over a critical section is one such low level primitive. Also, 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 created using your thread-library must run with interrupts enabled and shall use locks to implement mutual exclusion. So, you have to disable interrupts only for small portions in your thread library code and shall turn them on before passing control to one of the userspace threads.

4.3 Scheduling Order

Your thread library will have to maintain multiple queues. For example, it'll maintain a ready queue (queue of threads that are ready to run), queues of threads waiting to acquire a monitor lock, and queues of threads waiting for a signal. **All these scheduling queues should be FIFO**. Locks should be acquired by threads in the order in which the locks are requested (by `thread_lock()` or in `thread_wait()`). Remember that a correct concurrent program must work for all thread interleavings, so your disk scheduler must work independently of this scheduling order.

Here are some specifications:

1. 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.

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

3. 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 awoken thread(s) requests the lock when it next runs.

4. When a thread calls `thread_yield` it does yield the CPU.

The threads that are put on the ready queue get to run when the scheduler picks them next, according to the FIFO ordering.

4.4 Exiting the thread library

When there are no "runnable" threads in the system, i.e. the ready queue is empty, your thread library should execute the following code:

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

This is the same message that you encountered in MP2.

A thread finishes when it returns from the function that was specified in `thread_create()`. Remember to deallocate the memory used for the thread's stack space and context. (To avoid segfaults, do this AFTER the thread is really done using it).

4.5 Error Handling

Here we describe some of the error conditions that your library shall handle. 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, 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.

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 on the newsgroup if you're unsure whether you should consider a certain behavior an error.

4.6 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, etc., 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? (Remember the concept of deep-copy vs shallow-copy?) 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 (the `ucontext` struct is 348 bytes on Linux/x86).

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, and 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.

4.7 Writing 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. 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. 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 we are 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.

4.8 Some 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, we should be able to insert a call to thread_yield anywhere in your code that interrupts are enabled.

To compile an application (eg. disk.cc) that uses your thread library (thread.cc), use following command:

```
g++ -m32 thread.cc disk.cc libinterrupt.a -ldl
```

5 Problem

1. (40 points) You should implement a thread library that matches the interface given in `thread.h`, supplying the following procedures:

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

   The procedure `thread_libinit` is used to initialize the thread library. This should be done exactly once, and before calling any other thread functions. When `thread_libinit` is called it creates the first thread and initializes it to run the function pointed to by `func` on a single argument pointed to by `arg`. The calling process will never execute again after calling `thread_libinit`.

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

   The procedure `thread_create` is used for creating successive new threads. As with `thread_libinit` it will initialize the new thread to run the function pointed to by `func` on a single argument pointed to by `arg`.

```
int thread_yield(void)
```

The procedure `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. The main purpose of `thread_yield` is used to test the thread library, and you should not use it (although, later you will be expected to implement it). 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.

Synchronization and Mesa Monitors are implemented by the following:

```
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)
```

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 pair (lock number, cond number). Programs can use arbitrary numbers for locks and condition variables (i.e., they need not be numbered from 0 - n). The semantics of these function is as was discussed in class. 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.

There is one last function in `thread.h`:

```
void start_preemptions(bool async, bool sync, int random_seed);
```

The procedure `start_preemptions` is part of the interrupt library we provide (`libinterrupt.a`) and not part of the thread library you are to implement, 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. 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 (and only once) in the beginning of the function started by `thread_libinit()`.

6 Deliverables

1. Your thread library implementation - a single C++ file named "`thread.cc`"
2. A suite of test cases - each test case shall be a separate C++ program, written in a separate file (with a name ending in `.cc` with its own `main`).