Instructor: Ashvin Goel

Course Number: ECE344

Home
Lecture Notes
Lab Assignments
Discussion (piazza)
Grades (UofT portal)

Operating Systems

ECE344, FALL 2016
UNIVERSITY OF TORONTO

LAB 3: A PREEMPTIVE, USER-LEVEL THREAD PACKAGE

Due Date: Oct 24, 2016, 5 pm

In the previous lab, you implemented a cooperative, user-level thread package system in which `thread_yield` causes control to be passed from one thread to the next one. This lab has three goals.

First, you will implement preemptive threading, in which simulated "timer interrupts" cause the system to switch from one thread to another.

Second, you will implement two additional scheduling functions, `thread_sleep` and `thread_wakeup`. These functions will enable implementing **blocking** mutual exclusion and synchronization.

Finally, you will implement blocking locks for mutual exclusion, and condition variables for synchronization.

For this lab, we are not providing any additional code. You will be working with the code you have implemented in the `threads` directory in Lab 2.

TIMER SIGNALS

The user level code cannot use hardware timer interrupts directly. Instead, POSIX operating systems provide a software mechanism called *signals* that can be used to simulate "interrupts" at the user level.

Signals are a form of asynchronous, inter-process communication mechanism. A signal can be sent from one process to another, or from a process to itself. We will use the latter method to have the process, running your user-level scheduler, i.e., your thread library functions, deliver timer signals to itself.

The operating system delivers a signal to a target (recipient) process by interrupting the normal flow of execution of the process. Execution can be interrupted after any instruction. A process can register a signal handler, which is invoked when the signal is delivered. After the signal handler finishes executing, the normal flow of execution of the process is resumed. Notice the similarities between signals and hardware interrupts.

As signals can be delivered after any instruction, signal handling is prone to race conditions. For example, if you increment a counter (`counter = counter + 1`) during normal execution, and in your signal handler code as well, the increment operation may not work correctly because it is not atomic, and the signal may be deliver in between the instructions implementing the increment operation. To avoid this problem, you should disable signal delivery while the counter is being updated.

Please read a short introduction to signals to understand how they work in more detail. Make sure to read the "Risks" section or else you may not able to answer some questions below.

Now go over the code in the files `interrupt.h` and `interrupt.c`. You do not need to understand all the code, but it will be helpful to know how the code should be used. We will use the terms "interrupts" and "signals" interchangeably below.

void register_interrupt_handler(int verbose):

> This call installs a timer signal handler in the calling program using the `sigaction` system call. When a timer signal fires, the function `interrupt_handler` in the `interrupt.c` file is invoked. With the `verbose` flag, a message is printed when the handler function runs.

int interrupts_set(int enabled):

This function enables timer signals when enabled is 1, and disables (or blocks) them when enabled is 0. We call the current enabled or disabled state of the signal the *signal state*. This function also returns whether the signals were previously enabled or not (i.e., the previous signal state). Notice that the operating system ensures that these two operations (reading previous state, and updating it) are performed atomically when the `sigprocmask` system call is issued. Your code should use this function to disable signals when running any code that is a *critical section* (i.e., code that accesses data that is shared by multiple threads).

Why does this function return the previous signal state? The reason is that it allows "stacking" calls to this function. The typical usage of this function is as follows:

```
fn() {
    /* disable signals, store the previous signal state in "enabled" */
    int enabled = interrupts_set(0);
    /* critical section */
    interrupts_set(enabled);
}
```

The first call to `interrupts_set` disables signals. The second call restores the signal state to its previous state, i.e., the signal state before the first call to `interrupts_set`, rather than unconditionally enabling signals. This is useful because the caller of the function `fn` may be expecting signals to remain disabled after the function `fn` finishes executing. For example:

```
fn_caller() {
    int enabled = interrupts_set(0);
    /* begin critical section */
    fn();
    /* code expects signals are still disabled */
    ...
    /* end critical section */
    interrupts_set(enabled);
}
```

Notice how signal disabling and enabling is performed in "stack" order, so that the signal state remains disabled after `fn` returns.

The functions `interrupts_on` and `interrupts_off` are simple wrappers for the `interrupt_set` function.

int interrupts_enabled():

This function returns whether signals are enabled or disabled currently. You can use this function to check (i.e., assert) whether your assumptions about the signal state are correct.

void interrupts_quiet():

This function turns off printing signal handler messages.

To help you understand how this code works, we have provided you the `show_handler` program. Look at the `show_handler.c` file and make sure you understand the output of the `show_handler` program.

SETUP

You will be doing this lab within the `threads` directory that you created in Lab 2. So make sure to go over the setup instructions for Lab 2, if you have not done so previously.

Make sure to commit all your previous changes (or discard any uncommitted changes) in your local repository, and then run the following commands to get started with this lab.

```
cd ~/ece344
git tag Lab3-start
cd threads
make
```

PREEMPTIVE THREADING

Now you are ready to implement preemptive threading using the timer signals described above. However, before you do so, make sure that you can answer the following questions before proceeding any further.

1. What is the name of the signal that you will be using to implement preemptive threading?

2. Which system call is used by the process to deliver signals to itself?
3. How often is this signal delivered?
4. When this signal is delivered, which function in `thread.c` is invoked? What would this function do when it is invoked in the `show_handler` program?
5. Is the signal state enabled or disabled when the function in `thread.c` above is invoked? If the signal is enabled, could it cause problems? If the signal is disabled, what code will enable them? Hint: look for `sa_mask` in `interrupt.c`.
6. What does `unintr_printf` do? Why is it needed? Will you need other similar functions? Reread a short introduction to signals to find out.

Signals can be sent to the process at any time, even when a thread is in the middle of a `thread_yield`, `thread_create`, or `thread_exit` call. It is a very bad idea to allow multiple threads to access shared variables (such as your ready queue) at the same time. You should therefore ensure *mutual exclusion*, i.e., only one thread can be in a critical section (accessing the shared variables) in your thread library at a time.

A simple way to ensure mutual exclusion is to disable signals when you enter procedures of the thread library and restore the signal state when you leave.

Hint: think carefully about the invariants you want to maintain in your thread functions about when signals are enabled and when they are disabled. Make sure to use the `interrupts_enabled` function to check your assumptions.

Note that as a result of thread context switches, the thread that disables signals may not be the one enables them. In particular, recall that `setcontext` restores the register state saved by `getcontext`. The signal state is saved when `getcontext` is called (recall the `show_interrupt` function in the `show_ucontext.c` file is Lab 2), and restored by `setcontext`. As a result, if you would like your code to be running with a specific signal state (i.e., disabled or enabled) when `setcontext` is called, make sure that `getcontext` is called with the same signal state. Maintain the right invariants, and you'll have no trouble dealing with context switches.

It will be helpful to go over the manual pages of the context save and restore calls again.

Go ahead and implement preemptive threading by adding signal disabling and enabling code in your thread library in `thread.c`.

After you implement preemptive threading, you can test your code by running the `test_preemptive` program. To check whether this program worked correctly, you can run the following tester command (this script is run as part of testing Lab 3):

```
/cad2/ece344f/tester/scripts/lab3-01-preemptive.py
```

SLEEP AND WAKEUP

Now that you have implemented preemptive threading, you will extend your threading library to implement the `thread_sleep` and `thread_wakeup` functions. These functions will allow implementing mutual exclusion and synchronization primitives. In real operating systems, these functions would also be used to suspend and wake up a thread that performs IO with slow devices, such as disks and networks. The `thread_sleep` primitive blocks or suspends a thread when it is waiting on an event, such as a mutex lock becoming available or the arrival of a network packet. The `thread_wakeup` primitive awakens one or more threads that are waiting for the corresponding event.

The `thread_sleep` and `thread_wakeup` functions that you will be implementing for this lab are summarized here:

Tid thread_sleep(struct wait_queue *queue):

This function suspends the caller and then runs some other thread. The calling thread is put in a wait queue passed as a parameter to the function. The `wait_queue` data structure is similar to the run queue, but there can be many wait queues in the system, one per type of event or condition. Upon success, this function returns the identifier of the thread that took control as a result of the function call. The calling thread does not see this result until it runs later. Upon failure, the calling thread continues running, and returns one of these constants:

- `THREAD_INVALID`: alerts the caller that the queue is invalid, e.g., it is NULL.

- `THREAD_NONE`: alerts the caller that there are no more threads, other than the caller, that are available to run.

int thread_wakeup(struct wait_queue *queue, int all):

This function wakes up one or more threads that are suspended in the wait queue. These threads are put in the ready queue. The calling thread continues to execute and receives the result of the call. When "all" is 0, then one thread is woken up. In this case, you should wake up threads in FIFO order, i.e., first thread to sleep must be woken up first. When "all" is 1, all suspended threads are woken up. The function returns the number of threads that were woken up. It should return zero if the queue is invalid, or there were no suspended threads in the wait queue.

You will need to implement a `wait_queue` data structure before implementing the functions above. The `thread.h` file provides the interface for this data structure. As an optimization, note that each thread can be in only one queue at a time (a run queue or any one wait queue).

When implementing `thread_sleep`, it will help to think about the similarities and differences between this function and `thread_yield` and `thread_exit`. Make sure that `thread_sleep` suspends (blocks) the current thread rather than spinning (running) in a tight loop. This would defeat the purpose of invoking `thread_sleep` because the thread would still be using the CPU.

All the thought that you put into ensuring that thread preemption works correctly previously will apply to these functions as well. In particular, these functions access shared data structures (which ones?), so be sure to enforce mutual exclusion.

Go ahead and implement these functions in your thread library in `thread.c`.

After you implement the sleep and wakeup functions, you can test your code by running the `test_wakeup` and the `test_wakeup_all` programs. To check whether these programs worked correctly, you can run the following tester commands (these scripts are run as part of testing Lab 3):

```
/cad2/ece344f/tester/scripts/lab3-02-wakeup.py
/cad2/ece344f/tester/scripts/lab3-03-wakeupall.py
```

MUTEX LOCKS AND CONDITION VARIABLES

Now that you have implemented the `thread_sleep` and `thread_wakeup` functions, you will use them to implement mutual exclusion and synchronization primitives in your threads library. Recall that these primitives form the basis for managing concurrency, which is a core concern for operating systems, so your library would really not be complete without them.

For mutual exclusion, you will implement blocking locks, and for synchronization, you will implement condition variables.

The API for the lock functions are described below:

struct lock *lock_create():

Create a blocking lock. Initially, the lock should be available. Your code should associate a wait queue with the lock so that threads that need to acquire the lock can wait in this queue.

void lock_destroy(struct lock *lock):

Destroy the lock. Be sure to check that the lock is available when it is being destroyed.

void lock_acquire(struct lock *lock):

Acquire the lock. Threads should be suspended until they can acquire the lock, after which this function should return.

void lock_release(struct lock *lock):

Release the lock. Be sure to check that the lock had been acquired by the calling thread, before it is released. Wake up all threads that are waiting to acquire the lock.

The API for the condition variable functions are described below:

struct cv *cv_create():

Create a condition variable. Your code should associate a wait queue with the condition variable so that threads can wait in this queue.

void cv_destroy(struct cv *cv):

Destroy the condition variable. Be sure to check that no threads are waiting on the condition variable.

void cv_wait(struct cv *cv, struct lock *lock):

Suspend the calling thread on the condition variable `cv`. Be sure to check that the calling thread had acquired `lock` when this call is made. You will need to release the lock before waiting, and reacquire it before returning from this function.

void cv_signal(struct cv *cv, struct lock *lock):

Wake up one thread that is waiting on the condition variable `cv`. Be sure to check that the calling thread had acquired `lock` when this call is made.

void cv_broadcast(struct cv *cv, struct lock *lock):

Wake up all threads that are waiting on the condition variable `cv`. Be sure to check that the calling thread had acquired `lock` when this call is made.

The `lock_acquire`, `lock_release` functions, and the `cv_wait`, `cv_signal` and `cv_broadcast` functions access shared data structures (which ones?), so be sure to enforce mutual exclusion.

Go ahead and implement these functions in your thread library in `thread.c`.

After you implement these functions, you can test your code by running the `test_lock` and the `test_cv` programs. To check whether these programs worked correctly, you can run the following tester commands (these scripts are run as part of testing Lab 3):

```
/cad2/ece344f/tester/scripts/lab3-04-lock.py
/cad2/ece344f/tester/scripts/lab3-05-cv.py
```

HINTS AND ADVICE

You are encouraged to reuse *your own* code that you might have developed in the first lab or in previous courses to handle things such as queues, sorting, etc.

You may **not** use code that subsumes the heart of this project (e.g., you should not base your solution on wrappers of or code taken from the POSIX thread library). If in doubt, ask.

This project does not require you to write a large number of lines of code. It does require you to think carefully about the code you write. Before you dive into writing code, it will pay to spend time planning and understanding the code you are going to write. If you think the problem through from beginning to end, this project will not be too hard. If you try to hack your way out of trouble, you will spend many frustrating nights in the lab. This project's main difficulty is in conceptualizing the solution. Once you overcome that hurdle, you will be surprised at the simplicity of the implementation!

All the hints and advice from Lab 2 apply here as well.

Start early, we mean it!

TESTING YOUR CODE

You can test your entire code by using our auto-tester program at any time by following the testing instructions.

USING GIT

You should only modify the following files in this lab.

```
thread.c
```

You can find the files you have modified by running the `git status` command.

You can commit your modified files to your local repository as follows:

```
git add thread.c
git commit -m "Committing changes for Lab 3"
```

We suggest committing your changes frequently by rerunning the commands above (with different meaningful messages to the commit command), so that you can go back to see the changes you have made over time, if needed.

Once you have tested your code, **and committed it** (check that by running `git status`), you can tag the assignment as done.

```
git tag Lab3-end
```

This tag names the last commit, and you can see that using the `git log` or the `git show` commands.

If you want to see all the changes you have made in this lab, you can run the following `git diff` command.

```
git diff Lab3-start Lab3-end
```

More information for using the various git commands is available in the Lab 1 instructions.

CODE SUBMISSION

Make sure to add the `Lab3-end` tag to your local repository as described above. Then run the following command to update your remote repository:

```
git push
git push --tags
```

For more details regarding code submission, please follow the lab submission instructions.

Please also make sure to test whether your submission succeeded by simulating our automated marker.