Lab 3 - Preemptive Kernel

Operating Systems, CS-UH 3010

Assigned:	Oct 10, 2022	Due:	Oct 24, 2022

Do not share the start codes or virtual image provided in this lab, as they were kindly provided by Princeton without permission to redistribute.

This might be as complex as the one before but now you know what it takes to get through these assignments and you will better manage your time.

1 Overview

The previous lab was a simple but useful non-preemptive, multi-threaded operating system kernel. The main goal of this lab is to transform the non-preemptive kernel into a preemptive one. You will also change the system call mechanism from a simple function call to an interrupt mechanism as used in contemporary operating systems. Also, you will be required to add condition variables and semaphores to augment the synchronization primitives designed during the last lab. Time to read up on what condition variables are!

The kernel of this lab and future labs supports both processes and kernel threads. See the description for lab 2 on how the processes in our lab differ from the traditional notion of processes.

1.1 Getting Started

Some versions of gcc may not support the `-fno-stack-protector` flag; it is safe to remove this flag from Makefile if you are using one of those versions. If you are using the provided image on VirtualBox, it should be fine to directly use the provided Makefile.

We have provided the program: **tasks**, which prepares tasks (test cases) for your OS to run. Its usage is `./tasks taskset`. We have also provided five test sets:

1. `test_regs` observes the value of all registers both before and after a loop. Presuming that the process is preempted at some time during the loop, it will compare the values before and after, to see if any registers were improperly clobbered by preemption. Although this will test all registers (and flags), note that some classes of errors may manifest themselves rarely, and so passing this test does not mean that your code is perfect.

2. `test_preempt` creates two processes which spin as quickly as possible, but never yield or sleep. If you see that both counters are incrementing, then preemption is working (at least a little bit). Notice that for some reason, the counter growing speeds of two processes may be different, but the numbers of entry counts must be roughly the same if you are doing the round robin scheduling.
3. `test_blocksleep` creates two threads, and one goes to sleep. It tests four things: (i) that the scheduler survives when all processes are sleeping, (ii) that a process is not re-entered during sleep, (iii) that sleep lasts about the right amount of time, and (iv) the non-sleeping process is scheduled during the sleep.
4. `test_barrier` creates 26 threads and 1 process. The threads all sleep for a random period of time, and then `barrier_wait` on a common barrier. They repeat this process several times.
5. `tsk_test tests` priorities, locks, condition variables, semaphores, barriers, etc. Its output is self-explanatory. Note: it does not test the deadlock detection extra credit.

To test the barrier primitive on your OS, run `./tasks test_barrier` and then `make clean`. This will update the symbolic links in your main project directory so that, upon recompiling the project, your OS will run those tasks.

If you wanted to create your own test set, or to create a test set with no processes, you could copy one of these test sets to a new directory, and then modify it to suit your needs.

Use this test suite to test your kernel. We will use similar test cases when grading.

2 Preemptive Scheduling

The x86 interrupt mechanism is quite complicated, so we will present only the details relevant to this project. Interrupts can be triggered in several different ways: hardware, software, exceptions. The timer interrupt, IRQ 0, is a hardware interrupt, as is IRQ 7. The system call mechanism, which uses the int instruction, is a software interrupt. Exceptions are generated for a variety of invalid actions: dividing by zero, for example, will cause an exception.

The support code installs handlers for IRQ 0, IRQ 7, the system call interrupt, and the various exceptions. Your job for this part is to complete the handler for IRQ 0, `entry.S:irq0_entry`; the others are given to you as examples. The main file you will be working with is `entry.S`, but you may need to consult other files, especially `interrupt.[ch]` and `scheduler.[ch]`. In particular, the definition of the PCB structure has moved to `scheduler.h`.

The processor takes the following actions on an interrupt:

1. Disable interrupts
2. Push the flags register, CS, and return IP (in that order)
3. Jump to the interrupt handler

Once the interrupt handler has run, it returns control to the interrupted task with the `iret` instruction, which pops the instruction pointer and the flags register, then re-enables interrupts.

As with `entry.S:syscall_entry`, it may be some time before `entry.S:irq0_entry` actually performs the `iret`, since other tasks may execute in the middle. IRQ 0 is a hardware interrupt, so you should acknowledge it quickly using the macro `entry.S:SEND_EOI` – otherwise the hardware won't deliver any more timer interrupts. The other tasks `entry.S:irq0_entry` must perform are

1. Increment the 64-bit counter, `num_ticks`
2. Increment `entry.S:disable_count` to reflect the fact that interrupts are off.

3. If `nested_count` is zero, enter the kernel the way a system call would yield, else do nothing.
4. Decrement `entry.S:disable_count` in anticipation of `iret`

`nested_count` is a field in the PCB. For processes, it is 1 during a system call and 0 otherwise. For threads, it is always 1. You should study how interrupt.c changes this field during a system call.

This may not look very hard, but the tricky part is managing when interrupts are on, and when they are off. Your solution should satisfy the following two properties:

1. **Safety**: To avoid race conditions, interrupts should be off whenever your code is accessing kernel data structures, primarily the PCBs and task queues.
2. **Liveness**: Interrupts should be on most of the time so that processes that take too long are preempted.

3 Blocking Sleep

In the provided code, function sleep and `do_sleep` work by spinning on `num_ticks`. Unfortunately, this loop runs within the kernel, and so it cannot be preempted. As a result, if one task sleeps, all tasks sleep.

You must modify the provided implementation so that calls to sleep will block the current process (i.e. take it out of the ready queue until some appropriate time in the future).

4 Synchronization Primitives

The main file you will edit here is `sync.[ch]`, but you will need to consult `queue.[ch]` and `scheduler.[ch]`. You have to implement condition variables, semaphores, and barriers. We've given you an implementation of locks as a reference. These primitives must operate correctly in the face of preemption by turning interrupts on and off with the functions `enter_critical` and `leave_critical`.

In the descriptions below, if an operation is performed on an uninitialized structure, the results are up to you. All unblocks should place the unblocked thread at the end of the ready queue.

4.1 Condition Variables

`condition_init(cond)`	Initializes the specified memory to hold a condition variable.
`condition_wait(lock, cond)`	Releases lock, blocks until signalled, then reacquires lock. The calling thread is responsible for acquiring lock before calling this function. Please see the hints section.
`condition_signal(cond)`	Unblocks the first thread to have blocked on `condition_wait(lock, cond)`. If no thread is blocked on cond, then do nothing.
`condition_broadcast(cond)`	Unblocks all threads that have blocked on `condition_wait(lock,cond)`. The order in which threads are unblocked is not important.

4.2 Semaphores

`semaphore_init(sem, value)`	Initializes the specified memory to hold a semaphore whose starting value is value; a non-negative integer.
`semaphore_down(sem)`	If the semaphore is not positive, then block. Decrement the semaphore once it is positive. Please see the hints section 6.
`semaphore_up(sem)`	Increments the semaphore. Please see the hints section 6

Fairness requirement: in a trace with exactly n ups, the downs that complete successfully are the first n invocations. Calling `semaphore_init(sem, value)` counts as value ups.

4.3 Barriers

`barrier_init(bar, n)`	Initializes a barrier for use by n threads.
`barrier_wait(bar)`	Blocks until a total of n threads have called `barrier_wait(bar)`. The order in which threads are unblocked is unimportant. Immediately after the last call to `barrier_wait(bar)` returns, and possibly before the other calls have returned, the barrier is available for another round of waits. Please see the hints section 6.

5 Design Review

The design review is will be graded over 5 points, which will then contribute to 2 points of the overall lab.

1. `irq0_entry`: **1 pt**. Comparing to what you know of system calls (although irq0 is a hardware interrupt while system calls are software interrupts, they share with similar workflows), explain what this does and how to use it.
2. **Blocking Sleep: 1 pt**. You should be able to explain how to implement blocking sleep (how to make a task sleep and how to wake it up), as well as its relationship with the ready queue. If every process and every thread is sleeping, then the ready queue is empty, how can you handle this case?
3. **Condition Variable, Semaphore, Barrier: 3 pts.** For the above three synchronization primitives, specify 1) the data structure that you will use, and 2) the pseudo-code for each operation. The pseudo-codes provided in the hints are almost right, you can start from there to fix the issues they have. Specially, try to satisfy the fairness requirement for all primitives.

I strongly suggest that you study the new system call mechanism of this project and explain how it works. You should look at at least `entry.S`, `interrupt.c`, and `syslib.c`.

6 Hints

1. We put `-Wall -Wextra -Werror` flags in the Makefile to enable all warnings and make them errors, so your code must be warning free. When you start developing, you may turn off those flags for convenience, but don't forget to re-enable them for the final test.
2. Don't forget to link a test case using `./tasks taskset` before compiling.
3. We have provided a printf function, by which you can do some debugging.

4. C functions are allowed to trash registers %eax, %ecx, and %edx. Keep this in mind when calling them from assembly.

5. If interrupts are disabled, will the kernel clock num_ticks continue to increment?

6. This example suffers from a race condition:

```
void condition_wait(lock_t *lock, condition_t *cond) {
    lock_release(lock);
     /* 1 */
    enter_critical();
    block(&cond->wait_queue);
    leave_critical();
    lock_acquire(lock);
}
```

   At location 1, we may be preempted, then another thread can signal, resulting in a lost wakeup.

7. As does this:

```
void semaphore_up(semaphore_t *sem) {
    enter_critical();
    ++sem->value;
    unblock_one(&sem->wait_queue);
    leave_critical();
}
void semaphore_down(semaphore_t *sem) {
    enter_critical();
    if(sem->value <= 0) {
        block(&sem->wait_queue);
    }
    --sem->value;
    leave_critical();
}
```

8. Consider this trace: thread 1 calls down, but the value is 0, so it blocks; thread 2 calls up, unblocking thread 1; before thread 1 can run, however, thread 3 calls down; thread 1 now decrements the value to -1. Consider how to satisfy the fairness condition before applying the obvious fix.

9. And this:

```
void barrier_wait(barrier_t *bar) {
    enter_critical();
    if(--bar->value > 0) {
        block(&bar->wait_queue);
        ++bar->value;
    }
    else {
        unblock_all(&bar->wait_queue);
        ++bar->value;
    }
    leave_critical();
}
```

Suppose `bar->value` is initially 4, and thread 4 is the last of four threads to call `barrier_wait(bar)`. Thread 4 will unblock the other threads and increment `bar->value` to 1. Now, if thread 4 immediately calls `barrier_wait(bar)` again, it will return immediately, contrary to the specification.

10. We have provided a system call named `write_serial`, which writes a character to bochs' serial port. Additionally, we have configured bochs so that everything written to its serial port is in turn written to a file named serial.out on the real computer. This could be helpful for debugging. However, please ensure that the code you submit does not use `write_serial`, since we may use it for testing purposes.

7 Checklist

Note: this is not intended to be a thorough list; just a quick reference.

1. Preemptive Scheduling
   - implement `irq0_entry`
   - yields the current process
   - edit: entry.S
2. Blocking Sleep
   - Implement `do_sleep`
   - Compute some future time at which the process should unblock.
   - Remove the current process from the ready queue, save it somewhere else
   - Periodically check if these blocked processes can be added to the ready queue
   - edit: scheduler.c
3. Synchronization Primitives
   - Operations on uninitialized structures are undefined; you may handle such cases as you like.
   - Unblocks place the current process at the end of the ready queue
   - Must work with preemption
4. Condition Variables
5. Semaphores
6. Barriers
   - edit: sync.c

8 Grading Rubric

The project is worth 15 points. The breakdown is as follows:

Task	**Deliverables**	**Points**
Design review		2
Preempt	Processes can be preempted.	1
Register	Preemption preserves all registers.	1
Block sleep	Processes block on sleep.	0.5
Do_sleep	Threads block on do sleep.	0.5
All sleep	Scheduler survives when all processes are sleeping.	0.5
Timing	Sleep counts time correctly. Specifically, if your code passes the test blocksleep case with some acceptable error, it should be fine.	0.5
Wake one	Conditional signal wakes one.	0.5
Wake all	Conditional broadcast wakes all.	0.5
CV signal	Producer/consumer works when building a queue using your CV.signal.	0.5
CV broadcast	Producer/consumer works when building a queue using your CV.broadcast.	0.5
Semaphores	Producer/consumer works when building a queue using your semaphores.	1
Fairness		1
Barriers	Barriers successfully wait for all processes.	1
Auto-renew	Barriers are automatically renewed.	1
Priority	(Bonus) Priority with preemption. Tasks should be scheduled according to their priorities, and without starvation. Specifically, if you have task A with priority 100, task B with priority 200, task B should scheduled roughly twice as many as task A. We accept a fairly wide error here, say, $\pm 15\%$. Moreover, if you have task A with priority 100000, task B with priority 1, you should make sure task B somehow will get a chance to run.	
Priority	(Bonus) with yield	

To ensure compatibility with our tests, do the following:

- `./tasks tsk_test`
- `make clean; make`
- Fix any compilation errors without modifying files within `tsk_test`. Upon running, the screen should display which extra credits you have attempted. If the display is incorrect, then fix your `#defines` in common.h.
- We encourage you to print out diagnostics as a way to inspect your code. However, please do so in a way that leaves the output of the supplied tasks intact.

9 Submission

Please upload your group submission here: https://www.dropbox.com/request/2jxmURfjn1HNDNfA8xEz

10 Acknowledgments

Once again, we gratefully acknowledge the kindness of the staff of Princeton's COS 318 course for providing us with the material for (and most of the text of) this assignment.

Like always, best wishes and good luck!