Assignment 1 - Preemption

March 19, 2025

1 Background

In the current xv6 implementation, the scheduling of kernel threads relies on the kernel thread voluntarily relinquishing CPU control, i.e., by calling the `yield` method to mark itself as `RUNNABLE` and return CPU control to the scheduler.

In the code package for this assignment, the `void init()` function is the first piece of code executed after the kernel boots. It creates 8 worker threads that increment a shared variable `count`. Each worker thread first prints `starting`, then performs counting, sleeping for 500ms and printing the current progress every 1000 increments, and finally prints `exiting` and calls `exit` to terminate upon completion. The `init` function waits for all child threads to exit, then prints the value of `count` and exits.

Run the kernel using `make run`:

```
[INFO  0,-1] bootcpu_init: start scheduler!
[sched 0,-1] scheduler: switch to proc pid(1)
[INFO  0,1] init: kthread: init starts!
[sched 0,1] sched: switch to scheduler pid(1)
[sched 0,-1] scheduler: switch to proc pid(2)
[WARN  0,2] worker: thread 2: starting
[INFO  0,2] worker: thread 2: count 1000, sleeping
[INFO  0,2] worker: thread 2: count 2000, sleeping
...
[INFO  0,2] worker: thread 2: count 9000, sleeping
[INFO  0,2] worker: thread 2: count 10000, sleeping
[WARN  0,2] worker: thread 2: exiting
[sched 0,2] sched: switch to scheduler pid(2)
[sched 0,-1] scheduler: switch to proc pid(3)
[WARN  0,3] worker: thread 3: starting
...
[WARN  0,3] worker: thread 3: exiting
[sched 0,3] sched: switch to scheduler pid(3)
[sched 0,-1] scheduler: switch to proc pid(4)
[WARN  0,4] worker: thread 4: starting
...
[WARN  0,4] worker: thread 4: exiting
[sched 0,4] sched: switch to scheduler pid(4)
[sched 0,-1] scheduler: switch to proc pid(5)
[WARN  0,5] worker: thread 5: starting
...
[WARN  0,9] worker: thread 9: exiting
kthread: all threads exited, count 80000
[INFO  0,1] init: kthread: init ends!
```

We observe that after the scheduler starts, it first switches to the **init** thread (pid = 1), then sequentially executes the 8 worker threads, switching to the next thread only after each worker thread actively calls `exit`.

If we use `make runsmp` to start a kernel with 4 cores, although the kernel supports multi-core concurrency, the scheduling behavior remains sequential: threads 1, 2, 3, 4 execute first, followed by 5, 6, 7, 8, with switches occurring only when a thread voluntarily exits.

```
$ make runsmp | grep -E "sched|init"
[INFO  0,-1] bootcpu_init: start scheduler!
[INFO  2,-1] secondarycpu_init: start scheduler!
[INFO  1,-1] secondarycpu_init: start scheduler!
[INFO  3,-1] secondarycpu_init: start scheduler!
[sched 0,-1] scheduler: switch to proc pid(1)
[INFO  0,1] init: kthread: init starts!
[sched 0,1] sched: switch to scheduler pid(1)
[sched 0,-1] scheduler: switch to proc pid(2)
[sched 3,-1] scheduler: switch to proc pid(3)
[sched 2,-1] scheduler: switch to proc pid(4)
[sched 1,-1] scheduler: switch to proc pid(5)
[sched 0,2] sched: switch to scheduler pid(2)
[sched 0,-1] scheduler: switch to proc pid(6)
[sched 3,3] sched: switch to scheduler pid(3)
[sched 3,-1] scheduler: switch to proc pid(7)
[sched 2,4] sched: switch to scheduler pid(4)
[sched 2,-1] scheduler: switch to proc pid(8)
[sched 1,5] sched: switch to scheduler pid(5)
[sched 1,-1] scheduler: switch to proc pid(9)
[sched 0,6] sched: switch to scheduler pid(6)
[sched 0,-1] scheduler: switch to proc pid(1)
[INFO  0,1] init: thread 2 exited with code 114516, expected 114516
[INFO  0,1] init: thread 3 exited with code 114517, expected 114517
[INFO  0,1] init: thread 4 exited with code 114518, expected 114518
[INFO  0,1] init: thread 5 exited with code 114519, expected 114519
[INFO  0,1] init: thread 6 exited with code 114520, expected 114520
[sched 0,1] sched: switch to scheduler pid(1)
[sched 3,7] sched: switch to scheduler pid(7)
[sched 3,-1] scheduler: switch to proc pid(1)
[sched 2,8] sched: switch to scheduler pid(8)
[INFO  3,1] init: thread 7 exited with code 114521, expected 114521
[INFO  3,1] init: thread 8 exited with code 114522, expected 114522
[sched 1,9] sched: switch to scheduler pid(9)
[INFO  3,1] init: thread 9 exited with code 114523, expected 114523
[INFO  3,1] init: all threads exited, count 80000
[INFO  3,1] init: init ends!
```

Clearly, this scheduling implementation does not meet the requirements of multitasking scheduling in modern operating systems. Modern operating systems use preemptive scheduling (Preemption). When multiple ready processes need to run, the operating system allows them to take turns using the CPU. Even with a single CPU, the system can suspend the current process via preemption and start executing the next ready process.

Wikipedia: Preemption (computing) In computing, preemption is the act of temporarily interrupting an executing task, with the intention of resuming it at a later time. This interrupt is done by an external scheduler with no assistance or cooperation from the task. This preemptive scheduler usually runs in the most privileged protection ring, meaning that interruption and then resumption are considered highly secure actions. Such changes to the currently executing task of a processor are known as context switching.

2 Problem Description

In this assignment, please modify the given code to implement preemptive scheduling for kernel threads. You should call **yield** at every clock interrupt, causing the current process to relinquish the CPU and return to the scheduler.

This assignment is worth 4 points, with 3 Checkpoints and a report. The three checkpoints are worth 2 points, 1 point, and 1 point respectively, while the report is mandatory but does not contribute to the score. Your report must meet the following requirements:

- A report in PDF file format.
- You do not need to answer the reflection questions from the guide in the report.
- **Record the total time spent completing this assignment.**
- **Screenshots showing successful runs of each checkpoint.**
- (Optional) Describe any difficulties you encountered while completing this assignment, or suggest additional guidance you think this assignment might need.
- (Optional) You may share your thought process for completing this assignment in the report.
- The optional parts should be as concise as possible.

Your modified operating system should correctly execute until the `infof("init ends!");` statement without any PANIC messages, except for *init process exited* and *other CPU has panicked*.

2.1 Correct Output Hint

If your implementation is correct, the 8 child threads should **take turns** counting to 1000 before starting the next round of counting. Your logs should look like this:

```
[INFO  0,1] init: kthread: init starts!
[sched 0,1] sched: switch to scheduler pid(1)
[sched 0,-1] scheduler: switch to proc pid(2)
[WARN  0,2] worker: thread 2: starting
[INFO  0,2] worker: thread 2: count 1000, sleeping
[sched 0,2] sched: switch to scheduler pid(2)
[sched 2,-1] scheduler: switch to proc pid(4)
[sched 1,-1] scheduler: switch to proc pid(3)
[sched 3,-1] scheduler: switch to proc pid(5)
[WARN  2,4] worker: thread 4: starting
[WARN  1,3] worker: thread 3: starting
[sched 0,-1] scheduler: switch to proc pid(6)
[WARN  3,5] worker: thread 5: starting
[WARN  0,6] worker: thread 6: starting
[INFO  2,4] worker: thread 4: count 2000, sleeping
[INFO  0,6] worker: thread 6: count 5000, sleeping
[INFO  1,3] worker: thread 3: count 3000, sleeping
[INFO  3,5] worker: thread 5: count 4000, sleeping
[sched 0,6] sched: switch to scheduler pid(6)
[sched 1,3] sched: switch to scheduler pid(3)
[sched 2,4] sched: switch to scheduler pid(4)
[sched 0,-1] scheduler: switch to proc pid(7)
[sched 3,5] sched: switch to scheduler pid(5)
[WARN  0,7] worker: thread 7: starting
```

```
[sched 2,-1] scheduler: switch to proc pid(9)
[sched 1,-1] scheduler: switch to proc pid(8)
[WARN  2,9] worker: thread 9: starting
[sched 3,-1] scheduler: switch to proc pid(2)
[INFO  2,9] worker: thread 9: count 7000, sleeping
[INFO  0,7] worker: thread 7: count 6000, sleeping
[WARN  1,8] worker: thread 8: starting
[INFO  1,8] worker: thread 8: count 8000, sleeping
...
```

3 Code Explanation

In the code for this assignment, the *main.c* file uses `create_kthread` to create a kernel thread **init**. The prototype of this method is defined as follows:

```
int create_kthread(void (*fn)(uint64), uint64 arg);
```

This method allocates a PCB `struct proc` structure. It then sets `context.ra` to the `first_sched_ret` function and sets the context's s1 and s2 registers to the provided `fn` and `arg` parameters, respectively.

`first_sched_ret` is the method executed when the process is first scheduled by the scheduler. It releases the process's lock, enables interrupts, and jumps to the function `fn` passed during `create_kthread` to begin executing the specified kernel thread code. The `arg` parameter is optionally passed to `fn`. When executing the worker method, the CPU's interrupts should remain enabled, allowing clock interrupts to enter the Trap handler.

```
static void first_sched_ret(void) {
    // s0: frame pointer, s1: fn, s2: uint64 arg
    void (*fn)(uint64);
    uint64 arg;
    asm volatile("mv %0, s1":"=r"(fn));
    asm volatile("mv %0, s2":"=r"(arg));

    release(&curr_proc()->lock);
    intr_on();
    fn(arg);
    panic("first_sched_ret should never return. You should use exit to terminate kthread");
}
```

4 Guide Steps

4.1 Checkpoint. 1

Until the next instruction, use only `make run` to run the single-CPU kernel.

Modify the handling of the `SupervisorTimer` interrupt in `kernel_trap` to call the *yield* function, relinquishing continued execution of the current process.

Hint. 1 You may encounter a null pointer issue. Note that the `yield` function means relinquishing execution of the current process, which is reasonable only when a process is running on the CPU. The `yield` method dereferences the `struct proc*` returned by `curr_proc()`. However, when the scheduler is waiting for an interrupt in the `scheduler` method (`wfi`), this value is NULL. We should only yield kernel processes, not the scheduler.

Hint. 2 You will encounter the error *sched should never be called in kernel trap context.* This is because, in the original kernel scheduling model, preemption of kernel threads was not allowed—only user threads could be preempted. Upon entering a trap, we increment the `mycpu()->inkernel_trap` counter and use it to detect issues with locks or scheduling in the kernel trap context. For example, we do not expect to encounter another interrupt within the kernel's Trap Handler, a behavior known as Nested Interrupt. To resolve this, simply clear this value to zero before calling `yield()` and restore it afterward.

Checkpoint Passed. You should see in the logs that the 8 worker threads can run in turns. However, a kernel panic occurs when the first process exits. At this point, you should encounter the error *kerneltrap: not from supervisor mode* followed by a kernel panic. This indicates you have completed **Checkpoint. 1**.

4.2 Checkpoint. 2

Observe the `sstatus` register value in the Kernel Panic report. `SPP:U` indicates that the `sstatus.SPP` bit is 0, meaning the CPU was in U-mode before entering this trap. However, our code never expects to return to U-mode for execution, and the only way to return to user space is by clearing the `SPP` bit in `sstatus` before an `sret`, which our code never actively does.

Identify the reason why the kernel switched to U-mode and resolve the issue.

Hint. 3 Refer to the privileged architecture manual *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*, Section *4.1.1*, *Supervisor Status Register (sstatus)*, to identify when this flag is cleared by the CPU.

Hint. 4 This error `always` occurs after the first process exits. The `exit` function marks itself as `ZOMBIE` and, upon termination, invokes the `sched` method to switch to the scheduler. The scheduler then searches for the next `RUNNABLE` process to continue execution. However, all processes remain stuck at the `yield` point within the interrupt handler `kernel_trap`.

Checkpoint Passed. Your code can probably execute the `init` function on a single CPU and display the following messages:

```
[INFO  0,1] init: init ends!
[PANIC 0,1] os/proc.c:218: init process exited
```

This indicates you have completed **Checkpoint. 2**.

4.3 Checkpoint. 3

This checkpoint requires passing the multi-CPU pressure test.

Comment out two `logf` lines in *sched.c* to avoid excessive output; change `TICKS_PER_SEC` in *timer.h* to 400 to increase interrupt frequency; and modify `NTHREAD` to 15 and `SLEEP_TIME` to 50 in *nommu_init.c*.

Repeatedly run the multi-CPU operating system using **make runsmp**. You may encounter issues such as the program freezing or a kernel panic. Attempt to resolve these issues.

Hint. 5 Checkpoint 2 indicates that the values of some CSRs (Control and Status Registers) change before and after the execution of `sched`.

Hint. 6 Refer to the method you used to solve Checkpoint. 2 and read the privileged architecture manual, Section *3.3.2 Trap-Return Instructions*, for details about the `sret` instruction.

Checkpoint Passed. You can consistently pass the `make runsmp` test more than 10 times in a row. At the end of the output, you should see the following logs:

```
[INFO  1,1] init: init ends!
[PANIC 1,1] os/proc.c:218: init process exited
[PANIC 2,-1] os/trap.c:45: other CPU has panicked
[PANIC 0,-1] os/trap.c:45: other CPU has panicked
[PANIC 3,-1] os/trap.c:45: other CPU has panicked
```

5 Submission

Place your PDF report in the same directory as the Makefile file, then run `make handin`. This will generate a handin.zip archive. Upload this file to Blackboard.