Midterm Exam - EECS 211

Process Scheduling

Winter 2022

In this midterm exam you will modify a tick-based kernel to make the tick period dynamic and under the kernel's control. Your goal is to modify the kernel so that tick period is not fixed. You should have some policy for how the tick period changes based on the behavior of the processes.

You are required to include a short description of your adaptive tick interval algorithm.

1 Background

For this assignment you will use the xv6 kernel (https://github.com/mit-pdos/xv6-riscv) for the RISC-V 64 bit architecture. This includes all of the core components of an operating system kernel while still being accessible to hack on.

The kernel runs on QEMU (https://www.qemu.org/), a CPU emulator (similar to the LC3 VM, but more elaborate and can emulate more complicated processor architectures).

1.1 xv6 Kernel

The xv6 kernel includes the various components you would expect to be in an operating system kernel. The actual scheduler is in `proc.c`. You should be able to verify that the scheduler by default uses a round-robin scheduler.

The code for handling interrupts and exceptions is in `trap.c`. This is, for example, where the code that handles when a userspace process calls a syscall and the system traps to the kernel resides, as well as where interrupts are handled.

Userspace applications have a very standard set of syscalls, include `read`, `write`, `fork`, etc.

1.2 RISC-V

RISC-V is a new risc architecture with substantial documentation available online. Here are some basics which should be useful.

RISC-V cores have three privilege level modes: M (machine), S (supervisor), and U (user). A specific instruction is used to switch privilege modes. For example, the `mret` instruction switches from M mode to S mode, and the `sret` instruction switches from S mode to U mode. To go the other way, the `ecall` instruction traps to a higher privilege level.

The core boots into M mode. Some initial setup is done, and then the core switches to S mode where the kernel starts running. Of course applications execute in U mode.

The tick in xv6 is implemented using the machine-mode timer and configured initially in `start.c`. This is a simple timer which is just a continuously running clock and single compare register. When the clock value matches the compare register an interrupt is generated. Since the machine timer operates in M mode, this interrupt traps the CPU into M mode. The interrupt handler is specified by the `mtvec` register, which is set to the `timervec` function in the `kernelvec.S` file. The `timervec` handler adds the tick interval to the compare register to reset the timer for the next tick event. It then generates a software interrupt to notify the kernel in S mode that the tick occurred. These trap events are handled by the kernel in the `trap.c` file.

The different privilege levels can store state in the `mscratch`, `sscratch`, and `uscratch` registers. For example, machine mode can store 64 bits of state in the `mscratch` register. The xv6 kernel uses the `mscratch` register to store a pointer to some state reserved for the machine timer code. You can see in the `timervec` function how that is used. Note that the interval used for the tick is stored in the array pointed to by `mscratch`.

2 Setup

First you have to install some dependencies.

1. Clone the xv6 Kernel source.

```
    $ git clone https://github.com/mit-pdos/xv6-riscv
```

2. Install the RISC-V compiler.

```
    $ sudo apt-get install gcc-riscv64-linux-gnu
```

   OR

```
    $ brew tap riscv/riscv
    $ brew install riscv-tools
```

3. Install QEMU

```
    $ sudo apt-get install qemu
```

   OR

```
    $ brew install qemu
```

4. Test

```
    $ cd xv6-riscv
    $ make qemu
```

   That will load the emulator with the kernel, and by default the kernel loads the shell process (`sh`). This is a basic shell, but should be reasonably familiar.

   To exit QEMU you can enter `ctrl+a` and then `x` to exit QEMU.

2.1 Troubleshooting

Here are some of the problems that you *may* face (depending on the version of your Linux and GCC):

1. `error: unrecognized command line option '-mno-relax'; did you mean '-Wno-vla'?` This error is due to mismatch between the GCC versions. you may find the solution in this post useful: https://github.com/mit-pdos/xv6-riscv/issues/7

2. `make: qemu-system-riscv64: command not found`. This means there is a problem with either (i) installing the QEMU or (ii) the PATH variable of your shell. One solution is to install QEMU from its source code as follows:

   ```
       $ git clone https://github.com/qemu/qemu
       $ cd qemu
       $ ./configure --target-list=riscv64-softmmu
       $ make -j $(nproc)
       $ sudo make install
   ```

   Now you will see that the file `qemu-system-riscv64` is located under the `qemu/build/riscv64-softmmu` folder. Now you can modify the `Makefile` used by the xv6-riscv kernel (STEP 4 above) by changing the `QEMU` variable and hardcode the path to the compiled `qemu-system-riscv64` file.

2.2 How to modify the kernel:

To modify the kernel you can directly edit the source and then re-run `make qemu`. You can also add new applications in the `user` folder and then add them to the `UPROGS` variable in the `Makefile` and they will be available in the kernel.

To simplify the project, you should edit the `CPUS` variable in the `Makefile` and set it to 1. Run `make clean` to ensure the change propagates followed by `make qemu` to compile.

3 Deliverable

The objective of this project is to implement a dynamic tick interval policy. This should respond to the operation of processes in the system with the goal of reducing the number of ticks while preventing a process from monopolizing the CPU.

You should devise a scheme for verifying that the tick interval is being dynamically adjusted. Certainly one option is to add a `printf()` when the tick occurs and inspect that the frequency of the print messages changes. You can also write an application which calls `fork()` to create multiple processes that the OS has to schedule. If one is intermittently CPU bound (i.e. it has periods of high computation (say `while(1)`) and periods of waiting) and the other is IO bound, both processes should make progress but the tick interval should change.