COMP4430 - Operating Systems 2

Assignment 4

Due: *at Noon, April 5, 2017*

Introduction

This assignment has three tasks. In the second task, you will implement various scheduling policies. However, before that, we will implement an infrastructure that will allow us to examine (Task 3) how these policies affect performance under different evaluation metrics.

The first task is to extend the `proc struct` (see `proc.h`). Extend the `proc struct` by adding the following fields to it: `ctime`, `stime`, `retime` and `rutime`. These will respectively represent the creation time and the time the process was at one of following states: `SLEEPING`, `READY(RUNNABLE)` and `RUNNING`.

Tip: These fields retain sufficient information to calculate the turnaround time and waiting time of each process.

Upon the creation of a new process the kernel will update the process' creation time. The fields (for each process state) should be updated for all processes whenever a clock tick occurs (you can assume that the process' state is `SLEEPING` only when the process is waiting for I/O). Finally, care should be taken in marking the termination time of the process (note: a process may stay in the `ZOMBIE` state for an arbitrary length of time. Naturally this should not affect the process' turnaround time, wait time, etc.). Since all this information is retained by the kernel, we are left with the question of extracting this information and presenting it to the user. To do so, create a new system call `wait2` which extends the wait system call:

```
int wait2(int *retime, int *rutime, int *stime)
```

Input:

`int * retime` / `rutime` / `stime` – pointer to an integer in which `wait2` will assign:

- The aggregated number of clock ticks during which the process **waited** (was able to run but did not get CPU)
- The aggregated number of clock ticks during which the process was **running**
- The aggregated number of clock ticks during which the process was **waiting for I/O** (was not able to run).

Output:

`pid` of the terminated child process	if successful
`-1`	upon failure

Scheduling

Scheduling is a basic and important facility of any operating system. The scheduler must satisfy several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low-priority and high-priority processes, and so on.

The set of rules used to determine when and how to select a new process to run is called a scheduling policy. You first need to understand the current scheduling policy. Locate it in the code and try to answer the following questions: which process does the policy select for running, what happens when a process returns from I/O, what happens when a new process is created and when/how often does the scheduling take place.

First, change the current scheduling code so that process preemption will be done every quanta size (measured in clock ticks) instead of every clock tick. Add this line to `param.h` and initialize the value of `QUANTA` to `5`.

```
#define QUANTA <Number>
```

In the next part of the assignment you will add three different scheduling policies in addition to the existing policy. Add these policies by using the C preprocessing abilities.

Modify the `Makefile` to support `SCHEDFLAG` – a macro for quick compilation of the appropriate scheduling scheme. Thus the following line will invoke the `xv6` build with the default scheduling:

```
$ make qemu SCHEDFLAG=DEFAULT
```

The default value for `SCHEDFLAG` should be `DEFAULT` (in the `Makefile`).

Policy 1: Default Policy (`SCHEDFLAG=DEFAULT`)

Represents the scheduling policy currently implemented at `xv6` (with the only difference being the newly defined `QUANTA`).

Policy 2: First Come First Served (`SCHEDFLAG=FCFS`)

Represents a non preemptive policy that selects the process with the lowest creation time. The process runs until it no longer needs CPU time (IO/yield/block).

Policy 3: Multi-level Queue Scheduling (`SCHEDFLAG=SML`)

Represents a preemptive policy that includes a three priority queues. The initial process should be initiated at priority `2` and the priority should be copied upon `fork`. In this scheduling policy the scheduler will select a process from a lower queue only if no process is ready to run at a higher queue. Moving between priority queues is only available via a system call. Within a given queue the process selection is set by the current system default.

```
int set_prio(int priority)
```

Input:

`priority` – A number between `1-3` for the new process priority

Output:

0	if successful
1	otherwise

Policy 4: Dynamic Multi-level Queue Scheduling (`SCHEDFLAG=DML`)

Represents a preemptive policy similar to **Policy 3**. The difference is that the process cannot manually change it's priority. These are the dynamic priority rules:

- Calling the exec system call resets the process priority to `2` (default priority).
- Returning from `SLEEPING` mode (in our case I/O) increases the priority of the process to the highest priority.
- Yielding the CPU manually keeps the priority the same.
- Running the full quanta will result in a decrease of priority by `1`.

`yield` system call

Add the system call `yield`, which will yield execution to another process:

```
int yield()
```

Input:

`None`

Output:

0	if successful
1	otherwise

Sanity Tests

You will add two applications that test the impact of each scheduling policy. Similarly to several built-in user space programs in `xv6` (e.g., `ls`, `grep`, `echo`, etc.), you can add your own user space programs to `xv6`.

General Sanity Test

Add a program called `sanity` which takes a number (`n`) as argument, then it will `fork` (`3*n`) processes and wait till all of them finish, for each child process that end print its statistics. Each of the `3n` processes is of one of the three types:

- process with (`pid` mod 3 = 0) – CPU-bound process (`CPU`):
  - run `100` times dummy loop of `1000000` iterations
- process with (`pid` mod 3 = 1) – short tasks based CPU-bound process (`SCPU`):
  - run `100` times dummy loop of `1000000` iterations
  - after each dummy loop (of the `100`) `yield` the CPU
- process with (`pid` mod 3 = 2) – I/O bound process (`IO`):
  - to simulate the I/O waiting we will make dummy sleeps. makes `100` times: `sleep(1)`

Printing the Statistics

For each terminated process print in new line:

- the process id and its type (CPU / SCPU / IO)
- its wait time, run time and I/O time

  then after all `3n` processes terminates print the following:

- *Sleep time* – average time that a job was sleeping (for each process type group)
- *Ready time* – average time that a job to was waiting to CPU (for each process type group)
- *Turnaround time* – average time for a job to complete (for each process type group)

Priority Schedule Test

Add a another program called `SMLsanity` to test priority `3`. This should be a small test who prove that the scheduling order is prioritized as expected.

Hint: `fork` lots (let say 20) of CPU-bound processes, give each process different priority then print they termination time.

Analyze (for both parts)

Run the test for all the different policies, before checking the results try to predict the results for each policy, then check if the statistics results consists your prediction.

- Be ready to explain how you calculate the statistics and how they reflect the correctness of your scheduling policies

Tip: You have to call the exit system call to terminate a process' execution.