[redacted] [redacted]

Homework 3 : Lottery Scheduling

[redacted]

Abstract

In this homework, you will implement a lottery scheduling algorithm and measure the performance of the scheduling policies. Short-answer questions included in this homework serve to check your theoretical understanding of concepts.

[redacted]

Introduction

Scheduling is an important facility for any operating system. The scheduler must satisfy and optimize several conflicting objectives (discussed in class) by determining when and how a process should run. In this homework, you will implement lottery scheduling algorithm and the performance measurement of the scheduling policies.
In Part 1, you will implement a system call `wait_stat` where parent process will print the statistics of a child process when the child process is waited.
In Part 2, you will create a program `sanity` that creates some child processes and prints out the statistics for all children using `wait_stat`.
Part 3 is the time you implement the lottery scheduling algorithm. To review the scheduling algorithm, please see the slides Lec-9 Scheduling Algorithms, pp. 33-45.
Part 1 and Part 2 together help debugging for Part 3.
Short-answer questions are included in this homework. You will type your answers in Short Answer section on Anubis.

Part 1: Performance Measurement

The goal of this part is to implement infrastructure that will allow us to examine how policies affect performance under different evaluation metrics.

1. Extend `struct proc` in `kernel/proc.h` by adding the following fields of type `int`:

[redacted]

Field Name	What it represents
creation_time	The time that the process was created
termination_time	The time that the process was terminated
sleep_time	The total time that the process was in SLEEPING state
ready_time	The total time that the process was in RUNNABLE state
running_time	The total time that the process was in RUNNING state

- During a process's lifetime, a process can be in different states back and forth. *Total time* (tracked for `sleep_time`, `ready_time` and `running_time`) accumulates the duration the process was in a particular state.
- `RUNNABLE` is the xv6 term for `READY` from the slides. Other states mentioned here are named consistently.

2. Implement the update rules for the above rules as follows:

   - `creation_time`: Set when a new process is created. Think, which system call creates a new process? Apart from the processes created with that system call, how is the initial process created?
   - `sleep_time`, `ready_time`, `running_time`:
     When a clock tick occurs, increment the value of one of `{sleep_time, ready_time, running_time}` by 1. Such 1-unit increment synchronizes with a clock tick. Locate the exact place where a clock tick is incremented in `trap.c`, immediately after (or immediately before) the tick is incremented.
     There, you should loop over the process table, check if the process's state is `SLEEPING`, `RUNNABLE` or `RUNNING` and update the fields accordingly.
     The following is an example of going through the process table and checking each process's state.

     ```
     acquire(&ptable.lock); // Because the process table is shared,
                            // we need to acquire the lock before proceeding
     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
         // You might want to check for other things
         if(p->state == UNUSED) {
             // do something
         }
     }
     release(&ptable.lock);
     ```

   - `termination_time`: Think, when does a process terminate?
     Be careful when marking the termination time of the process. Note that a process may stay in the `ZOMBIE` state for an arbitrary length of time. Naturally this should not affect the process's turnaround time, wait time, etc.

   `ticks` is a global variable that you might refer to when retrieving the current time.

3. Since all this information is retained by the kernel we need to find a way to present it to the user. To do this, create a new system call, `wait_stat`, that has the following function prototype. You need to add this line to `kernel/defs.h`.

   ```
   int wait_stat(int* ctime, int* ttime, int* retime, int* rutime, int* stime);
   ```

   `wait_stat` should deference each pointer and sets their value to the value of the corresponding field on the current process. See how `wait` is implemented in `kernel/proc.c` to have an idea of how it works. Note that `proc` is a global variable that represents the current running process.

   The parameters are pointers to integers for `wait_stat` to store the results:

Function Arguments	What it will be assigned
ctime	The time that the process was created
ttime	The time that the process was terminated
retime	The total time that the process was in RUNNABLE state
rutime	The total time that the process was in RUNNING state
stime	The total time that the process was in SLEEPING state

You might find reviewing the slides from lecture 6 helpful.

- The return value is the pid of the waited process or -1 upon failure. (What does it mean by a process is waited?)
- Use argptr to fetch the arguments from the stack for system calls. Example usage:

```
int my_syscall(void) {
    int *ptr;
    if (argptr(
            0,              // Fetch the 0th argument;
            (char **) &ptr, // save it as a local variable ptr;
            sizeof(int)     // the fetched pointer points to a block of
                            // memory of sizeof(int) bytes.
        ) < 0) {            // When the return value is negative, argptr fails
        return -1;
    }
}
```

- wait_stat is very similar to wait. You can copy the body of wait and modify that when implementing wait_stat.
- Time is measured in ticks.

In the next part, we will write a user program which helps test wait_stat.

Part 2: Sanity Test

The goal of this part is to measure the performance of the scheduling algorithm (or policy) by implementing a **user program** sanity.

- The starter code of sanity.c is provided. Since sanity is a user program, the file sanity.c can be found under the user/ directory.
- When sanity program starts, the main process immediately creates 20 child processes. It must not wait for a process to exit before creating the next one. All child processes should be created at once.
- Each child process calls a time-consuming function. The function is provided, and feel free to change the argument to make it run faster when testing, but don't forget to reset it back to the given parameter before submitting. After the child process finishes its computation it will exit.
- The main process will call wait_stat and wait until all of its children exit.

  For every finished child, the main process will print its pid, creation time, termination time, ready time, running time, and sleep time.
- The main process will print the average wait time, average running time, average sleep time, and average turnaround time of all the processes.

- The printing formats are as follows:

  Per Process Format:

  ```
  printf("PID[%d] Creation[%d] Termination[%d] Ready Time[%d] Running Time[%d] Sleep
      Time[%d]\n", ...);
  ```

  Main Process Format:

  ```
  printf("Avg. Wait Time[%d] Avg. Running Time[%d] Avg. Sleep Time[%d] Avg. Turnaround
      Time[%d]\n", ...);
  ```

If `wait_stat` is implemented correctly, you should see some different non-zero values for each process, and they show up mostly in PID order. Note that it is expected to have 0 for sleep time because none of these processes is blocked.

Part 3: Lottery Scheduling

The goal of this part is to implement lottery scheduling replacing the current scheduling algorithm.

1. You need to understand the current xv6 scheduling policy. Locate the current xv6 scheduling policy in the xv6 source code (`kernel/proc.c`) and answer the following questions in your `ans.md` file:
   (a) What happens when a process returns from I/O in terms of state transition?
   (b) What does `void sched(void)` do?
   (c) How does the scheduler select a process for running? (See `void scheduler(void)`)
   (d) What happens when a process is created? (See `static struct proc *allocproc(void)`)
2. Your task is to replace the current scheduling policy with lottery scheduling policy. Extend `struct proc` in `kernel/proc.h` with a new `unsigned int` field `tickets`.
3. Generating a random number makes the scheduling non-deterministic[2] – that's the purpose of lottery scheduling. But for testing purposes, let's make the scheduling algorithm as deterministic as possible: an array `random_int` of 256 integers is provided in `proc.c`. Every time a random integer is needed, get the first unused integer from the array. If all 256 integers are used, then start from the front of the array again. Implement a function named `get_random()` that does the above. You might find defining a variable keeping track of the number of random numbers used helpful.
4. Tickets for each runnable process should be set every time the scheduler picks a new process to run.

   First, implement a function `set_and_get_total_tickets` that sets the number of tickets on each process, and returns the total number of tickets given.

   To generate the tickets for a single process, consider the following code snippet as example:

   ```
   process->tickets = get_random() % 999 + 1; // A number between 1-999
   ```

5. The winning ticket is a **random number** between 1 and the total number of tickets held by the runnable processes returned by `set_and_get_total_tickets`.

   ```
   winner = get_random() % set_and_get_total_tickets() + 1; // A number between 1 to the
       total number of tickets
   ```

[2] Non-deterministic Algorithm Wikipedia Page

6. A sample output from `sanity` for lottery scheduling is shown below. Please note that the non-deterministic factors are minimized by using a fixed array, but not eliminated, so your results may not be the same. That is fine because the results will be different based on the machine that runs the code.

```
$ sanity
PID[12] Creation[200] Termination[518] Ready Time[294] Running Time[24] Sleep Time[0]
PID[10] Creation[200] Termination[547] Ready Time[322] Running Time[25] Sleep Time[0]
PID[19] Creation[210] Termination[577] Ready Time[342] Running Time[25] Sleep Time[0]
PID[5] Creation[196] Termination[626] Ready Time[405] Running Time[25] Sleep Time[0]
PID[9] Creation[196] Termination[585] Ready Time[360] Running Time[25] Sleep Time[0]
PID[15] Creation[208] Termination[630] Ready Time[397] Running Time[25] Sleep Time[0]
PID[21] Creation[210] Termination[622] Ready Time[387] Running Time[25] Sleep Time[0]
PID[22] Creation[210] Termination[634] Ready Time[373] Running Time[26] Sleep Time[0]
PID[7] Creation[196] Termination[660] Ready Time[439] Running Time[25] Sleep Time[0]
PID[18] Creation[208] Termination[644] Ready Time[409] Running Time[25] Sleep Time[0]
PID[20] Creation[210] Termination[645] Ready Time[410] Running Time[25] Sleep Time[0]
PID[6] Creation[196] Termination[667] Ready Time[446] Running Time[25] Sleep Time[0]
PID[13] Creation[200] Termination[669] Ready Time[435] Running Time[26] Sleep Time[0]
PID[4] Creation[195] Termination[671] Ready Time[450] Running Time[25] Sleep Time[0]
PID[17] Creation[208] Termination[676] Ready Time[442] Running Time[26] Sleep Time[0]
PID[11] Creation[200] Termination[681] Ready Time[456] Running Time[25] Sleep Time[0]
PID[16] Creation[208] Termination[692] Ready Time[457] Running Time[27] Sleep Time[0]
PID[23] Creation[235] Termination[702] Ready Time[441] Running Time[26] Sleep Time[0]
PID[8] Creation[196] Termination[707] Ready Time[485] Running Time[26] Sleep Time[0]
PID[14] Creation[208] Termination[712] Ready Time[474] Running Time[30] Sleep Time[0]
Avg. Wait Time[411] Avg. Running Time[25] Avg. Sleep Time[0] Avg. Turnaround Time[438]
$
```

Note that the autograder on Anubis will only test if xv6 builds successfully and if `sanity` does not exit without errors. It is your responsibility to check if your code is working as intended.

Part 4

1. 1.a, 1.b, 1.c, and 1.d from Part 3.
2. What is a major difference between threads and processes?
3. Explain how the shell uses `fork()` and `exec()` to execute an arbitrary command.
4. Briefly explain how the OS switches between processes.

Save your answers in `ans.md`.