ECE492/592 – Operating Systems Design: Project #2

Due date: October 13, 2019

Objectives

- To learn the concepts and methods related to process management such as process creation, process priorities, process scheduling and context switch.
- To understand Xinu's implementation of process management.
- To implement different scheduling algorithms in Xinu and verify their correct operation on representative test cases.

Overview

This project focuses on process management. To help you proceed gradually, the project is divided into three parts.

In Part 1, your task is to learn Xinu's implementation of process management and perform some small coding tasks that will help you debug both Part 2 and Part 3. (**required for all students**)

In Part 2, you will implement Lottery Scheduling (LS) in Xinu. (**required for all students**)

In Part 3 (**required only for students taking the course at the graduate level – ECE592**), you will implement the Multi-Level Feedback Queue (MLFQ) scheduling policy in Xinu.

Before you proceed with your implementation, copy the whole `xinu` folder to a safe directory so to keep a clean version of Xinu. You should first implement Part 1. The code of Part 1 will be the baseline code used for Part 2 and Part 3. So, if you are taking the course at the graduate level, remember to save an extra copy of Part 1 before starting to code Part 2.

- If you are taking the course at the undergraduate level, you will submit a single code incorporating both Part 1 and Part 2.
- If you are taking the course at the graduate level, you will submit two versions of the code: (1) Part 1+2, and (2) Part 1+3.

Part 1: Understanding Xinu's process management and context switch.

The `xinu/system` folder contains the files related to process management and context switch. Study the files related to process creation (`create.c`), process scheduling (`resched.c`), context switch (`ctxsw.S`), process termination (`kill.c`), system initialization (`initialize.c`) and other related utilities (`ready.c`, `resume.c`, `suspend.c`, `chprio.c`, etc.).

Include in your report the answer to the following questions. Be clear and succinct.

Q1. What is the ready list? List all the system calls that operate on it.
Q2. What is the default process scheduling policy used in Xinu? Can this policy lead to process starvation? Explain your answer.
Q3. When a context switch happens, how does the `resched` function decide if the currently executing process should be put back into the ready list?

Q4. Analyze Xinu code and list all circumstances that can trigger a scheduling event.

Preliminary coding tasks:

CT1. Implement the function `syscall print_ready_list()` that prints the identifiers of the processes currently in the ready list. You don't need to document this function in your report.

CT2. Add to the PCB two fields:

- `uint32 runtime` – number of milliseconds the process has been running (i.e., in `PR_CURR` state).
- `uint32 turnaroundtime` – turnaround time in milliseconds.
- `uint32 num_ctxsw` – number of context switch operations *to* the process.

and update the code to keep these fields updated.

CT3. In this project, we will group processes in two categories: *system processes* and *user processes*. In particular, we will call "system processes" the processes spawned by default by Xinu (e.g., `startup`, `main`, `shell`, etc.), and "user processes" the ones used to model user applications. This second category includes the processes spawned by the `main` function. User processes can, in turn, spawn other user processes. Write a `create_user_process` function to explicitly spawn user processes (this function can invoke the standard `create` function internally). The `create_user_process` function should have the following interface (note: it must not include a *priority* parameter).

```
pid32   create_user_process(
      void          *funcaddr,      /* Address of the function      */
      uint32         ssize,         /* Stack size in bytes          */
      char           *name,         /* Name (for debugging)         */
      uint32         nargs,         /* Number of args that follow   */
      ...
)
```

CT4. Instruments the `resched` function so that, when the `DEBUG_CTXSW` constant is defined, it will print all the context switch operations in the following format:

```
ctxsw::<old-process-id>-<new-process-id>
```

To this end, you can use the C preprocessor `#define` and `#ifdef/#endif` directives. Specifically, next to the `ctxsw` call you will have a block of code as below:

```
#ifdef DEBUG_CTXSW
      [statements to print the ctxsw information above]
#endif
```

In order to log the `ctxsw` calls, insert the `#define DEBUG_CTXSW` directive at the beginning of the `resched.c` file.

Note: You should log only real context switch operations (i.e., the old and new process id must differ).

For example, the following is a possible output for 3 user processes and the default Xinu scheduler with the `DEBUG_CTXSW` constant defined.

```
Xinu for Vbox -- version #15 (xinu) Thu 27 Sep 07:08:44 PDT 2018

Found Intel 82545EM Ethernet NIC
MAC address is 08:00:27:9e:f4:08
   4425176 bytes of free memory.  Free list:
           [0x0015FA20 to 0x0009FFF7]
           [0x00100000 to 0x005F7FFF]
    105421 bytes of Xinu code.
           [0x00100000 to 0x00119BCC]
    132712 bytes of data.
           [0x0011CC60 to 0x0013D2C7]

ctxsw::0-2
ctxsw::2-0
ctxsw::0-3
ctxsw::3-0
ctxsw::0-4
ctxsw::4-0
ctxsw::0-3
ctxsw::3-4
ctxsw::4-0
ctxsw::0-3
ctxsw::3-4
Obtained IP address  10.0.3.15   (0x0a00030f)
ctxsw::4-5
ctxsw::5-6
ctxsw::6-4
ctxsw::4-5
ctxsw::5-7
ctxsw::7-5
ctxsw::5-8
ctxsw::8-5
ctxsw::5-0
```

CT5. Write a function

```
void burst_execution(uint32 number_bursts, uint32 burst_duration,
                     uint32 sleep_duration);
```

which simulates the execution of applications that alternate execution phases requiring the CPU (CPU bursts), and execution phases not requiring it (CPU inactivity phases). Specifically:

- `number_bursts` = number of CPU bursts
- `burst_duration` = duration of each CPU burst in milliseconds (all CPU burst have the same duration)
- `sleep_duration` = duration of each CPU inactivity phase in milliseconds (all CPU inactivity phases have the same duration)

Note: burst and sleep duration model the phase behavior of the application. However, the OS scheduler may cause the application to wait for periods of time longer than the sleep duration.

Part 2: Lottery Scheduling

Your goal is to implement Lottery Scheduling in Xinu.

1. Implement the following function to initialize or modify the number of tickets (`tickets`) assigned to a given process (identified by `pid`).

   ```
   void set_tickets(pid32 pid, uint32 tickets);
   ```

   You can include the `set_ticket` function in the same file where you have implemented the `create_user_process` function, and invoke it from `main.c`. By default, a user process should be initialized with no tickets.

2. Utilize the `rand()` pseudo-random number generator from the `stdlib.h` library to generate the random numbers required by the scheduler. Use the default random seed (this is equivalent to invoking `srand(1)`).

3. Modify Xinu's scheduler so to allow lottery scheduling of the "user processes". Note that:
   - System processes must be scheduled with higher priority (using Xinu's default scheduling policy) and not follow the lottery scheduling policy. The null process should run only when there is nothing else that can be scheduled to run. You can test your scheduler by invoking user processes from the `main()` function. If you don't schedule the system processes with higher priority, you won't be able to test your lottery-based scheduler effectively.
   - For simplicity, **you can keep Xinu's scheduling events**, and not limit scheduling decisions to the end of each time slice.
   - The textbook (http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf) provides a pseudocode for lottery scheduling. To make the implementation more efficient, sort the processes by descending number of tickets. If two processes have the same number of tickets, sort them by ascending process id (this is just a convention).
   - Recall that only processes in the ready list are eligible for execution. As a consequence, the tickets of processes that are not part of the ready list should not affect the scheduling decisions.

4. Analyze the fairness of your scheduler. To this end, write a test case file that spawns and runs two user processes with the same runtime, where the runtime is the execution time that the process would have if run from start to finish without preemption. Collect execution data with increasing runtime values. Plot the ratio between the actual execution time of the two processes when run together (i.e., execution-time$_{P1}$/execution-time$_{P2}$, where P1 and P2 are the two processes spawned) against the value of their runtime. You can select the number of data points and the runtime values as you like, provided that your selection leads to a meaningful plot. Include the plot and a brief discussion of it (no more than a couple of sentences) in the report.

Include in the report:

- A *brief* description of your implementation approach, indicating the files involved in the implementation of lottery scheduling.
- The fairness analysis of (4) and the `main.c` file you have written in order to perform that analysis.

Be clear and succinct.

Part 3: Multi-Level Feedback Queue (MLFQ) scheduling policy – (only for ECE592 students).

Your goal is to implement the Multi-Level Feedback Queue (MLFQ) scheduling policy in Xinu. The MLFQ scheduling policy operates according to the following rules:

- *Rule 1*: if Priority(A)>Priority (B), A runs
- *Rule 2*: if Priority(A)=Priority(B), A&B run in RR fashion
- *Rule 3*: initially a job is placed at the highest priority level
- *Rule 4*: once a job uses up its time allotment *TA*, its priority is reduced
- *Rule 5*: after some time period *S*, move all jobs in the topmost queue

Implement the MLFQ scheduling policy in Xinu. Your implementation must follow the following directions:

- As in Part 2, system processes should be scheduled separately (using Xinu's default scheduling policy). In addition, you should keep Xinu's scheduling events, and not limit scheduling decisions to the end of each time slice. In the absence of other scheduling events, you can let a process run until the end of a time slice even if it fully utilizes its time allotment within that time slice. Every time a process is moved to a different queue, its time allotment's utilization should be reset (in other words, the scheduler does not need to keep track of situations where a process overuses its time allotment before its priority is reduced).
- There should be 3 priority levels (i.e., 3 queues) **for user processes**.
- The length of the time slice should increase by a factor 2 when moving from higher to lower priority. The length of the time slice for processes at the highest priority level should be the same as for system processes.
- The time allotment (*TA*) and the priority boost period *S* (all expressed in milliseconds) should be configurable and defined in `include/resched.h` as follows:

  ```
  #define TIME_ALLOTMENT <TA>
  #define PRIORITY_BOOST_PERIOD <S>
  ```

  Think of meaningful values of these parameters (no need to include this in your report).
- When performing priority boost, reset the use of the time allotment for *all* processes. In addition, enqueue the processes in order of priority. For example: if before priority boost HPQ = {1, 5}, MPQ = {2, 6}, LPQ = {4, 3}, after priority boost HPQ = {1, 5, 2, 6, 4, 3} (HPQ, MPQ, LPQ being high, middle and low priority queue, respectively).

Use the `burst_execution` function to create use cases to validate the correct operation of your scheduler.

Include in the report: a succinct description of your implementation approach, indicating the files involved in the implementation of the MLFQ scheduling policy.

Submissions instructions

1. **Important:** We will test your implementations using different test cases (i.e., different `main.c` files). Therefore, do not implement any essential functionality (other than your test cases) in the `main.c` file. Also, turn off debugging output before submitting your code.
2. **Suggestion:** You have 3 weeks to complete this programming assignment. Complete one "Part" each week. In week 1, besides completing Part 1, read the whole assignment and assess the time you will required to complete it given your programming skills.

3. Go to the `xinu/compile` directory and invoke `make clean`.
4. As for the previous project, create a `xinu/tmp` folder and **copy** all the files you have modified/created into it (the `tmp` folder should have the same directory structure as the `xinu` folder).
5. Go to the parent folder of the `xinu` folder. You will submit Part 2 and Part 3 separately (the code of Part 1 should be incorporated in both Part 2 and Part 3). For each part (2 & 3), compress the whole `xinu` directory into a *tgz* file.

```
tar czf xinu_project2_part#.tgz xinu
```

6. Submit your assignment – including *tgz* files and report – through Moodle. Please upload only one *tgz* file **for each Part**. There is no need to print the report and bring it to class.

Grading rubric

		ECE492 students	**ECE592 students**
Part 1	*Q1-Q4*	8	5
	CT1	5	3
	CT2	3	2
	CT3	1	1
	CT4	1	1
	CT5	5	3
Part 2	*Implementation of lottery scheduling*	55 (20 for good effort)	35 (10 for good effort)
	Description of the implementation	10	5
	Analysis of fairness	12	5
Part 3	*Implementation of MLFQ*		35 (10 for good effort)
	Description of implementation		5