ECE592 – Operating Systems Design: Homework #2

Due date: October 3, 2017

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

In Part 1, your task is to learn Xinu's implementation of process management.

In Part 2, you will implement Shortest Time-to-Completion First (STCF) scheduling policy in Xinu.

In Part 3, you will implement a Multi-Level Feedback Queue (MLFQ) scheduling policy in Xinu.

Before you proceed with your implementation, please copy the whole `xinu` folder to a safe directory so to keep a clean version of Xinu. **You should start each part of this homework with a clean version of Xinu**.

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.

1. What is the ready list? List all the system calls that operate on it.
2. What is the default process scheduling policy used in Xinu? Can this policy lead to process starvation? Explain your answer.
3. When a context switch happens, how does the `resched` function decide if the currently executing process should be put back into the ready list?
4. Explain what a stack frame is and why it is needed during a function call.
5. The function `ctxsw` takes two parameters. Explain the use of these parameters. Which assembly instruction(s) set(s) the Instruction Pointer register to make it point to the code of the new process?
6. Analyze Xinu code and list all circumstances that can trigger a scheduling event.

Coding tasks:

1. 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.
2. If your answer to question #2 above is affirmative, write a `main.c` that implements a use case demonstrating the problem. Explain your use case and code in the report.

Part 2: Shortest Time-to-Completion First (STCF) scheduling policy

Your goal is to implement the Shortest Time-to-Completion First (STCF) scheduling policy in Xinu.

The time-to-completion of a process is defined as the difference between the estimated total processing time of the process and the CPU time already allocated to it.

As you know, the assumption that a scheduler knows the total processing time of a process *a priori* is often unrealistic and not generally applicable. However, in this exercise you will simulate tasks with a predefined execution time, allowing the scheduler to keep track internally of the time-to-completion of each process.

1. Write a custom process creation function `create_user_proc` that allows the creation of *user processes* with a predefined execution time `run_time` and different CPU usage patterns. This process creation function does not require explicitly assigning the process a priority. Except for the additional `run_time` parameter, `create_user_proc` should be invoked similarly to the `create` system call, and it should be declared as follows:

   ```
   pid32 create_user_proc(void *funcaddr, uint32 ssize, uint32 run_time,
                          char *name, uint32 nargs, …);
   ```

   The CPU usage pattern of the user process will depend on that of the function that it executes. Implement the following function:

   ```
   void timed_execution(uint32 run_time);
   ```

   to simulate a CPU intensive function executing for `run_time` milliseconds.

   Implements both `create_user_proc` and `timed_execution` within the same file, that you will name `create_user_proc.c`.

   Please note the following:

   - `run_time` must represent an estimate (which can be off by a clock timer interval) of the total time effectively required by the process to execute. In other words, `run_time` does not represent the difference between the completion and the creation time of a process, but rather the total time required by the function executed by it to complete if never preempted.
   - *Hint:* in order to implement timed user processes, you might need to modify portions of Xinu code other than the newly created `create_user_proc.c` file.

- You will use the `create_user_proc` function only to generate *user processes* with predefined execution time. *System processes* created by default by Xinu (e.g., `startup`, `main`, `shell`, etc.) should be created using the original `create` system call.

2. Modify Xinu's scheduler so to allow STCF scheduling of the timed processes. Please observe the following:
   - Since system processes are not timed, they must be scheduled with higher priority and not follow the STCF policy. You can test your scheduler by invoking user processes from the `main()` function (or from the shell). If you don't schedule the system processes with higher priority, you won't be able to test your STCF scheduler effectively.
   - In your implementation, you don't need to perform scheduling decisions only upon a new process creation (the way we have seen in class). You can keep the scheduling events used by Xinu.
   - User processes with the same time-to-completion can be scheduled in a round-robin fashion.

3. Write a representative test case for your STCF scheduler. To this end, write your own `main.c` file including invocations to the `create_user_proc` and `sleep` system calls. *Note*: you don't need to run the shell.

The execution should print out the following scheduling information **for all user processes** (**do not** include the system processes): (1) creation time, (2) completion time, (3) start and end time of each running cycle. Please use the following template (note: the output should not contain any spaces/tabs).

```
P<pid>-creation::<creation-timestamp(ms)>::<runtime(ms)>
P<pid>-termination::<termination-timestamp(ms)>
P<pid>-running::<start-timestamp>-<end-timestamp(ms)>
```

For example, the following interleaving

```
P8-creation::9::100
P8-running::10-30
P9-creation::32::20
P9-running::40-59
P9-termination:59
P8-running:60-128
P8-termination:128
```

could be produced in the presence of two user processes: one (pid=8) with a 100ms runtime, and the other (pid=9) with a 20 ms runtime (ignore the exact value of the timestamps shown – this example is just to show the format of the output).

Important – include in the report:

- A description of your implementation approach, indicating the files involved in the implementation of the STCF scheduling policy.
- A description of your test case, indicating whether its outcome is as you expected.

Please be clear and succinct.

Part 3: Multi-Level Feedback Queue (MLFQ) scheduling policy

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_L at a given level, its priority is reduced
- *Rule 5*: after some time period *S*, move all jobs in the topmost queue

As in Part 2, let's call *system processes* the processes created by default by Xinu (e.g., `startup`, `main`, `shell`, etc.), and *user processes* the other processes (which simulate processes spawned by the application).

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

- As in Part 2, system processes should be scheduled with higher priority (otherwise, they might interfere with the creation of user processes). There are different ways to implement this. One option is to have a separate, high-priority queue (not obeying the rules above) for system processes.
- There should be 4 priority levels (i.e., 4 queues) for user processes.
- The time allotment at the highest priority level (TA_{HPL}) and the priority boost period *S* (both expressed in milliseconds) should be configurable and defined in `include/resched.h` as follows:

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

  Please make sure that this constant definition is in `include/resched.h`.

- The time allotment should decrease by a factor 2 from level to level (i.e, $TA_{L-1} = ½\ TA_L$)

2. Write a representative test case for your STCF scheduler. To this end:
   - write your own `main.c`
   - write the following function to simulate interactive processes, and use it appropriately in your test case

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

   This function 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 (all CPU burst have the same duration)
   - `sleep_duration` = duration of each CPU inactivity phase (all CPU inactivity phases have the same duration)

Again, the execution should print out the following scheduling information **for all user processes** (do not include the system processes): (1) creation time, (2) completion time, (3) start and end time of each running cycle. Please use the same template as for Part 2.

As for Part 2, include in the report:

- A description of your implementation approach, indicating the files involved in the implementation of the MLFQ scheduling policy.
- A description of your test case, indicating whether its outcome is as you expected.

Please be clear and succinct.

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. Create a `xinu/tmp` directory and copy all the files you have modified/created into it (As in homework#1, maintain the `tmp` directory structure as the `xinu` folder). Note that the use of the `tmp` folder aims to help the TA quickly identify what files have been modified. So, make sure these files still have their original versions in corresponding folders so that the Xinu can be successfully compiled.
5. Go to the parent folder of the `xinu` folder. For each part, compress the whole `xinu` directory into a *tgz* file.

   ```
   tar czf xinu_homework2_part#.tgz xinu
   ```

6. Submit your assignment through Moodle. Please only upload only one *tgz* file **for each Part**.
7. Print your report and bring it to class on the due date.