# ECE492/592 – Operating Systems Design: Project #2

**Due date: October 18, 2018**

## 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. (**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 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 project 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: Lottery Scheduling

Your goal is to implement Lottery Scheduling in Xinu.

As you know, the assumption that a scheduler knows the total execution 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.

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.

   Implement both `create_user_proc` and `timed_execution` within the same file, and name the file `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. In this project, you can rely on Xinu's interrupt-based timer to make time measurements.
   - *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. Include in the `create_user_proc.c` file a function:

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

   to initialize or (dynamically) modify the number of tickets (`tickets`) for a given process (identified by `id`).

3. Utilize the `rand()` pseudo-random number generator from the `stdlib.h` library to generate the random numbers required by the scheduler.

4. Modify Xinu's scheduler so to allow lottery scheduling of the "user processes". Note that:
   - System processes must be scheduled with higher priority and not follow the lottery scheduling 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 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.
   - User processes with the same number of tickets can be scheduled in a round-robin fashion.

The execution should print out, **for all context switch operations performed**, the identification of the old and new running processes. *Suggestion*: add the required instructions next to the invocation of `ctxsw` function. Use the following format (note: the output should not contain any spaces/tabs).

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

For example, the following is a possible output for 3 user processes and the default Xinu scheduler.

```
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
```

5. Analyze the fairness of your scheduler. To this end, write a test case file that spawns and runs two user processes with the same `run_time` parameter value. Run your test case multiple times using increasing `run_time` 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 parameter `run_time`. You can select the number of data points and the value of the `run_time` parameter as you like, provided that you make a selection that leads to a meaningful plot. Include the plot and a brief discussion of it (no more than a couple of sentences) in the report.

**<u>Include in the report</u>**:

- A *brief* description of your implementation approach, indicating the files involved in the implementation of lottery scheduling.
- The fairness analysis of (5).

Please be clear and succinct.

## Part 3: Multi-Level Feedback Queue (MLFQ) scheduling policy – <u>(only for ECE592 students)</u>.

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). In addition, you can keep Xinu's scheduling events, and not limit scheduling decisions to the end of each time slice.

- There should be 3 priority levels (i.e., 3 queues) **for user processes**.

- The time slice should increase by a factor 2 when moving from higher to lower priority (i.e., $TS_{L-1} = 2\ TS_L$).
- The time slice at the highest priority level ($TS_{HPL}$), 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_SLICE <TSHPL>
#define TIME_ALLOTMENT <TA>
#define PRIORITY_BOOST_PERIOD <S>
```

- System processes should be scheduled using $TS_{HPL}$ (e.g., the same time slice as highest priority user processes).

2. Write the function

```
void burst_execution(uint32 number_bursts, burst_duration, 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 (all CPU burst have the same duration)
- `sleep_duration` = duration of each CPU inactivity phase (all CPU inactivity phases have the same duration)

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

Again, the execution should print out, **for all context switch operations performed**, the identification of the old and new running processes. Use the same template as for Part 2.

**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. For each part, 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.