Jacob Little

ECE 565 (001)

Instructor: Dr. Michela Becchi

Project 2 Report

Introduction

This report discusses the implementation of Lottery Scheduling and Multi-Level Feedback Queue (MLFQ) Scheduling of user processes in the Xinu operating system as well as answers to the questions presented in the project specification.

Question Answers

Q1. The `readylist` is a queue that holds the processes that are in the `PR_READY` state. They are ordered by priority (highest priority first) making it easy to compare and remove the highest priority processes first during scheduling. The `readylist` is referenced in `sysinit()` where it is first initialized, in `ready()` where it is used when a process enters the `PR_READY` state, and in `resched()` where it is used for comparing the currently running processes priority against the first item in the `readylist` and also for swapping the currently running process and the first item in the `readylist` when a higher priority process is in the `readylist`.

Q2. Xinu uses a priority based round-robin scheduling policy. This policy can lead to process starvation in the event that higher priority processes keep taking up all the runtime and never give a lower priority process the chance to execute because it will always be neglected for higher priority processes.

Q3. The `resched` function determines whether the currently running process has a higher priority than the first process in the `readylist` (since the `readylist` is ordered in descending priority, only the first process needs to be checked). If the currently running process has a higher priority, then it will continue to run and not be inserted into the `readylist`. But if the currently running process has a priority that is less that OR equal to the first process in the `readylist`, then it shall be inserted into the `readylist` and hand over execution to the first process in the `readylist`. Note that because the currently running process will change in the case of equal priority with the first item in the `readylist`, round-robin execution is enabled because any number of processes in the readylist with the same priority will be removed and reinserted in a round-robin fashion with each time slice.

Q4. All circumstances in which a scheduling event can occur:

- **system/**
  - > **rdssetprio.c** - `rdssetprio()` calls `resched()` after changing the priority of the currently running process.
  - > **clkhandler.c** - `clkhandler()` calls `resched()` once every amount of time set by `QUANTUM` passes for the sake of regular rescheduling.

- > **kill.c** - `kill()` calls `resched()` when a process kills itself.
- > **ready.c** - `ready()` calls `resched()` when a process enters the `PR_READY` state.
- > **receive.c** - `receive()` calls `resched()` when a process blocks for a message.
- > **recvtime.c** - `recvtime()` calls `resched()` when a process blocks a set amount of time for a message.
- > **resched.c** - `resched_cntl()` calls `resched()` when all defer requests have been stopped and the most recent defer attempt was successful.
- > **sleep.c** - `sleepms()` calls `resched()` when a process enters the `PR_SLEEP` state for a set amount of time.
- > **suspend.c** - `suspend()` calls `resched()` when a process enters the `PR_SUSPEND` state.
- > **wait.c** - `wait()` calls `resched()` when a process begins waiting on a semaphore.
- > **yield.c** - `yield()` calls `resched()` when a process yields voluntarily relinquishes the CPU.

P2. Lottery Scheduling

This problem involved modifying:

- **include/**
  - > **clock.h** - added ctr1000 for counting milliseconds.
  - > **kernel.h** - changed `QUANTUM` to 10 ms as per the project specification.
  - > **process.h** - added elements to the `procent` struct for `creationtime`, `runtime`, `turnaroundtime`, `num_ctxsw`, `user_process`, `tickets`.
  - > **prototypes.h** - added a few custom function prototypes for use in multiple files.
  - > **queue.h** - added inline definition for determining if a queue has only one element. Also updated `NQUENT` to account for `lotterylist`.
- **system/**
  - > **burst.c** - implemented a function that does a number of burst executions and sleeps in-between those bursts.
  - > **clkhandler.c** - increment `ctr1000` and `runtime` of the current process every millisecond.
  - > **clkinit.c** - initialize ctr1000 to 0 ms.
  - > **create.c** - added `create_user_process()` function which is identical to `create()` but has no priority argument and sets the `user_process` flag to `USER` instead of `SYSTEM`. User processes are also initialized to a static priority that is less than `INITPRIO` but

> greater than 0 (which is the priority of the null process). Also made a function to set the number of tickets for a process.

- > **initialize.c** - initialize new elements in `procent` for null process and initialize the `lotterylist` to hold user processes in the lottery.
- > **kill.c** - calculate `turnaroundtime` before process ends.
- > **ready.c** - insert user processes into the `lotterylist` instead of the `readylist`.
- > **resched.c** - implemented lottery scheduler.

In the `resched()` function, a few circumstances have to be considered:

- Old process is in the `PR_CURR` state
  - Old process: SYSYTEM -> New Process: SYSTEM
  - Old process: SYSYTEM -> New Process: USER
  - Old process: USER -> New Process: SYSTEM
  - Old process: USER -> New Process: USER
- Old process is NOT in the `PR_CURR` state
  - New Process: SYSTEM
  - New Process: USER

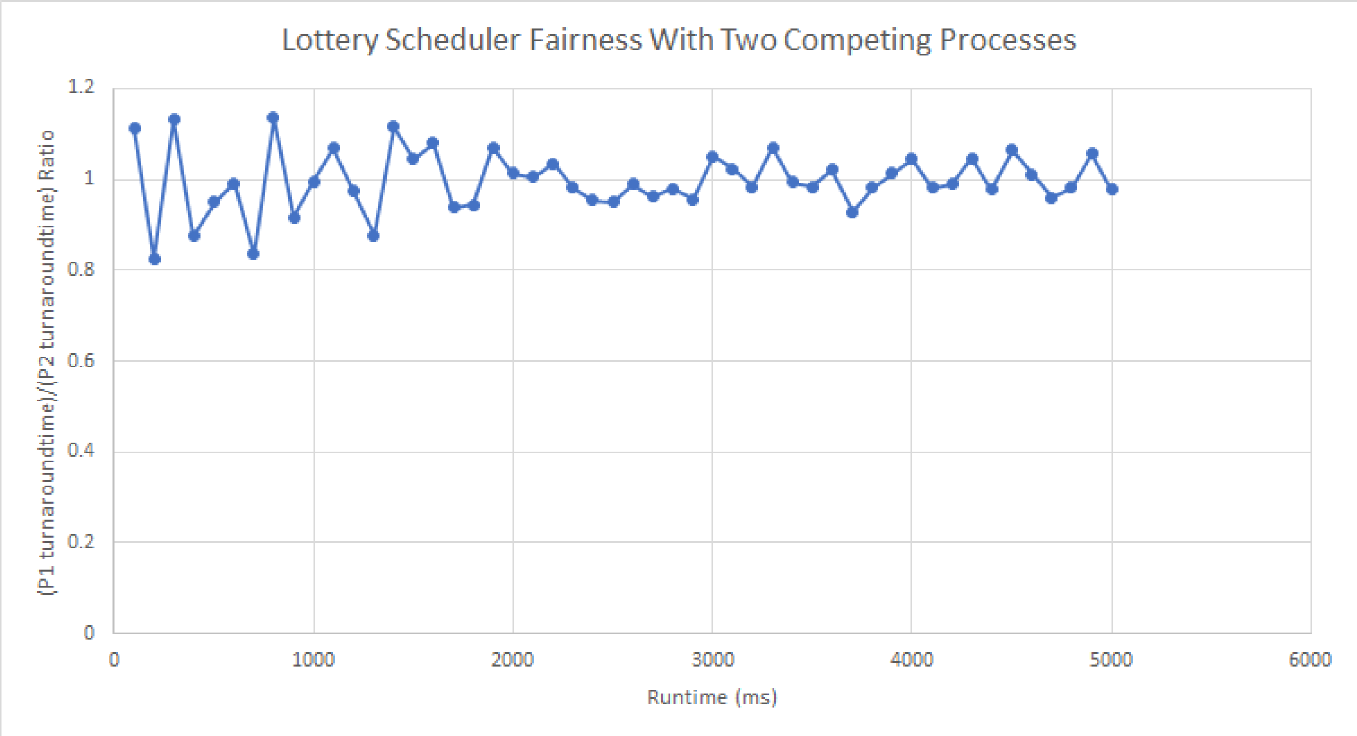
Each circumstance must be handled differently. For instance, if the old process is SYSTEM and the new process is USER, we need to put the old process back in the `readylist` (this will only happen for the null process) and we need to check the `lotterylist` (and do a lottery if there is more than one item in it) to determine new process.

User processes are only allowed to run when the null process would normally be the next process to run. This is done by checking if the new process has the pid `NULLPROC` and also checking if the `lotterylist` is nonempty. User processes also have a priority greater than the null process but less than `INITPRIO` so user processes will always be preempted by system processes.

A `lottery()` function is defined that returns the pid of the new process if a lottery is to be run (or if there is only one process in the lottery list, returns the pid of that process). Before performing the lottery, it tallys up the total number of tickets held by processes in the `lotterylist`, and then uses the project specification for iterating through the `lotterylist` to determine the winner.

Fairness Evaluation

A testcase was made that spawns two competing processes with increasing runtimes starting with 100 ms per process and ending with 5000 ms per process. This is a graph showing the ratio of the two processes' turn-around-times with increasing individual process runtime:



As we can see, as the indiviual process runtime increases, averaging takes effect and reduces the difference in turn-around-time between the two processes.

P3. Multi-Level Feedback Queue (MLFQ) Scheduling

This problem involved modifying:

- **include/**
  - > **clock.h** - added ctr1000 for counting milliseconds.
  - > **kernel.h** - changed `QUANTUM` to 5 ms as per the project specification.
  - > **process.h** - added elements to the `procent` struct for `creationtime`, `runtime`, `turnaroundtime`, `num_ctxsw`, `user_process`, `timeallotment`.
  - > **prototypes.h** - added a few custom function prototypes for use in multiple files.
  - > **queue.h** - Updated `NQUENT` to account for multi-level feedback queues.
  - > **resched.h** - added definitions for `TIME_ALLOTMENT` and `PRIORITY_BOOST_PERIOD`. Also added counters to keep track of priority boost and time slices for the medium priority queue and low priority queue.
- **system/**
  - > **burst.c** - implemented a function that does a number of burst executions and sleeps in-between those bursts.
  - > **clkhandler.c** - increment `ctr1000`, `runtime` and `timeallotment` counter of the current process, the priority boost counter, and for current process every millisecond. A flag is also set to indicate that an asynchronous scheduling event has NOT occured.

- > **clkinit.c** - initialize ctr1000 to 0 ms.
- > **create.c** - added `create_user_process()` function which is identical to `create()` but has no priority argument and sets the `user_process` flag to `USER` instead of `SYSTEM`. User processes are also initialized to a static priority for the high priority queue that is less than `INITPRIO` but greater than 0 (which is the priority of the null process).
- > **initialize.c** - initialize new elements in `procent` for null process and initialize the high priority, medium priority, and low priority queues to hold user processes that are ready. Also initialize the priority boost counter and lower priority queue time slice counter.
- > **kill.c** - calculate `turnaroundtime` before process ends.
- > **ready.c** - insert user processes into the the multi-level feedback queue that corresponds with it's current priority instead of the `readylist`.
- > **resched.c** - implemented MLFQ scheduler.

In the `resched()` function, the circumstances from Lottery still hold (SYSTEM->SYSTEM, SYSTEM->USER). The individual circumstances are handled identically to lottery.

User processes are only allowed to run when the null process would normally be the next process to run. This is done by checking if the new process has the pid `NULLPROC` and also checking if all multi-level feedback queues are nonempty. User processes also have a priority cooresponding to their priority queue that is greater than the null process but less than `INITPRIO` so user processes will always be preempted by system processes.

A `mlfq()` function is defined that returns the pid of the new process. As long as there are processes in a higher priority queue, those will run in a round-robin fashion. If a priority queue is empty, the next level down is checked. Each time slice, the time allotment for a process is checked and the process is demoted to the a lower level queue if its `timeallotment` counter exceeds `TIME_ALLOTMENT` (the process's `timeallotment` counter is also reset to 0). `timeallotment` of a process is not considered at the lowest priority queue.

Also, the time slice used for MLFQ scheduling is doubled as the priority level decreases. This is done by letting a user process continue it's execution (not be context-switched out of) if it is the same priority as the current level queue and a priority queue counter is not a multiple of 2 for the medium priority queue or a multiple of 4 for the low priority queue. The priority queue counter is reset to 0 in the event of an asynchronous sheduling event.

When the priority boost counter exceeds `PRIORITY_BOOST_PERIOD`, the `timeallotment` counters are reset for all user processes and the lower level queues are dequeued onto the high priority queue.