CS 140 MACHINE PROBLEM

Vincent Paul F. Fiestada

Introduction

This project is an implementation of a simulated process scheduler where “dishes” act as processes, and the “stove” acts as the CPU. In deciding the scheduling algorithm, I considered the Dish Scheduling Problem as a proxy for standard process scheduling in operating systems. Taking this into consideration, I decided to use a **Multiple Feedback Queue** for the scheduling.

Usage

The Dish Scheduler is implemented in C++, compiled and tested with the GNU C Compiler (GCC). If you already have GCC on your computer, you can go to the project directory and run the following command.

```
g++ scheduler.cpp -o mp.exe
```

Please note that the “gcc” command will fail to compile a C++ file. If you are running Windows 64-bit, there is a pre-compiled binary file included in the project directory, “mp.exe”. Some other files are described in the table below.

Filename	Purpose
output.csv	The output of the last simulation run
perf.log	Performance metrics from the last simulation run
mp.exe	(Default) compiled binary
scheduler.cpp	Main project file (Other files are included from this one)
recipes\ (Directory)	Contains all recipe files.

Goals and Objectives

In deciding the scheduling algorithm for the project, I wanted to treat it as a general process scheduling scenario. This was even though some facts about the problem specifically that offered opportunities for optimizing the scheduling beyond what would be practical in a computer process scheduling scenario. However, this implementation treats the kitchen scheduler as an exercise in the practice of theory. It ignores the fact that, for instance, that dishes have fixed recipes which would disable the need to guard against cheating of the algorithm.

This scheduler uses a version of the **Multiple Feedback Queue (MLFQ)**. Versions of this approach are used in many modern operating systems, including Windows NT and Solaris. Mac OS X uses a cooperative scheduling scheme based on MLFQ. The scheduler should hopefully achieve the following goals:

- A fast and simple scheduling algorithm that **adapts minimally** to the behavior of processes (dishes).
- Must not rely on the assumption that the what the process (dish) will do is fully known beforehand. In short, schedule intelligently **without perfect knowledge**.
- Allow lower-priority, interactive processes their "fair" share of the CPU and are not penalized for it.
- Ensure that higher-priority processes are accorded that higher priority in scheduling, but do not dominate or starve out lower-priority processes.
- Give processes with the same priority (almost) equal footing with the scheduler.
- Resolve the negative effect of aging on older processes.

Major Constraints

The following are the major limitations in the MLFQ approach to scheduling and to this implementation, specifically.

- The non-reliance on *a priori* knowledge about a process is helpful, and even necessary, in computing. But in a kitchen, it can be ignored since recipes are fixed and dishes cannot change their recipe actively.
- In extreme cases, a programmer can "game" the scheduling algorithm by making a process seem non-interactive 99% of the time and then suddenly, unexpectedly, become interactive in order to avoid aging. There is a very simple solution to this called a **Priority Boost**. But for the sake of simplicity and since dishes aren't expected to "game" the scheduler, it is not implemented in this project.
- Depends heavily on its parametrization, including the number of priority levels, corresponding CPU Burst quanta, and other values that cannot be easily decided. In short, it is **vulnerable to Ousterhout's Law** of arbitrary constants and defaults.

Algorithm Design

This implementation is based primarily on two descriptions of the algorithm, namely: "Scheduling: The Multi-Level Feedback Queue" by Arpaci & Dusseau, and the article on Wikipedia about the same. It has been tweaked but follows the general characteristics of MLFQ.

Data Structures Used

Suppose processes can have a priority of up to N. Set up a multi-level queue composed of N queues (one for each priority level). At any given time, a process that has already arrived and has not been terminated yet is in one and only one queue. A process can be moved from queue to queue. A process can only change position in a queue if it is first removed and then reinserted. Hence, if a process remains in the same queue, it can only move from the front to the back.

Selection

The following rules are followed in scheduling:

1. When a process with priority X arrives, it is pushed into queue X.
2. Processes in higher level queues are always run before lower processes.
3. Processes in the same level alternate in round robin.
4. A process selected to run is given a fixed quantum for CPU time. This quantum is dependent on the priority of the process. Higher priority processes have shorter quanta, lower priority

have longer quanta since they are selected less often. (This is where the issue of Ousterhout's Law arises).

5. When a process gives up control of the CPU or if its quantum is over, its **priority is lowered**. It is re-inserted into the next lower queue.
6. After a process completes an IO Burst (Preparation step), its **priority is increased**. It is re-inserted into the next higher queue unless it is already in the highest queue.
7. Once a process reaches the base or lowest level queue, it does not get demoted.

Pseudo Code

The following shows some pseudo code for the scheduling, including the most important bits about selection, promotion, and demotion.

```
sub schedule(queues)
      ▷demote last scheduled process
      if (stove  != null)
            dishes[stove].demote()
      end
      ▷Assume all processes have arrived
      ▷Select a process starting from the highest level queue
      for i := N to 1
            if (queues[i].size() > 0)
                  stove = queues[i].pop()
                  quantum = generateQuantum(dishes[stove].priority)
            else if (i == 1) ▷All empty
                  CPU_Idle()
            end
      end
end
sub prep(id)
      ▷ .. Prepare dish, do stuff
      ▷ After a prep step, promote a dish
      dishes[id].promote()
end
```

Performance Metrics

The following are some basic performance metrics regarding this particular implementation of MLFQ. In the examples shown, I used the following recipe files. Each recipe has 5 steps and each dish arrives within 5 time units of each other. The total time required for each dish is the same (102).

Process/Dish	Priority	Total CPU/Cook Time	Total IO/Prep Time
adobo	3	90	12
tinola	10	12	90
karekare	5	51	51

Best Case Scenario

The best case performance happens when the task list is monotonic and when all tasks are high-priority and highly IO-bound. In this scenario, the total time is lower than average (147). Stove utilization time and stove idle time are not good, but that is actually excusable because IO-bound or high-priority processes are expected to be interactive and not use the CPU all that much. The weighted average waiting time is impressive, though, at 46.6667.

Worst Case Scenario

The worst case scenario in MLFQ also arises when the task list is monotonic and when all tasks are low-priority and highly CPU-bound. When this is the case, the average waiting time is high and the turnaround time is also high.

For instance, when cooking three adobo's in succession, the total simulated time becomes much higher than expected (324). The stove utilization time is 270 with an idle time of 54. But the weighted average waiting time is also too high (297.095).

Average Case Scenario

The average case performance of the MLFQ scheduler is achieved when the following conditions are met. These conditions have been observed in the past to be generally true for most processes (Arpaci & Dusseau, 2014). This average case performance is also usually the norm as most systems will have a more or less balanced mix of low-priority and high-priority processes.

1) The number of priority levels is appropriate
2) Long CPU-bound Processes have lower priority
3) Short, Interactive (IO-bound) Processes have higher priority
4) There is a (balanced) mix of CPU-bound and IO-bound processes.

In this performance scenario, the stove idle time is reasonably small, stove utilization is maximized, the waiting time of processes is very short, and the overall simulation time to finish all processes/dishes is average.

A simulation run, for example, with adobo, tinola and karekare dishes arriving in that order, produces the following results: total time of 184, stove utilization of 153, stove idle time of 31, and a weighted average waiting time of 102.231. Minimal changes to the recipe files produce roughly the same result: high stove utilization and a short waiting time, as long as the four conditions above are met.

The finishing order of the dishes is also ideal. Tinola finished first, which was ideal because it was an IO-intensive, hence interactive process and should have a shorter turnaround time. Karekare finished next (quite close at tinola's heels), which was expected because it had lower CPU Time and

IO Time required but had an average priority. Adobo finished last which was acceptable because it was CPU-bound.

The average runtime, which is actually ideal since it is typical of modern systems and allows MLFQ to deliver decent performance, can be shown in the following graph. (Credits to Arpaci & Dusseau)