Process State Modeling

Operating Systems Assignment 4

Skyelar Craver & Steven Pitts

April 16 2019

By submitting this assessment, we hereby declare that we have neither given nor received any unauthorized help or used any unauthorized materials during this assessment, and that we have adhered to any additional policies regarding Academic Honesty set by Wentworth Institute of Technology.

[illegible] [illegible]

Part 1

Objective

To better understand how Operating Systems could manage multiple processes, a simulator was designed to model managing multiple processes. There are many variables to consider when designing such a scheduler. When to swap processes, and further, how many to swap in when a swap is triggered, could be critical to the optimum performance of the scheduler. Thus, an experiment was designed to determine the best combination of these variables worked the most efficiently for the simulator previously designed.

Methodology

1. Simulator will be retrofitted with ways to change when and how many processes are swapped
2. Simulator will also need to be updated with a system to simulate the latency and timing of actions to get a more accurate result
3. Test cases created to create workloads that may be realistic.
    a. One input file attempted to model a system booting and requiring a user logging in, and many processes waiting to be started at system up
    b. Another input file attempted to model a user utilizing two programs, such that two groups of processes were being swapped to and from
    c. The final input file modeled a heavily multithreaded GUI application. Each button would be assumed to run in a separate thread, and as a result interrupts would be random
4. The test cases for the experiment will be tested on replacement percentages of 80%, 90%, and 100% as well as testing one or two process swaps when a process swap is triggered.
5. The latencies reported by the simulator will be recorded both separately from each test case, and cumulatively by way of actual completion time using the 'time.h' library's clock() function.
6. Plot the simulation time for each simulation variable tested on a bar plot to visualize the speeds of the different variable combinations across the test cases
7. Compare the findings to a hypothesis and discuss the performance of the best and worst variable combinations with respect to different use cases based on the user models of the test cases.

Assumptions

- For each combination of replacement percentage, and number of swaps to make on replacement, there exists some realistic experiment where that combination is fastest.
- A completely random set of process operations may not necessarily be representative of a typical workload in a real operating system.

Part 2

Code Changes

The simulator used from Lab 2 had some minor bugs, such as incorrectly cutting off the 20th process by a malformed for loop. Another bug made new processes never correctly initialized. After the bugs in the simulator were patched, new code was added to prompt the user for percentage and number of processes to swap. These prompts were made interactive and would reject invalid responses. Additionally, some parts of code were further modularized out into functions for a more readable main.

```
int get_need_to_swap(int percent)
{
    int total_processes = 0;
    int blocked_processes = 0;
    for (int i = 0; i <= 20; i++)
    {
        // number of runnable processes varies by test case
        // simultaneously count runnable processes and blocked
        int blocked = (processes[i] == Blocked);
        total_processes += (blocked || processes[i] == Ready);
        blocked_processes += blocked;
    }
    // get the proportion using an integer division
    blocked_processes *= 100;
    int proportion_blocked = blocked_processes / total_processes;
    return (proportion_blocked >= percent);
}
```

Code example 1: function added for adjustable percentage process swapping

The most significant change made in the logic of the simulator was to replace the swapping logic to accommodate the user specified percentage and number to swap parameters.

An additional change made was to add a step to dispatching a process, such that if the targeted process was in a Suspended state, the swapping algorithm would be ran until the target process was swapped out with another process first. This would end up being one of the chief causes of latency added in the experiment.

Prediction

Different testing variable combinations should be better suited to some tasks, and less suited toward others. For this reason, the expectation would be that no one tested combination would be fastest for all test inputs. Tests that include quickly swapping between sets of inputs would logically do better with a lower required percentage. A workload where processes are ‘flowing’ through the system such that they are rarely revisited. From this, and the fact that the tests to be used cover these two use cases, the natural conclusion is that the 90% replacement would be the most successful. In terms of one versus two swaps, in a more realistic simulator double swaps would be expected to outperform single swaps. Unfortunately, the way process swaps are implemented in the simulator to be used is inefficient. As a result, the expectation is that single process swaps will be faster.

Formula

The simulator itself is programmed using these conditions. As such, the actual latency of execution should be enough for testing the hypothesis – so long as compiler optimizations are disabled to remove variance.

Part 3

Data

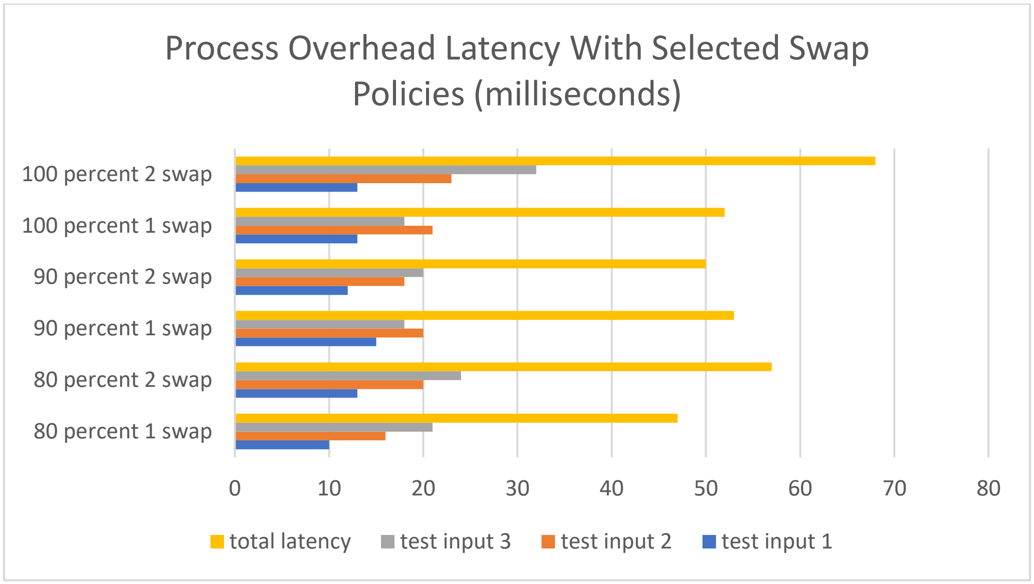

Figure 1: plotted overhead latencies of tests

(raw output included in submission)

Analysis

In many cases, missed processes were the same for many of the tests. The largest differentiators were the latency from bringing over multiple processes in a swap, and whether the swap policy caused a process to be swapped in before it was needed.

Part 4

Conclusion

The fastest overall combination was an 80% replacement margin, and a single process swap model. The slowest overall was the 100% replacement policy with double swapping. The fastest overall was not the fastest in all individual trials, with 90% replacement performing better with test input 3. Double swaps performed generally worse. In most cases the double swap not only increased latency by taking the time to perform the swap, but in some cases swapping out a process that is used sooner than the process it swapped in. Predictions line up well with these results, although the guess of 90% replacement producing the lowest latency was incorrect. A more efficient swapping algorithm would likely cause different results.