COS 431 Project 3 Report

Matthew Stickney

13 November 2010

1 General Questions

1.1 Q1

Why would the following program halt?

```
FK
FK
FK
FK
H
```

The program should always halt. Suppose the machine possesses unlimited memory; that is, a fork operation never fails. The first process will execute the first FK operation, creating a new child process. Both of these processes now have IC set to 1. Continuing this way, both processes now execute the second FK operation, creating two more processes. Each FK operation doubles the number of processes; with 4 operations, that creates $2^4 = 16$ processes, each of which then halts. Given that each process only needs 5 words of memory, the total memory consumption of this program is $5*16 = 80$ words, well within the machine's limitations. Even if memory were exhausted, forks would fail and the program would still halt. Since there are no loops, and forked processes always start execution at the next instruction, there is no case where this program fails to halt (take that, halting problem!).

1.2 Q2

The first-fit scheme comes with the advantage that it is a bit faster than best-fit, but comes at the cost of leaving more memory in a fragmented state. On the other hand, best-fit leaves less memory fragmented, but the fragments are much smaller, making them less likely to be usable.

2 Implementation-Specific Questions

2.1 Q1

Measuring the number of context switches in the dining philosopher's problem is not directly possible, since a successful solution never halts. Instead, at each context switch, we output the number of operations and context switches that have occurred. We then measure the ratio of operations to context switches, which gives a good indication of the context switching overhead incurred by the different programs.

In general, performance of the programs from Project 2 is about the same as it was in the previous project (despite some scheduler bugfixes). Most programs show little deviation in the ration of operations to context switches as granularity is changed, though the pipelined sort program seems to incur less overhead as granularity increases. The standard deviation for each program is less than 0.5, and the average ratio for each program ranges between 2 and 2.5 (with the exception of the pipelined sort, which has a significantly higher average ratio of 4.54).

The low deviation of ratios with different granularities is most likely the result of making heavy use of message passing. Since each message send and receive may cause a context switch, processes that frequently send messages may incur a context switch for both the sending and receiving process. If sends and receives are frequent, context switches may occur much more often than the scheduler's granularity requires, negating the higher throughput normally expected with higher granularities. Most programs show a noticably lower throughput with granularity 1 than with any other value. This suggests that while frequent message passing decreases throughput at higher granularities, it does not incur the same overhead as context switching after every operation. It is possible that at granularity 1 processes incur both the scheduler context switch and the send and receive context switch with every message pass, causing lower throughputs than other granularities.

See the spreadsheet for raw data and graphs.

2.2 Q2

Unlike the message passing implementations, the semaphor implementations of the dining philosophers problem have much wider spreads, with standard deviations of 10.08 and 13.11. The chief causes of the spread are the throughput spikes at granularities that are multiples of 5; while the other granularity values generally produced ratios between 2 and 5, granularities that were multiples of 5 produced ratios of 30 or 40 operations per context switch. This difference is most likely because the philosopher processes are 5 instructions long, so granularities that are multiples of 5 allow the process to acquire both forks, put both forks down, and return to the beginning of the process before switching the process out. As a result, neither process is ever blocked, so context switches incurred by suspending and resuming the process are not incurred.

Interestingly, the other granularity values do not generally offer a significant improvement in throughput over the message passing implementations. A few granularities produce ratios of between 5 and 10, and this slight benefit is likely the result of not using a process to represent a fork. Since there are no fork processes, the philosopher processes are the only ones to block, while the message passing implementations block both the philosopher process and the fork process.

2.3 Q3

The non-solution to the dining philosopher's problem deadlocks at granularities of 1, 3, 4, 6, 7, 9, 11, 12, 15, 16, 17, and 19. There doesn't appear to be discernible pattern here, though it is noteworthy that granularities that are multiples of 5 never deadlock, for similar reasons that they provide greater throughput (see Q2). The expected conditions for deadlock are any granularity such that a context switch occurs after the first PE operation, and the other process then runs its first PE operation. This is difficult to pin down numerically, since processes may be suspended after attempting a PE; processes do not strictly alternate execution after granularity operations, making it difficult to track relative IC positions.

It is also likely that some of the strange deadlock behavior is the result of extent scheduler bugs. There is at least one known bug still present in the scheduler, though it does not appear to be triggered when running this program.

See the spreadsheet for data and graphs.

2.4 Q4

See Q2.

2.5 Q5

As with previous problems, as the granularity increases, the ration of operations to context switches also increases. The most noticeable difference between buffer sizes is that the 10 and 15 element buffers have significantly higher throughput at higher granularities than the 5 element buffer. One possible reason for this is that if the producer or consumer process are running at different speeds, eventually the buffer will be exhausted and throughput will reduce to message passing between the two processes. For this set of data, the program was allowed to run for 2 seconds before recording the number of operations and context switches. Once the buffer is exhausted, the longer the process runs, the closer the throughput ratio will be to the message passing case; since our measurements are taken after a relatively brief time, the smaller buffer may have been exhausted sooner, leading to a lower throughput ratio.