Vladimir Urazov
CS 4210

Homework 2
Due February 24

1. Scheduling
   a. I suppose the answer to this would depend on what precisely we mean by "CPU utilization" – the percentage of time doing *something*, or the percentage of time doing something *useful*. In the former case, the problem becomes trivial, because we have a CPU-bound task, and so at all times, even if all the IO-bound tasks happen to be waiting on IO, we have something to do. Hence, I will assume we only count performance of useful tasks in the CPU utilization.
   That is quite simple for 1ms quanta. At the beginning of any quantum, we have at least one job on the ready queue – either the CPU-bound task, or an IO-bound task. If we start up the CPU-bound task, then it will be pre-empted after 1ms, and we will need to switch context. Otherwise, the IO-bound task will be done after 1ms, and we will need to switch context too. So, clearly, for any quantum, we have 1ms of useful work, and one context switch – 1.1ms of total work. Therefore, the useful work is $\frac{1}{1.1} \approx 0.91$, 91%.

   b. In this case, for every two quanta each IO task will get the CPU once, and the rest will be taken up by the CPU task. So, we will need a total of 11 context switches per two quanta, and therefore, the total amount of work we will do will be 21.1, and the CPU utilization will be $\frac{20}{21.1} \approx 0.95$, 95%. This is the worst-case scenario.
   In the best-case scenario, the IO tasks will be staggered, so that we have precisely one IO task at the head of the queue at the beginning of every quantum, followed by the CPU task. In that case, the IO task will get 1 ms, get context switched to CPU task, which will take another 10ms, and then get pre-empted. In this case, we have two context switches for every 11ms of useful work, and hence CPU utilization is $\frac{11}{11.2} \approx 0.98$, 98%.

   c. We can see from the above examples, that the more often we give time to the IO tasks, compared to the CPU task, the less is CPU utilization (due to the larger comparative number of context switches we need to do). But of course, if we give priority to CPU-bound tasks, then the IO-bound tasks might have to wait for their turn more than it takes to complete the IO portion of their task, and so then the utilization of the IO device would go down. Hence the conflict – more IO utilization might be mutually exclusive with more CPU utilization.

2. Path expressions:
   a. `path { foo + bar + baz } + bar end` – **may deadlock**, if the functions are executed in the following order: baz, foo, bar – which is allowed by the path.

   b. `path { bar ; foo ; baz } end` – **does not deadlock**.

   c. `path bar ; { foo } ; baz end` – **does not deadlock**.

   d. `path { baz + foo } + bar end` – **may deadlock if**, as the same sequence of function calls as in part (a) is allowed here as well.

   e. `path { bar ; baz } ; foo end` – **will deadlock**.

3. If a Solaris user-level thread is preempted while it is blocked in the kernel executing a system call, it will be interrupted by SIGLWP, and the system call will be restarted when the thread resumes execution after preemption.
   In the Psyche kernel, system calls do not need to be restarted. This is because of the protected procedure call system. When a client thread makes a system call and is blocked, the kernel will deliver an interrupt to a processor that can execute the system call. It will then wait for a 'reply from PPC' from the processor executing the system call. If this response comes before the client thread is preempted, then the client is simply unblocked and resumes execution. Otherwise, when the client wakes up after preemption, it will immediately receive an interrupt that once again blocks it and waits for the response from the processor executing the system call, at which point the client thread will be unblocked.