Jeffrey Everett
CSCI 3753
Spring 2017

Problem Set 1

Question 1:
The scheduler dispatches the processes in the following order:

P1 | P2 | P3 | P1 | P2 | P3 | P1 | P2 | P1 | P2 | P1 | P2 | P1

At each | symbol, a context switch is required to switch from one process to another.

The exact timeline of the scheduling is as follows:

- 0s - P1 starts
- 2s - P1 is preempted (8s remaining), context switch begins
- 2.2s - P2 starts
- 4.2s - P2 is preempted (6.5s remaining), context switch begins
- 4.4s - P3 starts
- 6.4s - P3 is preempted (1s remaining), context switch begins
- 6.6s - P1 starts
- 8.6s - P1 is preempted (6s remaining), context switch begins
- 8.8s - P2 starts
- 10.8s - P2 is preempted (4.5s remaining), context switch begins
- 11s - P3 starts
- 12s - P3 ends, context switch begins
- 12.2s - P1 starts
- 14.2s - P1 is preempted (4s remaining), context switch begins
- 14.4s - P2 starts
- 16.4s - P2 is preempted (2.5s remaining), context switch begins
- 16.6s - P1 starts
- 18.6s - P1 is preempted (2s remaining), context switch begins
- 18.8s - P2 starts
- 20.8s - P2 is preempted (0.5s remaining), context switch begins
- 21s - P1 starts
- 23s - P1 ends, context switch begins
- 23.2s - P2 starts
- 23.7s - P2 ends

Thus, P1 ends at 23s, P2 ends at 23.7s, and P3 ends at 12s.

There are 11 context switches, which collectively take 2.2s. So the percentage overhead due to context switching is 2.2s/23.7s = 0.0928 = 9.28%.

Question 2:
The scheduler dispatches the processes in the following order:
P1 | P2 | P3

Here, the | symbol merely acts as a delimiter; context switches are not required for batch mode systems.

The exact timeline of the scheduling is as follows:
- 0s - P1 starts
- 10s - P1 ends, P2 starts
- 18.5s - P2 ends, P3 starts
- 21.5s - P3 ends

Thus, P1 ends at 10s, P2 ends at 18.5s, and P3 ends at 21.5s.

The percentage overhead due to context switching is 0% because context switching is not used.

The batch mode system finished execution faster because it had no context switches. This would not be the case if one or more of the processes were I/O-bound.

Question 3:
The four types of exceptions are as follows:
- Trap: an intentional exception used to transfer control to the kernel (e.g. system call)
- Fault: a potentially recoverable error (e.g. seg fault)
- Interrupt: a hardware-level signal from an I/O device (e.g. disk write finished)
- Abort: nonrecoverable error (e.g. hardware failure)

Hardware interrupts are requested by devices like disks. They are asynchronous because they are requested by devices working independently (and using a different clock) from the CPU. Software interrupts are requested by currently executing processes. They are synchronous because both the requester and the handler are using the same clock.

Question 4:
The jump table stores pointers to all of the different system call handlers. When a system call is made, the system traps to the kernel. The kernel then looks in this jump table to determine what program to run to handle the system call.

Question 5:
Overlapping I/O with CPU processing can be advantageous because it ensures a greater utilization of system resources.

There are two ways to perform this overlapping: synchronous I/O calls and asynchronous I/O calls.

In synchronous I/O calls, the kernel initiates the I/O request (or queues it if the device is already being used) and then context switches to another process. Later, when the hardware interrupt from the device comes through that notifies the OS that the I/O request has been fulfilled, the kernel will context switch back to the process that initiated the request. Because other processes are being executed while the device is performing the I/O request, there is overlapping of CPU and I/O; this is beneficial because otherwise no work would be done while the original process was waiting for the I/O device.

In asynchronous I/O calls, the kernel initiates the I/O request (or queues it if the device is already being used) and then returns immediately. The process can then periodically poll the OS for information about the I/O call status. If the process does useful work between these polls (i.e. it’s not busy waiting), then the continued execution of the process is beneficial. Also, time-shared OSs will often preempt to another process, which is also capable of doing work while the I/O device is working. Thus, the overlap of I/O devices and the CPU is beneficial.

Question 6:

The steps are as follows:

1. The application calls write()
2. The CPU traps into the kernel and sets the mode bit to “0”
3. The kernel passes on the request to the relevant device driver
4. The device driver checks the status of the device
    a. If the device is busy
        i. The device driver adds the request to the device queue
        ii. The kernel will then context switch into other processes and execute them while waiting for the device to be ready
        iii. After each request that the device fulfills, the device controller will use a hardware interrupt to signal that it has completed its task. The interrupt handler will then invoke the device handler, which will advance the position of the relevant I/O request in the device queue
        iv. Eventually, the I/O request from the application process will be at the front of the queue and a hardware interrupt will come. The device handler will then initiate the request by interacting with the device controller.
        v. All these steps place the system state (with regards to the application of interest and its I/O request) in the same place as it would be if the device were available from the beginning.
5. The device fulfills its request
6. When it is completed, the device controller creates a hardware interrupt
7. The CPU detects this interrupt and invokes the kernel’s interrupt handler
8. The interrupt handler determines which device caused the intercept and then jumps to the device handler for that device
9. Data (such as how many bytes were written) is pulled from device controller
10. Data is returned to application process