Homework 1

Question 1

Which instructions should not be permitted in the user mode? Explain your answer in one or two sentences per each.

a) Perform a trap.

b) Read device register for a given device.

c) Read the time of day clock.

d) Disable all interrupts.

Answer: d) and maybe b)

a) User programs can perform a switch to kernel mode at any time either voluntarily by performing a system call, or involuntarily by performing an unexpected trap. The latter may happen, for instance, when the user program references a null pointer.

b) Whether this is permissible depends on the device and depends on the register. It is ok to read the status bits to see if a floppy is mounted. It is not ok to read a device register that holds data that may belong to another user.

c) Can be permitted in the user mode, because reading the time of day clock is not a privileged operation.

d) Should not be permitted in the user mode, because interrupts must be handled on behalf of the operating system, it's not safe for user programs to be able to service or ignore interrupts, nor is it safe for them to have raw access to devices.

Question 2

Describe the actions taken by the kernel to context-switch between two processes.

Answer:

1. In response to a clock interrupt, the OS saves the PC and user stack pointer of the currently executing process, and transfers control to the kernel clock interrupt handler,
2. The clock interrupt handler saves the rest of the registers, as well as other machine state, such as the state of the floating point registers, in the process PCB.

3. The OS invokes the scheduler to determine the next process to execute,
4. The OS then retrieves the state of the next process from its PCB, and restores the registers. This restore operation takes the processor back to the state in which this process was previously interrupted, executing in user code with user mode privileges.

Question 3

Briefly describe main differences between a process and a program.

Answer:

Program is a set of instructions stored on secondary memory like hard disk.

Process is usually defined as an instance of a running program, and consists of two components:

1) A kernel object--Process Control Block (PCB) that the operating system uses to manage the process. PCB is also where the system keeps statistical information about the process.

2) An address space that contains all the executable or DLL module's code and data. It also contains dynamic memory allocations such as thread stacks and heap allocations.

Question 4

When building an operating system for a modern mobile cellular telephone would you favor reverting to an older approach of building libraries of procedures to which application programs were linked directly or to are more modern time-sharing and multi-process type OS with kernel and user modes?

Answer:

Older type OS is better for single user and resource limited devices like a modern mobile cell phone. The approach of building libraries of procedures and directly linking application programs allows one user to do one thing effectively at one time. More modern time-sharing and multi-process type OS is redundant and too heavy weighted for a modern mobile cell phone.

Homework 2

Question 1 Threads

Q: Name one situation where threaded programming is normally used, and *briefly* describe two reasons why.

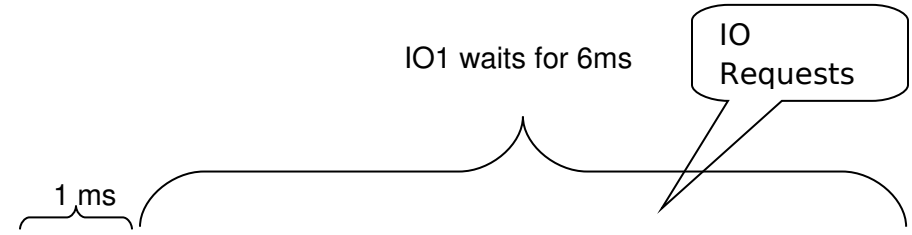
A: Threaded programming would be used when a program should satisfy multiple tasks at the same time. A good example for this would be a program running with GUI; the User Interface can be handled by one thread while the other thread is working on required background computations, or a thread can be used to complete the I/O work of an application. Two reasons for using threads are, firstly they increase the performance by running background processes in parallel with other operations and (as in the example with the I/O) they ensure that the regions which are not dependent on regions with latencies, would not be affected by these latencies.

Question 2 Scheduling

Consider a system running 3 I/O bound threads and a single CPU bound thread. The three I/O threads are first on the run queue and issue constant streams of messages to three unique I/O devices; it takes the thread 1 ms to make a request, each I/O device 6 ms to service each request, and the threads yield immediately to wait for the devices. The CPU bound thread runs forever. Assume the system is running a pre-emptive round-robin scheduler and each context switch takes 1 ms.

a) Define the percent ``Useful CPU Usage'' of the system to be the ratio of (Total CPU time) minus (Time spent performing context switches) to (Total CPU time). What is the percent useful CPU usage if the scheduler runs with a time quantum of 1 ms?

A:



IO1	CS	IO2	CS	IO3	CS	CPU	CS	IO1	CS
IO2	CS	IO3	CS	CPU	CS	…			

As seen above, for every 1ms execution a 1ms Context Switch takes place:

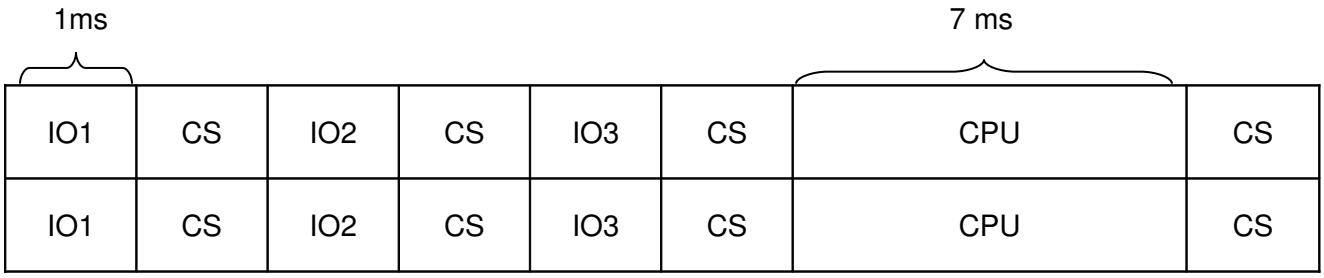
Total CPU Time: 16 ms

→ Useful CPU Usage: (16ms - 8ms) / 16 ms = 50%

Context Switches: 8 ms

b) We implement the system with a time quantum of 1 ms but the owner of the CPU bound thread complains about poor performance. We blame the time spent in context switches. Propose a new time quantum value that increases useful CPU usage by at least 20 percent.

A: We should increase the time quantum: let's make it 7.



Total CPU Time: 28 ms

→ Useful CPU Usage: (28ms - 8ms) / 28 ms = ~71%

Context Switches: 8 ms

c) We implement the system with the new time quantum value but receive complaints from the owners of the I/O bounded threads that the new system is ``worse'' than the old system. What do they most likely mean by ``worse'', and briefly explain why it is happening.

A: As the time quantum is larger, the IO-bounded threads have to wait longer for the CPU-bound thread. For example in the first example they were waiting only 1 ms after their 6ms wait was over, in the second example they have to wait 7ms.

Question 3 Atomic Operations

In class, we saw that an atomic test and set operation can be used to implement critical sections on a machine supporting parallelism (with multiple physical CPUs). But suppose that you were given a different atomic instruction called decrement. This instruction decrements a variable and leaves the initial value in register 0. Could decrement be used to implement critical sections? Show us how or explain why not.

A: In test-and-set(x) where x is a binary or boolean variable, x is set to true (atomically) and the old value of x is returned. The use of test-and-set for implementing critical sections is as follows:

```
class spinlock:
     # this implementation is based on test and set

     def __init__(self):
          self.lockval = 0

     def lock(self):
          while test-and-set(self.lockval):
               pass
          # if we return, we have successfully acquired the lock

     def unlock(self):
          self.lockval = 0
```

This atomic decrement instruction provides an atomic read-decrement-write operation that initially seems suitable for a similar implementation:

```
class spinlock:
     # this implementation is based on atomic decrement

     def __init__(self):
          self.lockval = 0

     def lock(self):
          success = False
          while not success:
               oldval = atomic-decrement(self.lockval):
               success = (oldval == 0)

     def unlock(self):
          self.lockval = 0
```

But without a corresponding atomic-increment instruction, this implementation is INCORRECT; it suffers from failure when the lock value overflows.

Question 4 Semaphores

a) Name one situation where semaphores should be used instead of the basic atomic test and set operations. Justify your answer in one sentence.

A: Using semaphores would cancel out the busy-waiting when atomic test-and-set is used, so it is better to use semaphores when there is only one CPU serving every application.

b) Name one situation where the basic atomic test and set operations should be used instead of semaphores. Justify your answer in one sentence.

A: In the implementation of semaphores themselves, the basic atomic test and set is necessary.

Question 5 Modified Monitors

Suppose we replace the wait() and signal() operations of monitors with a single construct await(B), where B is a general Boolean expression that causes the process executing it to wait until B become true. Explain why, in general, this construct cannot be implemented efficiently.

A: When a process is leaving the monitor, it may or may not leave the variables in such a state to make B true. Consequently, all waiting threads need to be awoken to test if their predicate holds. Thus, monitor exits would be very expensive operations, requiring waking up potentially hundreds or thousands of waiting threads. Condition variables are a much more directed way to wake up just the right number of targeted threads.