CS 537: Intro to Operating Systems (Fall 2017)

Worksheet 2 - Scheduling Mechanisms

Due: Sep 20th 2017 (Wed) in-class OR email to Zhewen before 5:30 pm

1. **Limited Direct Execution**

The following are the list of events (in no particular order) that may happen when a system call or a timer interrupt happens while a user process is executing.

a. Process A: trap into OS via `int $64`
b. Process A: calls read()
c. Process B: continues execution
d. OS: return from trap (into B)
e. OS: handle trap
f. OS: call switch routine
g. OS: send read to disk
h. OS: restore registers(B) from PCB(B)
i. OS: switch to kernel-stack (B)
j. OS: save registers(A) to PCB(A)
k. OS: put A to sleep (i.e. A's state = blocked)
l. Hardware: restore registers(B) from kernel-stack(B)
m. Hardware: jump to B's PC
n. Hardware: save registers(A) to kernel-stack(A)
o. Hardware: timer interrupt
p. Hardware: move to user mode
q. Hardware: jump to trap handler
r. Hardware: move to kernel mode

What are the events that may happen as per the limited direct execution protocol? Choose and **sort** them in order for the following two scenarios. Just write the **lower-case letters** for each event in order.

(a) Process A switches to B, because it calls **read()** and **blocks**.

(b) Process A switches to B, because a **timer interrupt** happens.

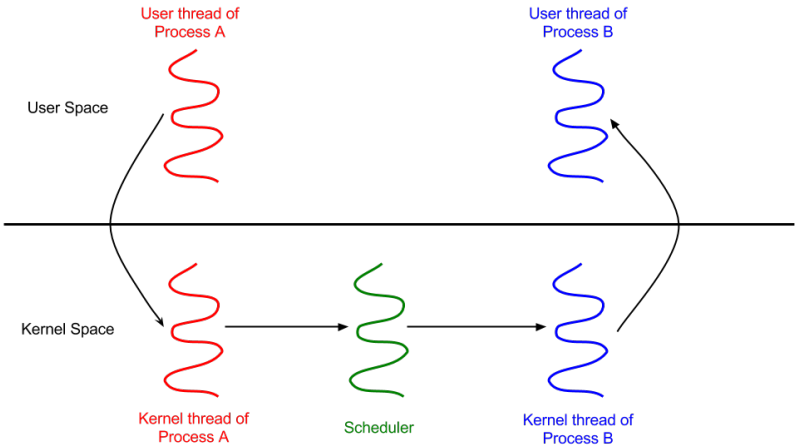
2. **Context switch**

a. When a context switch happens (e.g. from process A to process B), the hardware saves the *registers(A)* to *kernel-stack(A)* (in a structure called *trap frame*) and the OS stores the *registers(A)* to *PCB(A)*. Similarly, to begin executing process B, the OS restores *registers(B)* from the *PCB(B)* and the hardware restores the *registers(B)* from the *kernel-stack(B)*. Why do **both** the hardware and the OS save/restore the registers of a process during a context switch?

b. The control flow during a context switch from process A to process B is shown in the figure below.

i. How many times during this process does the value of **stack pointer** (%esp register) change?

ii. What are the different stacks that the stack pointer points to? You may assume that initially the stack pointer was pointing to process A's user stack.



3. **Interrupts, System Calls and Traps**
The following statements may contain some mistake(s). Can you ***find*** the mistake(s), *underline* them and ***correct*** them? If there are no mistakes, you may simply write **OK**.

   a. Traps, like system calls, and interrupts, such as from a programmable timer or disk, are handled very similarly in OSes. For example, when each occurs, the hardware saves some state (such as the program counter) and jumps into the OS. At that point, the OS handles the trap or interrupt. When finished, the OS returns, through a normal return instruction, and then retries the trapping or interrupted instruction.

   b. An interrupt is an asynchronous signal to the processor that some external event has occurred that may require its attention. As the processor executes instructions, it checks for whether an interrupt has arrived. If so, it completes or stalls any instructions that are in progress. Instead of fetching the next instruction, the processor hardware discards the current execution state and starts executing at a specially designated interrupt handler in the kernel.

   c. User processes can also transition into the operating system kernel voluntarily to request that the kernel perform an operation on the user's behalf. A *system call* is any procedure provided by the kernel that can be called from user level.

4. **Trap Handling, IDT, and Kernel Stack**

   a. When an interrupt or a system call trap occurs, the operating system must take different actions depending on whether the event is a file read system call, or a timer interrupt. How does the processor (hardware) know what code to run to handle that system call or interrupt?

   b. Why is the interrupt descriptor table stored in kernel memory rather than user memory?

   c. Why does each process has its own kernel stack? Wouldn't it be simple to have a single kernel stack for *all* processes?