Operating Systems Design
Fall 2010 Exam 1 Review

Paul Krzyzanowski
pxk@cs.rutgers.edu

Question 1

To a programmer, a system call looks just like a function call. Explain the difference in the underlying implementation.

The main difference is that a system call invokes a *mode switch* via a trap (or a system call instruction, which essentially does the same thing).

[other distinctions could be the way parameters are passed and that the specific system call is identified by a number since all system calls share the same entry point in the kernel]

Question 2

Explain what is meant by *chain loading*.

Chain loading is the process of using multiple boot loaders. A primary boot loader loads and runs a program that happens to be another boot loader. That, in turn may load a third boot loader or an operating system.

Why do we use chain loading? Often, there are severe size constraints on the primary boot loader (e.g., with a BIOS, it has to fit in under 440 bytes of memory). The primary boot loader may load a secondary boot loader that may give you options of which OS to load, be able to check for errors, and be able to parse a file system.

Question 3

Is it possible to construct a secure operating system on processors that do not provide a privileged mode of operation? Explain why or why not.

No. Since there is no distinction between privileged (kernel) or unprivileged (user) mode, a user process can do anything that the operating system kernel can do, such as accessing I/O devices or disabling interrupts. A process can, for example, read or modify any port of the disk or keep other processes from running.

Important! The most common wrong answer was explaining that you cannot have an operating system without privileged/unprivileged modes. This is not true. Many early processors, such as the 8086, did not have privileged modes. A process was able to access all of memory, modify interrupts, and access any I/O devices if it wanted to.

Question 3

Is it possible to construct a secure operating system on processors that do not provide a privileged mode of operation? Explain why or why not.

No. Since there is no distinction between privileged (kernel) or unprivileged (user) mode, a user process can do anything that the operating system kernel can do, such as accessing I/O devices or disabling interrupts. A process can, for example, read or modify any port of the disk or keep other processes from running.

Important! The most common wrong answer was explaining that you cannot have an operating system without privileged/unprivileged modes. This is not true. Many early processors, such as the 8086, did not have privileged modes. A process was able to access all of memory, modify interrupts, and access any I/O devices if it wanted to.

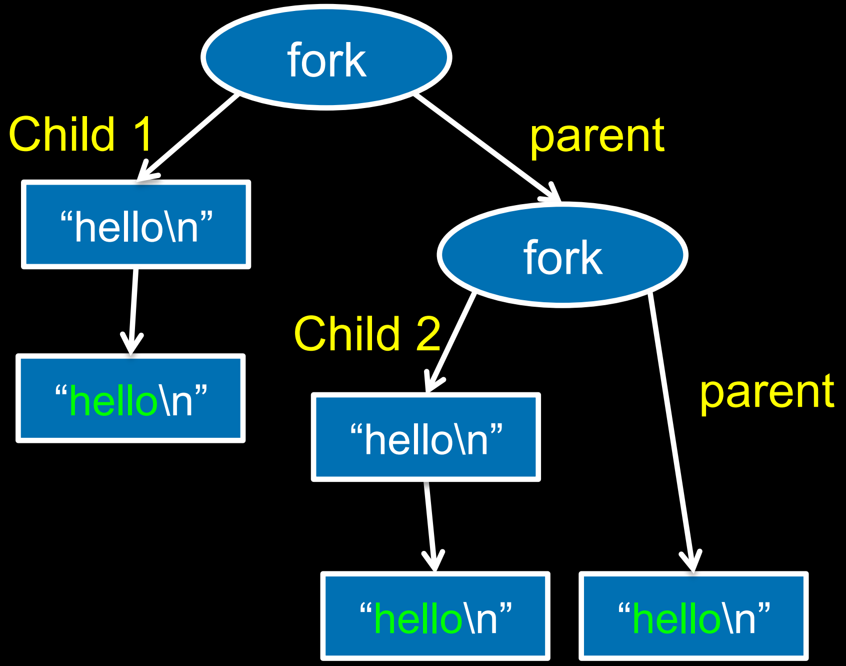
Question 4

How many times does the following program print hello?

Child 1 →
Child 2 →
all →

```
#include <stdio.h>
#include <unistd.h>

main()
{
    if (fork() == 0)
        printf("hello\n");
    else if (fork() == 0)
        printf("hello\n");
    printf("hello\n");
}
```



Answer: 5
fork() creates a copy of the process. The parent gets a process ID as the return from fork and the child gets 0. The first fork() creates a child process that prints “hello”, skips the “else” and prints the last “hello”. The second fork() is run by the parent and creates a second child. This second child prints “hello” in the second “if” statement and then prints the final hello. The parent, meanwhile, continues on to print the “hello” in the last statement.

Question 5

What are two advantages of threads over processes?

1. Creating threads and switching among threads is more efficient.
2. Some programming is easier since all memory is shared among threads – no need to use messaging or create shared memory segments.
3. Depending on the implementation, a separate (or custom) scheduler may be used to schedule threads. This is more common for user threads.

Question 6

What is the advantage of having different time slice lengths at different levels in a multilevel feedback queue?

We expect interactive (I/O intensive) processes not to use long stretches of the CPU (short CPU bursts). We reward these by giving them a high priority for execution. Processes that have longer CPU bursts get progressively longer CPU bursts BUT increasingly lower priority levels. This means that they'll get scheduled less often but, when they do, they will get to run for a longer time. This reduces the overhead of context switching.

Question 7

What is busy waiting? How can it lead to priority inversion?

Busy waiting, also known as a *spin lock,* is the situation when a thread is looping, continuously checking whether a condition exists that will allow it to continue. It is often used to check if you can enter a critical section or to poll a device to see if data is ready (or written/sent).

Busy waiting can lead to priority inversion under the following situation:
Process A is a low priority process that is in a critical section.
Process B is a high priority process that is executing a spin lock trying to enter the critical section.
A priority scheduler always lets B run over A since it's a higher-priority process. A never gets to run so it can release its lock on the critical section and allow B to run. The situation is called priority inversion because B is not doing useful work; it is essentially blocked by A from making progress.

Question 8

We looked at using test & set locks to lock access to a critical section by using a shared variable:

```
while (locked) ; /* loop until locked==0 */
locked = 1; /* grab the lock */
/* do critical section */
locked = 0; /* release the lock */
```

Unfortunately, this was buggy because there was a race condition between getting out of the while loop and setting locked to 1. Will this code fix it? Explain why (or why not).

```
wait:    while (locked);
         if (locked == 1) goto wait; /* fix wait condition */
         locked = 1;
         /* do critical section */
         locked = 0; /* release the lock */
```

No. The race condition still exists. In the original code, we risk being preempted before we set “locked=1”. In the revised code, we have the exact same risk. The “if” statement does exactly the same thing as the while loop.

Question 9

Multiprogramming (vs. multitasking) allows the operating system to:

(a) interrupt a process to run another process.

(b) switch execution to another process when the first process blocks on I/O or makes a system call.

(c) allow a single process to take advantage of multiple processors.

(d) distribute multiple processes across multiple processors in a system.

Answer: (b). The distinction between multiprogramming and multitasking is that multitasking allows preemption. Multiprogramming relies that a process relinquishes the CPU by calling the kernel (a system call; windows had a *yield* system call that did nothing but relinquished the processor).
(c) Is true for both multiprogramming and multitasking; (a) is true for multitasking; (d) may be true for both if it's a multiprocessing system.

Question 10

A semaphore puts a thread to sleep:

(a) if it tries to decrement the semaphore's value below 0.

(b) if it increments the semaphore's value above 0.

(c) until another thread issues a notify on the semaphore.

(d) until the semaphore's value reaches a specific number.

Answer: (a) The two operations on a semaphore are down(s) and up(s). Down(s) decrements the semaphore s but does not allow its value to go <0. If it will, then the value stays at 0 but the process blocks until another process does an up(s).

Question 11

You have three processes on a system with the following performance requirements:

process A runs every 90 milliseconds, using 30 milliseconds of CPU time.

process B runs every 60 milliseconds, using 20 milliseconds of CPU time

process C runs every 500 milliseconds, using 10 milliseconds of CPU time

Use rate-monotonic analysis to assign priorities to the three processes. Assume that priorities range from 0 to 90, with 0 being the highest and that no other processes will be running on the system. Which of the following is a valid set of priority assignments?

(a) $P_B=20, P_A=40, P_C=60$

(b) $P_A=5, P_B=60, P_C=90$

(c) $P_C=20, P_B=40, P_A=60$

Answer: (a) Rate monotonic priority assignment simply sorts processes by their period of execution. Higher frequency processes get a higher priority.

Question 12

A processor switches execution from user mode to privileged mode via:

(a) a software interrupt.

(b) a programmable I/O.

(c) a hardware interrupt.

(d) memory mapped I/O.

Answer: (a) or (c). Tricky since there are two valid answers. A CPU will enter kernel (privileged) mode via either a hardware interrupt or a trap (a software interrupt).

N.B., if the question asked how a *process* switches from user to kernel mode, then then answer would be (a).

Question 13

Implementing preemption in operating systems relies on:

(a) a programmable interval timer.

(b) being able to switch to kernel mode.

(c) having a modular operating system design.

(d) programmable I/O.

Answer: (a) To get preemption to work, you need to be able to get control away from the currently running process. Programming an interval timer to generate periodic interrupts forces the operating system to get control at regular intervals so that it can decide whether or not to preempt the running process.

Question 14

The following is not a character device:

(a) Webcam.

(b) Printer.

(c) DVD.

(d) Mouse.

Answer: (c) We didn’t cover devices except in the most introductory manner. A block device is one where you can address fixed-size blocks. It stores data persistently and can hold a file system. Any other device is either a network device or a character device. With these devices, the data stream is always changing and there is no concept of addressing the storage.

Question 15

The master boot record (MBR):

(a) loads the operating system from a boot partition.

(b) loads a boot loader from a boot partition.

(c) identifies disk partitions so the BIOS can load a boot loader from the proper partition.

(d) contains code that is run as soon as the system is powered on or reset.

Answer: (b) The MBR resides in the first 440 bytes of the first disk block and is loaded by the BIOS. It contains code that identifies the partitions on the disk (up to 4) and loads a boot loader from that partition.
Not (a): it does not load the OS directly.
Not (c): The BIOS does not load the boot loader; the first stage boot loader in the MBR contains the code that loads the next-stage boot loader.
Nod (d): The BIOS, sitting in flash memory, runs when the system is powered on.

Question 16

Which of these is not a component of the operating system?

(a) Boot loader.

(b) Process scheduler.

(c) System memory manager.

(d) File system driver.

Answer: (a) The boot loader is used to load the operating system but is not needed once the OS is loaded.

Question 17

Which component of a process is not shared across threads?

(a) Register values.

(b) Heap memory.

(c) Global variables.

(d) Program memory.

Answer: (a) All memory except is shared across threads that belong to the same process. The unique component per thread is the register set: all the processors registers, including the stack pointer and program counter.

Question 18

The alternative to programmed I/O is: (PIO is for data transfer)

(a) memory-mapped I/O.

(b) interrupt-driven I/O.

(c) independent I/O.

(d) direct memory access (DMA).

Answer: (d) PIO requires using the CPU to read data from device registers that are mapped onto the system's memory. The alternative is DMA, in which the device controller accesses the memory bus to transfer data to/from memory.

Not (a): Memory-mapped I/O is the same as PIO: the processor has to issue instructions to transfer data to/from the device memory.
Not (b): Interrupts may occur with either mode to let you know that the device is ready. The term "interrupt-driven I/O" doesn't really make much sense.
Not (c): No such thing.

Question 19

Which is not a valid process state transition?

(a) running → blocked.

(b) running → ready.

(c) ready → running.

(d) blocked → running.

Answer: (d) After a process is no longer in the blocked state (whatever it was waiting on happened), it is moved to the READY state. It's up to the scheduler to move a process from the READY to the RUNNING state.

Question 20

A process control block (PCB) exists only for processes in:

(a) the ready state.

(b) ready and running states.

(c) ready and blocked states.

(d) ready, running, and blocked states

Answer: (d) The PCB exists for processes in all states until the process exits and the parent picks up its exit via *wait*.

Question 21

The memory map of a multithreaded process looks similar to that of a single-threaded process except that the multithreaded one has:

(a) a copy of the data segment per thread.

(b) a stack for each thread.

(c) a heap for each thread.

(d) a heap and stack for each thread.

Answer: (b) Each thread requires a separate stack to be allocated for it (from the memory that is shared among all threads). The heap contains dynamically allocated memory (e.g., via *malloc* or *new*) and is shared by all. The data segment contains global variables and is also shared.

Question 22

In a thread-aware operating system, a process control block (PCB) no longer includes:

(a) saved registers.

(b) process owner.

(c) open files.

(d) memory map.

Answer: (a) Since each thread has its own register set, the registers have to be saved per thread in the Thread Control Block (TCB). The owner, process memory, and open files are shared among all threads in a process so they can still be tracked in the Process Control Block.

Question 23

Mailboxes:

(a) make it easy to support multiple readers.

(b) make it possible to support multiple writers.

(c) improve efficiency since messages do not have to copied to an intermediate entity.

(d) all of the above.

Answer: (a) The sender does not have to address a specific receiver but instead sends a message to a mailbox. One or more readers can then read messages from the mailbox.

Mailboxes support (a) and (b) but it’s not difficult to use direct messaging to support (b): multiple writers simple send messages to the same reader.

(c) Is wrong since messages do get copied with mailboxes.

Question 24

A CPU burst is:

(a) an example of priority inversion where a low priority process gets access to the CPU.

(b) a temporary increase in the priority of a process to ensure that it gets the CPU.

(c) an unexpected increase in a process' need for computation.

(d) the period of time that a process uses the processor between I/O requests.

Answer: (d) A CPU burst is the period of time that a process is executing instructions before it gets to a *waiting* state on some event (usually I/O but also things like messages or semaphores).

Question 25

Compared to a non-preemptive scheduler, a preemptive scheduler can move processes from the:

(a) *running* to the *blocked* state.

(b) *ready* to the *running* state.

(c) *blocked* to the *ready* state.

(d) *running* to the *ready* state.

Answer: (d) A preemptive scheduler can stop a process from running and have another pro

Question 26

A round robin scheduler:

(a) favors processes that are expected to have a short CPU burst.

(b) favors high priority processes.

(c) dynamically adjusts the priority of processes based on their past CPU usage.

(d) gives each process an equal share of the CPU.

Answer: (d) A round robin scheduler is basically first-come, first-served with preemption: a process can run until it blocks or until its time slice expires. Then it's put at the end of the queue and every other process that's ready to run gets to run.

No estimation of CPU burst, no priorities.

Question 27

A shortest remaining time first scheduler:

(a) dynamically adjusts the quantum based on the process.

(b) favors processes that use the CPU for long stretches of time.

(c) gives each process an equal share of the CPU.

(d) tries to optimize mean response time for processes.

Answer: (d) A SRTF scheduler tries to estimate the next CPU burst of each process by weighing past CPU bursts. It then sorts processes based on this estimated burst time. This causes short-cpu-burst processes to be scheduled first and optimizes the average response time.

Question 28

Computing the weighted exponential average of previous CPU cycles is used for:

(a) determining the length of a quantum for the process.

(b) allowing a priority scheduler to assign a priority to the process.

(c) having a round-robin scheduler sort processes in its queue.

(d) estimating the length of the next cycle

Answer: (d) This is used to estimate the next CPU burst time. The assumption is that the CPU burst period will be related to previous CPU burst periods.

Question 29

Process aging:

(a) helps estimate the length of the next compute period.

(b) increases the runtime of a process.

(c) is a count of the total CPU time used by the process and is stored in the PCB.

(d) improves the chances of a process getting scheduled to run.

Answer: (d) Process aging is when you temporarily increase the priority of a low-priority process that has not been run in a while to ensure that it gets a chance to get scheduled to run.

Question 30

Which statement about multilevel feedback queues is false? Multilevel feedback queues:

(a) assign processes dynamically into priority classes.

(b) give low priority processes a longer quantum.

(c) favor interactive processes.

(d) prioritize processes into classes based on their estimated CPU burst

Answer: (d) A multilevel feedback queue scheduler does not compute an estimate of the CPU burst. It simply drops a process to the next lower priority queue if the process used up its entire time slice in the current CPU burst. If it did not, then it remains in its current queue.

Question 31

A hard deadline is one:

(a) that is difficult to estimate.

(b) where the computation has diminishing value if it is missed.

(c) with an unknown termination time.

(d) that cannot be missed.

Answer: (d) The key difference between *hard* and *soft* real-time systems is that hard deadlines are those where there is no value to the process if it misses its deadline. Soft real-time systems will do a best effort job at trying to meet the deadline but all is not lost if the result is a bit late.

The End