Operating Systems

Week 4 Recitation:
Exam 1 Preview – Review of Exam 1, Spring 2014

Paul Krzyzanowski
Rutgers University
Spring 2015

Spring 2014: Question 1

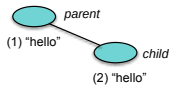
How many times does this code print "hello"?

```
main(int argc, char **argv) {
        int i;
        for (i=0; i < 3; i++) {
                fork();
                printf("hello\n");
        }
}
```

- Each call to fork() causes a process to create a child process

Spring 2014: Question 1 (continued)

```
main(int argc, char **argv) {
        int i;
        for (i=0; i < 3; i++) {
                fork();
                printf("hello\n");
        }
}
```



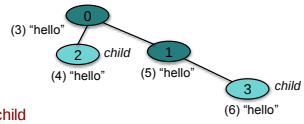
i=0

Process 0 forks child (process 1)

Parent & child print "hello"

Spring 2014: Question 1 (continued)

```
main(int argc, char **argv) {
        int i;
        for (i=0; i < 3; i++) {
                fork();
                printf("hello\n");
        }
}
```



i = 1

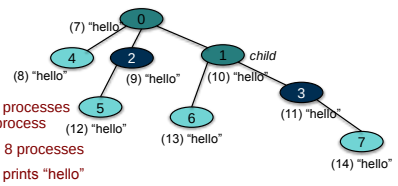
Original parent and child each fork a process

Now we have 4 processes

Each of them prints "hello"

Spring 2014: Question 1 (continued)

```
main(int argc, char **argv) {
        int i;
        for (i=0; i < 3; i++) {
                fork();
                printf("hello\n");
        }
}
```



i = 2

Each of the 4 processes forks a child process

Now we have 8 processes

Each of them prints "hello"

Total "hello" messages = 2 + 4 + 8 = 14

Spring 2014: Question 1 (continued)

How many times does this code print "hello"?

```
main(int argc, char **argv) {
        int i;
        for (i=0; i < 3; i++) {
                execl("/bin/echo", "echo", "hello", 0);
        }
}
```

- execl overwrites the current process by loading the program /bin/echo.
- The for loop is gone!
- Answer: 1

Spring 2014: Question 3

Your system supports messages but does not offer semaphores. Implement semaphore operations using messages.

Assume that messages use a mailbox. You may assume a unique mailbox per semaphore (i.e., semaphore s corresponds to mailbox s). Sending a message is a non-blocking operation. Receiving a message is non-blocking only if there is a message ready to be read.

Hint: you may send and receive empty messages (Ø).

a. Create a new semaphore *s* and initialize its value to *N*.
   init_semaphore(s, N);

b. up(s)

c. down(s)

Spring 2014: Question 3a

Create a new semaphore s and initialize its value to N
init_semaphore(s, N)

A semaphore can be represented by a message queue.

The value, N, is the number of items in the message queue

Create a new semaphore ≡ create a new message ⇒ `new_msg(s);`

Semaphore: counts # of *downs* before a sleep

Message: sleep when receiving a message that isn't there

To receive *N* messages before sleeping, fill the mailbox with *N* messages

```
for (i=0; i<N; i++)
    send(s, Ø);
```


Spring 2014: Question 3b

Implement ***up(s)***

What does *up(s) do?*

Wake one process up if ≥1 processes are sleeping on s .

Otherwise increment s

We can get the same behavior by adding a message to the mailbox:

If a process is waiting, it will receive a message & wake up

If no process is waiting on s, then s gets one extra message

```
send(s, Ø);
```


Spring 2014: Question 3c

Implement ***down(s)***

What does *down(s) do?*

If s == 0 then go to sleep.

Otherwise decrement s

We can get the same behavior by receiving a message from the mailbox:

If no message is in the mailbox, sleep and wait for one.

Otherwise, take a message. There will now be one fewer message.

The contents of the message do not matter and are discarded.

```
receive(s);
```


Multiple-choice questions

Spring 2014 – Part II

4. Multiprogramming is:

(a) An executable program that is composed of modules built using different programming languages.

(b) Having multiple processors execute different programs at the same time.

(c) Keeping several programs in memory at once and switching between them.

(d) When a program has multiple threads that run concurrently.

Spring 2014 – Part II

5. With a legacy PC BIOS, the Master Boot Record:

(a) Identifies type of file system on the disk and loads the operating system.

(b) Contains the first code that is run by the computer when it boots up.

(c) Contains a list of operating systems available for booting.

(d) Contains a boot loader to load another boot loader located in the volume boot record.

The master boot record is a 512-byte disk block.
It identifies disk partitions (dividing one disk into multiple logical disks) and contains code to load N consecutive blocks from the start of one of these partitions (the boot partition).

Spring 2014 – Part II

6. Which of the following is a policy, not a mechanism?

(a) Create a thread.

(b) Prioritize processes that are using the graphics card.

(c) Send a message from one process to another.

(d) Delete a file.

A mechanism is the implementation – *how something is done*
E.g., switching a process, writing to a file, multi-level queues for scheduling

A policy specifies *what is do be done*
E.g., which process goes next, who's allowed to write to a file, what priority should be assigned to a process?

Spring 2014 – Part II

7. Which of the following does NOT cause a trap?

(a) A user program divides a number by zero.

(b) The operating system kernel executes a privileged instruction.

(c) A programmable interval timer reaches its specified time.

(d) A user program executes an interrupt instruction.

The kernel is already running in privileged mode, so executing a privileged instruction will not cause a violation.

Spring 2014 – Part II

8. A context switch always takes place when:

(a) The operating system saves the state of one process and loads another.

(b) A process makes a system call.

(c) A hardware interrupt takes place.

(d) A process makes a function call.

In (b) and (c), the same process may continue to run.
In (d), the kernel is not even involved.

Spring 2014 – Part II

9. A dedicated system call instruction, such as SYSCALL, is:

(a) Faster than a software interrupt.

(b) More secure than a software interrupt.

(c) More flexible than a software interrupt.

(d) All of the above.

SYSCALL is less flexible – there is only one destination address defined.

However, it is faster since it does not require looking up an address in a main memory table.

Spring 2014 – Part II

10. Which of the following is *not* a system call?

(a) Duplicate an open file descriptor.

(b) Get the current directory.

(c) Decrement a semaphore.

(d) Create a new linked list.

A linked list does not involve access to protected resources.

At times, however, creating a linked list may involve a request for memory. This, indirectly, may result in a system call. However, (d) is the best answer of the four choices.

Spring 2014 – Part II

11. A process control block is:

(a) A structure that stores information about a single process.

(b) The kernel's structure for keeping track of all the processes in the system.

(c) A linked list of blocked processes (those waiting on some event).

(d) A kernel interface for controlling processes (creating, deleting, suspending).

Spring 2014 – Part II

12. A process exists in the *zombie* (also known as defunct) state because:

(a) It is running but making no progress.

(b) The user may need to restart it without reloading the program.

(c) The parent may need to read its exit status.

(d) The process may still have children that have not exited.

The process no longer executes.
The kernel reclaimed any memory that it used and closed any open files.
However, the process entry is still around in the process list so that information about the process can be read.

When a process does a wait(), it gets the exit code of the child process and the PCB for that process is deleted.

Spring 2014 – Part II

13. Which state transition is *not* valid?

(a) Ready → Blocked

(b) Running → Ready

(c) Ready → Running

(d) Running → Blocked

A process needs to do something to get to a blocked state.
When it's in a ready state, it's not running and can't create an action that will put it into a blocked state.

Spring 2014 – Part II

14. Threads within the same process *do not* share the same:

(a) Text segment (instructions).

(b) Data segment.

(c) Stack.

(d) Open files.

Each thread must have its own stack.
The stack holds return addresses for functions that were called as well as local variables.

Spring 2014 – Part II

15. A race condition occurs when:

(a) Two or more threads compete to be the first to access a critical section.

(b) The outcome of a program depends on the specific order in which threads are scheduled.

(c) A thread grabs a lock for a critical section, thus preventing another thread from accessing it.

(d) Two threads run in lockstep synchronization with each other.

A race condition is a timing bug.

Spring 2014 – Part II

16. Which of the following techniques avoids the need for spinlocks?

(a) Event counters

(b) Test-and-set

(c) Compare-and-swap

(d) All of the above.

Semaphores, blocking messages, event counters, and condition variables (monitors) enable a process to become blocked on a critical section so it does not have to repeatedly check to see if it is allowed to continue.

Spring 2014 – Part II

17. Priority inversion occurs when:

(a) A low priority thread has not been given a chance to run so its priority is temporarily increased.

(b) The scheduler allows a low priority process to run more frequently than a high priority process.

(c) Two or more threads are deadlocked and unable to make progress.

(d) A low priority thread is in a critical section that a high priority thread needs.

A high priority process may be sitting in spin lock, unable to get out of it because it is waiting for a low priority process to get out of a critical section and release the lock.

However, the low priority process never gets scheduled because the high priority process is always ready to run.

The condition is called ***inversion*** because IF the scheduler would allow the low priority process to run, it can exit the critical section and allow the high priority process to make progress.

Spring 2014 – Part II

18. What's the biggest problem with spinlocks?

(a) They are vulnerable to race conditions.

(b) They are fundamentally buggy.

(c) They waste CPU resources.

(d) They rely on kernel support and cannot be implemented at user level.

They are tight for() loops that check to see if a process is allowed to get a lock.

A process can waste an entire time slice checking the lock while no useful work gets done. It is more efficient to put the process to sleep (blocked state) and wake it up when the lock is released.

Spring 2014 – Part II

19. A condition variable enables a thread to go to sleep and wake up when:

(a) The value of the variable is greater than or equal to some number N.

(b) Another thread sends a signal to that variable.

(c) Another thread increments the variable.

(d) Another thread reads the variable.

Condition variables are the simplest synchronization primitive.

One thread goes to sleep by waiting on a specific condition variable:
wait(condition_variable)
Another thread can wake that thread up by signaling the condition variable:
signal(condition_variable)

Spring 2014 – Part II

20. Preemption is when an operating system moves a process between these states:

(a) Running → Ready

(b) Running → Blocked

(c) Ready → Blocked

(d) Blocked → Running

Preemption is when the kernel stops a running process and performs a context switch to let another process run.

The process that was stopped is placed in a *ready* state and the scheduler will eventually let it run at a later time.

Spring 2014 – Part II

21. The disadvantage of round-robin process scheduling is:

(a) It gives every process an equal share of the CPU.

(b) It can lead to starvation where some processes never get to run.

(c) It puts a high priority on interactive processes.

(d) It never preempts a process, so a long-running process holds everyone else up.

A round-robin scheduler treats all processes equally.

This is fair but is often not what we want.
We want interactive processes to get higher priority so they get scheduled more frequently and can be more responsive.

Spring 2014 – Part II

22. The downside to using a small quantum is:

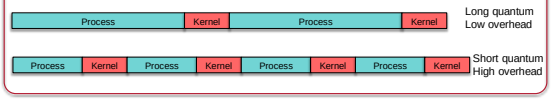
(a) A process might not get time to complete.

(b) The interactive performance of applications decreases.

(c) Some processes will not get a chance to run.

(d) Context switch overhead becomes significant.

The smaller the quantum (time slice) the larger is the % of time that the kernel spends deciding who gets to go next.



Spring 2014 – Part II

23. A time-decayed exponential average of previous CPU bursts allows a scheduler to:

(a) Estimate when each process will complete execution and exit.

(b) Compute the optimum number of processes to have in the run queue.

(c) Pick the process that will be most likely block on I/O the soonest.

(d) Determine the overall load on the processor.

The time-decayed average guesses that the next CPU burst will be similar to previous CPU bursts.

This allows a scheduler to pick the process with the shortest expected CPU burst.

Spring 2014 – Part II

24. Process aging is when:

(a) A long-running process gets pushed to a lower priority level.

(b) A process that did not get to run for a long time gets a higher priority level.

(c) A long-running process gets pushed to a higher priority level.

(d) Memory and other resources are taken away from a process that has run for a long time.

Process aging is a trick to avoid starvation.
A low priority process may never get to run.
With process aging, its priority will be increased so that it gets a chance to run.

The simplest implementation of process aging is to give all processes a high priority periodically – then have the scheduler lower the priority of those that use the CPU a lot.

Spring 2014 – Part II

25. The goal of a multilevel feedback queue is to:

(a) Keep the priority of interactive processes high.

(b) Gradually raise the priority of CPU-intensive processes.

(c) Ensure that each process gets the same share of the CPU regardless of how long it runs.

(d) Allow the scheduler to provide feedback to the process on how often it is being run.

A multilevel feedback queue punishes processes that use up their time slice by lowering their priority.

Interactive processes, which do not use up a whole time slice but block on an I/O request get to stay at the same priority level.

Spring 2014 – Part II

26. With soft affinity on a multiprocessor system, the scheduler will:

(a) Try to use the same processor for the same process but move it if another processor has no work.

(b) Associate a process with a specific processor and ensure it always runs on that processor.

(c) Use a single run queue so that there is no ongoing association between processors and processes.

(d) Periodically reset the association between all processes and processors.

Hard affinity = the kernel will ONLY reschedule the process to run on the same CPU it used the previous time it got to run.

Soft affinity = the kernel may rebalance the workload among CPUs.

Spring 2014 – Part II

27. Process A has a deadline of 100 ms and requires 80 ms of compute time.

Process B has a deadline of 80 ms and requires 50 ms of compute time.

Process C a deadline of 50 ms and requires 10 ms of compute time.

In what order will a *least slack scheduler* schedule these processes?:

(a) A, B, C

(b) C, B, A

(c) B, A, C

(d) C, A, B

Slack = deadline – compute time

A: slack = 100 – 80 = 20 ms

B: slack = 80 – 50 = 30 ms

C: slack = 50 – 10 = 40 ms

A least-slack scheduler always picks the process with the smallest slack time.

The End