CSc 452: Principles of Operating Systems
Fall 24 (Lewis)

Test 2

Thu 7 Nov 2024

Solutions

Name: _____ NetID: _____

Question	Points	Score
Page 1	20	
Page 2	20	
Page 3	20	
Deadlock Conditions	20	
All False	20	
Total:	100	

NetID: _____

1. (a) (7 points) In Phase 2, you used a Mailbox, from an interrupt handler, to send a message from the interrupt handler to a process that needs to be woken up. Why was it critical that you used `MboxCondSend()` instead of `MboxSend()`?

> **Solution:** The interrupt handler, since it is an asynchronous interrupt, **must not block** - and thus, we cannot call `MboxSend()`. Instead, we use the `Cond` version so that, if the function would have blocked, we will simply fail (immediately) instead.

(b) (8 points) When we discussed deadlock, I said that one classic strategy for preventing deadlock was to grab locks in a certain order. Which of the four deadlock conditions did this prevent? **Explain how it prevents the condition!**

> **Solution:** This prevented **Circular Wait.** It prevents the condition because if a process A blocks waiting for a given lock, then the owning process B cannot possibly be blocked on any lock which A already holds. If B is blocked on a lock, it must be on a lock which is even later in the ordering, and thus not a lock that A holds.

(c) (5 points) What is the difference between blocking a process, and performing a context switch?

> **Solution:** Blocking means that the process refuses to consume CPU time, for a while. A context switch simply means that the process is not on the CPU for now, but maybe it will return to the CPU soon.

2. (a) (7 points) The Banker's Algorithm requires that you give the locking system a list of **all** of the locks you will ever need; it locks them all at once, atomically.

   Why does this make deadlock impossible?

   Solution: We never **Hold and Wait.** That is, the process goes from the state of having no locks, to the state of having **all** of the locks that it wants. While the process may block arbitrarily long, it will never block while already holding any locks.

   (b) (8 points) What is a synchronous interrupt? Also, give at least one example of an event that causes a synchronous interrupt.

   Solution: A synchronous interrupt is one which must be handled **immediately,** because it is caused by the process. Classic examples include syscalls, page faults, invalid instruction exceptions, etc.

   (c) (5 points) What is a "zombie" process?

   Solution: A process which has died, but which the parent has not (yet) collected the status. It still holds an entry in the process table, but will never execute, ever again.

3. (a) (5 points) What is a race condition?

Solution: Any situation where the outcome of a calcalation might vary based on accidents of how quickly various threads run, or in what order.

(b) (5 points) What is the difference between a process and a program? Give an example of a common situation which helps illustrate the difference.

Solution: A program is a file on disk; it's a **plan** for how a computation might happen. A process is a **particular instance** of the program. This distinction is critical because we often are running multiple instances of the same program at once - such as multiple copies of `bash`.

(c) (5 points) Explain how you can use a Mailbox to implement sleep/wakeup. Be specific about what you call in each situation, and also how you initialize the system.

Solution:

- Allocate a mailbox when the sleeping process decides it needs to sleep; it has no mail messages inside.
- Store the mailbox ID in some data structure
- Recv (which blocks)
- The "waking" process finds the mailbox ID, and does Send on the mailbox
- The blockign process wakes up, and then frees the mailbox

NOTE: The solution works for single-process wakeup. However, another reasonable design is possible, which applies when you might have many processes queued up, waiting for some common operation, which happens from time to time:

- During **program init,** allocate a mailbox (with no messages inside).
- Any process that needs to block can Recv() on the shared mailbox.
- Any process that wants to wake up one element of the queue calls Send() on the shared mailbox.
- The mailbox is never freed.

(d) (5 points) Explain how you can use a Mailbox to implement a lock Be specific about what you call in each situation, and also how you initialize the system.

Solution: Version 1:

- During init, allocate a mailbox, with a capacity of **at least** 1 mail slot. Immediately Send() one message to it.
- Recv() to lock.
- Send() to unlock.

Version 2:

- During init, allocate a mailbox, with a capacity of **exactly** 1 mail slot. Do not Send() anything to it.
- Send() to lock; if another process has already sent a message, then this will block.
- Recv() to unlock.

4. (20 points) We said that there were 4 conditions that must all be true, in order for us to have deadlock. Explain each of them, with a sentence or two.

Circular Wait

Solution: All of the processes involved in the deadlock are arranged in a cycle; for example, A is waiting on B, B on C, and C on A.

No Preemption

Solution: It is not possible to take a resource back, after it has been given to a process.

Hold and Wait

Solution: All of the processes involved in the cycle hold at least one resource while blocked, waiting for another.

Mutual Exclusion

Solution: A resource cannot be owned by two processes at the same time.

5. Each of the statements below is **False.** Explain why.

(a) (5 points) When virtual memory is turned on, the program must be careful to translate each memory access to use the proper physical address, before it actually touches memory.

> **Solution:** The process never translates its own virtual into physical addresses! That is done by the CPU, and is always invisible to the program.

(b) (5 points) Page faults are sent by the CPU when a process attempts an illegal access to virtual memory, such as following a wild pointer. The OS reports a page fault to the user as a "segfault."

> **Solution:** A page fault simply means that the page tables don't allow the operation which was attempted. This doesn't mean that it actually is a bug. Often, the OS will notice that the access was valid, but that something needs to be done to get memory ready for the process - such as loading up a page from disk, or performing a COW copy.
>
> The OS only sends a "segfault" when it determines that the program attempted to do something illegal.

(c) (5 points) When a user-mode process completes its `main()` function, it returns to kernel mode, and then eventually destroys the process.

> **Solution:** It is **impossible** to return to kernel mode; a user-mode process is stuck there forever - except if it performs a system call. (So the way that a program dies is by calling a syscall that kills it.)

(d) (5 points) Suppose that we have defined an order for our locks, in order to prevent deadlock. It is illegal for any process to even attempt locks out of order, since if it blocks it can participate in deadlock.

> **Solution:** It is permissible to **attempt** to lock out of order, so long as you don't block on the lock if it fails - including unlocking all existing locks upon failure.