**CSc 452: Principles of Operating Systems**
Spring 23 (Lewis)

# Test 2

Thu 30 Mar 2023

**Solutions**

Name: ______________________ NetID: ______________

| Question | Points | Score |
|---|---|---|
| Page 1 | 25 | |
| Page 2 | 25 | |
| Page 3 | 23 | |
| All False | 27 | |
| Total: | 100 | |

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) (7 points) What is the difference between starvation and deadlock?

   **Solution:** In starvation, a process is not making progress, but it is **possible** that it might do so in the future. But with deadlock, it is **impossible** for the process to ever make progress, ever again.

   (c) (11 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.

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) (7 points) In a virtual memory system, why are page sizes always powers of 2?

   **Solution:** Because we are going to use bit-masking and shifting to break the addresses into their "page number" and "offset" parts. Thus, the page size must be a power of 2 - since if it was anything else, we would have to perform division, instead of masking and shifting.

   (c) (11 points) Explain how COW (Copy-On-Write) works in a virtual memory system. Make sure to describe what the page tables contain before the copy, and how they change.

   **Solution:** In the beginning, there is only one physical page, and multiple processes have maps of it - but all of the page table entries are marked as **readonly.**

   When one of the processes tries to write to the page, we get a page fault, and the OS determines that a write was permissible. So the OS allocates a second physical page, and duplicates the data.

   Once the data is duplicated, the process that wanted to write is pointed at the duplicate page, but this time with write permissions. The write is re-tried and succeeds.

3. (a) (8 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.

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

   **Solution:** An interrupt which is caused by, and related to, the currently-running process. It must be handled and resolved before the current process can make any further progress.

   (c) (7 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.

NetID: ____________________

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

---

(a) (9 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) (9 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) (9 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.)