CSc 452: Principles of Operating Systems

Spring 23 (Lewis)

Test 3

Tue 9 May 2023

Solutions

Name: ____________________ NetID: ____________

Run $\LaTeX$ again to produce the table

1. (5 points) The class responded mightily when I asked you all to fill out your TCEs. I'm giving you all 5 freebie points on the exam - 5 "correct" points, that you get just for turning in your exam!

NetID: ____________________

2. (a) (8 points) What is the difference between synchronous and asynchronous interrupts? Why is it illegal to block a process from inside an asynchronous interrupt? Why is it perfectly OK to block a process from inside a synchronous interrupt?

Make sure to answer all three parts.

> **Solution:** An asychronous interrupt is one that is related to a device. The process that it interrupts has nothing to do with the device, and thus it would not be fair or correct to block the process to handle a device event.
>
> On te other hand, a synchronous interrupt, such as a system call or page fault, is about something which has ust happened on the specific process that got interrupted; thus, blocking the process is perfectly reasonable, as it is often part of how we handle the event.

(b) (4 points) What is a TLB?

> **Solution:** This is a cache of page table entries, which resides inside the CPU. It exists to make virtual-to-physical address conversions faster.

(c) (4 points) In virtual memory, why are page sizes always powers of 2?

> **Solution:** We need to use division and modulo to split the virtual address into the page number and offset. Division by power of 2 is trivial, as it simply becomes bit masking and shifting.

3. (a) (4 points) What is the difference between blocking a process, and performing a context switch?

   Solution: Blocking a process means that the process chooses to not execute, perhaps for a long time. It will no longer use the CPU until some event occurs, which causes it to be unblocked.

   A context switch removes the process from the CPU, but often it may be temporary. A switched-out process (which is not blocked) will still use the CPU, as soon as it is allowed, at some point in the near future.

   (b) (4 points) What is a disk seek, and why do we try to avoid them? (Assume that you are using a traditional spinning disk, and not an SSD.)

   Solution: We move the "head" from one track to another. Compared to the CPU, this is **incredibly** slow, and so we try to minimize how many we do, and how far they have to move.

   (c) (4 points) What is a "shadow" process table? Why, in our simulator, didn't we simply add new fields to the original process table?

   Solution: A "shadow" process table is a parallel data structure, which contains more data about the process, which is not stored in the main process table. We had to use this in our projects because the old Phase(s), which declared the process table, could not be modified; therefore, it was impossible for us to add fields for our new code.

4. In the following code, I give three processes, which send messages to each other. (To simplify the code, they never include data payloads, so I've omitted those parameters to `MsgSend()`,`MsgReceive()`.) Unfortunately, the code is broken - so deadlocks are not only possible, they are likely!

```
  init:
mbox1 = MboxCreate(1,0);    // max slots 1, buffer length 0
mbox2 = MboxCreate(1,0);
mbox3 = MboxCreate(1,0);
k = 2;

  P1:                  P2:                  P3:
MboxSend(mbox1);     MboxSend(mbox1);     MboxSend(mbox1);
MboxReceive(mbox2);  MboxSend(mbox2);     MboxReceive(mbox3);
k = k+1;             k = k-7;             k = k*2;
MboxSend(mbox3);     MboxReceive(mbox1);  MboxReceive(mbox1);
                     MboxReceive(mbox1);
```

The questions about this code are on the **next page.**

These questions regard the code on the **previous page.**

(a) (2 points) All three processes race to send a message to `mbox1` on the first line. Explain why only one of them will succeed at first, and the others will block, at least for a while.

Solution: `mbox1` only can hold a single queued message. Thus, once one process sends a message to it, no other process can send until some process receives.

(b) (4 points) Explain why deadlock occurs if P1 wins the race to `mbox1`.

Solution: If P1 wins the race to `mbox1`, then P2,P3 are both blocked on their respective first lines of code. P1 blocks on its second line, but this will never make progress because the only process that sends to `mbox2` is P2 line 2.

(c) (7 points) If P2 wins the race to `mbox1`, then deadlock will not occur; explain why it is impossible, in that situation, for P3 to update `k` before P1 does.

Solution: Short Answer - Sufficient and Concise
P3 does not update `k` until it receives from `mbox3`. That mailbox is only send to by P1, and that only **after** it has updated `k`. Thus, if both processes happen to update `k`, P1 must do it first.

Long, Detailed Answer
If P2 wins the race, then both P1,P3 will block until P2 receives messages from `mbox1`. It does so **after** it has updated k; thus, it is the first to do so.

P2 also sends a single message to `mbox2`; this doesn't immediately unlock anything, but will be important soon.

After P2 receives from `mbox1` the first time, either P1 or P3 wake up; however, each immediately attempts to receive from a mailbox. If P1 wakes up, then it will succeed in receiving from `mbox2`, which P2 previously sent to; it will then update `k`.

However, if P3 wakes up when P2 receives, then it will block on `mbox3`; while this looks similar to the case with P1, in this case, it will not proceed because no one has written to `mbox3`. Thus, it will block; after P1 wakes up later (and updates `k`), after which P1 sends to `mbox3`. This wakes up P3, which now updates `k`.

Thus, in either case, P3 waits on P1, in such a way that P1 always updates `k` before P3 does.

5. (a) (4 points) What is a trampoline function?

   Solution: It is a wrapper function, which calls another function, but provides some sort of extra purpose that enhances the function, like setting up the environment before the function runs, handling its return value, etc.

   (b) (4 points) In our project, we disabled interrupts in our functions, in order to solve race conditions. Explain why this was "as good as" grabbing a lock - provided that we are only running on a single CPU.

   Solution: If we disable interrupts, then the only way that a context switch can happen is if we explicitly choose to call the dispatcher ourselves. Thus, we don't have to worry about races - there are no other blocks of code that might run during our critical sections.

   (c) (4 points) I've said that a monitor provides some sort of protection against race conditions, but they do it in a strange way. How does a monitor provide mutual exclusion for your critical sections?

   Solution: The "lock" is gained **automatically,** when we call into a function that is part of the monitor. We hold the lock the whole time, until we return.

6. (8 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: The procsses which are blocked are arranged in a cycle.

No Preemption

Solution: We are not allowed to steal resources (locks or whatever) back from a process once they have been granted. The process must **choose** to release them.

Hold and Wait

Solution: Each process in the cycle holds at least one resource, and is also blocked waiting for one.

Mutual Exclusion

Solution: Resources cannot be shared across multiple processes.

NetID: ____________________

7. (a) (4 points) In UNIX, what is a signal and what is a signal handler?

    Solution: A signal is a software interrupt, which is delivered to a certain process. A singla handler is a function, registered inside that process, which is called when a certain class of signal is delivered to the process.

   (b) (4 points) In a typical OS, how many "zero [ages" are there, physically? How many are there, virually?

    Solution: Physically, there is only one. However, there are **many** in the virtual space - in fact, each process can have many!

   (c) (4 points) As part of UNIX's `fork()`, we make heavy use of the Copy-On-Write feature of virtual memory. Why is this so important?

    Solution: Many processes either die or call `exec()` relatively soon after they have been created. If we duplicated the entire virtual space every time we created a child, we would spend a lot of CPU time doing useless work. COW defers the work; we only copy the pages that we must.

   (d) (4 points) What does it mean when we say that virtual memory (and often, swap) are "overcommitted?"

    Solution: We have allowed more allocations than we can actually support; we hope that the processes will not actually make use of everything they have asked for - at least, not all at once.

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

(a) (3 points) Non-blocking functions cannot be interrupted by devices, unless they are related to the work being done by the function.

Solution: Non-blocking has nothing to do with interrupts; a non-blocking function can definitely be interrupted for a moment, after which it keeps doing its work. However, it never **blocks** - meaning that it either keeps using CPU, or it returns.

(b) (3 points) A user program, if it has access to atomic instructions, can implement semaphores (both `P()` and `V()`) without needing to make any system calls.

Solution: No, `P()` must be able to go to sleep, and `V()` must be able to wake it up - which certainly require system calls.

(c) (3 points) When we link our programs to standard libraries (such as the C standard library, provided by our OS), those standard libraries run with enhanced permissions - or sometimes even run in kernel mode directly.

Solution: No, standard libraries are ordinary code, running in user mode, which simply runs as part of the process (even if it so happens that their origin was from some "standard" code).

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

(d) (3 points) In UNIX, all virtual pages in a process's address space are `mmap()`s of files - although some of them are private copies that may diverge from the file contents over time.

Solution: Some pages are "anonymous," meaning that they were never connected to any file. Such pages have a nominal content of all zeroes to start with; if the user writes to any such page, COW will duplicate it to allocate physical space.

(e) (3 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's legal to **attempt** a lock, but not to block! So we can use a "trylock" on the lock; if it works, we proceed. If it fails, however, we must unlock every lock and start over from scratch.

(f) (3 points) In a Reader-Writer lock, we can never have readers and writers running at the same time, but it is OK to have lots of readers running, or lots of writers.

Solution: A RW lock does not allow multiple writers at once!