CS 162   Spring 1997   Midterm 1 solutions

1. Why turn off ints in project 2 but not project 1?

In project 1, the shared data that might need protection are used *only* in
system calls, and one system call can't be preempted to run another one. In
project 2, the shared data are used both by system calls and by events
initiated by interrupts (such as scheduling initiated by clock interrupts,
but also I/O device interrupts). The interesting case is when a
setrunqueue() interrupts a setrunqueue() in progress. For example, suppose
that someone blocks reading from the disk. Then a call to setrunqueue() is
initiated by the scheduler because of a clock tick. When the disk transfer
is complete, an asynchronous device interrupt wants to wake up the process
sleeping on the data buffer, so it makes a second call to setrunqueue(),
interrupting the first in progress.

We gave one point for an answer that explained one project but
not the other. We also gave one point for an answer saying that
the clock-level events must be protected against system-call events;
it's actually the other way around (and also one interrupt against
another).

The worst zero-point answer, which we saw far too often, was
"project 1 implements semaphores, and semaphores provide synchronization,
so we don't also need to turn off interrupts." Semaphores are an
abstraction that provide synchronization *to the users of the semaphores*
but the *implementation* of semaphores does have to worry about making
sure that's the case! Specifically, if the semaphore mechanism we
invented was to be used by the kernel itself as well as by user
programs, we would indeed have to turn off interrupts there.

Other zero-point answers were "system calls are atomic" and "system
calls can't be interrupted." System calls can't be *preempted by
another user process* but they can, and often are, interrupted by
bottom-half activities such as I/O interrupts.

2. How to fix the CTSS scheduler?

The answer I was expecting is this: Increase the priority of a process
when the user completes a line *and the process was blocked waiting for
keyboard input*. This solution is closest in spirit to the original

CTSS scheduler's approach.

Since it doesn't matter what priority a process has while it's blocked, we can, without changing the behavior of the scheduler, increase the priority as soon as the process blocks for keyboard input, without waiting for the user to type something. This, too, is a fine two-point answer.

We also accepted "Increase the priority when a process blocks," but that isn't really quite right. A process can block for many reasons other than interacting with the user. Something like compiling a large program is likely to be I/O-bound much of the time, but that doesn't mean it should have the same priority as something like a text editor. (The TAs and I had a lively debate during the grading about whether a process that blocks trying to output to the screen should count as interactive.) People who said this were probably misled by the fact that BSD increases the priority whenever a process blocks *for more than one second*. This BSD feature is a heuristic, a guess; they use it because in these days of remote computing via networks it's not easy to know whether a network block is ultimately waiting for a human being at the far end. The heuristic is that I/O blocks for devices other than human beings are usually cleared in much less than a second; people are the only computer peripherals that work so slowly!

> We gave one point for solutions of the form "do either X or Y" where one was right and the other wrong. There were hardly any one-point single solutions.
>
> The most common zero-point solutions focused on the specific issue of carriage return characters, e.g., "only raise the priority if the user types other characters in between carriage returns." This completely misses the point; the problem isn't that the user is typing *empty* lines, but that the user is typing lines that the program isn't reading! It wouldn't be any better if the user typed in the text of Macbeth. Other zero-point answers involved timing-based kludges, like increasing the priority only for some limited number of characters per second. All such solutions seem to indicate a lack of understanding that the problem is about the nature of an interactive program.

3. Rename by copying.

The answer we expected was this: If a file has more than one name because of hard links, after a rename-by-copying there will be two distinct copies of the file, instead of one copy with a different name.

We also accepted several other correct answers, although most of them are about obscure or unlikely situations:

* The modification time will be different.
* If there isn't enough free disk space for the copy, it won't work.
* If the system crashes in the middle, there will be two copies.
* In some cases the file's ownership might change.

We gave one point for answers such as "for a moment there will be two copies," which I suppose is user-visible but only if you're really trying hard. We also gave one point for answers that essentially just restated the question, like "This version will allow renaming across file systems." (By the way, a lot of you don't seem to know exactly what a filesystem is. During the exam there were several questions like "do you mean a directory?" A filesystem is a local disk partition, or a directory tree mounted from a remote filesystem, or one of those layered or portal filesystems discussed in the BSD book. The system administrator can "mount" a filesystem at a particular point in the directory hierarchy, so that, for example, we could mount a floppy disk filesystem as /floppy and then open its files as /floppy/myfile.)

The most common zero-point answers were about increased fragmentation (it's just as likely to be decreased, and besides, that's not really what we mean by a user-visible change. The user can only observe indirect effects of that (slow response), and can't determine the reason for the effects. Another zero-point answer was that the file might disappear out from under a reader who has the file already open at the moment of the rename. That just isn't true; Unix won't really delete the old inode until all those readers close the file.

4. Bridge synchronization.

Solution A: The problem with this solution is that it's a bad simulation because it will not allow the maximum possible use of the bridge. Suppose two cars are on the bridge, and a car is waiting to go in each direction. When one car leaves the bridge, it signals a condition variable. One of the waiting cars will get that signal, but we can't control which one. If it's the car waiting to go in the opposite direction, that car can't go because there is still one car on the bridge moving against it, so the

signal is wasted. Another inefficient situation is if the last car leaves
the bridge, and two or more cars are waiting to cross in the other direction.
We should allow three of them to enter the bridge at once, but this program
allows only one car to enter.

The easiest solution is to change the signal() in ExitBridge() to call
broadcast() instead. Then every waiting car will wake up and decide for
itself whether it is now safe to cross.

(Another more complicated solution would be to assign a separate condition
variable for each car, and maintain explicit queues for the two directions,
and signal exactly the right car(s) when one leaves the bridge.)

    The program as given does *not* lead to starvation or deadlock.
    When the last car leaves the bridge, so that a car can enter from
    either direction, some car will get the signal and succeed in
    entering. The delays caused by this program are temporary.

Scoring: Three points for changing signal() to broadcast() and giving
a good explanation. Two points for the code change *or* the explanation
but not both. Also two points for a three-point solution along with an
additional incorrect explanation. One point for a partial explanation,
for saying "correct but inefficient" without further explanation,
or for a two-point solution along with an additional incorrect explanation.

    Typical zero-point solutions were adding explicit locks (the
    monitor takes care of that) and changing the test in the while loop.

    We gave zero points for saying "correct." You may argue that if
    the program doesn't fully use the capacity of the bridge it is a
    mere inefficiency, and to be "incorrect" it would have to let cars
    crash into each other, but the point of programs like this is to
    simulate the real situation. If allowing one car at a time to use
    the bridge were okay, you would have solved the homework problem
    by putting a lock around the entire Arrive-Enter-Exit sequence!

Solution B: This one was so complicated that we graded it very leniently.
The problem with this code is that ArriveBridge() increments carCount even
if the car can't enter the bridge because it's already full. Suppose
three cars are on the bridge and three more cars arrive and have to wait.
Since only three cars can leave the bridge, carCount will never get below
3 and so ExitBridge() will never signal either condition variable.

Also, since we've incremented carCount before testing it, the following while loop should check (carCount > 3) rather than >=.

The most natural solution to these problems would be to move the line

```
        carCount = carCount + 1;
```

down below the second while loop. Unfortunately, this introduces a new, even worse problem: Suppose that three cars are on the bridge, more are waiting in the same direction (therefore waiting at the not_full while loop), and a car is waiting in the other direction (at the safe while loop). A car leaves the bridge and broadcasts not_full, but it happens that the process that runs next isn't a waiting car; it's the next car on the bridge, which again broadcasts not_full. By chance, the scheduler again picks a car on the bridge (there's only one left) rather than a waiter. The third car leaves the bridge and broadcasts both safe and not_full. The scheduler then happens to run the car that's waiting in the other direction. Since the bridge is now empty, that car gets through both while loops and enters the bridge. Now the scheduler picks a car that's waiting in the original direction. Since the direction has now changed, it's not safe for that car to enter the bridge. But it has already gotten past the first while loop! It now assumes that the direction is okay, and checks only the capacity. It enters the bridge and hits the other car.

To solve this problem we have to rearrange things so that an entering car makes both tests without leaving the monitor, something like this:

```
void ArriveBridge(int direc) {
    while (1) {
        if (direction != direc && carCount != 0)
            wait(safe);
        else if (carCount >= 3)
            wait(not_full);
        else break;
    }
    direction = direc;
    carCount = carCount + 1;
}
```

but at this point the solution is essentially similar to solution A, and might as well use a single condition variable. (This one is slightly more efficient in that when a car leaves the bridge but others are still there, wrong-direction waiters aren't woken up.)

Scoring: We gave three points for any single problem identified and fixed.
We gave one point to anyone who said the code is incorrect.

5. Segmented virtual address translation

a. 00000000: Segment 0 page 000 offset 0000
Segment 0 implies page table A. In that table, virtual page 000 is
physical page CAFE. So the result is CAFE0000.

b. 20022002: Segment 2 page 002 offset 2002
Segment 2 is invalid, so the result is "bad virtual address."

c. 10015555: Segment 1 page 001 offset 5555
Segment 1 implies page table B. In that table, virtual page 001 is
physical page D8BF. So the result is D8BF5555.

Scoring: one point each. We accepted solutions in which the
hexadecimal values were translated to binary. (We gave two
points to the one paper in which the hex values were *partly*
translated to binary!)

The most common wrong answer was to add the offset to the
physical page number. I'm guessing that these people were
confused by the phrase "the offset is added to the page
origin" on page 134 of Tanenbaum. "Page origin" means the
first address in the page. For example, the first address
in page number D8BF is D8BF0000. It's not the same as the
page number.

6. Disk write direct from user memory.

The answer we wanted was this: Just because the virtual address that
the user provides is valid, that doesn't mean that all the other addresses
in the buffer are valid. The buffer might cross a page boundary. So the
operating system must check the validity of every virtual page within the
buffer, and translate each to its physical address. (Contiguous virtual
pages need not correspond to contiguous physical pages!)

We also accepted these correct answers:
* The pages of the buffer must be locked in memory so that they aren't

paged out before the transfer is complete.
* There may not be enough contiguous free disk space to do the
  transfer all at once. [It's not clear that this will really be a
  problem; the operating system probably already knows how to transfer
  multi-record blocks of data to a file.]
* The user program might change the data in the buffer while the transfer
  is still in progress.

> We gave one point for the answer "the page might not be paged in" because we think that that would be part of translating the virtual address to a physical address.
>
> We gave zero points for the answer "it's the hardware that translates virtual to physical addresses." That's true when the central processor wants to make a memory reference, but I/O device controllers don't have access to the TLB inside the processor! The operating system must supply the I/O device with a physical address.

TOPS-10, one of the PDP-10 operating systems, really did use this approach for I/O. The kernel had to think about all the issues raised here; to deal with the problem of synchronization with the user process, each user buffer had a header that included an in-use flag, rather like the flag on a mailbox, to let the user know when the buffer could be reused (or, for input, when the buffer contained unread data). The advantage is that this method avoids the need for the processor to copy the data from a kernel buffer into the user's address space, so it saves processor time. But most systems today don't use this approach because of its complexity. Instead they use "double buffering," which means that the data to be written is first buffered in the user process (by the standard I/O library), then copied to another buffer in the kernel, and then written to the disk from there.

7. Effects of error in lottery scheduling.

There is no long-term effect, because the next time that process runs, these compensation tickets will be taken away. (Perhaps new compensation tickets will be issued if the process doesn't complete its time slice, but those will be calculated correctly.

> This was a three-point question. We gave two points to papers that included the correct answer above and also said something else that isn't true.

We gave no points for "none, because the process will eventually terminate and give back all its tickets" because this answer shows a failure to understand that compensation tickets only last until the process next runs.

We also gave no points for answers about potential overflows, because an overflow is equally possible if the number of tickets is calculated correctly.

Of course there will be an effect if the system *always* computes the compensation tickets incorrectly for some process, but that's not what the question asked.

David P. wants me to point out that a system designer might choose to give out too many compensation tickets on purpose, when a process is started, as a way to favor interactive jobs. Once the process has been run, if it's interactive it will get compensation tickets in the usual way. But when the program hasn't been run yet at all, it hasn't blocked before the end of a time slice, so on a heavily loaded system the already-running interactive processes will run instead of it.