Solutions
15-410, Spring 2025, Homework Assignment 1.

1 Tape drives (4 pts.)

Consider a system with four processes and seven tape drives. The maximal needs of each process are declared below:

Resource Declarations

Process A	*Process B*	*Process C*	*Process D*
5 tape drives	*5 tape drives*	*3 tape drives*	*2 tape drives*

Imagine the system is in the state depicted below. List one request which the system should grant right away, and one request which the system should react to by blocking the process making the request. Briefly justify each of your answers.

Who	*Max*	*Has*	*Room*
A	*5*	*1*	*4*
B	*5*	*2*	*3*
C	*3*	*1*	*2*
D	*2*	*1*	*1*
System	*7*	*2*	-

Solution

If D requests 1 tape drive, the request should be granted, as there is the following safe sequence:

1. D returns all 2 tape drives. The system now has 3.
2. C requests 2 tape drives, then returns all 3. The system now has 4.
3. B requests 3 tape drives, then returns all 5. The system now has 6.
4. A requests 4 tape drives, then returns all 5.

If A requests 2 tape drives, the system should block A. Otherwise, the system would have no more resources, but every thread could request more resources. Therefore, there is no thread which could come first in a safe sequence, so a safe sequence would not exist.

2 Device queue (6 pts.)

2.1 1 pt

What is the purpose of the `!device_running` busy-wait loops in the driver code? What problem could happen if they were deleted?

The point of the `!device_running` busy-wait loops is to make sure the device has detected and acted on the `device_start` command before the `enq()` function returns. If `enq()` were to return too soon, the host might invoke `enq()` again before `device_running` had a chance to flip to non-zero, in which case `enq()` would overwrite (and lose) the true head-of-queue item with a new head-of-queue item.

2.2 1 pt

Are those busy-wait loops "ok"? Why or why not?

For a busy-wait loop to be "ok" it should have a bounded time which is short. If you read device-driver code in various kernel code bases, you may well see busy-wait loops similar to what is found in `enq()`, which are written on the assumption that hardware devices will respond promptly to simple commands. There are two issues with those loops. First, a bug in a device might result in a CPU spinning forever, which in turn

might result in the entire kernel wedging. But a second, more subtle, issue is that as CPUs get faster the time it takes for an I/O device to respond to even a "simple" command, measured in terms of CPU instructions, goes up. It is possible for a single device to be deployed in two or three generations of machines, in which case what was a very short delay when the device driver was first written might turn into an arguably-long delay. If the delay for a hardware device is very short compared to the time to switch to another thread for a while, a busy-wait loop (with an escape hatch for a stuck device) may be appropriate, but if there *is* time to switch to another thread, that would probably be better.

(The credit for this sub-part will be assigned based on the quality of your explanation, not whether you told us "yes" or "no.")

2.3 4 pts

There is a concurrency problem with the "code base" shown above. Briefly state something that could go wrong and then present a trace which supports your claim. You may use more or fewer lines in your trace.

Briefly stated, it is possible that the device may go idle "at the same time as" *enq()* is deciding between appending a new item to the existing queue (if the device *is* still running) versus starting a new queue (if the device has gone idle). If this happens the wrong way, the new item can end up attached to the end of a queue that is being ignored and will be garbage-collected—in other words, the item will be "lost."

Execution Trace

time	`enq()`	`device()`
0		perform(9)
1	device_running?	...
2	cur?	...
3	cur?	cur=cur→next
4	cur==0	assert()
4	device_running?	cur==0
5	prev→next=ip	device_running=0
6	clean?	device_start?
7	return	device_start?
8		device_start?
9		device_start?

As Edsger Dijkstra discovered in 1960, this is an annoying problem (see "EWD1303").

In terms of difficulty, this homework is easier than typical exam questions on similar topics. However, this homework represents some of the reasoning we expect you to carry out in order to detect and explain race conditions (and/or deadlocks).