HW #3

Part A a) Problem set 1 (Bigger files)
b) Problem set 4 (EZPark)
c) Problem set 9 (Ben's Kernel)

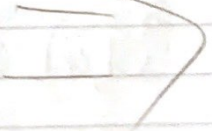
a) Problem Set 1 (Bigger files)

Q1.1) B and D

Both increasing the number of bytes/block from 512B to 2048B AND replacing one of the direct block numbers in each inode to an additional triple-indirect block number would increase the maximum file size.

Q1.2) B

Decreasing the size of all block numbers in almost any case will work for this situation. As the more block numbers you can save in the inode, the larger the file size limit will be, but you cannot have any 2^{16} block numbers as in the case for C



HW #5 (cont.)

b) Problem Set 4 (EZ Park)

Q4.1) A and C

We have race conditions in both updating available and numcars that can violate multiple correctness specifications.

Q4.2) Not entirely. Alyssa's code enforces atomicity of procedure only for RELINQUISH_SPOT(), however in FIND_SPOT(), her RELEASE() below line 7 releases the lock after a single iteration of the for loop, even if a spot is not found. It needs to instead acquire the lock inside the for loop above the if statement.

Q4.3) The RELEASE() after line 7 is necessary as if a spot is never found, the FIND_SPOT() RPC will never release the lock so that RELINQUISH_SPOT() can ever open up more spots.

Q4.4) I believe this version does meet specifications.

⇒

HW #3 (cont.)

b) Q4.5) This version does not meet specifications as FIND_SPOT() never releases the lock and therefore will only allow 1 car a parking spot and no other call will ever get to access available[] or numcars.

Q4.6) If the most recent "slim version" is implemented this won't work at all, assuming we use the previous working version, then B is correct and the client will wait until at least one car relinquishes a spot, but it is possible it will never get a response if there are many concurrent requests, it may never be the thread to acquire the lock. While A is technically true, B is more true as it is possible to wait forever as a result of the race to acquire the lock by multiple threads of FIND_SPOT().

Q4.7) A and D, as it may be processing multiple requests from the same client and could end up giving multiple spots to the same car, resulting in numcars being larger than the number of actual cars in the parking lot.

HW #3 (cont.)

b) (Q4.8) A Just as described in chapter 5, thread switches require the thread state to be entirely saved. That state includes the PMA, PC, SP, and several registers.

c) Problem Set 9 (Ben's Kernel)

(Q9.1) n is the gate that the SVC call is intending to enter kernel mode through which is saved in a register so that the kernel can read the intended gate name.

(Q9.2) A, C, and D

The kernel changing kpmar changes the current address space. Changing the user-mode bit will switch between user and kernel so it also does it. And the application attempting to write to upmar or kpmar will cause an exception which will change modes.

(Q9.3) C It will return to where it had left in user mode which will be in the procedure YIELD after the SVC call.

HW #3 (cont.)

C Q9.4) A and B because address space separation is always an enforcement of modularity, and the user-mode bit stops the applications from being able to modify their address spaces so they cannot exploit other program's memory.

Q9.5) C and D because now that we have a timer interrupt, the user could've entered kernel mode from any instruction location when the timer triggered the interrupt.

Q9.6) A, B, and C A and B the same as in Q9.4; and now C because a thread cannot hold the processor if it does not call YIELD, which would cause other threads to starve.

Q9.7) A, B, and C

A - T1 could read tmp then get preempted, then end up adding 1+2 giving it 3.

B - if either thread is interrupted after loading into R0, then comes back, increase of 4+8=12

C - This is expected as long as nothing is preempted between loads.

HW #3 (cont.)

c Q9.8) C Now that the code executes atomically, only powers of 2 will print.

Q9.9) A Since 100 is in the range 100-112 is the interrupt triggers inside one thread running those instructions, it will get its EIPC set to 100 while the other thread enters that range.

Q9.10) A and C

A - this is how it will run with no preemption
C - If preempted after instruction 104 finishes, a will have b saved in it, then when the section restarts, a=2 Then we will get b also equals 2.
B & D cannot happen as a will always change no matter how many times the thread is preempted and subsequently restarted.