Operating Systems Quiz 1 ECE344, Winter 2022

Duration: 1 hour

Examiner: D. Yuan

Instructions

Examination Aids: This is an open book exam.

All questions have been provided in this exam booklet. You need to provide your answers in Quercus.

If any of the questions appear unclear or ambiguous to you, then make any assumptions you need, state them and answer the question that way. If you believe there is an error, state what the error is, fix it, and respond as if fixed.

Be brief and specific as possible. Most questions should only need 1 sentence to answer. Time is tight, and you can't afford to write long answers. Marks will be deducted for incorrect statements in an answer.

You'll get 20% of the mark for each question where you leave the answer empty.

Work independently.

MARKING GUIDE

Q1:	(2)
Q2:	(2)
Q3:	(3)
Q4:	(3)
Q5:	(5)
Q6:	(5)
TOTAL:	(20)

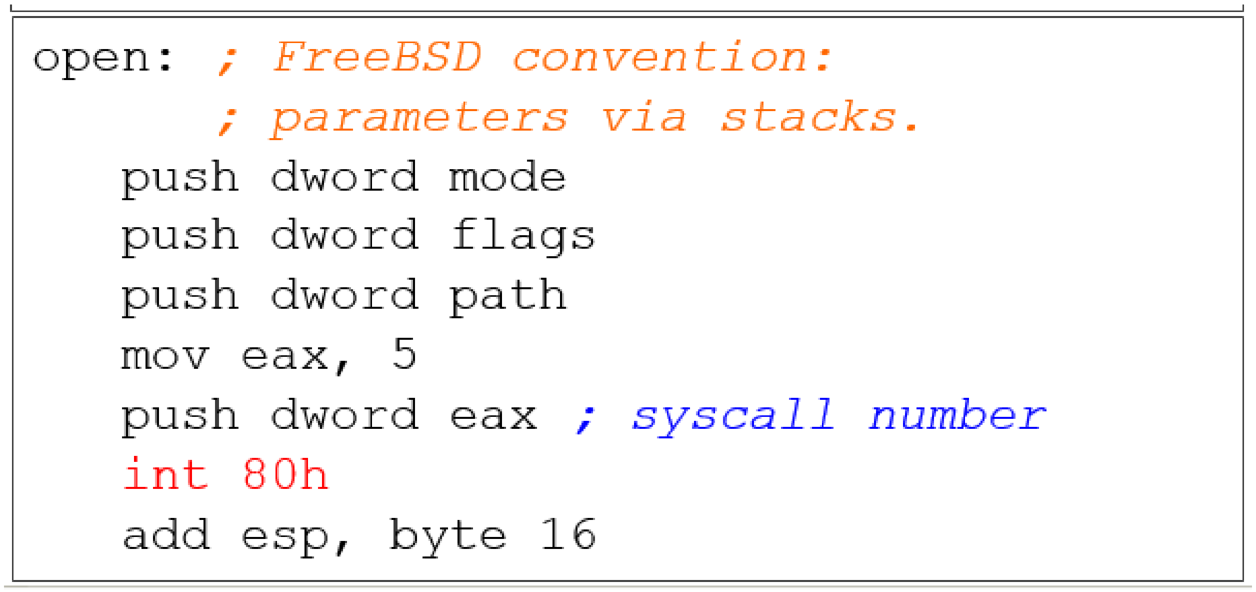
Question 1 (2 points). The latency of memory is about 100ns, whereas the latency of the hard drive is ~10ms. Name one implication of this performance gap to the design of the OS?

Question 2 (2 points). Protected instructions.
For each of the instructions below, answer (1) whether it is a protected instruction, and (2) why.

(1) Changing CPU's mode bit (i.e., changing kernel/user mode)

(2) Halt

Question 3 (3 points): System call.
Consider the following code snippet that was discussed during lecture, and answer the following questions:



```
open: ; FreeBSD convention:
      ; parameters via stacks.
   push dword mode
   push dword flags
   push dword path
   mov eax, 5
   push dword eax ; syscall number
   int 80h
   add esp, byte 16
```

(1) Do these instructions execute in user mode or kernel mode?

(2) What does the last instruction, "add esp, byte 16", do?

(3) These instructions are typically hand-crafted by the developers of OS or library, instead of being produced by the compiler. Why?

Question 4 (3 points): `fork() and exec()`
Assume that on a Unix OS, the PID of a newly created process will be its parent's PID + 1 (and in case that this PID is already used, the OS will try parent's PID + 2, parents' PID +3, and so on). A PID will be immediately available for reuse once the process terminates and its parent has reaped it. Assume that the PID of the initial process running the code below has PID 100. In addition, initially no process on this system has a PID greater than 100.

```
Consider the following program:

void main() {
  int pid = fork();
  if (pid == 0) {
    pid = getpid();
    if (pid < 103) {
       exec("a.out"); // a.out is the executable of this program
    }
    printf("L1: %d\n", pid);
  } else {
    waitpid(pid);
    printf("L2: %d\n", getpid());
  }
}
```

Write all possible outputs of this program.

Question 5 (5 points): **Concurrency Programming**

Consider the following code:

```
int count;

void worker () {
  printf ("%d", count++);
}
```

```
int main(int argc, char ** argv) {
  int i;
  count = 1;
  for (i = 0; i < 3; i++) {
    int tid = thread_create(worker, NULL);
  }
}
```

Answer the following questions.

(A) Describe all possible outputs of this program.

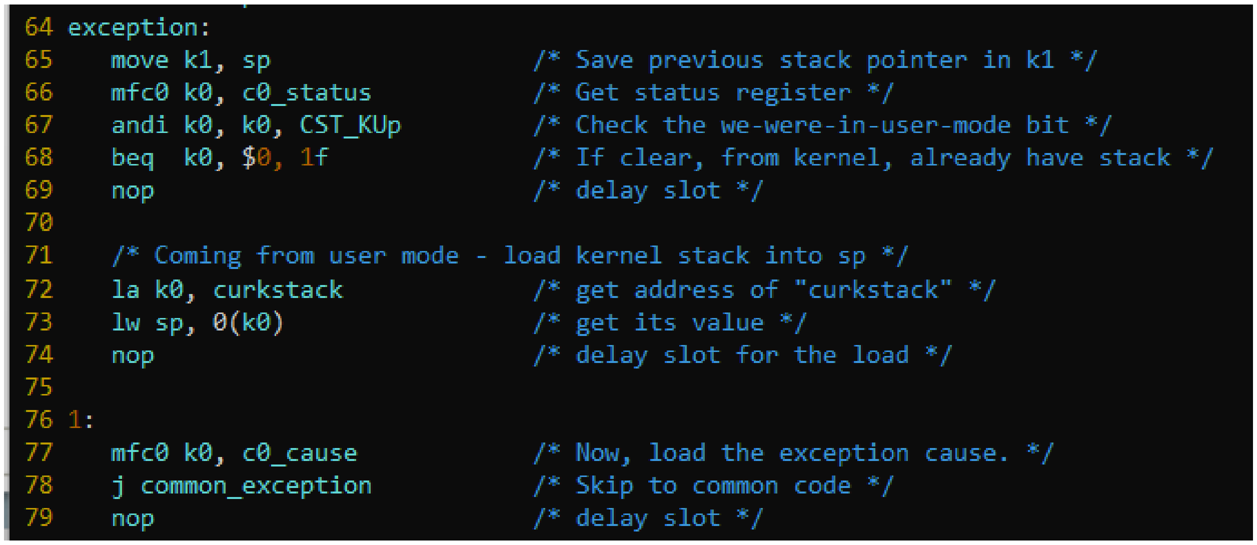
(B) If we want to ensure the output is always "123", while any thread can print any of the three numbers, what's the **minimal** patch? By minimal it means both the minimal lines of code and minimal critical region. You could use the following interfaces:

- thread_yield()
- thread_exit()
- thread_join()
- lock_acquire()
- lock_release()

(C) If we want to ensure the output is always "123", and "1" is printed by the first thread that was created, "2" is printed by the second thread, and "3" is printed by the third thread. What's the **minimal** patch (to the original code) that would work? You can only use the interfaces listed in (B).

Question 6 (5 points): OS161.

Consider the following code snippet from exception.S of OS161:



```
64 exception:
65    move k1, sp                 /* Save previous stack pointer in k1 */
66    mfc0 k0, c0_status          /* Get status register */
67    andi k0, k0, CST_KUp        /* Check the we-were-in-user-mode bit */
68    beq  k0, $0, 1f             /* If clear, from kernel, already have stack */
69    nop                         /* delay slot */
70
71    /* Coming from user mode - load kernel stack into sp */
72    la k0, curkstack            /* get address of "curkstack" */
73    lw sp, 0(k0)                /* get its value */
74    nop                         /* delay slot for the load */
75
76 1:
77    mfc0 k0, c0_cause           /* Now, load the exception cause. */
78    j common_exception          /* Skip to common code */
79    nop                         /* delay slot */
```

(1) When is this code snippet executed?

(2) Is the processor in user mode or kernel mode when this code is executed?

(3) Line 67-68 checks whether the processor was in user mode or kernel mode **before** this code gets executed. Why could the processor have been in kernel mode?

(4) What does `curkstack` store?

(5) Why are lines 72-73 needed when we were "coming from user mode", but not when we weren't?