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.

Please be brief and specific as possible. Clear, concise answers will be given higher marks than vague, wordy 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

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

OS can afford to trap to handle every I/O, whereas it cannot for memory. As a result, I/O operations are system calls, whereas memory load/stores are not protected instructions.

Another implication is that, given the slowness of disk, the OS will context switch when a process is blocked on I/O operation.

[Marking:] One of the above answers will receive full mark. But students need to give OS-specific answers. A generic answer, where you can just replace OS with application in the answer, is not a correct answer.

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)

Protected. Because otherwise a user program can change to kernel mode and gain access to everything.

(2) Halt

Protected. Because even if the user process finishes its work, there could be other processes waiting to use the CPU. Therefore the OS shouldn't allow the user process to turn off the CPU.

[Marking]: 1 for each sub-question; among which 0.5 mark for protected/non-protected, and 0.5 mark for the reason.

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?

User mode.

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

It frees up the space on the stack for the 4 words (mode, flags, path, and 5).

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

Because these are specific to the OS's convention for handling system calls.

[Marking]: 1 for each sub-question. If a student gives the correct answer, but further says sth wrong, -.5 mark.

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, so on and so forth). A PID will be immediately available for reuse once the process terminates and its parent has reaped it. Assume that

`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.

L1: 103
L2: 102
L2: 101
L2: 100

There is only one possible output.

-

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++) {
    thread_create(worker, NULL);
  }
}
```

```
Answer the following questions.
      (A) Describe all possible outputs of this program.
      111
      112
      113
      122
```

123

Marking: 1 mark

- Note 233 or 223 is not possible. If student answer these on top of correct answers, -0.5

(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()

```
int count;
+ lock_t lock;

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

Marking: 2 marks

- If student gives the same answer as (C), 1 mark.

(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, not the code in (B)) that would work? You can only use the interfaces listed in (B).

```
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);
+   thread_join(tid);
  }
}
```

Marking: 2 marks

- Use thread_join but still keep the locks: -1

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?

Whenever an event occurs.

Marking:

- **If students say "exception" it is OK.**

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

Kernel mode.

(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?

Because an interrupt could have occurred when the OS is executing.

(4) What does `curkstack` store?

The stack pointer of the kernel's stack.

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

Because if we were coming from the kernel mode, then we already have sp pointing to the kernel stack. If from user mode, then sp points to the user's stack, hence we need to change it to the kernel stack first.