CS 423Operating System Design: Midterm answers and grading rubric

Jack Chen

Question 1

Threads are known as “cheap” processes (cheap means lightweight; not stingy). What makes threads cheaper [2 points]? What are the “states” shared by two threads of the same process and what are not [2 points]?

Less time for context switches
Less time for creation
Share many resources
Etc.

Shared: heap, code, data segments.
Not shared: stack, registers, program counter

Question 2

We have discussed how to use the test-and-set instruction to implement a lock on a uniprocessor in the class. Can we use the test-and-set instruction to implement a lock on a multiprocessor environment [2 points]? Explain why or why not [3 points].

Yes

Test-and-set is an atomic instruction that writes to a single memory location and return the old value stored in that memory location.

Question 3

We discuss Linux's Completely Fair Scheduler (CFS) in the class. What does fairness mean in CFS [1 point]? Is CFS really "completely" fair? If not, can you show us an example in which CFS is not that fair [4 points]?

https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt

Sleeping threads, threads doing I/O could receive unjust virtual time accumulation.

Question 4

We are designing a new kernel for a new machine. The processor in this system uses 36 bits for virtual and physical addresses. There have three options for different page sizes:
(1) 4096 bytes (4 KB), (2) 8192 bytes (8 KB), and (3) 65536 bytes (64 KB). Each of these choices leaves a different number of bits available for the virtual page number. You hear that Jack and Andrew are arguing about which choice allows the largest amount of virtual memory to be addressed. Please explain to them which option, if any, provides the largest amount of virtual memory to be addressed [1 point] and what is the largest virtual memory address [4 points]?

They are all the same.

2^36

Question 5

```
// global variables
struct lock* lock
struct cv* done
void andrew() {
    thread_create("gojackgo", NULL, 0, jack, NULL);
    lock_acquire(lock);
    cond_wait(done, lock);
    lock_release(lock)
    /* do real work */
    do_work();
}
void jack() {
    /* A bunch of initialization stuff */
    config();
    lock_acquire(lock);
    cond_signal(done, lock);
    lock_release(lock);
    /* do real work */
    do_work();
}
```

Do you think the code is correct? Briefly justify your answer. [1 point] If you agree that there is a problem, briefly describe how you would change the code to fix this problem [4 points].

Not correct.

Move lock_acquire(lock) in andrew() one line above.

Question 6

Remember the two ideas of memory management based on segmentation and paging, respectively. Paging splits the address space into equal sized units called pages. While segmentation splits the memory into unequal units that may have sizes more meaningful or appropriate to the program. Knowing these two ideas and reasoning about their pros and cons, Andrew and Jack (as smart grad students) plan to combine these two ideas to build a more advanced memory management system. Andrew proposes to first perform segmentation and then paging, while Jack proposes to first do paging and then segmentation. Whose idea leads to a better designs [1 point]? Please justify your answer by comparing the two designs [4 points].

Andrew's way is better

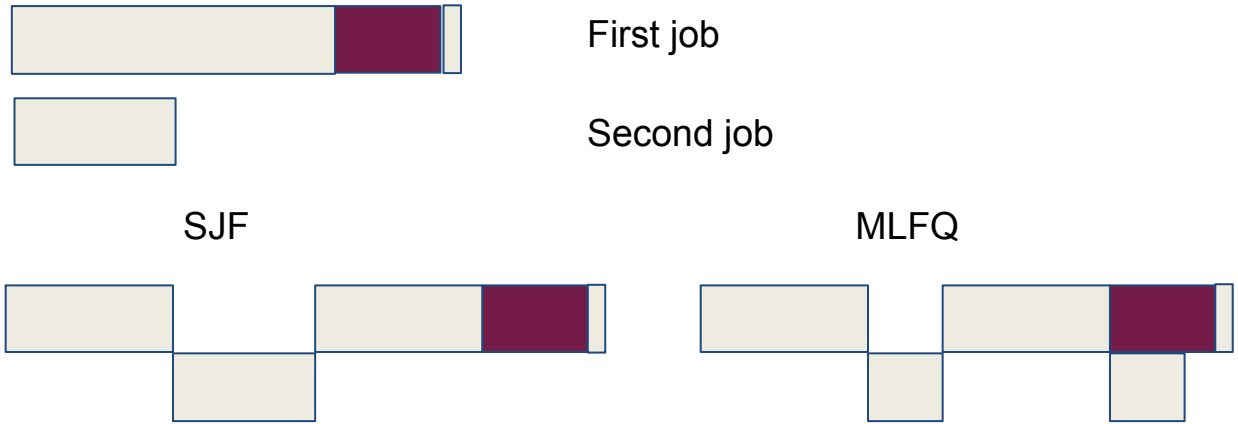
If segments, such as data segment, code segment, and heap exist in each page, it is more difficult to keep track of them in terms of address translation, more difficult for creating segmentations for those segments, etc.

Question 7

We have discussed both Shortest-Job-First (SJF) and Multi-Level Feedback Queue (MLFQ). In the presence of I/O, can MLFQ gives a faster completion time than an SJC scheduler that preempts blocking process [2 points]? Please give an example [3points].

MLFQ can be faster.

Example:



Question 8

```
1 /* Machine-independent context switch code.
2 * curthread is a pointer to the thread data structure of
3 * of the currently executing thread */
4 static void mi_switch(threadstate_t nextstate) {
5    int spl = splhigh();
6    if(curthread != NULL && curthread->t_stack != NULL) {
7       assert(curthread->t_stack[0] == (char)0xae);
8       assert(curthread->t_stack[1] == (char)0x11);
9       assert(curthread->t_stack[2] == (char)0xda);
10      assert(curthread->t_stack[3] == (char)0x33);
11   }
12   if(nextstate == S_READY) {
13      result = q_addtail(runqueue, cur);
14   } elseif(nextstate == S_SLEEP) {
15      result=array_add(sleepers, cur);
16   } else{
17      result=array_add(zombies, cur);
18   }
19   next = scheduler();
20   curthread = next;
21   /*
22    * Call the machine-dependent code that actually does the
23    * context switch.
24    */
25   md_switch(&cur->t_pcb, &next->t_pcb);
26   splx(spl);
27 }
```

In its uniprocessor version, the following snippet shows the machine-independent context switch code:

Can you guess what are Line 5 and Line 26 used for [1 point]?

Enable and disable interrupt

Can we move Line 12-18 to the location after Line 25 (but before Line 26)? Why or why not? [2 points]

No, after md_switch(), the state could change

What do you think is the purpose of Line 6-11? [2 points]

Sanity check.

The implementation of md_switch() at Line 25 is machine dependent (implemented in Assembly). Why can't it be machine independent? [3 points]

Different hardware have different supports for context switch.