Asa Hayes CSCE-313-199 Ahmed June 4, 2020

**Quiz 2**

Part 1: True/False (10pts)

1. After handling a fault successfully, the CPU goes (when it does go back) to the instruction

immediately after the faulting one. **False**

2. Interrupts are asynchronous events. **True**

3. Memory limit protection (within a private address space using base and bound) is

implemented in the hardware instead of software. **True**

4. Memory limit protection checks are only performed in the User mode. **True**

5. Translation Look-aside Buffer (TLB) is a cache for popular (i.e., recently used) page table

entries. **True**

6. Divide by 0 is an example of a fault. **False**

7. Every process has its own page table. **True**

8. A process cannot access its own page table. **False**

9. Trap is a type of synchronous exception. **True**

10. Faults are unintentional but possibly recoverable. **True**

Part 2: Which of the following are privileged operations allowed only in Kernel mode? 11. (10pts)

a) Setting 0 to a large chunk of memory (i.e., using the memset function) **non-Kernel, valid for usermode**

b) Modifying the page table entries **Kernel**

c) Disabling and Enabling Interrupts **Kernel**

d) Using the “trap” instruction **non-Kernel**

e) Directly accessing I/O devices **Kernel**

f) Handling an Interrupt **Kernel**

g) Issuing a system call **non-Kernel, system call is a request to briefly change to kernel**

h) Changing the processor’s execution mode to User mode **Kernel, otherwise would be in user mode already**

i) Divide by zero **Kernel, as an abort exception**

j) Clearing the Interrupt Flag **non-Kernel, requires return from kernel mode after successful completion of interrupt**

Part 3: Short Answer (80pts)

12. [5 pts] Why is the process state (i.e., PC, SP, EFLAGS, general registers) kept in the Kernel Interrupt Stack before handling an Interrupt? Why could we not store it in the user memory? What is the risk?

The process state is kept in the KIS before an interrupt (a so that the program can resume as normal once the interrupt is done, and more importantly b) so that it is not accessible or readable. The risk of keeping the process state in user mode memory is that it could be read by harmful code -whether malicious or just badly written- to alter either the state of the process or attempt to manipulate the memory being accessed by that object. If unintentional this could corrupt memory or crash the system, but if not, the program could be manipulated to do something other than its regular purpose.

13. [5 pts] Describe the attack scenario of how an interrupted process can manipulate the process state in the above question. Note that this is from the Textbook 1 – we did not discuss this in class.

During an interrupt, while the interrupted program may not be able to continue, an alteration may be made to create an attack from another thread. If things are not properly secured, another process may be able to switch out the target of a system call in a program before the interrupt finished being processed. For example, a process on a separate thread could modify the location of a filename to change it to some harmful code after the call is sent but before the kernel-level system can process it, thus running the contents of the malicious file instead of the intended one.

14. [5 pts] While implementing the process state diagram, what is the problem of having only 1 queue for all blocked processes waiting for all events? ﻿What is the solution to this problem? Describe with an example.

The main problem introduced by this fact is that there needs to be a useful way of determining how to sort the queue as to reduce the average program response time as much as possible. If this is not done, simply having the processes wait in the order they went in would be very likely to make the overall system run slowly and be unresponsive while it works thru the queue in plain sequential order. There are several possible solutions for sorting, with one currently appearing as the best for keeping all aspects balanced. Using the multi-level quantum approach, the one queue is converted into multiple smaller queues layered on top of each other. To keep things flowing quickly, each queue has a progressively higher quantum as you go down. Quicker processes stay at the top level and the low quantum allows for many of these smaller processes to be carried out quickly, while longer programs will have more time to run with the larger quantum at lower layers.

15. [5 pts] What is the difference between the “New” state and the “Ready to Run” state in the process state diagram?

The “new” state refers to a program that been created, but cannot run yet. This is due to initialization not being finished for said program, as it requires some amount of computational effort to set up before it can properly cycle thru the “ready-running-waiting” loop. Once it is done, it can be pushed to the “ready” state, but until then, the vast majority of “ready” programs are ones that can start immediately on the CPU once their turn comes, without I/O or other preparation. “Ready” processes are essentially existing processes that were in the middle of processing before they were stopped as their quantum was over.

16. [5 pts] Is a transition from the “Blocked” state to directly to the “Exit” state possible in the process state diagram? How?

No, a “blocked” process cannot directly proceed to exit, as the checks that would cause it to exit are only present in the other steps. Even in the case of an obvious, unhandleable error like divide-by-zero for example, the process would still need to get to “ready” and actually run before the system tosses it out once it throws the associated abort, with the abort setting the flag for the process to exit.

17. [5 pts] Assume that the following physical memory is full with already allocated 5 pages as shown below (i.e., it is 20KB in capacity). Describe what happens if process 2 wants to allocate and use another page. What changes in the page tables and the physical memory? (Image not copied)

As Process 2 is currently active and needs the memory, one of the pages corresponding to Process 1 is virtualized (copied to virtual memory). This frees up space for process 2 to allocate the now empty physical block for its needs. Once Process 2 is done, the new block will then itself be copied to Process 2’s virtual memory to allow for Process 1’s virtualized block to occupy that space.

18. [10 pts] The following are steps in a “sequential” Interrupt handling. (Image not copied)

(a) What changes would you make in the steps below so that “nested” Interrupts can be handled?

For nested loops, we would need to alter how these steps handle both user-kernel switching and interrupt masking. To do this, we would follow regular procedure until the end of step 5 (i.e. once all state flags have been copied), but then re-activate interrupts before the software handling happens. From here on, organization of interrupt “depth” must be done using the interrupt stack, but the result of calling further interrupts inside of the handler code should be to where each successive interrupt switches not from user to kernel, but from one level of kernel to a deeper, higher-priority level.

[Note: B and C assume steps are same as displayed on handout, not as described in 18a.]

(b) [3 pts]: Can you interchange steps 2 and 3? Why or why not?

No. As step 3 requires accessing the Kernel Interrupt Stack, you would need to switch to kernel mode first, so these steps cannot be switched.

(c) [2 pts]: Can we interchange step 1 with step 2? Why or why not?

No. Masking interrupts needs to be done before any mode switching, else multiple conflicting interrupts might corrupt the interrupt table’s values.

19. [20 pts] (a) Write the steps in a system call to read a byte from the keyboard. Refer to page 31 of Lecture 05 as a starting point. However, note that this will require more steps because the user may not type anything in the keyboard immediately. What is the state of the process while it is waiting for the keyboard input? How does it come back to the CPU? Discuss with the help of Process State Diagram.

1. Program reaches point to make system call. (Start > ready > running)

2. Generate interrupt; copy user memory & data down to kernel (running > blocked)

3. Kernel handler must wait until an input is received to carry out its code (blocked, can’t proceed without input.

4. Upon receiving input, interrupt handler code continues and returns the desired result (blocked > ready)

5. Value of the keyboard byte that was called for are now copied back over to usermode process and finishes the program. (ready > running > exit)

(b) Now repeat the above when the user presses Ctrl+C from the keyboard without providing a valid input? How does this process navigate the process state diagram?

The process is the same as above, but the user-level code calls for another interrupt upon receiveing an invalid result for the received keyboard bytes, so steps 2-4 are repeated with the same listed associations on the Process State Diagram until a proper input for the keyboard is provided while the process is blocked and waiting.

20. [15 pts] What is the output of the following program? Assume that the first processes pid=x, and every subsequent process’s pid increases by 1, because that is how usually PIDs are assigned. Explain the output that you observe after running the program using a diagram. (Code not copied)

PID: 75

PID: 76

PID: 76

PID: 77

PID: 77

PID: 78

PID: 76

PID: 79

PID: 75

PID: 80

PID: 80

PID: 81

PID: 75

PID: 82

Diagram on next page

For the diagram, each line is the depth (-- is one level deeper), and the index of the loop in the parent process

75 (parent), i=0

--76, i=0

--76, i=1

----77, i=1

----77, i=2

------78, i=2 (78 done, i=3 after this, no more loops or forks)

----77, i=2 (77 done, i=3 after this, no more loops or forks)

--76, i=2

----79, i=2 (79 done, i=3 after this, no more loops or forks)

--76, i=2 (76 done, i=3 after this, no more loops or forks)

75, i=1

--80, i=1

--80, i=2

----81, i=2 (81 done, i=3 after this, no more loops or forks)

--80, i=2 (80 done, i=3 after this, no more loops or forks)

75, i=2

--82, i=2 (82 done, i=3 after this, no more loops or forks)

75, i=2 (75 done, i=3 after this, no more loops or forks)

On the last iteration of a process (when i=2), it creates 1 fork which then immediately exits as it cannot iterate, then said process resumes, cannot iterate anymore either, and closes.