Student Name:..…………………………………………………………………. UIN:………………………………..

**Student Score / 100**

True/False Questions [10 pts]

1. After handling a fault successfully, the CPU goes (when it does go back) to the instruction immediately after the faulting one *(False, it goes to the current one again)*
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 *(True)*
9. Trap is a type of synchronous exception *(True)*
10. Faults are unintentional but possibly recoverable *(True)*
11. [10 pts] Which of the following are privileged operations allowed only in Kernel mode?
12. Setting 0 to a large chunk of memory (i.e., using the memset function)
13. Modifying the page table entries
14. Disabling and Enabling Interrupts
15. Using the "trap" instruction
16. Directly accessing I/O devices
17. Handling an Interrupt
18. Issuing a system call
19. Changing the processor’s execution mode to User mode
20. Divide by zero
21. Clearing the Interrupt Flag

Short Questions

1. [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?

* *Because a user program can modify whatever the kernel keeps in its stack. Now, note that the current thread is stuck while the interrupt is handled. So, it must be another thread with access to the private address space who can be scheduled in another CPU/core and make modifications while the Kernel is in the interrupt handler.*

1. [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.

* *Although the PCB is in user memory, another process cannot access it because each process is limited to its own private address space. However, the interrupted process is stuck until the interrupt handling finishes, and thus cannot cause any damage to its own PCB. Therefore, the culprit must be another thread of the same process (another thread of the same process has full access to the entire private address space) that is scheduled in another processor/core who can modify the PCB and fool the kernel to jump somewhere else or do something undesired.*

1. [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 is inefficiency. When an event happens, with a single queue, the Kernel must look through the queue to find all processes who wait for the event. This makes each such operation O(n). Per-event queue is much more efficient because the kernel can just wake up the next process from the FIFO queue.*

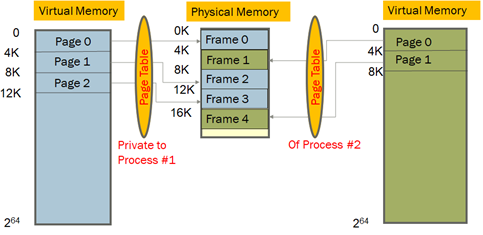
1. [5 pts] What is the difference between the "New" state and the "Ready to Run" state in the process state diagram?

* *New has only the Process Control Block (PCB) created. However, the program executable image is yet to be loaded.*

1. [5 pts] Is a transition from the "Blocked" state to directly to the "Exit" state possible in the process state diagram? How?

* *Yes. Because either the user or the parent process has just killed this process. In addition, the device itself can crash causing the program to crash.*

1. [5 pts] Assume that the following physical memory is full of 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?



*An unpopular frame in the physical memory is chosen according to some policy (e.g., Least Recently Used), that frame is evicted to the disk, the corresponding page table for the corresponding process is updated to reflect this change (i.e., that the page now belongs in the disk), that freed up frame is associated with the missing page that the faulting process wants, change reflected in its page table and finally the fault returns to the faulting instruction.*

1. (a) [10 pts] The following are steps in a “sequential” Interrupt handling. What changes would you make in the steps below so that “nested” Interrupts can be handled?

**Hardware** does the following:

1. *Mask further interrupts*
2. *Change mode to Kernel*
3. *Copy PC, SP, EFLAGS to the* ***Kernel Interrupt Stack*** *(KIS)*
4. *Change SP: to the KIS (above the stored PC, SP, EFLAGS)*
5. *Change PC: Invoke the interrupt handler by vectoring through the Interrupt Vector Table (i.e., overwrite PC with the handler PC)*

**Software** (i.e., the handler code) does the following:

1. *Stores the rest of the general-purpose registers being used by the interrupted process*
2. *Does the rest of interrupt handling operation?*

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

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

Answer:

*(a) You can re-enable Interrupts right before going to Software or as the first step of software. You cannot eliminate disabling interrupts because that is necessary for making sure that half the information about an Interrupt is not overwritten by another incoming Interrupt.*

(b) Yes, because the h/w does not care what mode it is in to execute some kernel mode operation (e.g., access KIS).

(c) Yes, for the same reason as (b)

*More note: Step 1 provides a lock-like mechanism to protect all the remaining steps in hardware so that another interrupt does interfere with these and erase the record of the interrupted process. If interrupts are not disabled first, another interrupt can take control away after we are done saving half of the required information and they can now be overwritten.*

**foo(){  
 open(“test”,“rw”);**

**}**

User Program

**open(arg1,arg2){  
 push SYSOPEN   
 trap  
 return**

**}**

User Stub

**open\_handler(arg1,arg2){  
 //do operation**

**}**

Kernel

**open\_handler\_stub(){  
 //copy args from user memory   
 //check args  
 open\_handler(arg1,arg2)  
 //copy return value to user mem.  
 return**

**}**

Kernel Stub

(1)

(6)

(2)

(5)

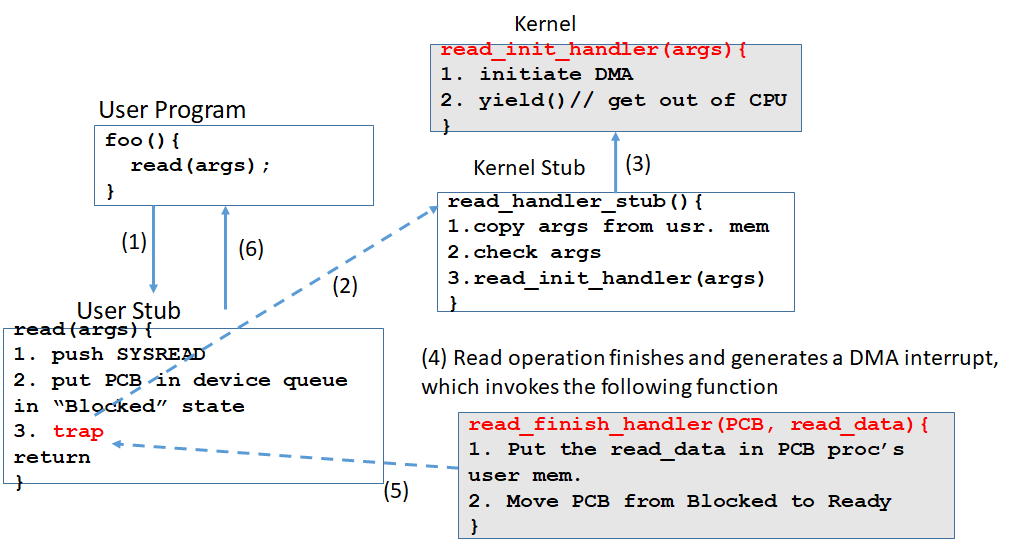
(3)

(4)

1. [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.

*-Answer: The difference from the scenario in the Lecture 05 page 31 and the one here is that although opening a file is elevate mode operation, it does not necessarily take the latency for a I/O operation. So, the process did not need to be in “Blocked” state for the example shown in lecture. However, this current problem will require the process to go to the “Blocked” state first. That is the primary problem to solve here.*

*The following would be one of several possible answers:*



(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?

*Answer: The above key combination (Ctrl+C) generates a signal (i.e., SIGINT) that kills the process while it was in the Blocked state. That means, step 1-3 happen as they are in the above. The difference happens in step (4) where the signal comes before DMA finishes. This signal is forwarded to the process, which results in the process being terminated.*

*[Note: in the Signals lecture later in the semester we will see the mechanism of how a process terminated upon receiving a signal. That explanation is not expected at this point].*

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. Can you explain the output that you observe after running the program?

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

void main (){

    for (int i=0; i<3; i++){

        fork();

1000

        printf ("PID: %d\n" getpid());

        wait (0);

    }

}

1007

1006

1005

1004

1003

W

1002

1001

The numbers above the boxes show the PIDs of the processes as they are created, assuming the first one is 1000. The number x inside the P(x) shows when this print will happen. P(1) from both 1001 and 1001 says that either process can print it first and then the other one. So, the order in box diagram is as follows. You can put any one from these boxes first and the other one second. But the boxes happen in the order show.

1000  
1007

1005  
1006

1000  
1005

1001  
1004

1002  
1003

1001  
1002

1000  
1001