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

**Student Score / 80**

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**
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 **FALSE**
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**
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?

The kernel requires the process state registers to return to the main process. If there is an issue during the interrupt, the user memory register could be lost and the cpu would not be able to return to the process.

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.

Blocked processes are waiting for some event to occur, such as an I/O operation. A queue is a FIFO structure. If a process at the back of the queue and ready, it will have to wait till it gets to the front. A round robin process will fix this by checking each process one at a time to see whether it has been unblocked and let it move forward.

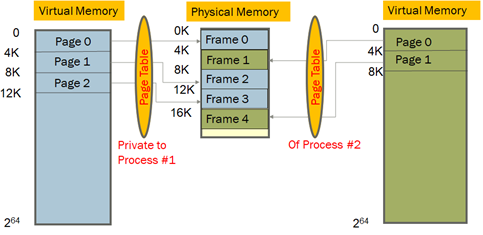
1. [2 pts] What is the difference between the "New" state and the "Ready to Run" state in the process state diagram?

A program in the new state has been initialized but does not have memory assigned to it. The ready to run processes are processes that are scheduled by the scheduler in the ready queue.

1. [3 pts] Is a transition from the "Blocked" state to directly to the "Exit" state possible in the process state diagram? How?

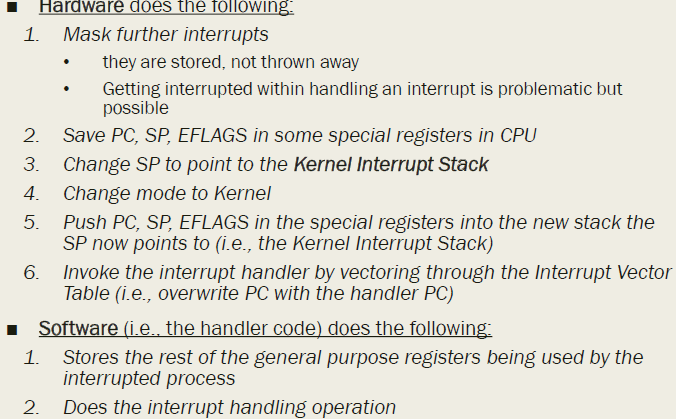
A transition from the blocked state to the exit state would be possible if the process is terminated by being killed explicitly.

1. [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?



The oldest page should be kicked out to disk and put the newest page in the RAM.

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? Reenable interrupts between the software steps 1 and 2.



(b) [3 pts]: Can you interchange steps 4 and 6? Why or why not?

The steps cannot be interchanged as the process cannot invoke interrupt handler if not in kernel mode.

(c) [2 pts]: Can we interchange step 1 with step 2 or 3? Why or why not?Step 1 can be interchanged with step 2 without any problems as the actions are exclusive. However, step 1 and step 3 cannot be interchanged as step 3 changes the Stack Pointer. This pointer needs to be saved in step 2 before changing it otherwise it would be impossible to return to the original process.

1. [15 pts] Write a C/C++ program that executes the command “ls” with standard output redirected to a file “a.txt”. The same program then asks the user to enter a search key (using cin), reads the a.txt file to find it and reports whether t he search key was found among the list of files in the current directory or not. Note that you must use an exec() function to run the “ls” command and before that, you must redirect STDOUT to the file a.txt. Also, at the end, you must report the result in the STDOUT. You cannot use the “system()” function. Make sure to test your program.

Attached as parent\_child.c

19 [10 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();

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

        wait ()

    }

}

After the program is run, the process IDs are printed. When fork() is called, a child process is initiated with the same PC counter, which runs concurrently from the next line. When wait() is called, if the process has any child processes, it pauses until all child processes terminate. The processes continue to create their own children until the value of i in the child goes above 2. The value of i is unique in each process. Iterating i in one process does not iterate it in all the processes. Therefore, the pid is printed for each process that has value of i below 3. When it goes above 2, the process terminates, and the parent runs and creates children till it goes above 2 and terminates. This program is recursive and outputs 14 times.

