CSCE 313 **QUIZ 2** Spring 2020

Student Name:……..Siyuan Yang……. UIN:……826006958……..

Student Score ____ **/ 100**

True/False Questions [10 pts]

1. **[True]** After handling a fault successfully, the CPU goes (when it does go back) to the instruction immediately after the faulting one
2. **[True]** Interrupts are asynchronous events
3. **[True]** Memory limit protection (within a private address space using base and bound) is implemented in the hardware instead of software
4. **[True]** Memory limit protection checks are only performed in the User mode
5. **[True]** Translation Look-aside Buffer (TLB) is a cache for popular (i.e., recently used) page table entries
6. **[False]** Divide by 0 is an example of a fault
7. **[True]** Every process has its own page table
8. **[False]** A process cannot access its own page table
9. **[True]** Trap is a type of synchronous exception
10. **[True]** Faults are unintentional but possibly recoverable

11. [10 pts] Which of the following are privileged operations allowed only in Kernel mode?

a) Setting 0 to a large chunk of memory (i.e., using the memset function)
b) Modifying the page table entries
c) Disabling and Enabling Interrupts
d) Using the "trap" instruction
e) Directly accessing I/O devices
f) Handling an Interrupt
g) Issuing a system call
h) Changing the processor's execution mode to User mode
i) Divide by zero
j) Clearing the Interrupt Flag

Short Questions

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?

Because user stack is unprotected from user manipulating the old return address, SP and EFLAGS, etc. However, the same user code would be stuck and not make progress. Other processes cannot manipulate these because they are outside the address space. Other threads in the same process will have access to this user stack and thus will be able to change these crucial values.

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.

When the kernel finishes handling the request, it resumes execution of the interrupted process by restoring its program counter, restoring its registers, and changing the mode back to user-level.

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.

While implementing the process state diagram, a blocking system call informs the kernel that a process is waiting as the system call returns to its caller only when a specific external event occurs. The solution to this problem is a single blocked queue containing all process that are blocked regardless. For instance, a separate queue for each terminal device and inbound IPC is an example.

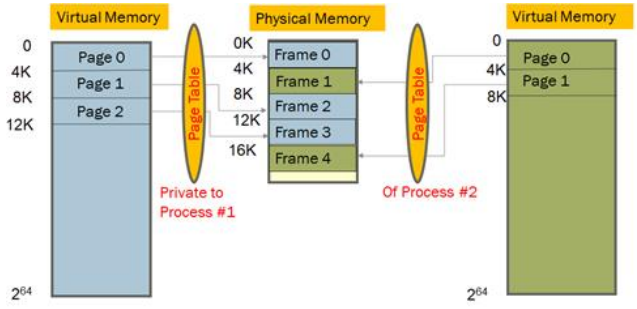
15. [5 pts] What is the difference between the "New" state and the "Ready to Run" state in the process state diagram?

In the new state, the process is about to be created but not yet created, it is the program which is present in secondary memory that will be picked by OS to create the process. However, in the ready state, the process is loaded into the main memory. It is ready to run and is waiting to get the CPU time for its execution.

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

Yes. When a parent process terminates the child process, the transition is done from blocked state to exit state. First, the process goes to the suspended state since the process is terminated to free some space in main memory. After the suspended state, the process goes to the blocked state. If there is any faut situation in the process in the blocked state, the process will directly go to the exit state.

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?



When a process requests allocate and use another page, the operating system will map the virtual address provided by the process to the physical address of the actual memory where the data is stored. The page table will store the new mappings of virtual addresses to physical addresses with each mapping also known as page table entry.

18. (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?

1->4->2->3->5

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?*

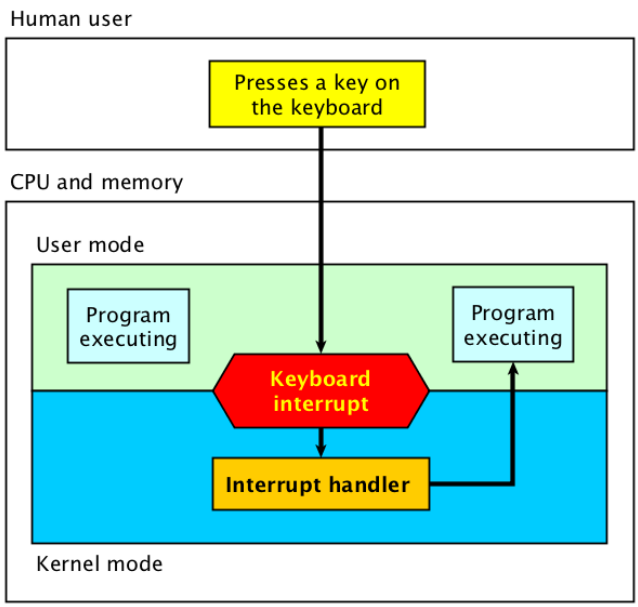
(b) [3 pts]: Can you interchange steps 2 and 3? Why or why not?

No. Since EFLAGS contain the current mode of the processor, we cannot interchange steps 2 and 3 because that is when EFLAGS is backed up to a temporary place.

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

No. If we do not mask further interrupts first, other interrupt might come before the currently interrupted process state can be saved completely. Then, it would be impossible to go back to this process after changing mode to Kernel.

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.



To read a byte from the keyboard, a device will repeatedly check the state of the input device until it detects a key press. Steps can be summarized as below: 1) A program cannot proceed until an external device becomes ready. 2) The program checks the status of the external device. 3) The program can now proceed and read a byte from the keyboard.

The state of the process waiting for the keyboard input is called busy-waiting as the program continuously polls the device without doing anything in between checks. The system will use a

conditional loop so that CPU can repeatedly load the value of the status register to check if new input is available. While waiting for input, the CPU doesn't do any work.

(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?

While pressing Ctrl+C from the keyboard, an interrupt will occur, and control is automatically transferred to the interrupt handler executing in kernel mode. The interrupts generated by other hardware devices outside the CPU make it possible for the CPU to execute the program of copying the data while waiting for user input.

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.

```
#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 ();
    }
}
```

The output of the program will be 8 PID printed one by one in random order. Starting from x, the PID value will increase by 1 until reaching x+7. The reason is that the fork() function returns the child PID for the parent and returns 0 for the child. It depends on to which process it returns. The diagram is shown below: