Department of Computer Science and Engineering, Indian Institute of Technology Palakkad

CS3110: Operating Systems Lab

19th September, 2019 **Viva prelims** **4:30pm - 5:00pm**

1. **(Stages 16)** The `STATUS` field in the terminal status table is sufficient to keep track of whether the terminal is busy or is free. Then, what are the reasons to have a `PID` field in the terminal status table? What will happen if this field is not set?
2. **(Stage 16)** Suppose you have implemented upto stage 16.
   - Imagine a situation where the terminal status was free and eight user processes made read system calls to read from the terminal. Is it guaranteed that they enter user mode back in the same order as they made the read system call? Why?
   - `TerminalWrite` function calls both the `AcquireTerminal` and `ReleaseTerminal` functions whereas the `TerminalRead` function calls only `AcquireTerminal`. Then, how is the terminal status again made free after receving the data in the input port?
3. **(Stage 17)** In the implementation of exec system call, there is a call to `Exit Process` function of process manager module; which in turn calls `Free User Area Page`. Instead of resetting SP to the beginning of user area page in `Free User Area Page` function, it is done in the `Exec System Call` Handler (after returning from `Exit Process` and adjusting the memory free list entry and `MEM_FREE_COUNT`). What will go wrong if SP was reset in `Free User Area Page` function?
4. **(Stage 17)** In the implementation of `Exec` system call, how is it ensured that the execution of the new program loaded starts with the correct IP value?
5. **(Stage 18)** Suppose the disk interrupt handler is modified, so that it makes only one of the processes in `WAIT_DISK` state (specifically, the process whose pid is in the Disk Status Table) to ready state. Then, what will go wrong?
6. **(Stage 18)** What are the different ways in which a process can enter WAIT_DISK state? How does it come out of the WAIT_DISK state in each of these ways, and what does it do in each of these cases when it eventually gets scheduled on the processor?
7. **(Stage 18)** Suppose you have implemented upto Stage 18. Suppose a user process is making an exec system call. Before starting to execute the newly loaded program in its user mode, which are the places in the kernel code that may cause the process (with the caller's pid) to wait?
8. **(Stage 19)** At this stage, demand paging is implemented only one code page of a process is loaded into memory initially. Consider a user program which has two code pages and no heap pages. When an exception due to page fault occur for this process for the first time, what would be the values of `EC` and `EPN` registers? At this time, is it possible to predict the value of `EIP` register precisely? Why or why not?
9. **(Stage 19)**
   - Why can code pages be shared without any issues? Indicate example(s) where such page sharing was previously used? What information is stored about such sharing and where? Why isn't the PID of the processes sharing a page stored?
   - Under what circumstances is `ExitProcess` invoked from the exception handler?
10. **(Stage 19)** How is it ensured during a page fault exception handling that after loading the required page to memory, the execution in user mode restarts with the same instruction which caused the exception?

11. **(Stage 20)** Suppose we want to modify the fork system call implementation such that after a successful fork, the child process is started in the user mode before returning to the parent process. What changes would you need in your kernel code to implementing this?

[END]