Midterm Test   
CSCE 313 Summer 2020  
120 points

Date: June 10, 2020

Student Name: …Asa.Hayes…………………………….

UIN:…525003952………………………….…………………

I have adhered to Aggie Code of Honor

Signature:……Asa.Hayes…………………………………………………….

TRUE/FALSE Questions [1 pts each]

1. In multiprogramming, multiple jobs are preloaded in memory to save time switching them. **True**
2. Interrupts are mandatory to bring “paused” programs back to CPU. **True**
3. Multiprogramming is ineffective when all running programs are made of “CPU-only” operations. **True**
4. Interrupts are checked between every two CPU instructions. **False, only if interrupt is called**
5. **Efficiency** is the secondary goal of an OS, right after the primary goal of **Convenience**. **True**
6. During every context switch, the scheduler runs to decide which process to run next. **False**
7. Context switch can cause significant overhead if implemented inefficiently. **False, overhead from bad/over usage, overhead of switch itself is not high**
8. Every process gets its own private address space that is isolated from other processes. **True**
9. Kernel is mapped to every process’s address space. **True**
10. Memory protection (to protect kernel) is implemented in hardware. **True**
11. Opening a file is an example of Asynchronous Interrupt. **True, file operations are I/O**
12. A **Trap** always return control to the immediately following instruction. **True**
13. Implementation of a System Call requires context switch and mode elevation. **False, not necessarily**
14. A Fault returns control (if it does at all) to the same instruction. **True**
15. A page fault may lead to a segmentation fault. **True**
16. A Round Robin (RR) scheduler is always better than FIFO. **False**
17. SRTF guarantees shortest Average Response Time (ART). **True**
18. FIFO is a fair scheduling method. **False**
19. Increasing higher device utilization increases the system throughput. **True**
20. Larger RR time quantum may lead to unresponsive behavior for some I/O bound tasks (e.g., iPad touch events, mouse clicking). **True**

Short Questions (be precise, long answers are not expected/desired)

1. [10 pts] Describe how I/O operation by 1 program can happen simultaneously with CPU operations by other program(s). What is the main technique and what hardware component(s) is/are involved in this?

This process is referred to as multiprogramming. In multiprogramming, if the currently running program has to wait for awhile for a response from some operation in the I/O area of the system, that program is essentially put on hold until the I/O operation is done. Its process state (registers, instructions, etc.) in the CPU is saved into hardware memory, and in its place is loaded the process state of a waiting program that can (at least partially) run within the time that I/O is processing the first program’s process. Afterwards, the second process’s state is saved if it isn’t done, and the first process’s state is load back into the CPU to continue.

1. [5 pts] Can a multi-programmed system work without Interrupts? Why or why not?

No. A multi-programmed system requires separation between I/O operations and the main CPU. Without interrupts or something functionally similar, there would be no way to separate I/O operations out from regular CPU function, and there would be no way to pause just the main operations during I/O operations to let other processes have a turn at the CPU.

1. [15 pts] Describe the steps necessary to perform the read() system call for getting a 100-byte record from a disk file. What different states the process (performing the read operation) can be in? Discuss using the process state diagram.

1. During a currently running process (in “running” state), the read() command is called and initiates the associated system call.

2. This pushes the parameters of read(), that being place to read form, buffer to read into, and number of bytes to copy, to the interrupt stack and then calls the interrupt. The program transitions from “running” to “blocked” until the interrupt is finished.

3. With the parameters in, the code for read() is then set to the current working registers and pushed to the kernel where t can be fully executed.

4. After the code is in the kernel, it is directed to the relevant system call handler, and the indicated memory is read into the buffer properly.

5. Once the read() operations are finished, the now-filled buffer and other parameters are then moved back into usermode into the interrupt stack.

6. Now that the interrupt has received its desired input, the results of read() are moved back into the original process and moves the counter to the process’s next instruction as read() is now finished. The process moves from the blocked state to the ready state, and can continue with the results of read() in its memory once the scheduler picks it up.

1. [15 pts] Assuming each page is 4KB, how many page faults the following program will generate? Can the number of faults be different from one system to another? Explain your answer.

class Header{

public:

    int data;

    bool status;

};

int main (){

    int size = 4 \* 1024 \* 1024; // 4 mega bytes

    char \* memory = new char [size]; // allocate memory

    Header\* h = (Header\*) memory;

    Header ho;

    // copy header object ho onto memory count times

    int count = 512 \* 1024;

    for (int i=0; i<count; i++){

        \*h = ho;

        h++;

    }

}

This code would generate 1024 page faults. The regular calculation of [allocated memory size / page size = # of page faults] still applies. While the loop might seem to only go thru 0.5MB, remember that each Header object takes 8 bytes, and will thus take up 8x more space than count’s value. This leads to it taking up the entire allocated space. As it does no go past the allocated space, there is no extra page fault to lead into a segment fault. Differing amounts of page faults from one system or another is not possible, as all objects and types are by necessity the same size on all systems for compatibility. The only thing that might change amount of page faults per system is if different systems had different sizes of pagefile, but that’s not exactly system-dependent as page file size can be adjusted by the user in most cases.

1. [20 pts] Consider the following program. Assume that process IDs are always increment-by-1 and the number depends on which order they are created. For instance, if process X is created after process Y, the PID of process X is greater than that of Y. Now, what will be the output of the following program? How many processes will be created? Describe with help of a process tree and based on how wait() function works.

You can assume that the first process’s ID is 1000 and add 1 for each subsequent process in the order they are created

**for (int i=0; i<3; i++){**

**int f = fork();**

**if (i == 0 || i == 2)**

**wait(0);**

**cout << "PID = "<< getpid() << endl;**

**}**

Program Output (Assuming initial parent PID = 1000):

PID = 1001

PID = 1001

PID = 1002

PID = 1003

PID = 1004

PID = 1001

PID = 1002

PID = 1000

PID = 1000

PID = 1005

PID = 1006

PID = 1007

PID = 1000

PID = 1005

Processes Created: 8

Note: Diagram Attached as Figure A.

1. [20 pts] What will the file file.txt contain after running the following program? What is in the standard output? Explain your answer by drawing the Descriptor Tables (DT) and File Table (FT) for all processes (i.e., from this program) right after executing lines 5, 9, and 11. (Note that most points are allocated to the correct explanation. Just mentioning the file content/output w/o explanation will not earn you any point).
2. int fd = open("file.txt", O\_CREAT|O\_WRONLY|O\_TRUNC, 644);
3. write (fd, "Howdy world", 6);
4. if (!fork()){
5. dup2 (fd, 1); //redirect stdout
6. cout << "Mars, I am here to conquer." << endl; //draw tables
7. }else{
8. wait(0);
9. dup2 (fd, 1);
10. lseek (fd, 25, SEEK\_SET); // draw tables
11. cout << "to learn and grow."<< endl;
12. write(fd, "Thanks.", 7);//draw tables
13. }

Description (Tables per step included as Figure B):

Line 1 opens/creates ‘file.txt’ with parameters to limit and specify read/write/other utilities, and is then filled with “howdy ” on line 2. Not with “howdy world”, as it only writes in 6 chars.

At line 4, the dup(fd,1) command redirects the path of the program’s output from stdout (i.e. prints to terminal) to writing path for ‘file.txt’. Line 5’s cout command is now pointing to ‘file.txt’ instead of standard out, so the string gets sent there. ‘file.txt’ now contains: “howdy Mars, I am here to conquer.”

At line 9, the position of the cursor (where new text begins to be inserted into the file) is changed to 25, placing it right before the start of “conquer”. The cout on line 10 then inserts the given string at that point, overwriting “conquer”. ‘file.txt’ now contains: “howdy Mars, I am here to to learn and grow”. The “to to” could be fixed if the cursor was set to 22 instead of 25 in the lseek command.

At line 11, since the cursor is now past the point where it would overwrite anything and that the writelength specified now contains the whole string, this write command functions as normal. Final contents of ‘file.txt’ are: “howdy Mars, I am here to to learn and grow.\nThanks.” [\n indicating moving to a new line.]

As it has not been switched back, the pointer to the file is still in standard output. You would need to use “int terminal\_location = dup(1);” to save the pointer to the terminal before calling dup(fd, 1) on lines 4 or 8, and then calling dup2(terminal\_location, 1) to restore standard output at the end of the file.

23. [15 pts] Assuming 0.5 sec overhead per context switch (i.e., time between 2 user processes) in a 1-CPU-1-core system, schedule the following workload and compute Average Response Time (ART) under:  
a. Shortest Remaining Time First (SRTF) (preemptive)  
b. Round-Robin (RR) with time quantum=2sec

|  |  |  |
| --- | --- | --- |
| Process | Arrival Time(s) | CPU Service Time(s) |
| P1 | 1 | 3 |
| P2 | 3 | 4 |
| P3 | 2 | 2 |
| P4 | 2 | 1 |
| P5 | 4 | 3 |

Diagrams and Work included as Figure C.

a) ART(SRTF; preemptive) = 6.3s

b) ART(RR; q=2s) = 8.3s