CS 111 Midterm exam

Kevin Zhang

TOTAL POINTS

85 / 100

QUESTION 1

1 Page replacement for multiprogrammed environment 10 / 10

✓ - 0 pts Correct

QUESTION 2

2 Purpose of trap instruction 10 / 10

✓ - 0 pts Correct - switch from user mode to kernel mode, with handling context switching

QUESTION 3

3 Scheduling policies and turnaround time 10 / 10

✓ - 0 pts Correct

QUESTION 4

4 Interface compatibility 10 / 10

✓ - 0 pts Correct

QUESTION 5

5 Benefits of paged virtual memory 10 / 10

✓ - 0 pts Correct

QUESTION 6

6 Performance metrics under load 10 / 10

✓ - 0 pts Correct

QUESTION 7

7 Protecting critical sections 5 / 10

✓ - 3 pts Missing one answer. (Note: locks, semaphores, completion events, spin locks, etc. count as only one technique, since all of them ultimately rely on the lock to achieve consistency.)

✓ - 2 pts You discuss atomic instructions only in context of implementing locks, which is another technique you mention.

QUESTION 8

8 Swapping vs. paging 10 / 10

✓ - 0 pts Correct

QUESTION 9

9 Paged VM fragmentation 10 / 10

✓ - 0 pts Correct

QUESTION 10

10 Synchronization code bug 0 / 10

✓ - 10 pts Proper semaphore implementation will prevent such a race.

Midterm Exam
CS 111, Principles of Operating Systems
Summer 2018

Name: Kevin Zhang

Student ID Number: 104939334

This is a closed book, closed note test. Answer all questions.

Each question should be answered in 2-5 sentences. DO NOT simply write everything you remember about the topic of the question. Answer the question that was asked. Extraneous information not related to the answer to the question will not improve your grade and may make it difficult to determine if the pertinent part of your answer is correct. Confine your answers to the space directly below each question. Only text in this space will be graded. No question requires a longer answer than the space provided.

1. In a multiprogramming system, why is a simple clock page replacement algorithm applied to the full set of pages likely to lead to poor performance?

Clock page replacement models LRU (least recently used) algorithm which tries to select the least recently used page to evict from the page table or TLB. However, in a multiprogramming setting, multiple processes are being run and they all potentially store some of their pages in the page table/TLB. However, this conflicts with round robin scheduling because each time slice is allocated for a different process. When a different process is run, obviously processes that are not running will never be used. Therefore every time a new process runs, the old process pages will all be evicted so when round robin comes back to said process, it will have to page fault for all of the processes. This results in extremely costly context switches that defeats the purpose of caching and doesn't exhibit any locality.

2. What fundamental requirement for a modern general purpose operating system is met by providing a trap instruction in hardware? How does provision of this instruction enable the OS to meet that requirement?

Providing a trap instruction in hardware allows the OS to execute privileged instructions. The OS can set up the trap table at boot so that any privileged instructions coming from the user or any process will trap into the hardware instruction and execute the previously pre-configured trap handlers. The OS gets to decide what to do with traps, be it asychronous errors or synchronous traps.

Regardless the trap instruction is fundamental to allow the OS to handle and execute privileged instructions.

3. Why does the Shortest Job First scheduling policy typically result in better turnaround time than the First Come, First Served policy?

Shortest Job First scheduling typically allows users to circumvent the issue of the queue starting off with an extremely long process. In this case, since turnaround time is calculated as arrival time to the finish time, a long process at the front of the queue will get executed and this time will be added to the turnaround time of shorter length jobs. However Shortest Job First, preempts these longer jobs so that shorter jobs can complete so overall average completion (turnaround) time is shorter b/c shorter jobs finish first and longer jobs turnaround times don't get affected much.

4. Describe a way to allow for otherwise incompatible interface changes without sacrificing backwards compatibility. Why might such changes be valuable?

The way to allow for incompatibe interface changes is version control. Updating the API with new interface change can be packaged in a new version of the software but this doesn't sacrifice backwards compatibility because users can still adhere to older versions of the API without any loss of functionality. These changes are valuable because developers might want to expand or enhance their software in new directions that might nullify or deprecate old API functionality. These changes are extremely valuable to the growth of a development group's technology.

5. Describe two benefits of virtualizing memory addresses at the page level, including why virtual memory provides that benefit.

Virtualizing memory addresses allows easier relocation of pages and segments of memory b/c all references are relative to a base address. Virtual memory converts to physical addresses through a base address and if a memory space is relocated, all the computer has to do is change the base address and all references in that program will be valid again. Furthermore, virtualization at the page level allows the different code segments to be separated, not necessarily occupying the same contiguous chunks to better utilize memory and reduce fragmentation. Paging like this gives the illusion of an infinite segment of memory for each process because some pages that are infrequently used can be swapped out onto the disk. Each process will then no longer have to worry about running out of memory. Moreover paging hugely reduces fragmentation to an average of 1/2 page per process

6. When load increases, throughput drops below maximum ideal values, but response time quickly grows to infinity. Why do these two metrics exhibit different behavior under the same condition of increasing load?

Response time grows to infinity because the load surpasses the maximum limit of the computer, forcing the OS to drop processes in order the preserve performance and prevent thrashing. This effectively means the process will never be executed. Throughput on the other hand, is the number of processes finished per second which is only affected negatively because the memory is overbounded so there is more context switching, page swapping, page faulting, and in general, thrashing occurring. Both metrics negatively impact performance metrics but response time is better when its low while throughput is better when it's high.

7. Describe three techniques we can use to ensure proper behavior of critical sections, briefly indicating why each of them can achieve that effect.

compare and swap: Check/compare to a designated variable to see if it has changed, and enter critical section if it hasn't. Don't enter the critical section otherwise. This also prevents multiple executions of the critical section at once since only one can change the lock at a time.

conditional variables: Variables that are shared. When one process/thread enters critical section, it sets that variable and any other process/thread that enters the critical section is blocked in a waiting queue. When the first finishes, a process is taken out of the queue. This ensures only one process is in the critical section at any given time.

Spin Locks: Spin CPU clock cycles until the desired resource is free, indicated by a flag and then enter the critical section. Prevents multiple executions of critical section at once since all waiting processes/threads are spinning while only one is executing.

8. What is the difference between swapping and paging? What is each of these two techniques used for?

Swapping is the technique of swapping out process' unused pages onto the disk to conserve memory and allow other processes to coincide in memory. It also allows virtualization b/c it essentially gives each process the illusion of infinite memory.

Paging on the other hand is just a technique of memory management where data, including process segments (text, data, stack, etc) is split up into fixed sized pages of memory that can be accessed more easily through a page table/TLB. Paging decreases fragmentation dramatically and allows for a fast and efficient way of splitting up memory. Using page tables/TLB, we can also access memory in a fast, structured way. Both swapping and paging are memory management techniques but paging is a way memory is set up while swapping is a technique to move pages or other memory units like fixed memory pieces into and out of memory. Swapping helps to ease memory pressure, free up unused or infrequently used memory; and provide the illusion of infinite address space. Paging is a way to structure memory and form a structured way of accessing memory.

9. What kind of fragmentation will a paged virtual memory system experience? For each segment that a process requires, how much of that kind of fragmentation will a paged virtual memory system exhibit, on average?

A paged virtual memory system experiences internal fragmentation where the last page of a sequence of pages might have some excess storage memory that is unused. On average, fortunately, paging is very space efficient and only exhibits around 1/2 a page of internal fragmentation per process.

10. The following C code is intended to use semaphores to control reading and writing from a circular buffer by two different threads. It has a serious synchronization bug. Find the bug and describe (in words – you need not write the actual code) what further synchronization operations are required to fix it. NOTE: Obviously the code shown below is not a complete program. The necessary pieces missing to create a complete program, such as a main routine and thread creation code, are not relevant to the answer. Also, I am looking here for a synchronization bug. If you find and specify some other bug that does not have synchronization issues, you will not get any credit.

```
struct semaphore pipe_semaphore = { 0, 0, 0 }; /* count = 0; pipe empty */
char buffer[BUFSIZE]; int read_ptr = 0, write_ptr = 0;

/*  This code is run by thread 1.  */

char pipe_read_char() {
    wait (&pipe_semaphore );                    /* wait for input available */
    c = buffer[read_ptr++];                     /* get next input character */
    if (read_ptr >= BUFSIZE)                    /* circular buffer wrap */
            read_ptr -= BUFSIZE;
            return(c);
}

/* This code is run by thread 2.*/

void pipe_write_string( char *buf, int count ) {
    while( count-- > 0 ) {
            buffer[write_ptr++] = *buf++;       /* store next character */
            if (write_ptr >= BUFSIZE)   /* circular buffer wrap          */
                    write_ptr -= BUFSIZE;
            post( &pipe_semaphore );            /* signal char available */
    }
}
```

There might be a race condition when posting/waiting for the pipe semaphore where if, in the unlikely, yet still possible, event that both threads run the semaphore wait/post at once, In other words; if the pipe semaphore waits but before it exits that critical section, the post occurs, pipe_read_char thread will not be lifted from the FIFO queue. Therefore, the pipe_read_char() thread will be asleep forever. To fix this a simple mutex lock that locks the wait and post critical sections will allow the synchronization to run w/o errors.