Midterm Exam
CS 111, Principles of Operating Systems
Summer 2018

Name: Yuemping Song

Student ID Number: _____

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?

The processes may be scheduled in a round-robin fashion, which does not cooperate well w/ the clock algorithm that is an approximation to LRU. Even if the scheduler is not RR, there is a non-trivial probability that the amount of physical memory is less than the sum of the multiple processes'/programs' working sets. Therefore, thrashing might occur, because the page the algorithm would like to evict from physical memory onto swap space is accessed by a process that will execute soon but has not run since the last sweep of the "clock hand."

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?

The OS needs to virtualize the CPU, providing each process the illusion that it has exclusive ownership while guaranteeing protection against errant or malicious programs and creating a mechanism for implementing various scheduling policies that favors different performance metrics such as throughput and response time.

The trap instruction provides protection. Trapping into the OS changes the CPU from user mode to privileged mode. So the OS is able to run certain privileged instructions on behalf of untrusted user processes and provide such capabilities as a service via system calls. Also, traps for timer interrupts and asynchronous I/O provides hardware support for OS services w/ much less overhead than if they were implemented in software.

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

SJF ensures that the most number of processes run to completion per unit time, therefore it greedily optimizes average turnaround time.

First-Come-First-Served optimizes throughput by keeping context switches at a minimum. It is not great for turn around time though, because it is non-preemptive. if a time-consuming process is scheduled, subsequent interactive requests cannot run until the scheduled process terminates or blocks, causing slow turn around time for late-arriving, interactive jobs. SJF avoids this problem because upon receiving interactive requests, it preempts the compute-intensive job and schedule the late-comers instead.

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

One may create dynamically loaded libraries and allow each user program to load the version of the library best suited for its use. Eliminating certain deprecated parts helps simplify the interface and makes it easier to use. On the other hand, breaking other people's production code, however outdated, is not a commercially-viable option. Therefore, having many versions of dynamic libraries is a necessary compromise.

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

Pages are a more flexible unit of allocation than fixed-size partitions or variable length segments (which is difficult to expand b.c. it has (base and bounds)
to be continuous). that completely eliminates external fragmentation, while obtaining minimal internal fragmentation. furthermore, pages may be moved in memory because the page table provides a level of indirection, where as in a partitioning approach, moving a chunk invalidates all the pointer values within.

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?

throughput drops because as load increases, the system is spending more and more time handling incoming requests and less doing useful work.

(avg) response time goes to infinity because as system works near capacity, its queue of requests to be processed gets longer and longer. the convoy causes average response time to explode. when a system is totally overwhelmed by the deluge of requests, it simply drops requests it cannot handle, leading to truely infinite response time.

Making the difference b/w the two metrics here is convoy effect, which degrades response time over time, even if load is kept constant.

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

1). The simplest way of having a safe critical section is performing all the operations in it in a single atomic instruction provided by hardware. This works because atomicity is ensured at a hardware level.

2). A second way is to use mutexes, a form of lock. It is built from atomic operations provided by hardware but can ensure large sections of user code is atomic by atomically checking and setting certain flags known to the mutexes.

3). A third way is semaphore, which has a counter and a queue. A requesting process is blocked if the counter drops below zero when decremented and a posting process increments counter, and awakes a process from the queue if there is any.

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

Swapping refers to moving pages on or off a disk region known as swap space from or to physical memory. Paging refers to storing page table info in a fast cache in MMU called TLB.

Swapping is used to extend the size of VM to include disk because physical mem alone usually is not enough. When a process access a page not present in PM, it incurs a page fault. the MMU then swaps the page from disk into mem. Paging is used to aid in memory address translation. Because page table resides in MEM, accessing a memory address would incur too much overhead if we also need to go to RAM to consult page table. Instead, a portion of the page table is cached in TLB, if a process requests an address not in TLB, a TLB miss happens and MMU goes to main memory, and brings in the PTE to TLB, a process called "paging" and retries the memory operation.

semphore achieves tomicity by combining ch wit condition ariables. Different ffects can be chieved by initializing different values

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 VM system will only exibit internal fragmentation. It has no external fragmentation because page (frames) are fixed-sized chuncks (4kB) that fills physical memory and swapp space. It still experiences internal fragmentation because memory requests for space less than the size of a page must be satisfied w/ a full page.

for each segment, suppose the request is of size N. We will allocate $\lfloor \frac{N}{4kB} \rfloor$ full pages and only the last page will be partially filled w/ $N \bmod 4kB$ bytes. Because the residure can be any where between 1 byte and 4095 byte, the expected wastage is exactly half of a page, = 2048 bytes in our case, per segment.

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 */
    }
}
```

Incrementing and decrementing a semaphore should be protected with mutex already held. The semaphore is initialized to 0. Suppose T_1 decrements and T_2 increments at the same time. A race condition occurs and the result is not deterministic. Suppose T_1 reads, T_2 reads, and increments, T_1 decrements and goes to sleep. ~~T_2 runs and posts the semaphore, because T_1 hasn't gone to sleep, T_2 doesn't awaken it.~~ T_2 completes, T_1 runs and sleeps forever. There is a wake/sleep race. The fix is to protect waiting and posting w/ mutex to make those operations atomic. Because the execution has to be interleaved for this bug to happen, making either wait or post atomic would actually suffice.