Midterm Exam
CS 111, Principles of Operating Systems
Fall 2018

Name: Clayton Chu

Student ID Number: 104906833

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. Why is proper choice of a page replacement algorithm critical to the success of an operating system that uses virtual memory techniques? What is the likely difficulty if a poor choice of this algorithm is made by the OS designer?

The proper choice of a page replacement algorithm is critical because if worse algorithm choice can possibly evict a page that gets many references. This is costly since it will have to fetch the page back into memory/TLB again. For example, choosing a random-based algorithm vs. choosing an LRU-approximation algorithm (like clock algorithm): Random algorithm has a chance of evicting a very important page, wasting more time to bring it back into TLB again. The clock algorithm is not likely to do so since it removes "old" pages (assumption is that the page is not in demand if it wasn't recently used) (hindering performance)

2. Spin locks can cause performance problems if not used carefully. Why? In some cases, a thread using a spin lock can actually harm its own performance. Why?

Spin locks can cause performance problems because a thread that cannot acquire a lock will waste CPU cycles for its entire time slice on checking if the lock variable changed (which it won't).

A spin lock can also harm its own performance because the constant checking during its entire time slice wastes time for itself and other threads, meaning that it creates situations where it gets the lock later compared to if it had just yielded once it started spinning. Also, since spin locks are usually unfair (except for fetch-and-add), the thread could be losing the competition with other threads to acquire the lock, resulting in starvation.

3. Assume you are running on a virtual memory system that uses both segmentation and demand paging. When a process issues a request to access the memory word at address X, one possible outcome in terms of how the address is translated and the content of the address is made available is: the address is valid, the page is in a RAM page frame, and the MMU caches the page table entry for X, resulting in fast access to the word. Describe three other possible outcomes of the attempt to translate this address and the actions the system performs in those cases.

Case 2: Address is valid and the page frame is not in RAM
- The page lies on disk and it gets brought into memory
- A TLB miss occurs, the page is found in memory, and the TLB keeps track of the mapping
- A TLB hit occurs b/c the mapping exists inside the TLB, resulting in fast access to the data

Case 3: Address is invalid
- Trap is issued and trap handler handles invalid memory access
- Process is terminated

Case 4: Address is valid and is found in a memory segment instead of a frame
- Consults a data structure containing pointers to various physical memory chunks
- Goes to the appropriate pointer and retrieves the data

4. When a Linux process executes a fork() call, a second process is created that's nearly identical. In what way is the new process different? Why is that difference useful?

The new process has its own stack/registers/address space and different PID. This is useful because then the child process's space doesn't interfere with the parent process's space and the PID allows you to choose what to do differently with the parent and child.

5. If your OS scheduler's goal is to minimize average turnaround time, what kind of scheduling algorithm are you likely to run? Why?

You are likely to run a shortest-time to completion-first or shortest-job-first approximation algorithm. Turnaround time is measured by the difference between completion time and arrival time. Thus, STCF/SJF is optimal because it runs the shortest job first before the longer process or preempts the longer process to complete the shorter process, allowing for more processes to complete in the same time interval. Since these processes require an oracle to be optimal, however, it is much more realistic to use an algorithm that orders processes based on history of turnaround time, on the assumption that history predicts the future behavior of processes well.

↓ It is unlikely to know how long a process exactly runs

6. Assume you start with an operating system performing paged memory allocation with a page size of 4K. What will the effects of switching to a page size of 1K be on external and internal fragmentation? Describe one other non-fragmentation effect of this change and why it occurs.

Switching to a page size of 1k will not affect external fragmentation because external fragmentation only exists for variable page sizes. It will minimize internal fragmentation because smaller page size means less likely to have wasted space inside it (↑page more likely to be completely used).

Decreasing page size while maintaining the same amount of total memory will result in increased page table entries and a longer lookup time in page tables. This is because page size and number of pages (which is also number of page table entries) are inversely related.

7. An operating system can provide flow control on an IPC mechanism like sockets, but cannot provide flow control on an IPC mechanism like shared memory. Why?

Sockets can be used to provide communication between processes as far from each other as across networks. It requires some kind of input-output byte stream that processes communicate through (often represented by a temporary file with certain policies), allowing for the OS to enforce policies for the traffic through the stream.

However, this doesn't apply for shared memory because processes directly fetch and write data directly, instead of sending it through a stream. The shared memory is directly added into processes' address spaces, which means processes have full control over what happens. If processes have full flow control over it, then the OS can no longer regulate that part of memory with full authority.

8. Why are application binary interfaces of particular importance for successful software distribution?

ABIs are important because different devices use different ISAs. An ABI is what provides the proper instruction mapping for APIs so that APIs may be portable for many different devices. If ABIs didn't exist, the software would have to be written for every specific ISA, taking too much time and effort, implying an unsuccessful software distribution.

9. Which memory management technique allows us to solve the problem of relocating memory partitions? How does it achieve this solution?

Relocating memory partitions can be solved by defragmentation. It can be achieved by temporarily using chunks of available disk space to swap chunks of memory. It goes through each chunk of memory and puts some on the available space, reordering chunks that belong to each other together. When suitable, the algorithm will take appropriate chunks from the available space and put them directly adjacent (unlike partitions) to related chunks. This ensures no external fragmentation between independent chunks and better spatial locality (until new memory chunks start getting used).

10. The following multithreaded C code contains a synchronization bug. Where is it? What is the effect of this bug on execution? This is not a full program, but only a part of a program concerning some synchronization functionality. The fact that it's not a full program ISN'T the bug. 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.

```
sem_t balance_lock_semaphore;
int balance = 100;

. . . /* Unspecified code here */

sem_init(&balance_lock_semaphore,0,0); /* Initialize the balance semaphore
*/

char add_balance(amount) {
    sem_wait (&balance_lock_semaphore ); /* wait to obtain lock on balance
variable */
    balance = balance + amount;
  sem_post(&balance_lock_semaphore);  /* Release lock after updating
balance */
}

void subtract_balance( amount ) {
    balance = balance – amount;
}

. . . /* More unspecified code here */

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

add_balance (deposit);

. . . /* More unspecified code here */

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

if (balance >= withdrawal) {
    sem_wait(&balance_lock_semaphore); /* wait to obtain lock on balance
variable */
    subtract_balance (withdrawal);
  sem_post(&balance_lock_semaphore);
}

/* More unspecified code */
```

10. The if statement checking balance and withdrawal should be inside the sem_wait and sem_post. If the OS context switches while inside add_balance but before the balance is actually added, then thread 2 will check (balance >= withdrawal) on an old balance. Then, context switch back to thread 1, the balance is added and the lock released. Then, context switch back to thread 2: the withdrawal does not depend on the new balance value, so withdrawal may or may not happen.

Ex. balance = 100

Thread 1: about to add 10 to balance
context switch
Thread 2: wants to withdraw 101. However, balance < 101 so it skips.
Context switch
Thread 1: balance now 110.
Context switch
Thread 2: doesn't withdraw

The expected behavior should've been balance = 100
then balance = 110
then balance = 9 because of withdrawal after deposit