CS 111 Midterm

Kevin Sung De-Ming Tan

TOTAL POINTS

95 / 100

QUESTION 1

1 Page replacement algorithm choice 10 / 10

✓ **- 0 pts** Correct

- 10 pts Incorrect/no answer

- 5 pts Incorrect/no explanation of why algorithm choice matters

- 5 pts Incorrect/no explanation of likely difficulties upon poor algorithm choice

- 2.5 pts Explanation of why algorithm choice matters unclear/needs more detail

- 2.5 pts Explanation of poor algorithm choice's consequences unclear/needs more detail

QUESTION 2

2 Spin lock performance 10 / 10

✓ **- 0 pts** Correct

- 10 pts Incorrect/no answer

- 5 pts Incorrect/no explanation of how spin locks cause performance problems

- 5 pts Incorrect/no explanation of how a thread can harm its own performance

- 2.5 pts How spin locks cause performance problems unclear/needs more detail

- 2.5 pts How a thread can harm itself with spin locks unclear/needs more detail

QUESTION 3

3 Virtual address translation 10 / 10

✓ **- 0 pts** Correct

- 3 pts Missing one case

- 6 pts Missing two cases

- 1 pts The page table doesn't get full in the sense of being too full. At most, it contains an entry for every page.

- 2 pts You never "search" a disk for a page. You always know exactly where it is.

- 2 pts You don't search page tables for invalid addresses, since they won't be there.

- 3 pts Third case same as example case.

- 1 pts And what happens in the third case?

- 2 pts If the page is supposed to be somewhere and can't be found anywhere, that's an OS crash, not a page fault. This must never happen.

- 3 pts I/O does not occur in the middle of handling an address translation.

- 1 pts First outcome results in page fault.

- 1 pts MMU cache page table entries, not pages

- 10 pts Diagram does not describe cases.

- 7 pts Imprecise description of situation and actions for all three cases.

- 2 pts What precisely do you mean by "system will continue"?

- 1 pts Entire page table isn't cached in MMU. Individual entries are.

- 1 pts In third case, if page isn't in RAM, you have to pay to get it from disk. Context switches may result, but that's not the main activity required.

- 1 pts How does the system "add a page to the frame"?

- 10 pts You did not answer the question

- 1 pts In case 3, cache what in the PTE?

- 2 pts You don't make an invalid page valid by simply allocating a page frame.

- 3 pts MMU must not allow one process to access another process' pages, regardless of their address.

- 3 pts TLB doesn't cache actual pages.

- 2 pts What is the consequence of case 2?

- 1 pts If a page is on disk, it will not have an entry in the TLB.

- 6 pts Cases 2 and 3 are not requests to translate an address.

- **3 pts** Dirty bit is only relevant for page replacement, not address translation.
- **3 pts** We don't move an invalid page into a process' working set because it issued an address in the page.
- **1 pts** Page on disk is listed in page table, just with present bit not set.
- **2 pts** If page is not in a RAM page frame, it's on secondary storage and access will be very slow.
- **2 pts** Valid bit and present bit have different meanings.
- **2 pts** In first case, must get page off disk into a page frame
- **3 pts** First case won't happen.
- **1 pts** More details on first case.
- **3 pts** Third case won't happen.
- **4 pts** Click here to replace this description.

QUESTION 4

4 Results of fork 10 / 10

✓ **- 0 pts** Correct

- **2 pts** Does not mention pid difference/ return code
- **5 pts** Unclear about differences between parent and child
- **10 pts** Completely wrong
- **3 pts** Insufficient explanation
- **1 pts** Does not mention utility of return code/ pid in differentiating between parent and child
- **1 pts** fork() call in child returns 0 not 1 or something else
- **10 pts** No answer
- **4 pts** Does not provide any explanation for why stated difference is useful
- **2 pts** Copy-on-write, not always
- **2 pts** Child does not have a PID of zero, that is the return value from fork()
- **0 pts** correct

QUESTION 5

5 Scheduling for turnaround time 5 / 10

- **0 pts** Correct
- **10 pts** No answer
- **5 pts** RR does not finish short jobs quickly, thus does not optimize average turnaround time.
- **5 pts** Non-preemptive algorithms allow long job to keep new short jobs waiting.
- **5 pts** Did not specify which algorithm to use.
- **2 pts** SJF or STCF? Which?
- **3 pts** STCF over SJF, due to preemption issue.

✓ **- 5 pts FIFO chooses early arrivers over short jobs, harming average turnaround time. One long job could kill your average.**

+ **4 pts** Preemption is indeed necessary
- **8 pts** This approach does not consider that running short jobs first reduces average turnaround time.
- **4 pts** Earliest deadline first only applies to RT scheduling.
- **3 pts** STCF will do better, if one has a good estimate of job run time.
+ **2 pts** Good explanation.
- **8 pts** Not clear what algorithm you mean. Poor explanation of why to use it.
- **4 pts** Insufficient explanation.
- **4 pts** Without knowledge of job run times, MLFQ will probably do better than your choice.
+ **2 pts** Mentioned SJF, but did not favor over other incorrect choices.
- **3 pts** Preemptive or not?

QUESTION 6

6 Changing page size 10 / 10

✓ **- 0 pts** Correct

- **3 pts** No external fragmentation with either page size.
- **1 pts** More details on internal fragmentation effect.
- **3 pts** Less internal fragmentation, not more, none, or the same.
- **2 pts** More details on non-fragmentation effect
- **3 pts** No discussion of external fragmentation
- **4 pts** No discussion of another effect
- **1 pts** As long as the pages are in RAM, the speed of access won't be much different.
- **4 pts** This effect will not occur.

- **4 pts** Page size does not really affect allocation requests.
- **3 pts** With paging, need not use method like best/worst fit.
- **4 pts** Thrashing is not directly related to page size. It is based on actual memory use.
- **3 pts** Non-contiguous allocations across page frames already happens with 4K pages.
- **1 pts** More details on external fragmentation effect.

QUESTION 7

7 Flow control and shared memory 10 / 10

✓ **- 0 pts Correct**
- **5 pts** Flow control for sockets not explained/incorrect
- **5 pts** Absence of flow control for shared memory not explained/incorrect
- **2.5 pts** Flow control for sockets unclear
- **2.5 pts** Absence of flow control for shared memory unclear
- **10 pts** Incorrect
- **1 pts** Sockets aren't unidirectional
- **1 pts** Sockets don't imply 2 machines

QUESTION 8

8 ABIs and software distribution 10 / 10

✓ **- 0 pts Correct**
- **3 pts** Does not mention that ABIs specify how an application binary must interact with a particular OS running on a particular ISA
- **3 pts** Does not mention the need for fewer versions of code / If OS is made compliant then code compiled to an ABI will run on any compliant system
- **5 pts** Unclear about what an ABI is
- **2 pts** Does not mention lack of requirement for user compilation
- **3 pts** Unclear answer
- **2 pts** Needs more detail
- **10 pts** Wrong

QUESTION 9

9 Relocating partitions 10 / 10

✓ **- 0 pts Correct**
- **1 pts** More generally, virtualization (both segmentation and paging) allows relocation.
- **8 pts** Virtualization is the key to relocation.
- **7 pts** Swapping alone won't do it. You need virtualization of addresses.
- **10 pts** Totally wrong. Virtualization is the technique.
- **4 pts** Insufficient explanation.
- **10 pts** No answer.
- **2 pts** Insufficient explanation
- **2 pts** TLB is just a cache. General answer is virtualization.
- **0 pts** Not really called "address space identifiers," but the concept is right
- **3 pts** this is virtualization, not swapping.
- **4 pts** Other way around. To relocate, you change the physical address, not the virtual address.
- **7 pts** Incorrect explanation of the aspect of virtualization that allows relocation.

QUESTION 10

10 Semaphore bug 10 / 10

- **0 pts** Correct
- **10 pts** Incorrect
- **0 pts** Balance checked against withdrawal before obtaining semaphore: balance could decrease between check and lock if unspecified code contains decrement to balance

✓ **- 0 pts Balance checked against withdrawal before obtaining semaphore: balance could decrease between check and lock if concurrent run of thread 2**
- **5 pts** Balance checked against withdrawal before obtaining semaphore: incomplete assumptions
- **10 pts** Assumed bug in unspecified code
- **1 pts** semaphore should be initialized with 3
- **3 pts** b = b+a not being atomic is irrelevant here and cannot cause a bug
- **2 pts** Another strange part [...] <- That comment is incorrect

Midterm Exam
CS 111, Principles of Operating Systems
Fall 2018

Name: Kevin Tan

Student ID Number: 704826225

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?

Choice matters because for page replacement, our goal is to delay the next page fault for as long as we can. This implies an optimal algorithm exists, but we would need to know the future for it. So, we do our best to approximate using algorithms like clock algorithms. The likely difficulty if we choose a poor algorithm that constantly evicts pages that we need in the near future is that we will be constantly swapping pages ~~to and~~ from disk. This is called thrashing and is a severe hindrance to 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?

A spin lock is essentially just a loop that we cannot break out of until a condition is met. This causes performance problems because we require CPU cycles to spin (i.e. continually check if our condition has been met yet) instead of doing actual work. They're also susceptible to bugs... if you have a bug that never fulfills the condition we're spinning on, the program may loop forever! A thread using a spin lock also harms its performance since it can spend more CPU time checking to see if its condition has succeeded than doing actual work. This is especially bad if a thread with a priority is spinning while waiting for a thread with a lower priority to fulfill the condition. The high priority thread will be scheduled often but it will just spin... the low priority thread may even starve if we have a poor thread scheduling algorithm so the high priority thread will take even longer... if it ever unlocks at all!

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.

① The address is invalid (ex. the page is unallocated or the process does not have the right permissions for this page), and the OS kills the process.

② The MMU already cached the valid address translation (VPN and its corresponding PTE) in the TLB. The OS doesn't even need to consult the page table in RAM and it just does the translation, resulting in even faster access to the word.

③ The address is valid but the page is not currently in RAM. If RAM is currently full, the OS needs to pick a page to evict. Then, it makes an I/O request out to disk for the page so it can be swapped in. The process may block while waiting. When we get the page, the page table and TLB are updated, and now we have access to X.

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 a different process ID (PID) than its parent. This is useful because the processes may go on to do different things (e.g. the child exec()s a program and the parent listens for output) and we may need a way for a third process, for example, to send different signals to both (which we couldn't do if they had the same PID). In addition, of one process has multiple children, it can use the PIDs to differentiate between them which is important for when it needs to reap specific children.

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

FIFO
STTC
EDF

If we know about the time all of our jobs would take, we would run earliest deadline first, a non-preemptive algorithm that schedules jobs with the shortest time to completion first. This minimizes turnaround time (time to completion - time of arrival) because on average we schedule short, fast jobs first which gives them very good turnaround time. If we don't know about the time each job takes, we would just use FIFO. The first job that comes in is the first one run, so we minimize its turnaround time. FIFO is also non-preemptive and avoids the overhead time associated with context switching.

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.

It won't have much of an effect on external fragmentation since pages are typically contiguous in memory anyway, i.e. we will not cause memory to be chopped up, leaving unused blocks of memory in between so we just split each 4K page into 4 1K pages. If we are running many processes that do not require much memory, e.g. just 100 bytes, internal fragmentation may decrease since less of the virtual address space, and hence less of physical memory, will be unused/wasted by a process. We also can consider a process that needs 5K of memory. Before, it needs 2 pages and wastes 3K. Now, it needs 5 pages and no waste, which is a big win for decreasing internal fragmentation. Another effect, however, is decreased performance. For processes with address spaces larger than RAM, or perhaps processes with large working sets, if they "jump around" their memory by accessing many addresses that are far apart we'll incur a lot of page faults and need to swap many more pages than we would need to with the 4K pages.

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?

With sockets, processes must read and write to them, which inherently implies flow control. Processes ask the OS to perform reads/writes, and the OS can block processes so they can wait and send interupts when data is ready. With shared memory, however, processes have mapped a shared file into their virtual address spaces and any can modify/read it at a given time. The OS has no part in blocking processes on a read or polling for writes; it's up to the processes to agree upon protocols and data structures for communication.

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

An ABI is an interface, that binds a program's source code to a specific instruction set architecture (ISA). ISAs define conventions in machine code and are usually device-dependent. Devices with different ISAs will probably not be able to run the same binary for a program. However, when distributing software we don't want to distribute source code and have people compile it locally for their particular ISA. Knowing ABIs ahead of time, we can compile multiple different binaries and just distribute those, so consumers can continue to just download the program and run it.

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

Virtualization, or virtual memory, allows us to solve this. The problem of relocating memory partitions is that when we do so, a process using that memory has its addresses invalidated. However, virtual memory allows us to have virtual addresses in a process that we can translate to physical addresses. The physical address may be computed based on the base address of a segment, which is typically stored in a special hardware register. When we relocate the memory, all we have to do is update the base address and all of the virtual memory addresses within the process will remain valid.

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 bug is in the code run by Thread 2, particularly in the (balance >= withdrawal) condition. The key is that we are reading balance, a shared variable, before we have acquired the lock. Consider the following example: Thread 1 is performing add_balance (100), Thread 2 is performing subtract_balance (50), and balance is currently 0. There is a race condition on whether Thread 1 updates the balance variable before Thread 2 performs the comparison. If Thread 1 updates balance before Thread 2 reads, then balance == 100 >= 50 and the withdrawal occurs. However, if Thread 2 performs the comparison first, balance == 0 $\not\geq$ 50 and the withdrawal isn't executed, even if the 100 deposit was intended to happen first! Again, the effect of this bug on execution is it introduces a race condition that impacts the correctness of the program.