CS 111 Midterm

Rohit Dhanaraj Tavare

TOTAL POINTS

74.5 / 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 8 / 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 9 / 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 10 / 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 7 / 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 7.5 / 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 3 / 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 0 / 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: Rohit Tavare

Student ID Number: 704 984 314

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 process of replacing a page involves writing the evicted page to disk if it has been modified, ~~any~~ and getting the content of the new page from disk. When a page is not found ~~it~~ a page fault ~~it~~ is thrown and the OS has to handle it ~~accordingly~~ by loading a page into memory. ~~then~~ Reading/writing to disk is an ~~expensive~~ slow task and dealing with a page fault can be computationally expensive. Thus we want to make the ~~~~ best use of our disk reads and writes by evicting ~~to~~ the least ~~~~ frequently used pages, and get the fewest page faults possible. A poor choice of this algorithm will force the OS to constantly jump to the trap table to handle page faults, and constantly read/write to disk, spending less time executing the process.

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 cause performance problems because they waste CPU cycles. In the short term they may be more efficient than the overhead of yielding or putting a thread to sleep; however ~~~~ if run for a long time can seriously degrade performance. If the scheduler is using ~~~~ a MLFQ or similar algorithm, the use of a spin lock may cause the thread to ~~drop~~ drop in priority, reducing chance of being run. Furthermore, since the CPU uses speculative execution, ~~over time~~ the CPU will have to flush its state when the spin lock finally completes, as the CPU would most likely assume statistically the spin lock would continue.
Spin locks also fail to make use of concurrency; other threads could to calculations in the cycles the spin lock uses up.

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.

One possible outcome is that the address is already in the TLB for fast access. The page translation is quickly found, and the address is valid, and the frame is found in RAM and the data is returned.

Another outcome is that the address is valid, however the page's invalid bit is set, indicating it is not in ~~[illegible]~~ RAM. A ~~[illegible]~~ page fault is thrown. In one case, there is available space in RAM, so the ~~page~~ frame is loaded from disk, and the MMU caches the entry. In another case a page must be stolen from another process, so a page eviction algorithm chooses a ~~page~~ frame to write to disk, and the requested frame is loaded, and the entry is cached.

A final outcome is a page ~~[illegible]~~ in VM requests an entry with an invalid bit set in the TLB, or not ~~[illegible]~~ of the current process, or does not follow the read/write rules for that page. A trap is thrown, and the OS will most likely ~~the~~ assume the process is ~~a~~ malicious and kill it.

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?

In Linux, a forked child process has an exact same copy of the virtual address space of the parent process. The child gets its own process ID, and Process Control Block, but the main difference is the copy-on-write feature. This allows the child to use the parent's virtual address space as a template for its own, without affecting the parent process as every ~~change the child makes is written into a copy of the data~~. time ~~[illegible]~~ the child writes to memory, it copies the data so that the child does not change the data that the parent process is looking at. As stated earlier, this makes for an efficient way to set up a new process using the parent as a template.

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

~~If the scheduler's goal is to minimize turnaround time, it would implement a shortest job first (SJF) algorithm. SJF is a non-preempting algorithm meaning that when a ~~process~~ job starts, it runs until completion. This ~~maximizes~~ minimizes turnaround time since there is a large overhead to context-switching due to having to save the CPU state, process VM, stack pointers etc, which occurs during preempting. Furthermore pre-empting delays a process completion~~

To minimize average turnaround time, a Shortest Time to Completion First algorithm must be employed. This is a preempting algorithm that runs the shortest jobs currently in the queue. This allows short jobs to have a very small turnaround time, and does not impact longer jobs as much. Though pre-empting has an associated overhead with context-switching, the algorithm avoids the caravan effect of non-preempting algorithms that can severly affect turnaround time. Furthermore it is more realistic that not all jobs are scheduled at the start.

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.

When switching from a page size of 4K to 1K, internal fragmentation decreases. Internal fragmentation occurs when not all of a page is entirely used. For a 4K page, any page with <4K of data suffers from internal fragmentation, whereas a 1K page only suffers when <1K of data is stored, which is less likely than storing less than 4K.

External fragmentation is also reduced, as the smaller granularity of pages will allow the system to make use of small gaps of unallocated memory that would otherwise be unusable. External fragmentation is when the physical memory has many small segments of unallocated memory too small for any process to use.

Another non-fragmentation effect of this is that page faults are more likely to occurr, reducing the effectiveness of & spatial locality. Before, in a 4K page, only 1 page fault is needed to load the same amount of data that would take 4 page faults in the 1K page system. The system would have better locality for loops that iterate through a single 1K page of data in the new system, while the old system exhibited good locality for 4K of data.

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?

IPC mechanisms like sockets, pipes etc. are primarily treated as communication mechanisms like streams and thus have system calls and ABI support to provide flow control.

However, memory sharing is not treated like these other IPC mechanism because it is dealt with ~~the memory~~ memory management, which is none-the-wiser that the memory is shared. That typical non-shared memory has no flow control mechanisms, and since shared memory is treated the same as unshared memory, the OS cannot provide any flow control. It is all up to the user and the process. Also, data in IPC mechanisms like sockets is consumed one by one, and flow control can be applied in a queue-like mechanism, where shared memory is entirely readable at ~~any point~~ every point once written.

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

ABI's are critical for software distribution because it promotes great portability. By creating an ABI, the OS developer 'signs' an 'interface contract' indicating that they will support this interface across all hardware allowing software developers to not ~~the~~ focus on the many classes of hardware, but on the abstracted interfaces the OS provides. Further more, the abstraction and information hiding in OS interfaces lets programmers create more robust software that is not hardware dependent.

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

Coalesceing is the memory management technique that allows us to solve relocating memory partitions. Coalesceing means to traverse the free list for free space, and copying ~~large~~ segments of allocated memory ~~to next~~ to be next to each other, moving the un allocated segments as one large group.

In pseudo-single write multi-read memory like SSDs, a nearly empty partition is found and a nearly full partition is moved to fit into the empty partition. This lets the SSD to format large blocks of memory, since it cannot format individual bytes.

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 <u>synchronization</u> 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 */
```

Rohit Tavare
204 884 316

10) The synchronization bug with this program is that the semaphore is initialized to a value of 0, and no thread does a sem_post to the lock before thread 1 or 2 run their code. The issue that results is that thread 1 and 2 block indefinitely. When the value of a semaphore is 0, then the value is decremented and the thread waits. The thread can continue only when another thread does a sem_post and wakes a waiting thread. The way to fix this bug is to make a binary semaphore by initializing the semaphore to a value of 1. This means the first thread to reach the lock will decrement the value to 0, and continue. That thread will either increment the value to 1 and leave before the other thread, or have the other thread wait, and wake the thread when leaving the critical section.