CS 111 Midterm

Rajiv Anisetti

TOTAL POINTS

98 / 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 8 / 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: Rajiv Anisetti

Student ID Number: 904801422

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 bottleneck in regards to paging is the page fault process. Retrieving pages from disk is expensive. Because of this, the performance of our virtual memory system relies on how often we experience a page fault when attempting to access a page. Due to this, if we have a good page replacement algorithm, we will experience less page faults due to our choice of which pages to keep within our physical memory. If a bad choice is made, then we will likely experience many page faults. This will incur severe performance issues, as we will be spending too much time paging, making our virtual memory implementation have too much overhead costs.

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 waste CPU cycles simply checking if they can acquire the lock. In the case where multithreading is does not utilize other CPU cores, a context switch to a thread currently spinning may never acquire the lock in that time cycle, as it is hogging the CPU of the thread that may be attempting to finish executing its critical section. Thus, it is repeatedly checking a value that will never change (in that time cycle). Evidently, this can cause performance issues, as it is wasting CPU time that could be used towards getting closer to its unlocking and execution, and thus performance.

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.

- the address is valid, the page is NOT in a RAM page frame, resulting in a page fault. The OS must bring in the page in from disk, the MMU caches the PTE and retries the instruction, resulting in a fast access on retry.
- the address is invalid, so the system issues a trap instruction and will most likely kill the process, as it attempted to reference a page not relevant to its process (maybe through a segfault)
- the address is valid, the address translation is already cached and found within the TLB. The physical address can be extracted very fast, and doesn't need to recache the PTE.

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 is different in that it does not have its own copy of the parent process' data. This is due to the fact that a copy-on-write policy is used here. This is useful because data portions of a process can be large and expensive in terms of time/space to copy. Copy-on-write ensures that the data is only copied if either attempts to write to the data. The new process is also different in the fact that the fork() call returns 0 to the child thread. This is useful in triggering a different control flow for a child thread, which could perhaps call the exec() function to run another program, rather than the same program as the parent. We can implement a shell in this way or use other methods to take advantage of the child process.

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

Turnaround time is defined as a jobs completion time minus its receival time. If we were to prioritize this metric, shortest job first (SJF) or shortest-time-to-completion-first (STCF) would be good options. We are prioritizing the finishing of jobs rather than fairness. Also, we are now less subject to the convoying effect of FIFO, (and we can finish shorter jobs in lower turnaround time thus lowering our average turnaround time. Round Robin would be unfavorable as well, as it would preempt our jobs before finishing repeatedly, which is more fair, but at odds with performance.

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.

Internal fragmentation will be lowered, as we are now allocating less space per page, so there is less space that will not be utilized after requesting a page. However, external fragmentation will most likely not be effected, as paging eliminates this due to its fixed-size memory splitting. Because we now have more pages, we must keep a larger page table than before and we can encounter page faults more often, as our virtually located segments are now mapped across 4K as many pages, so our scope of locality is hindered, thus, this would hurt performance due to more excessive page faults as well.

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?

Shared memory cannot be as regulated by the OS. The write changes are nearly instantaneous and there is no sense of identity/trust of who is writing. In the case of the socket or pipe, there is a flow of data that can be regulated by intermediary steps in order to control the flow. Shared memory does not exhibit such behavior so there is no fashion of controlling the flow, and if there were, the OS couldn't resolve identity anyway.

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

Application binary interfaces bind an API to an instruction set architecture. Because different machines implement different ISAs, the use of an ABI eliminates the need to recompile from source files for different systems. As long as two systems support the ABI, the software can simply be downloaded with no need for recompilation. This extremely useful in software distribution, as we do not need to distribute source files/libraries that need to be retranslated in order to run the corresponding software, which allows distribution to those outside of compilation knowledge, who just want to download the software and run it.

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

The main problem of relocating memory partitions is the fact that all references to the relocated memory must be corrected, which is an extensive and unfeasible task. However, a simple fashion of virtualizing memory through base pointer solves this problem well. For the different segments of an address space (code, data, stack, el.), we can keep base registers to the beginning of their physical address locations. When we want to relocate a partition, we can move it as needed and resolve our memory references by simply adjusting the base register to that partition to the new location. Because we are adding our virtual offset to the base register, the same virtual memory locations are now mapped to the corresponding physical memory locations. Thus, the implementation makes use of hardware in order to store virtual addresses as offsets from a base address. Relocating a partition just changes the base address.

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 */
```

The issue with this multithreaded code is the initialization of the semaphore. If thread 1 runs first and wants to acquire the lock, it will decrement the semaphore count from 0 to -1, see that the value is negative and wait. The same phenomena will occur for thread 2, and neither will ever end up calling sem_post(), so we are caught in a deadlock. The correct initialization would set the semaphore count to 1 initially in the following manner: sem_init(&balance_lock_semphore, 0, 1). Now when either thread first attempts to grab the lock, sem_init() will decrement the count to zero, which is nonnegative and the thread can acquire the lock, later calling sem_post() and no longer falling prey to deadlock.