CS 111 Midterm

Luke Hyun Jung

TOTAL POINTS

64 / 100

QUESTION 1

1 1 6 / 12

- **0 pts** Explains problem with virtual memory AND physical addressing AND difficulties in preventing memory access violation
- **3 pts** Does not explain potential memory access violation
- **2 pts** Does not address virtual memory
- **10 pts** Denies possibility without giving a reason why
- **10 pts** Insists it's possible without explaining why/how

✓ **- 6 pts Explains normal bootstrapping process but not problems with accessing physical memory and violations.**

- **8 pts** Explains normal bootsequence without using Ubuntu as part of boot strap. Does not point out problems with memory, virtualization or physical addresss translation
- **12 pts** Blank answer
- **3 pts** Does not explain difficulty in accessing physical memory
- **6 pts** Denies possibility, but does not show the 3 main reasons why. Points out differences in disk reading.

QUESTION 2

2 28 pts

2.1 2a 3 / 3

✓ **- 0 pts Correct**

- **1.5 pts** Needs more detail/clarification
- **3 pts** No answer/incorrect

2.2 2b 3 / 3

✓ **- 0 pts Correct**

- **1.5 pts** Needs more detail/clarification
- **3 pts** Incorrect/no answer

2.3 2c 10 / 10

✓ **- 0 pts Correct**

- **10 pts** Incorrect/no answer
- **5 pts** Needs more detail/clarification
- **5 pts** On the right track, but incorrect

2.4 2d 0 / 6

- **0 pts** Correct

✓ **- 6 pts Incorrect/no answer**

- **3 pts** Needs more detail/clarification
- **2 pts** On the right track

2.5 2e 6 / 6

✓ **- 0 pts Correct**

- **6 pts** Incorrect/no answer
- **3 pts** Needs more detail/clarification
- **3 pts** On right track, but not correct

QUESTION 3

3 3 7 / 12

- **0 pts** Good reasoning.
- **3 pts** Reasoning has some flaws or not good/complete enough.

✓ **- 5 pts Reasoning has some flaws or not good/complete enough.**

- **7 pts** Reasoning is not good/correct/complete.

QUESTION 4

4 18 pts

4.1 4a 7 / 12

- **0 pts** Correct
- **12 pts** Incorrect/Not done

- **2 pts** Incorrect use of a system/function call

✓ - **5 pts** **Code unclear (several calls incorrect)**

- **4 pts** Incorrect critical section
- **6 pts** Unclear Explanation
- **7 pts** No code given

4.2 4b 6 / 6

✓ - **0 pts** **Correct**

- **1 pts** incorrect function/system call
- **3 pts** Code unclear
- **3 pts** Explanation unclear
- **6 pts** Incorrect/ Not Done
- **2 pts** Incorrect critical section
- **4 pts** No code

QUESTION 5

5 5 8 / 15

- **0 pts** Correct answer with correct explanation, using all the system calls specified

✓ - **2 pts** **ping/pong incorrect**

✓ - **2 pts** **does not perform expected behavior**

- **1 pts** missed a fork
- **2 pts** missed read/write
- **1 pts** missed pipe/close
- **4 pts** setup incorrect

✓ - **3 pts** **Ais not parent of B always/ new B spawned each time/Only first ping-pong works**

QUESTION 6

6 6 8 / 15

+ **15 pts** Good reasoning!

✓ + **4 pts** **The discussion where DQ has a higher utilization than RR makes sense.**

+ **4 pts** The discussion where RR has a higher utilization than DQ makes sense.

+ **4 pts** The discussion where DQ is more fair than RR makes sense.

✓ + **4 pts** **The discussion where RR is fairer than DQ makes sense.**

+ **4 pts** Saying they are both fair, or fairness discussion is not good/correct/complete enough.

+ **2 pts** Did some analysis on utilization, but not correct.

+ **5 pts** Couldn't fully understand writing. Please type down your answer then request regrading. Thanks!

+ **0 pts** Empty.

+ **2 pts** Only a few words.

UCLA Computer Science 111 (winter 2019) midterm
180 minutes total, open book, open notes,
no computer or any other automatic device

Name: Luke Jung Student ID: 904-982-644

1	2	3	4	5	6	total

1 (12 minutes). Dr. Eniac is nostalgic for the good old days when standalone programs ruled the world and there were no operating systems. Eniac decides to add two system calls to her copy of the Linux kernel running on an x86-64 machine. The first system call, 'void readsector(long S, long A);' is like the read_ide_sector function discussed in class: it reads a single 512-byte sector from sector S of the primary disk drive into the physical memory location numbered A. The second system call 'void _Noreturn execute(long A);' terminates all currently-running processes and then directs the Linux kernel to jump to location A.

With these two system calls, is it possible for Dr. Eniac to attach a fresh disk drive to her computer, initialize it appropriately, and then treat running Ubuntu as part of an bootstrap process intended to run some other operating system? If so, briefly explain how booting Dr. Eniac's machine would work; if not, briefly explain why not. Either way, state any assumptions you're making.

It is possible, since booting takes the following steps of reading new memory to execute, then replace all current programs with a new one. So, using the programs given, Dr. Eniac would first hook up the disk drive, run "void readsector(...)" where our Ubuntu OS software is written, and then use the "execute" function to jump into code and run the single process on the disk for bootstrapping. where ubuntu is written

2. Consider the 'close' function on the SEASnet GNU/Linux servers.

2a (3 minutes). What is the API for 'close'?

close takes in a file descriptor integer and closes the file descriptor, by unlinking it. It returns 0 on success. close(int fd);

2b (3 minutes). What is the ABI for 'close'?

ABI is the more portable interface written in binary, so it still does the same function as the API. The ABI would take in a binary address on the file wanting to be closed and unlink it.

2c (10 minutes). Consider the following x86-64 assembly language code:

```
foo:    notl    %edi
        jmp     close
bar:    .quad   close
```

Does this assembly language code correspond to the following C-language source code? If not, why not? If so, explain why any seeming discrepancies are not really discrepancies.

```
#include <unistd.h>
typedef int (*func_ptr) (int);
func_ptr const bar = close;
int foo (int n)
{
  return bar (-1 - n);
}
```

yes it does, due to the first machine instruction doing the computation -1-n and it then jumps to the close function.

Bar is also implemented correct because the typedef at the top maps the function close to bar.

The discrepency are minute because the machine code still does the expected behavior.

2d (6 minutes): Does the assembly language code follow the ABI for 'close', the API for 'close', or both, or neither? Briefly explain.

Follows API for close, because API close is int close(int) which bar is typedef for. The ABI uses memory locations, so no

2e (6 minutes): Does the ABI for 'close' use hard modularity or soft modularity? Briefly explain.

Assuming ABI simply closes an address, uses soft modularity because can close any file and run data transfer among other processes

3 (12 minutes). As the Arpaci-Dusseaus explain, in GNU/Linux an N-thread process has N stacks, one for each thread. Now, a thread accesses its stack only while running, and suppose our system has thousands of threads but only two CPU cores, so at most two threads can run at any time. Can we save memory by having only two stacks? More generally, if there are R CPU cores, can we save memory by having only R stacks instead of N stacks? If so, give a good reason why GNU/Linux doesn't save memory in this way. If not, explain why not.

No, the reason each thread has its own stack is so that the thread knows what action to do next. While threads share memory and other hardware, each thread needs its own stack and registers so that they can run and do each task at hand. The reason GNU/Linux doesn't save memory by having only enough stacks as CPUs is because the whole point of threading is so that multiple pieces of program are run in parallel. If they shared same stack, the threads would run through same instrs, waiting for each thread, minimizing the purpose of threading. Using mult stacks gives each thread independence to finish tasks.

4. In the Linux kernel, the 'read' system call returns -1 and sets errno to EINTR (with no other side effects) after a signal is handled during 'read' and the signal handler returns. Another possible API design (let's call it "Linux B") would have 'read' continue to do its work in that situation, just as ordinary code does, and return -1 only if a true I/O error occurs.

4a (12 minutes). Give realistic code that benefits from the Linux design; your code should stop working (or should not work nearly as well) on a "Linux B" system. Briefly explain the critical section in your code, or explain why it has no critical section.

In any scheduler, we want to implement fairness and efficiency. Therefore if we have a large file to be reading from and want to stop the reading and choose a different task to read, a linux system could stop and return to do next insn, while linux b would ignore it and we couldn't move on. For example, code like

signal(SIGINT, Sighandler);
read(a);...
read(b);

If we were reading file a but wanted to switch to b, linux would stop, say didn't finish, and run b. But, linux B would read(a), get signaled, continue, and read(b) concurrently. Both read(a) and read(b) are critical sections, meaning when run together, the buffer containing the read would be all jumbled.

4b (6 minutes). Give realistic code that would run well on a "Linux B" system but not so well on plain Linux. Again, briefly explain the critical section in your code, or explain why it has no critical section.

If we were doing reads on a file that needed to be done as quickly as possible, and we weren't waiting on anything else, Linux B would work well because the read would always complete, regardless of how many signals are passed

For ex.

read("file needs to be done right now");
Signal(SIGINT, Sighandler);

Here, since our goal is we need to read and parse through code quickly, not allowing signals to interrupt can be a good thing. Our critical section here again is the read.

5. (15 minutes). Suppose we want to arrange things so that two GNU/Linux proceses A and B never execute at the same time. That is, B is idle whenever A is running, and A is idle whenever B is running; it is OK if neither A nor B is currently running. Describe how to use the 'close', 'fork', 'pipe', 'read', and 'write' system calls to implement this. Implement the C functions 'setup', 'ping', and 'pong' so that:

* A calls 'setup ()' to set up the arrangement, with A being the parent and B being a newly-created, idle child of A,
* A calls 'ping ()' to let B run.
* B calls 'pong ()' to let A run.

Keep your functions as simple as possible. You can assume that A and B are both perpetual processes; that is, that neither exits.

```
setup( ){
int pipefile[2];
int filelist[2];
if(pipe(pipefile)==0){
    filelist[0]=0;
    filelist[1]=1;
}
pid_t p=fork();
close(filelist[0]); close(filelist[1]);
}
void pong(void){
    int wstatus;
    waitpid(-1, &wstatus, 0)
    if(WIFEXITED(wstatus))
        return;
}
```

```
void ping(void){
exit(0);
}
```

While my C code may be off, the

- close sys call closes the file list at a specific file descriptor, here by closing the ends of the pipe.
- fork creates a new process right after itself and runs two processes, one with a pid of the child and the child with Pid =0.
- pipe creates a rd side and wr side and can use those for read write commands
- read is used to read part of the read side of pipe, gaining info from what was written
- write: writes output to write side of pipe, pushing info to read side.

6 (15 minutes). In class we assumed that in a round-robin (RR) system every process consumes an entire quantum before being preempted. However, in real life a process can yield the CPU voluntarily before the quantum has expired using system calls like sched_yield, thus letting some other process run for the rest of the quantum (or until it also voluntarily yields). The "burst time" of a process is the amount of CPU time that it most recently consumed while not yielding the CPU voluntarily. A process's burst time grows whenever it has the CPU, stays the same while it is not running, and is reset to zero whenever the process resumes after yielding the CPU voluntarily.

Suppose we modify a single-CPU RR scheduler to use a dynamic quantum as follows: at every timer interrupt, the scheduler changes the quantum's length to be the minimum of the current burst times of all the processes currently in the system. Call the resulting scheduler the DQ scheduler (DQ is short for dynamic quantum).

Compare the DQ scheduler to the RR scheduler with a fixed 10 ms quantum. Consider both utilization and fairness. Assume a job mix where each process's burst time can vary dynamically. For example, what sort of job mixes will do better with RR? with DQ?

Assuming that after a yield, the process doesn't run again, so the burst time never goes to 0. This is because if the quantum is 0, the no process could run because none would get any time.

For a RR, we get average utilization due to context switches every 10ms and good fairness because every process is given some amt of time.

Using a DQ, we get a better utilization, because the quantum is going to get higher so less context switches and worse fairness because there will be a higher wait time as the quantum increases.

For a set of burst times that vary dynamically, a DQ method would do better since as the less time reading programs get to finish first and the Quantum increases for the bigger processes.

But for a more uniform job mixes, DQ performs the same or worse than RR because it has overhead of computing the minimum burst time. While RR would be more consistent and more fair.