CS 111 Midterm

Eric Balagtas Perez

TOTAL POINTS

35 / 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 0 / 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 3 / 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 9 / 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 0 / 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 **2 / 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 **5 / 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**

+ 3 Point adjustment

"Somehow" - how? by asking B to read. Points given for explanation.

QUESTION 6

6 6 **4 / 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 few words.

UCLA Computer Science 111 (winter 2019) midterm
~~180~~ 182 minutes total, open book, open notes,
no computer or any other automatic device

Name: Eric Perez Student ID: 605181484

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.

Yes. We would have a MBR in that fresh disk drive. This would have information about how and where the OS is located in the fresh disk drive. We would boot the normal Ubuntu with the BIOS looking at the old drive's MBR. After this Ubuntu is loaded, we could run a program that read, the sector on the fresh disk drive and after call void _Noreturn execute(long A); to terminate Ubuntu and run to other OS. This assumes we already have a program that calls these system calls. The whole process of having Ubuntu as a middle man just adds extra overhead when we could have booted the other OS straight from after the BIOS.

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

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

It takes in a file descriptor integer and closes it.

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

ABI is the low level interface between a program and the OS. Close does a system call and removes the reference to the open file descriptor.

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);
}
```

No, we do not even get the parameter 'n' and return '-1-n'.

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

None. It is not even 'closing' anything. It is simply returning -1-n.

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

It uses hard modularity since it uses a system call. It is not just a function call.

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.

Threads are like processes. They share memory, a file descriptor table, and a signal handler table. But, each thread has its own instruction pointer, registers, stack, thread ID, signal mask, errno, and state.

We cannot just save memory by having only two stacks, or R stacks instead of N, we would lose all the stack information relevant to the threads. Threads don't share a stack. If we only have two stacks, the threads would edit these stacks and run into possible race conditions.

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.

This design would be better in most cases except in the situation from part 4b. It is good practice to just return -1 to indicate an error and have the signal handler return.

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.

Code that would run well on this is code that must be finished, no matter the signal. If we are writing at the same time we are reading, we should try to finish despite the signal, or else the write would be incomplete. If we are deleting from the file we are reading from, Linux 'B' would be good. It will try to finish, no matter the interrupt.

```
// read from line
// delete line
// write to file
```
} Critical section.

5. (15 minutes). Suppose we want to arrange things so that two GNU/Linux processes 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.

```
void setup() {
    // pipe set up
    // fork
}

void ping() {
    // A stops writing
    // A closes pipe end
}

void pong() {
```

We could implement this in such a way that either process will have an unclosed end of a pipe. When we want A to run, somehow make B hang on a pipe. B has to read from a pipe when the write end is open. When we want B to run, we do the opposite. B writes to the write end of a pipe and A would be stuck. We would keep making new pipes and closing them when switching.

```
}

int main (int argc, char ** argv) {
    // child
    if () {

    }
    // parent
    if () {

    }
}
```

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?

Better with Round Robin:

We know that the quantum length is the minimum of the current burst times of all the processes currently on the system. If there are currently a lot of long jobs and then a lot of quickly yielding processes constantly come in, the quantum length will always be low. It is not fair for the longer jobs. They will take a while to finish, while the jobs that yield a lot will benefit.

So, we should run this mix on Round Robin.

better with DQ!

When we care about the jobs that yield often, it is better to use DQ. The quantum time will be low, so there is less idle time for these often yielding jobs that yield quicker than 10ms. This would be good if the I/O for these jobs are quick, so it could be more fair for the often yielding jobs.