CS 111 Midterm

TOTAL POINTS

90.5 / 100

QUESTION 1

1 1 12 / 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 1.5 / 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 4 / 6

- 0 pts Correct

- 6 pts Incorrect/no answer

- 3 pts Needs more detail/clarification

✓ **- 2 pts On the right track**

The assembly code is not involved with the C API.

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 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 12 / 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 15 / 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 12 / 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
100 ~~180~~ minutes total, open book, open notes,
no computer or any other automatic device

Name:_____ Student ID:_____

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.

~~This bootstrap process would indeed work, if we could make a few assumptions:~~

1. Dr. Eniac has a priori knowledge about the ~~size~~ and layout of her computer's physical memory ~~as well~~ as the size of the standalone program.
2. The new system calls introduced to the Linux kernel do not do any permission checks with regards to ~~the~~ memory address A.

~~She could write a program that calls readsector()~~

The bootstrap process would not work. In order for the CPU to jump into a fresh new operating system, the new operating system must be first loaded into physical memory. The Linux kernel, through security measures like ASLR, prevents userspace programs from finding out what a virtual memory address actually corresponds to in physical memory. However, even assuming a priori knowledge about the layout of physical memory, and that readsector() is able to bypass any permission checks when writing to A (which it is able to do as a system call running in the kernel), readsector() can only read the PRIMARY disk drive of the system — i.e., the disk with Ubuntu installed. (continued)

If, on the other hand, the fresh new disk somehow becomes the "primary" disk and thus readable through readsector(), then with additional knowledge of the layout of the new disk, Dr. Eniac could conceivably use readsector() and execute() to start reading and executing machine code stored on the new disk. execute() would have to ensure all CPU protections enabled at system boot before jumping to that physical memory ~~and~~ though. (I.e., enable real mode (are disabled) on x86 machines.)

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

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

The interface visible through C of the function:

int close (int fd); as well as a contract of how it operates.

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

The API, as well as the calling convention used on the system (e.g. the argument fd is stored at %edi, and return code is stored at %eax.)

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);
}
```

It corresponds exactly to the C code.

0. The NOT instruction flips all bits in the operand, which is equivalent to the operation (-1-n) assuming n uses 2's complement.
1. the call to bar is in fact a call to close(), as bar() is a const function pointer pointing to close.
2. Rather than doing call + ret in foo merely modifies its argument and does a jump. close() This is a valid way to call a function, known as "tail call" in compiler theory. will then call ret on behalf of foo().
3. bar is a global variable in C, and it is a global w/ the same value in ASM.

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

It follows the ABI for 'close' as it assumes that the argument n is passed through the register %edi, which is part of the calling convention (and thus ABI). It does not look at the API for 'close' as it does not obviously treat its input as a file descriptor; it's inconceivable why anyone would try to do bitwise arithmetics on a FD.

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

It uses soft modularity, as other parts of the process may tinker with the operations of close [redacted]. E.g. A separate thread may be able to modify the return address of close() to point to another address in addressable memory while close() is running, thereby changing close()'s behavior.

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.

Yes., in that one could conceivably construct an OS with only R stacks, working

but there are overwhelming reasons not to do that.

1. security/modularity: if many threads share a single stack, then all those threads will be able to access each other's memory, even if they belong to distinct processes causing a huge hole in the modularity barriers between processes. Bugs like stack smashing would affect other processes as well.

2. preemptive scheduling. The premise of preemptive scheduling relies on a kernel's ability to pause and restart a thread easily. However, if many threads use the same stack, it is difficult to pause (i.e., context switching) a thread and subsequently restart it when any other thread may write to that same stack while executing. The OS in this case would not be able to reliably restart the thread.

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.

```
#include "unistd.h"    etc.
#include "signal.h"
char *buf = NULL;

void sighandler(int signum) {   // reset buffer on SIGHUP
    free(buf);
    buf = malloc(1024);
}

int main(void) {
    signal(SIGHUP, sighandler);
    buf = malloc(1024); if (!buf) { perror("malloc"); exit(1); }
    if (read(0, buf, 1024) < 0) {
        perror("read");
        exit(1);
    }
    ...
}
```

critical section: read() relies on that buf points to a chunk of memory available to the process. However, if a SIGHUP arrives during read(), then buf pointer may get invalidated. On Linux, this code would exit with an informative error. However on Linux B, read() will continue writing to buf, possibly causing a segmentation fault.

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.

```
#include's omitted

int setting = 0;
void sighup_handler(int signum) {   // re-read setting from disk on SIGHUP
    int fd = open("/etc/setting", O_RDONLY);
    if (fd >= 0) {
        char c;
        if (read(fd, &c, 1) > 0) {
            setting = c - '0';
        } close(fd);
    }
}
int main(void) {
    signal(SIGHUP, sighup_handler);
    char buf[1024];
    ssize_t s = read(0, buf, sizeof(buf));
    if (s < 0) { perror("read"); exit(1); }
    /* process data in buf depending on setting */
}
```

There is no critical section here, as the action read() does not depend on setting, which is modified in sighup_handler(). It would run well on Linux B, as there is no ill effect that could be caused by sighup_handler running while a read() is being done in main(). It would not run well on Linux, as the user would expect the program to keep running despite the ... to update setting.

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.

```
int pipefd_a[2]; int pipefd_b[2];
void setup(void) {
    int pipefd_a[2], int pipefd_b[2];
    if (pipe(pipefd_a) || pipe(pipefd_b)) { perror("pipe"); exit(1); }
    pid_t p = fork();
    if (p == 0) {
        // in B
        close(pipefd_b[0]); close(pipefd_a[1]);
        char c;
        while (read(pipefd_a[0], &c, 1) > 0) {
            // B is now running. Do things.
            pong();
        }
    } else if (p > 0) {
        // still in A
        close(pipefd_b[1]); close(pipefd_a[0]);
        char c;
        do {
            // A is now running. Do things.
            ping();
        } while (read(pipefd_b[0], &c, 1) > 0);
    } else { perror("fork"); exit(1); }
}
void ping() { write(pipefd_a[1], "a", 1); }
void pong() { write(pipefd_b[1], "b", 1); }
```

// will block while A is running

// will block while B is running

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?

The RR scheduler remains the fairest, as it ensures that any process will run after at most $10N$ milliseconds, where N is the number of processes on the computer. On the other hand, since the quantum changes dynamically with DQ, there is essentially no fairness guarantee.

- Consider a job mix with all CPU-intensive jobs. In this case, processes will all have a large wait time due to the quantum being longer, due to large burst times, under DQ. This weakens DQ's fairness. However, DQ increases system utilization, as less time is spent on context switches and more on doing useful work. On the other hand, RR sacrifices utilization for fairness. In this scenario, DQ is better, as clearly responsiveness is not essential with all CPU-intensive jobs
- Consider a job mix with many CPU-intensive jobs but one job that yields regularly (~9ms). In this case, DQ approximates RR with a short quantum, and the two schedulers act similarly.
- Consider a job mix with many jobs that take around 9ms before yielding, but one job that only runs for 1ms before yielding. In this scenario, RR is better, as it allows all jobs to run to completion w/o forcing context switches. DQ forces a quantum of 1ms, breaking slightly larger jobs into chunks, thereby decreasing utilization due to context switches. At the same time, the responsiveness gained by DQ is marginal, as human perception hardly differs for a 1-ms quantum vs. a 10-ms quantum.