1.

Eggert Answer:

you specify a bogus address. You issue the system call and it scrambles part of the memory.

user calls read into A, trashes part of kernel

execute(A). Takes boot sector A and jumps into. it. The question is independent of VM.

You issue a system call to read into it. It returns back into it. Maybe that is the process table. Maybe your boot thing won’t work anymore. You can’t do it as directly as that

I'm accepting answers regardless of whether they say it's possible or if they say it's not, as long as the major problems that arises are addressed:

- memory access violations while loading the new OS

- difficulty in accessing physical memory and the addressing problems

- problem with virtual memory and memory address translation (only 2 points for this)

In addition, credit is given if:

- the bootstrapping process using Ubuntu is explained

Full Credit Answer:

The bootstrap process would not work. In order for the CPU to jump into a 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 the 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.

If, on the other hand, the fresh new disk somehow becomes the "primary" disk and thus readable through readsector(), then with the 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 are disabled before jumping to that physical memory though. (I.e., enable real mode on x86 machines.)

2.

2a) ‘close’ is a function that takes an integer (file descriptor) and returns an integer depending on if it managed to close it (-1/ERRNO if failure occurred, 0 if successful).

2b) It’s essentially similar to the API, but at the binary level. To give an example with a bit more detail, you could have your argument in edi, your result in rax, with ERRNO on failure. Note that it should be apparent in the answer whether the function call level close is being referred to or the system call level close, as the system call version of this would differ slightly.

2c) Yes, it does correspond- this code is actually the direct result of the assembly being fed into a C compiler, so there should not be any negotiation about whether it actually works. There are a variety of potential discrepancies, so I’ll just list a couple of the most likely ones here.

>The source code says -1 – n, and the machine code doesn’t have a subtract in it anywhere- this is because both the source and assembly code are actually just flipping the bits of x in their own ways. Hence in the end they correspond.

> We can see the source code calls bar, but the machine code never uses bar and jumps straight to close. This is okay because the compiler has figured out that it’s a constant pointer and will never change, so it will just dereference it at compile time and not bother to do it. Putting it more simply, in the end they’re performing the same action because close is going to get called either way, so the C and assembly correspond to each other.

2d) It’s not really the C API because, well, this is the assembly language level. Saying the ABI would be more accurate if you wanted to go that route. Strictly speaking, however, what it’s following is the assembly language API.

2e) An important factor here is: is this ‘close’ the system call or function call? As we know, system call is hard modularity, function call is soft modularity. The system call of close is being called like a function call here, though, so we see soft modularity.

3.

Yes, we can. We could store the N-R stacks to disk, so they won't take up memory. However, this means context switching would be much slower as disk I/O is orders of magnitude slower than RAM.

You could also say no, then you have to give a valid and reasonable explanation.

(Saying yes by merging N stacks into R stacks is not correct as it does not save memory.)

4.

Eggert Answer:

The idea here is you read from a slow device like a keyboard. You are in a read eval print loop. Read something from user and do it over again. In your program, you might want to say if someone does ctrl c then say interrupt. This program will have trouble with this semantics of Linux B. Calls B. Read will return from 47 bytes. Yes, there was a signal, but eventually the guy typed something. Did a signal arrive while I was reading. There is a global variable every time you did an IO. Win for Linux A

Win for Linux B. Just before a program exits. Just wants to count the number of interrupts. That benefits. If you interrupt it, it goes and interrupts the counter. At the end, you print it out and are done. Doesn’t like Linux A.

Full Credit Answer:

4a)

#include <unistd.h>

#include <signal.h>

// etc.

char \*buf = NULL;

void sighandler(int signum) { // reset buffer on SIGHUP

free(buf);

buf = malloc(1024);

}

int main(void) {

signal(SIGHUP, sighandler);

// start of critical section

buf = malloc(1024);

if (!buf) {

perror("malloc");

exit(1);

}

if (read(0, buf, 1024) < 0) {

// end of critical section (branch A)

perror("read");

exit(1);

}

// end of critical section (branch B)

...

}

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.

#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 signal to update the setting.

5.

Implementation of setup:

1. Must set up a pipe before fork

2. Must call fork

3. Must call close on the end of the pipe that's not being used

4. Must start the ping-pong process

Implementation of ping and pong:

Must block by calling read and blocking till a write is called

Must unblock the other process by writing to the pipe.

- partial credit given for use of each system call

- partial credit given if explanation is sound but code is incomplete/wrong

x credit is not given when fork and wait is used: that is not the requirement here.

6.

Here to get full points, you have to discuss both the utilization and fairness on two cases.

1. If the time slice for DQ is shorter than 10ms, then DQ has lower utilization due to more context switches, but higher fairness as processes have a higher chance to run in a fixed time range.

2. If the time slice for DQ is longer than 10ms, then DQ has higher utilization due to fewer context switches, but lower fairness.