CS 3214 Summer 2020 Test 1 - Solutions

July 5, 2020

Contents

Rules

- This exam is open book, open notes, and open Internet, but in a read-only way.
- You are not allowed to post or otherwise communicate with anyone else about these problems.
- You are required to cite any sources you use, except for lecture material, source code provided as part of the class materials, and the textbook.
- If you have a question about the exam, you may post it as a *private* question on Piazza. If it is of interest to others, I will make it public.
- Any errata to this exam will be published prior to 12h before the deadline.

1 Basic OS Functions (10 pts)

1.1 CR3 (5 pts)

On the x86 architecture, the CR3 control register contains the physical address of the current page directory, which is a data structure that translates the current process's virtual addresses to physical addresses. This data structure is directly consulted by the processor's MMU (memory management unit). For instance, in a given process, the virtual address 0x400000 may be translated to physical address 0x13ffe000. According to the Intel manual, the value of the CR3 control register can be assigned with a mov instruction after loading the desired value in a register, for instance, `mov %rax, %cr3` would move the physical address contained in `%rax` to `%cr3` and activate the mappings contained in a page directory at the chosen address.

Is it possible to create (i.e., compile or assemble, and successfully link) a user program in Linux that uses such a mov instruction to assign a value to CR3? If so, what would happen if this user program were executed with the `exec()` system call? Justify your answer either way.

Solution: This question is a variation of the "dual mode" operation mini demo. It is definitely possible to create (compile or assemble, and successfully link) such a program, for instance, by compiling the following source code:

```
int
main()
{
    asm("mov %rax, %cr3");
}
```

When run, it will cause a segmentation fault because a mov instruction that affects the CR3 register must be a privileged instruction. If it were not, user programs could gain access to arbitrary physical memory by reprogramming the MMU, including that of the kernel and of other users' processes.

Answering this question didn't require detailed knowledge about the page directory, MMU, or CR3 itself. The description given in the problem should have been sufficient to realize that it can be manipulated only via privileged instructions, which in a system exploiting dual-mode operation cannot be executed from user mode when they occur in a program.

Note that the stipulation "user program" and the mentioning of being executed with the "exec" system call made it clear that the program in question is running in (unprivileged) user and not in (privileged) kernel mode.

1.2 What happens to programs that forget to exit (5 pts)

In one of the video demos on low-level I/O, I showed the following program:

```
/*
 * A short assembly program that makes 2 system calls to output
 * "Hello, World\n" to its standard output stream
 *
 * Build with
 *  gcc -c hello.S
 7   *  ld hello.o -o hello
 8   *
 9   * Written by G. Back for CS 3214 Summer 2020
10   */
11  #include <asm/unistd.h>
12
13      .section .rodata
14  hello:
15      .string  "Hello, World\n"
16      hello_len = . - hello
17
18      .text
19      .globl _start
20  _start:
21      // call write(1, hello, sizeof hello);
22      mov $__NR_write, %rax
23      mov $1, %rdi
24      lea hello(%rip), %rsi
25      mov $hello_len, %rdx
26      syscall
27
28      // call exit(0)
29      mov $__NR_exit_group, %rax
30      mov $0, %rdi
31      syscall
```

What would happen if I removed lines 28 – 31 and ran this program again? Explain your answer.

Solution: As you can try out, removing it causes a segmentation fault:

```
$ ./hello
Hello, World
Segmentation fault (core dumped)
```

The reason is that the CPU will continue to execute the instruction at the address after the write syscall instruction. Depending on how the program is built, different content could be located at that address (or, if it's the last page in the executable, the process's valid addresses may end). In this case, the read-only string `Hello, World` appeared right after the code in memory. The byte pattern `0x48, 0x65, 0x6c` representing the first 3 characters of the string `Hello, World` represent opcode 0x48 (which is the rex.W prefix for 64-bit instructions, which in this case is ignored since it does not apply to the next instruction), followed by the instruction `gs insb (%dx),%es:(%rdi)` with encoding 0x65 0x6c which happens to be a privileged I/O instruction.

The more general point is that the OS, when it gives a process control of the CPU, will let the process execute whatever instructions it fetches from its memory. If a process fails to call exit(), the CPU will simply continue to fetch and execute instructions from memory - likely to the point where these "instructions" are either illegal instruction encodings, privileged instructions, or instructions that cause other faults such as memory access violations. The OS will take action only if/when the user program executes those instructions (or attempts to), and then terminate the process with a signal. In other words, the protection facilities provided by a system employing dual-mode operation are strong enough that the OS does not have check or verify whether a program is well-behaved and calls exit() when it is done ahead of time. This is an important technique that ensures that user programs are executed by the CPU at full speed, it is also called limited direct execution.

2 Process States (10 pts)

I wrote a program called `fork.c`. After compiling this program, I started it using the cush shell. I observed that the cush shell does not return to the prompt unless/until the user types Ctrl-C. About 17 seconds after starting the program, in a separate terminal, I typed `ps f` and observed the following process tree:

```
32658 pts/19   Ss     0:00 cush
 9839 pts/19   S+     0:00  \_ ./fork
 9840 pts/19   S+     0:00      \_ ./fork
 9842 pts/19   T+     0:00      |   \_ ./fork
 9841 pts/19   R+     0:17      \_ ./fork
```

Reconstruct the `fork.c` program I wrote. Pay attention to the (Linux) process states, the CPU usage, and the parent/child relationships displayed by `ps`.

Solution: The fork process has 2 children, one of which has a grandchild, thus the general structure must be `if (fork()) { fork(); } else { fork(); }`

2 of the four processes are in the S state (BLOCKED, "Sleeping" in Linux), which can be accomplished simply by calling `sleep(2)` (or by having the process call some variant of the `wait*()` system call, etc.) One process is in the running state for 17s, which indicates some kind of computation, which is easiest accomplished with an infinite loop. The 4th process is in the T state, which in Linux means it's been stopped by a stop signal. This can be accomplished by sending, for instance, SIGSTOP to itself via `kill(2)` or `raise(3)`.

```c
#include <unistd.h>
#include <signal.h>

int
main()
{
    if (fork()) {
        if (fork())
            sleep(100);
        else
            while (1)
                ;
    } else {
        if (fork())
            sleep(100);
        else
            raise(SIGSTOP);
    }
}
```

3 Unix System Calls (15 pts)

Shells use Unix system calls to set up programs they run and redirect their input/output in ways desired by the user. Sometimes, programs themselves wish to make use of similar functionality without requiring access to a shell program.

Translate the following specification into a C function `redirect_command_to_string`.

- The function should execute a new program in a new process. The arguments for the new program are given as a NULL-terminated array of character pointers 'cmd'. The function should read up to 'maxsize' bytes of the program's standard output into a buffer allocated with `malloc()` and zero-terminate the buffer. The function shall not return until the program has terminated. After it terminates, the function should return a pointer to the program's output which the caller is responsible for freeing.
- The function must use only functions that are described in Section 2 of the Unix manual, except for `malloc(3)` and if needed `execvp(3)` and `printf(3)`.
- The PATH variable shall be used to find the executable of the program requested. The function shall not affect the wait status of other child processes that may be in existence at the time of the call. The function shall not leak file descriptors.
- If the command cannot be found, the function shall return `command not found` as a zero-terminated C string.
- The function's behavior is undefined if any resources are exhausted such as the limit on the number of processes or file descriptors.

If this function is integrated in a C program `redstring.c` as shown below:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

static char *
redirect_command_to_string(char * const cmd[], int maxsize)
{
}

int
main(int ac, char *av[])
```

```
{
    printf("Output of %s is %s", av[1], redirect_command_to_string(av+1, 300));
}
```

then interaction with it at the command line may look like this:

```
$ ./redstring date
Output of date is Tue Jun 30 12:21:20 EDT 2020
$ ./redstring ls -dl ..
Output of ls is drwxr-xr-x 14 gback gback 4096 Jun 28 17:46 ..
$ ./redstring data
Output of data is command not found
```

Solution: A minimal solution that elides error checking is given below:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/wait.h>

/* Use only system calls to implement reading the first
 * maxsize bytes of the standard out of a forked child
 * process. (Except for execvp and malloc.)
 * Since the specification allowed for undefined behavior
 * in the presence of errors, error checking is elided.
 */
static char *
redirect_command_to_string(char * const cmd[], int maxsize)
{
    int fd[2];
    pipe(fd);   // create a pipe for parent/child communication
    int child = fork(); // fork child
    if (child > 0) {    // in parent
        close(fd[1]);   // close unused fd
        char *ret = malloc(maxsize+1);  // allocate memory
        // read content. This has to occurs first, because the
        // pipe must be closed before waiting for the child.
        // To tolerate short reads if the child program uses
        // unbuffered output and sends content in smaller
        // chunks, we use a loop
        ssize_t total_bytesread = 0;
        while (total_bytesread < maxsize) {
            ssize_t bytes_read = read(fd[0],
                                      ret + total_bytesread,
                                      maxsize - total_bytesread);
            if (bytes_read <= 0)    // EOF or error
                break;
            total_bytesread += bytes_read;
        }
        assert(total_bytesread <= maxsize);
        ret[total_bytesread] = '\0';
        // Now close the pipe to ensure that the child process
        // receives a SIGPIPE signal if it attempts to write
        // more into the pipe
        close(fd[0]);
        // Finally, wait for this child (restricted to only
        // the situation when this child has exited, not
        // stopped, and not affecting other children)
        waitpid(child, NULL, 0);
        return ret;
    } else {    // child process
        close(fd[0]);       // close unused fd
        dup2(fd[1], 1);     // dup write end of pipe to stdout
        close(fd[1]);       // close now unused fd
        execvp(cmd[0], cmd);    // execute child
        printf("command not found\n");  // handle fault case
        exit(1);
    }
}

int
main(int ac, char *av[])
{
    printf("Output of %s is %s", av[1],
        redirect_command_to_string(av+1, 300));
}
```

Since you had the chance to work offline on this question and use tools such as valgrind and your own child programs to test, I applied a higher standard than for a sit-down exam and looked for a fully correct solution. For instance, you can test with `./redstring yes yes` (which shouldn't get stuck), or `./redstrong ./shorty` where `shorty.c` reads:

```
#include <unistd.h>

int
main()
{
    write(1, "Hello ", 6);
    sleep(1);
    write(1, "World\n", 6);
}
```

Also, note that calling `wait()` instead of `waitpid()` might reap other children, which the specification disallowed. Failing to close all file descriptors in the parent means that the program would run out of file descriptors if it calls the function repeatedly, which is also a defect.

4 Linking (15 pts)

4.1 Symbol Types (7 pts)

`symtab.o` is a relocatable object module compiled from `symtab.c`. Its symbol table contains:

```
0000000000000004 C a
0000000000000000 D b
0000000000000000 R c
0000000000000000 T d
0000000000000007 t e
0000000000000000 b f
0000000000000004 d g
```

Reconstruct `symtab.c`. Note that the numbers in the first column may differ.

Solution: A possible reconstruction is shown below:

```
int a;
int b = 1;
const int c = 2;
void d() { }
static void e() { }
static int f;
static int g = 3;
```

4.2 Best Practices (8 pts)

Below, you will find two files `main.c` and `walk.c` which are part of a small program that uses separate compilation. This miniproject does not use best practices when it comes to how to structure and encapsulate code that is separately compiled and then linked. Your task is to fix these mistakes, subject to the following restrictions.

You may

- Change the modifiers (e.g. `static`, `extern` of both variables and functions)
- Replace definitions with declarations as needed
- Introduce one or more header files as needed
- Move declarations to/from header files as needed

You may not

- Remove global variables (replace them with getters or by passing pointers as function call arguments, etc.)

- Change the file in which each function is defined (i.e., simply placing all code in a single .c file is not a valid solution.)

```
/* main.c */
#include <stdio.h>

typedef void (*walk_fun_t)(char *);

extern void walk_dir(char *dirname, walk_fun_t fun);
int number_of_entries_found;

void
print(char *name)
{
    printf("%s\n", name);
}

int
main(int ac, char *av[])
{
    walk_dir(av[1], print);
    printf("%d entries\n", number_of_entries_found);
}
```

```
/* walk.c */
#include <sys/types.h>
#include <dirent.h>

typedef void (*walk_fun_t)(char *);
int number_of_entries_found;

/* Enumerate all files and directories contained in the directory
 * given by `dirname` and invoke `fun` on each of the entries.
 */
void
walk_dir(char *dirname, walk_fun_t fun)
{
    DIR * dir = opendir(dirname);

    struct dirent *entry;
    while ((entry = readdir(dir))) {
        number_of_entries_found++;
        fun(entry->d_name);
    }
    closedir(dir);
}
```

Solution: The following issues should be addressed:

- Lack of a header file walk.h for exported functions and variables by walk.c
- The header file should contain declarations for variables, functions and in this case, also needed type definitions
- Removal of the reliance on the common symbol
- Declaring functions not used outside main.c using `static`
- Alternatively, moving `print` as an inline function into walk.h (using either static inline or correct use of extern inline with extern inline declaration in walk.c).

```c
typedef void (*walk_fun_t)(char *); // shared typedef

// declaration of global variable
extern int number_of_entries_found;

// declaration of function (extern is optional)
extern void walk_dir(char *dirname, walk_fun_t fun);
```

```c
/* walk.c */
#include <sys/types.h>
#include <dirent.h>
#include "walk.h"

// definition of global variable owned by this file
int number_of_entries_found;

/* Enumerate all files and directories contained in the directory
 * given by `dirname` and invoke `fun` on each of the entries.
 */
void
walk_dir(char *dirname, walk_fun_t fun)
{
    DIR * dir = opendir(dirname);

    struct dirent *entry;
    while ((entry = readdir(dir))) {
        number_of_entries_found++;
        fun(entry->d_name);
    }
    closedir(dir);
}
```

```c
/* main.c */
#include <stdio.h>
#include "walk.h"

static void     // only used in main.c, should be static
print(char *name)
{
    printf("%s\n", name);
}

int
main(int ac, char *av[])
{
    walk_dir(av[1], print);
    printf("%d entries\n", number_of_entries_found);
}
```

With these changes, compilation with

```
gcc -Wall -Wmissing-prototypes -Wl,--warn-common main.c walk.c
```

will be successful.

(More changes are possible, such as finding a name with prefix `walk_` for the exported `number_of_entries_found` variable, etc.)