University of Windsor

School of Computer Science
Faculty of Science
COMP-2560: System Programming (Fall 2021)

Lab#	Date	Title	Due Date	Grade Release Date
Lab08	Week 08	**Bootstrap a Program File**	Nov. 10, 2021, Wednesday 4:00 AM EDT	Nov. 17, 2021

The main objective of this lab will be to learn how a program file is bootstrapped into memory and become a process. We have already explained that shell does this task for us by doing system calls. Shell is an application-level (user-level) program. If the shell can bootstrap a program, why not other application-level programs.

Step 1. Library Routine to Bootstrap Program: `system`

Let's write a simple program that accepts two numbers from the input and outputs the sum in the output. From the previous lab (Lab06), by input and output, we mean the standard input (fd=STDIN=1) and output (fd=STDOUT=2):

```
hfani@alpha:~/lab08$ vi main_add.c
#include <stdio.h>
#include <stdlib.h>
int result;
int main(int argc, char *argv[]){
        int a = 0;
        int b = 0;
        a = atoi(argv[1]);
        b = atoi(argv[2]);

        result = a + b;

        printf("%d + %d = %d\n", a, b, result);
        return 0;
}
```

As seen, the `main` function is used in its complete format, accepting `argc` as the number of arguments (parameters) from the shell, and `argv` as the list of lists of characters (strings), and returning an integer number to show the exit status. Since all the arguments from the shell to the `main` function of the program are characters, we have to convert them into equivalent numbers. We used the library routine `atoi` (ASCII to integer) as seen in line#7 and 8. Note that argv[0] is always the name of the program itself.

A sample run of the above program will be:

```
hfani@alpha:~/lab08$ ./main_add 3 40
3 + 40 = 43
```

We know that shell is bootstrapping our program into the memory and making it a process. However, there is `system` function in library routine `stdlib.h` that allows executing any valid command string on a shell but from another program. This is similar to bash scripts, where we put the shell's command lines in a script and give it to the shell for execution. But using `system` function, we do the same using C programming language. The following program bootstraps the above program using `system` function:

```
hfani@alpha:~/lab08$ vi system_bootstrap.c
#include <stdlib.h>
int main(void){
        system("./main_add 2 6");
}
```

We compile and run this new program bootstrapper:

```
hfani@alpha:~/lab08$ cc system_bootstrap.c -o system_bootstrap
hfani@alpha:~/lab08$ ./system_bootstrap
2 + 6 = 8
```

As seen, we ask the shell to bootstrap the `system_bootstrap` program that bootstraps the `main_add` program that sums two numbers (shell → `system_bootstrap` → `main_add`)

Are `system_bootstrap` and `main_add` programs executed under the same process, that is, have the same process ID? To answer this question, we can do a system call to the kernel and ask for the process IDs using `getpid()`, available in `unistd.h` header. We can change `main_add` and `system_bootstrap` program to print out their process IDs foremost and then continue with the rest of their instructions:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int result;
int main(int argc, char *argv[]){
        printf("main_add pid: %d\n", getpid());
        int a = 0;
        int b = 0;
        a = atoi(argv[1]);
        b = atoi(argv[2]);

        result = a + b;

        printf("%d + %d = %d\n", a, b, result);
        return 0;
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main(void){
        printf("system_bootstrap pid:%d\n", getpid());
        system("./main_add 2 6");
}
```

After rebuilding `main_add` and `system_bootstrap` programs, a sample run will output:

```
hfani@alpha:~/lab08$ cc main_add.c -o main_add
hfani@alpha:~/lab08$ cc system_bootstrap.c -o system_bootstrap
hfani@alpha:~/lab08$ ./system_bootstrap
system_bootstrap pid:1080957
main_add pid: 1080959
2 + 6 = 8
```

You see that their process IDs are different; they were executed under different processes. *This means that in our UNIX-based/like system, we can run multiple processes simultaneously!* This is contrary to our assumptions during our course. So far, we've assumed only one process can be executed in memory (besides the kernel and the shell.)

Wait! We still need the shell to bootstrap our `system_bootstrap` program.

Step 2. System Call to Bootstrap Program: `exec`

UNIX standard header, `unistd.h`, has 7 different variations of a system call to ask directly from the kernel to bootstrap a program. As we promised in class, we choose the one that works with the file descriptor; that is, we open the program file and give its file descriptor to the kernel to bootstrap the program. Here is an example:

```
hfani@alpha:~/lab08$ vi exec_bootstrap.c
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
int main(void){
        printf("system_bootstrap pid:%d\n", getpid());

        int fd = open("./main_add", O_RDONLY);
        printf("fd of main_add: %d\n", fd);

        char *newargv[] = {"./main_add", "3", "6", NULL};
        char *newenviron[] = {NULL};

        int res = fexecve(fd, newargv, newenviron);
        printf("%d\n", res);
}
```

The `main_add` is the program files that is built in previous section. As seen in `newargv` variable, we programmatically provide new arguments to the `main_add` program. Although we are supposed to provide 3 arguments, we have to provide an additional `NULL` argument (Why?) `fexecve` accepts a pointer to a copy of the environment variable table too, which we passed `NULL` for simplicity. The return value of exec system calls is nothing if successful, and -1 on error:

```
hfani@alpha:~/lab08$ cc exec_bootstrap.c -o exec_bootstrap
hfani@alpha:~/lab08$ ./exec_bootstrap
system_bootstrap pid:1224784
fd of main_add: 3
main_add pid: 1224784
3 + 6 = 9
```

Wait a sec! If you closely look at the PIDs, they are the same. Here is why by our reference book:
"When a process calls one of the exec functions (e.g., exec_bootstrap), that process is completely replaced by the new program (e.g., main_add), and the new program starts executing at its main function. The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process — its text, data, heap, and stack segments — with a brand-new program from disk" Advanced Programming in the UNIX Environment, Chapter 8, Section 10.

Note that we still need the shell to bootstrap the `exec_bootstrap`.

Step 2. A Program that Generates Program

We already know that we can create a new file by `open()` or `creat()` system calls. The new file can be a C program. Also, we know how to bootstrap another program using the `system` function or `fexecve` system call. So, we can bootstrap `cc` program to build a C program. Putting it all together, we can dynamically generate a program, build it, and run it using a bootstrapper.

Let's write a bootstrapper using `system` function that generates a program for summing two numbers, builds it, runs it, and prints out the result:

```
hfani@alpha:~/lab08$ vi system_math_bs.c
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main(void){
        printf("system_bootstrap pid:%d\n", getpid());
        char *program = "#include <stdio.h>\n"
                        "#include <stdlib.h>\n"
                        "#include <unistd.h>\n"
                        "int result;\n"
                        "int main(int argc, char *argv[]){\n"
                        "printf(\"main_add pid: %d\\n\", getpid());\n"
                        "int a = 0;\n"
                        "int b = 0;\n"
                        "a = atoi(argv[1]);\n"
                        "b = atoi(argv[2]);\n"

                        "result = a + b;\n"

                        "printf(\"%d + %d = %d\\n\", a, b, result);\n"
                        "return 0;\n"
                        "}\n";
        printf("%s\n", program);
}
```

As seen, we are creating the text body of a program inside another program. Please note the following:

1) Newline character `\n` in the text body of the program should become normal text by `\\n`. Otherwise, it would be interpreted as the newline character in the bootstrapper program.
2) The quotation `"` in the text body of the program should become normal text by `\"` Otherwise, it would be interpreted as opening or closing quotation in the bootstrapper program.

The `\` is called *escape character,* and we use it whenever we want to treat a character as a normal text, not as a special character in C program.

Now, we can write the text body of the program into a file, build it by bootstrapping `cc` compiler using `system` function, and then call it using the same `system` function:

```c
#include <fcntl.h>
#include <string.h>
int main(void){
        printf("system_bootstrap pid:%d\n", getpid());
        char *program = "#include <stdio.h>\n"
                        "#include <stdlib.h>\n"
                        "#include <unistd.h>\n"
                        "int result;\n"
                        "int main(int argc, char *argv[]){\n"
                        "printf(\"main_add pid: %d\\n\", getpid());\n"
                        "int a = 0;\n"
                        "int b = 0;\n"
                        "a = atoi(argv[1]);\n"
                        "b = atoi(argv[2]);\n"

                        "result = a + b;\n"

                        "printf(\"%d + %d = %d\\n\", a, b, result);\n"
                        "return 0;\n"
                        "}\n";
        //printf("%s\n", program);
        int fd = creat("./math.c", S_IRUSR | S_IWUSR);
        printf("fd for math.c file:%d\n", fd);
        write(fd, program, strlen(program));

        system("cc ./math.c -o math");
        system("./math 10 40");
}
```

As seen in the last few lines, we:

1) Create a new file for the program, called `math.c`, and we write the text body into it.
2) We build it into the program file `math` using `cc` compiler using `system` function.
3) We bootstrap the program using `system` function for `10` and `40`.

Here is the result:

```
hfani@alpha:~/lab08$ ./system_math_bs
system_bootstrap pid:1337484
fd for math.c file:3
main_add pid: 1337495
10 + 40 = 50
```

Step 3. Lab Assignment

You must redo step 2 considering the following conditions:

1) The bootstrapper program should accept 3 arguments from a user:
   a. Math operation: `+, -, *, /`
   b. First operand
   c. Second operand
2) The bootstrapper should generate a new body text for the program based on the user's input values, build it, and run it using `fexecve` system call only. You are not allowed to use `system` function.
3) For text manipulation or printing out the results, you can use library routines, e.g., `string.h`, `stdio.h`.

A sample run of your final submission for `-` operation on `10` and `30` would be:

```
hfani@alpha:~/lab08$ ./exec_math_bs - 10 30
exec_math_bs pid:1370081
fd for math.c file:3
math pid: 1370081
10 - 30 = -20
```

A sample run of your final submission for `/` operation on `10` and `30` would be:

```
hfani@alpha:~/lab08$ ./exec_math_bs / 10 30
exec_math_bs pid:13018681
fd for math.c file:3
math pid: 13018681
10 / 30 = 0
```

Note that the PID of the `exec_math_bs` and the PID of `math` programs are the same when using `exec` system calls. The sample code for steps 1 and 2 have been attached in a zip file named `lab08_hfani.zip`.

1.1. Deliverables

You will prepare and submit the program in one single zip file `lab08_uwinid.zip` containing the following items:

(90%) `lab08_uwinid.zip`
- (70%) `exec_math_bs.c` => the bootstrapper builds and runs with no error for all 4 operations.
- (20%) `results.pdf/jpg/png` => the image snapshot of the program run for all 4 operations
- (Optional) `readme.txt`

(10%) Files Naming and Formats

Please follow the naming convention as you lose marks otherwise. Instead of `uwinid`, use your own account name, e.g., mine is hfani@uwindsor.ca, so, `lab08_hfani.zip`