CS 5600 Project 1

1. Overview

A modern compiled program is quite complex, pulling in thousands of lines of include files and using dozens of shared libraries in addition to "hidden" code the compiler adds to the executable for things like exception handling, threads, and other startup and shutdown tasks. In this assignment we'll ignore all this, and create some executables using only the files in your directory.

In this homework you will implement simple versions of several basic operating system functions:

- invoking system calls
- program loading
- stack/context switching

You are provided with the following files: - "syscall.S" - defines a `syscall` function, not surprising used to invoke system calls without using the system libraries - "sysdefs.h" - all the `#defines` you'll need to use the "syscall.S" and the other assembly files - "elf64.h" - defines for parsing executable file headers, so that your program can load an executable into memory and then execute it - "vector.S", "call-vector.S" - creates a miniature system call table, so that those programs can access functions in your main program - "switch.S", "stack.c" - switches stacks, to create multiple threads, and prepares a stack frame for starting a thread - "Makefile" - contains somewhat unusual compile commands so that all of this will work.

1.1. Part 1 – read / write / exit

In this part you are going to write a "bare" program which reads input and writes output, without using any of the C standard library, C runtime library, or include files. You'll write functions 1. to invoke the `read` and `write` system calls, and other functions 2. to read lines of input and write strings to output, using the `read()` and `write()` functions which you wrote, and then 3. combine them into a simple loop to echo what you typed and exit on "quit".

1.2. Part 2 – Program loading / syscalls

For the second part you'll write a simple "single-user operating system" that runs specially-compiled "micro-programs". These programs don't perform I/O on their own, but instead call functions in *your* code via a system call table. Your program will act as a simple shell, reading a line, splitting it into words, loading a program and starting it with arguments.

1.3. Part 3 – Context Switching

This part will create two processes (really threads), process 1 and process 2, and load an executable into each one, placing each one at a different position in memory. You will give each a stack, and implement three functions: `yield12` and `yield21` which switch from thread 1 to thread 2, and from 2 to 1, and `exit` which switches back to the main program.

1.4. Project rules in general (for all projects)

1.4.1. Teams and collaboration

You are expected to work in teams of 2 to 3 students; you will submit **one copy** of the homework **per group** and receive the same grade. Feel free to discuss the ideas in the homework with other groups, but absolutely no sharing of code across groups is allowed. All your code should come from your own fingers typing on the keyboard – if you're copy-and-pasting it from somewhere (from another team or from the Internet) it's likely to be academic dishonesty.

If you want to work on the code with your teammate using services such github, you must place the code in a **private repository** at all times which only your team has access to.

1.4.2. Submitting your work

You should place all relevant files to the project in a folder and create a zip file by zipping the folder itself. You **should not include object files or executable binary files** that are generated during the compilation process. The zip file should be submitted via **Canvas. Only one of the team members may submit the work:** if one member uploaded the file on Canvas, the other team member will automatically see the same submission. **Call your files Project1_group#.**

1.4.3. Progress report

You will be required to submit a progress report partway through the assignment via Canvas. This is a simple text describing the work that you have done so far and your plans for finishing the project. It can include things like “discussed design of part X”, or “implemented Y function”, etc. It does not have to be long and can be just a few bulletin points. **Each student on the team should describe in their own words what they specifically did on that part.**

1.4.4. Testing

You are expected to implement tests for your code. Depending on the project, you may not need to write test cases as part of the submission, but the rule of thumb is to always test your code thoroughly. Code that hasn’t been tested doesn’t work. Therefore if you didn’t test your code but it passes the tests for grading, then it’s a lucky accident and doesn’t deserve as good a grade as if you actually tested it and know it works. Run your tests under `valgrind`, as well – it will catch a lot of bugs before they actually happen. Use GDB for debugging.

1.4.5. Code Quality

Unlike most workplaces or open-source projects, the coding standards for this class are minimal. They are:

1. Initialize every variable when you declare it.
2. Get rid of all compiler warnings. There’s almost always a good reason the compiler is complaining – fix it.
3. Indent your code consistently and reasonably.
4. Use reasonable variable and function names. If you change what a function does, **change the name of the function to reflect that**.
5. Write reasonable comments when needed: delete comments that has nothing to do with your code and if the code is self-explanatory you do not need comments.

Other than #1 and #2, these can all be described as **Don’t write ugly code**. Take some pride in your work.

1.4.6. Grading

Things you can lose points for:

- Failure to achieve any progress by the progress report date (slip day does not apply for progress report).
- Really ugly code, compiler warnings, uninitialized variables.
- Failure to implement all of the requested functionality.
- Test failures (on the grading tests)
- Not enough testing.

2. Before you begin

Fully read through the project description. To work on each part, you will need to read and understand the given code. Each file includes some useful comments, but you will need to study extra materials on your own to fully understand what is going on. Here are some useful resources.

- GNU assembly syntax: https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax
- Linux man pages (you can search for syscall interfaces): https://linux.die.net/man/
- ELF file specification: https://refspecs.linuxfoundation.org/elf/elf.pdf
- System V application binary interface: https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf

Finally, there are office hours offered by the TAs and myself. Take advantage of office hours and ask questions.

3. Part 1 - read/write/exit

Write a program which:

- reads a line of input from standard input (file descriptor 0)
- if the line starts with 'q', 'u', 'i', 't' then exit
- otherwise print the line to standard output (file descriptor 1) and loop

You will need to implement the following in "part-1.c":

- system call wrappers for `read`, `write`, and `exit`
- logic to read a line of input using `read`. (I suggest reading 1 byte at a time and sticking it in a buffer; if it's '\n' then add a zero on the end and you're done)
- a function to print a string using `write`. (again I suggest doing it a byte at a time)
- a simple loop, with a check after reading a line to see if the user entered "quit".

You can assume that input lines are never more than 200 bytes long. Remember that you can't return from the `main` function – you have to call `exit`.

You'll probably want to factor out `readline` and `print`, since you'll need them as separate functions in parts 2 and 3.

Remember that you can't use any functions other than the ones that you write yourself. That means no `printf` – use GDB for your debugging (and no `strcmp` for string comparisons, or `malloc` for allocating memory).

Compiling

You can compile part 1 with the command `make part-1`. You can "clean up" (i.e. delete the compiled executables and intermediate files) with the command `make clean`.

Testing

You won't need to do a lot of testing for this one - it either works or it doesn't. Before you submit your work I would suggest you run it under `valgrind`, which will tell you if you're reading or writing memory that you didn't intend to:

```
user@linux:proj1$ valgrind ./part-1
==12338== Memcheck, a memory error detector
==12338== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==12338== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==12338== Command: ./part-1
==12338==
Hello, type lines of input, or 'quit':
> this is a test
you typed: this is a test
> quit
==12338==
```

```
==12338== HEAP SUMMARY:
==12338==     in use at exit: 0 bytes in 0 blocks
==12338==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==12338==
==12338== All heap blocks were freed -- no leaks are possible
==12338==
==12338== For lists of detected and suppressed errors, rerun with: -s
==12338== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
user@linux:proj1$
```

Submission

Your submission for this part will be the contents of "part-1.c" and any code you wrote for testing if it is in a different file. Include a README.txt file in the submission folder to indicate which files you have modified/created to finish part 1.

4. Part 2 - Program loading / syscalls

Write a program which:

- reads a line of input
- splits it into words
- exit if the first word is "quit"
- load the file named by the first word into memory
- provide "system calls" for the loaded program
- call the loaded program's entry point.
- repeat

You'll find a function for splitting a line in "part-2.c". You'll need your read/write/exit system call implementations from part 1, and the `readline` and `print` functions you created (assuming you factored that code out).

You'll need to write several more system calls

- `open`, `close`. (you can assume `open` always takes 2 arguments)
- `lseek`, to read specific parts of the executable file
- `mmap`, to allocate memory at specific addresses, and `munmap` for when you're done

I've provided prototypes for each of the system calls; writing them should be straightforward.

Loading an executable file into memory is a lot trickier. You might want to look at the file "elf-example.c" first, which prints out the information we need from the executable - the "entry point" (i.e. function to call to start execution), and the entries in the program header table which are marked `PT_LOAD`.

ELF files have two sets of headers - "section headers", which get used by the compiler and linker, and "program headers", which specify how to load it into memory; we can ignore the section headers. To find the program headers you read the ELF header (`struct elf64_ehdr`) at the beginning of the file into memory - the program headers an array of `struct elf64_phdr`; if you read the ELF header into a variable named `e_hdr`, then that array has `e_hdr.e_phnum` entries starting at offset `e_hdr.e_phoff` bytes into the file.

You will be loading a *relocatable* executable, which starts at address 0 but can actually be loaded starting anywhere in memory. I would suggest loading it at 0x80000000, which will keep it far away from your program. (the rest of your program is statically linked at 0x400000, and the stack will be at an address somewhere a bit below 0x800000000000) Each program section of the executable specifies a virtual address (`phdrs[i].p_vaddr` in the ELF example code); the program will run correctly as long as you add the same constant to each virtual address to determine where to load it into memory (remember that you'll have to add the constant to the entry point address, as well).

Additional Details

You'll use the mmap system call to allocate memory. Although the segments may have varying protections in the ELF header, we'll allocate them all as R/W/X to make things simpler.

```
void *buf = mmap(address, size, PROT_READ | PROT_WRITE |
                 PROT_EXEC, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
if (buf == MAP_FAILED) {
    print("mmap failed\n"); /* your "print" function */
    exit(1);                /* your "exit" function */
}
```

mmap operates in terms of 4096-byte pages, so you'll need to "round up" the section size to the next multiple of 4096 – there's a macro in sysdefs.h that you can use:

```
int len = ROUND_UP(size, 4096);
```

You'll need to use lseek and read to read each program segment into the allocated memory chunk:

```
lseek(fd, offset, SEEK_SET);
read(fd, address, size_in_file);
```

Note that for some sections (e.g. data) the header gives two sizes – the size of the memory region to allocate (p_memsz), which we round up to a multiple of 4K, and the number of bytes to read from the file (p_filesz). The second number may be smaller, which means that we only read that amount of data in from the file, and leave the rest of the memory region as zero (that's how it comes from mmap).

Now you'll need to make a function call to the program entry point. You'll set a function pointer to the value of e_hdr.e_entry (well, that value plus the offset you used with all your other sections), and then call it:

```
void (*f)();
f = hdr.e_entry + offset;
f();
```

When you load the program into memory you should keep a list (well, an array) of the memory addresses that you allocated and their lengths, so that you can munmap them when you're done. If you don't, you'll probably crash the second time through your command loop, when you try to allocate memory at the same location a second time.

First part - the wait executable

All that the wait executable does is waste time, by looping a billion times or so, and then return, which lets you tell that you loaded and ran it properly (if it finishes immediately, you didn't).

A hint for getting this working - assuming you factored out a function to load and run an executable (you did, didn't you?), you can put off writing your command loop and just hard-code a call to load and run it, e.g.:

```
exec_file("wait");
```

(you're going to have to write a readline function for the next part, which you can re-use in your command loop.)

Second part - system calls

For this part you'll need to do two things:

Write a command loop. This will

- read a line of input,
- split it into words,
- save the 2nd, 3rd etc. word as arguments for the micro-program, and
- load and execute the program named by the first word (and preferably doesn't crash if you type the wrong thing, which you'll know if open returns a value less than zero)

Write three "system calls"

- `readline(char *buf, int n)` – this reads up to `n-1` bytes into `buf[0]`, `buf[1]`, ..., then adds a terminating zero byte at the end. It stops reading into `buf[]` when either (a) the last character read was '\n', or (b) `n-1` bytes have been written
- `print(char *buf)` – print the null-terminating string to standard output – i.e. go through `buf[0]`, `buf[1]`, ..., writing out each non-zero byte and then stopping at the zero byte that terminates the string.
- `char *getarg(int i)` - returns argument `i`, or 0 if there weren't that many arguments.

Normally you would use a standard library function to split the line into parts (`strtok` or `strsep`), so I've provided you with a function that does it for you. It's OK to handle a max of say 10 arguments, including `arg0` — the current micro-programs don't go higher than 2.

You should be able to test your Part 2 implementation by running `hello` and `ugrep`; `hello` does what you expect it to, and `ugrep` should behave similarly to the `grep` program, outputting lines of input which match its argument (instead of end-of-file, it will exit when it sees a blank line).

```
user@linux:proj1$ ./part-2
> wait
> hello
Hello world!
> ugrep xyz
ugrep: enter blank line to quit
a line without the pattern
a line without the pattern
a line with 'xyz' in it
-- a line with 'xyz' in it
next line blank to quit:
> quit
user@linux:proj1$
```

Testing

There aren't many cases to test here:

1. Does it run `wait`, `hello`, and `ugrep` without crashing?
2. Can you run them multiple times, in different orders?
3. If you type something that isn't one of the three microprogram names, or "quit", does it crash?
4. If you type "quit", does it exit without crashing?

I suggest running these tests under `valgrind`, as that will help ensure that you don't have any "hidden" bugs that only show up when the TA is grading your assignment.

Debugging

`readelf`. The `readelf` command prints out all the headers in an executable; you can use that to check whether your code is reading the information correctly:

```
code$ readelf -a wait
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  [...]
  Entry point address:               0x1000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          26264 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         11
  [...]
```

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
[...]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000321 0x0000000000000321  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x00000000000000aa 0x00000000000000aa  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x0000000000000050 0x0000000000000050  R      0x1000
[...]
```

There are also a few GDB tricks which make it a lot easier to debug this assignment:

Loading an additional symbol file. If you crash in the middle of the microprogram, you want the debugger to know what's going on. You can load symbols for the microprogram, as long as you know the offset in memory where you loaded it – in the example here it's 0x1000000. (Note that the address you give has to be the offset **plus 0x1000 (entry point addr)**):

```
code$ gdb part-2
GNU gdb (Ubuntu 8.3-0ubuntu1) 8.3
 ...
Reading symbols from part-2...
(gdb) b do_getarg
Breakpoint 1 at 0x401600: file part-2.c, line 153.
(gdb) run
Starting program: .../part-2
> ugrep foo

Breakpoint 1, do_getarg (i=0) at part-2.c:153
153 {
(gdb) add-symbol-file ugrep 0x1001000
add symbol table from file "ugrep" at
    .text_addr = 0x1001000
(y or n) y
Reading symbols from ugrep...
(gdb) bt
#0  do_getarg (i=0) at part-2.c:153
#1  0x00000000010010ab in main () at ugrep.c:36
#2  0x00000000004014d1 in execfile (file=0x7fffffffe378 "ugrep", offset=16777216) at part-2.c:110
#3  0x0000000000401825 in main () at part-2.c:208
(gdb)
```

Without that second symbol file, GDB would be totally confused:

```
(gdb) bt
#0  do_getarg (i=0) at part-2.c:153
#1  0x00000000010010ab in ?? ()
#2  0x0000000000000000 in ?? ()
(gdb)
```

Single-instruction stepping. To figure out what's going wrong when you call a function pointer, you may need to single step an instruction at a time, because the same bug that's keeping your code from working is probably keeping GDB from putting line-by-line breakpoints in the right places. First we'll tell GDB to print out the next instruction after every command (`display/i $pc`), then we'll use the `stepi` command to verify that it's calling into the code of the `wait` microprogram properly:

```
(gdb) display/i $pc
1: x/i $pc
=> 0x4014b6 <execfile+801>: mov    -0x128(%rbp),%rdx
```

```
(gdb) stepi
109     void (*f)(void) = hdr.e_entry + offset;
1: x/i $pc
=> 0x4014bd <execfile+808>: mov    -0x150(%rbp),%rax
(gdb)
0x00000000004014c4  109     void (*f)(void) = hdr.e_entry + offset;
1: x/i $pc
=> 0x4014c4 <execfile+815>: add    %rdx,%rax
(gdb)
109     void (*f)(void) = hdr.e_entry + offset;
1: x/i $pc
=> 0x4014c7 <execfile+818>: mov    %rax,-0x68(%rbp)
(gdb)
110     f();
1: x/i $pc
=> 0x4014cb <execfile+822>: mov    -0x68(%rbp),%rax
(gdb)
0x00000000004014cf  110     f();
1: x/i $pc
=> 0x4014cf <execfile+826>: callq  *%rax
(gdb)
0x0000000001001000 in ?? ()
1: x/i $pc
=> 0x1001000:   endbr64
(gdb)
0x0000000001001004 in ?? ()
1: x/i $pc
=> 0x1001004:   push   %rbp
```

Your main program (whether part-1, part-2, or part-3.c) is going to live somewhere around 0x401000 in memory; here we see code computing the entry point value `f`, then making a function call to 0x1001000. (even without loading symbols, you can use `objdump -d wait` on the command line to verify that the first two instructions are `endbr64` and `push %rbp`.)

Oh, and I don't know what `endbr64` means, either, or at least I didn't until I got curious after writing up this example. If you're looking at assembly language code while you're debugging, you usually don't have to really understand it for it to be helpful - for example in this case we really only need to make sure that the addresses are correct.

Submission

Your submission for this part will be the contents of "part-2.c" and any code you wrote for testing, if it is in a different file. Indicate in the README.txt file which files you have modified/created to finish part-2.

5. Part 3 - Context Switching

For this last part we'll throw away the command loop, and we'll load and run two processes which switch back and forth between each other:

```
user@linux:proj1$ ./part-3
program 1
program 2
program 1
program 2
program 1
program 2
program 1
```

```
program 2
done
user@linux:proj1$
```

Program loading

We'll load two different programs, `process1` and `process2`, at two different locations in memory. I would suggest modifying your load-and-execute function from Part 2 to (a) take an offset as a parameter, and (b) return the entry point rather than calling it. Don't worry about unmapping the memory when you're done - we're just going to run things once and then exit, so Linux can clean everything up for us when we're done.

Context switching

We'll need to allocate stacks for the two processes - 4096 bytes should be more than enough. You can either use global arrays:

```
char stack1[4096];
char stack2[4096];
```

or you can allocate them with `mmap`:

```
void *stack1 = mmap(0, len, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

(the zero address tells `mmap` to pick an unused address.)

We'll use the `setup_stack_0` from `stack.c`, and the `switch_to` function defined in `switch.S`:

```
extern void switch_to(void **location_for_old_sp, void *new_value);
extern void *setup_stack0(void *_stack, void *func);
```

`setup_stack0` takes a pointer to the **top** of a region of memory (not the bottom - in our case we would pass `stack1+4096`), and returns a stack pointer which will cause `switch_to` to invoke `func` with no arguments.

We need to implement the following system calls: - `yield12` - this switches from process 1 to 2 - `yield21` - this switches from process 2 to 1 - `uexit` - this switches back to the original process stack (see below)

There are three different stacks - the original process stack that the operating system gave us, and the stacks we allocated for processes 1 and 2 - and we'll need locations (global variables) for storing pointers to each of them. We're going to switch from the OS stack to the process 1 stack; it will then switch to process 2, and they'll switch back and forth a few times, then one of them will call `uexit` which will switch back to the main OS stack and we can call the real `exit` system call to finish up.

Testing

There isn't much to test here - there aren't any parameters or variable inputs to the assignment, so it either runs to completion without crashing, generating the output shown above, or it doesn't.

Debugging

There's a lot to debug here. Unfortunately, `valgrind` won't be of much use for this part of the assignment, as the moment you switch the stack pointer it thinks you have a horrible bug.

You'll want to use the same tricks for loading additional symbols - in this case you can load them for both the "process1" and "process2" executables - and for single-stepping by instruction.

Finally, you may find yourself crashing with this error:

```
Program received signal SIGSEGV, Segmentation fault.
switch_to () at switch.S:34
34      mov  0(%rax),%rax
```

That's an assert in the `switch_to` function, telling you that you've tried to switch to a bogus stack. Both `setup_stack` and `switch_to` leave a flag value on the top of the stack; if you try to switch to a stack that doesn't have this flag, it crashes **before** the switch happens. (Once you switch stack pointers, there's no way to tell where you called `switch_to` from, or to look at the variables in that function.)

Submission

Your submission for this part will be the contents of "part-3.c" and any code you wrote for testing if it in a different file. Indicate in the README.txt file which files you have modified/created to finish part-2.

6. Final zip file submission

Your zip file should include all necessary files (include all given files even if you have not modified them) to build the executable for part-1/2/3 and the README.txt file. See Section 1.4.2. for more instructions.