In this homework you will write a miniature operating system, which runs inside of a Linux process and can load tiny programs from the filesystem and execute them. You are given some code in your repository to start with; you will extend this code in order to implement your solution.

Rules

I expect you to work in teams of two students; you will submit one copy of the homework and receive the same grade. Feel free to discuss the ideas in the homework with other groups; however you must write answers to written questions in your own words, and absolutely no sharing of code across groups is allowed. You do not need to explicitly submit your assignment – your grade will be based on the code in your repository at the time it is due.

Please commit your changes frequently – at least once every two days while you are working on the project, and preferably much more often. These checkins serve to document that you and your teammate wrote the code you are submitting, rather than copying it from someone else. **Failure to check in frequently will result in a lower grade.**

You will be required to submit a **progress report** partway through the assignment – this is a text file in your repository (named **`progress.txt`**) describing the work you have done so far and your plans for finishing the project.

Remember to use the **git push** command to upload your changes after committing.

Things you can lose points for:

- Failure to achieve any progress by the progress report date, or failure to commit your code frequently.
- Violations of class C coding standards – uninitialized pointers, unreadable code (e.g. totally not indented), lack of comments, use of inline assembler.
- Failure to implement all of the requested functionality.

Note that sharing code across groups or significant differences between checked-in work and your progress report description will be considered academic dishonesty.

Programming Assignment materials and resources

You will download the skeleton code for the assignment from the CCIS repository server, trac.ccs.neu.edu, using the '`git clone`' command:

```
git clone https://github.ccs.neu.edu/cs5600-01-f15/team-nn-hw1
```

where '*nn*' is your team number. Periodically you will commit checkpoints of your work into your local repository using '`git commit`':

```
git commit -a -m 'message describing the checkin'
```

and push the commits to the central repository:

```
git push
```

You should be able to complete this homework on any 32-bit x86 Linux system (**NOT 64-bit)**; however, the official environment is the CS-5600 virtual machine which you used for Homework 0, and which may be downloaded from http://pjd-1.ccs.neu.edu/files/CS5600-f15-Ubuntu32.ova

Contents

This assignment contains the following files:

homework.c - the main file. It contains a number of helper functions, and the assignment itself is described in comments in this file. The main() routine is written to dispatch to the various parts of the assignment; thus to run the code corresponding to question 1 you will compile and then run the command:

```
./homework q1
```

compile-it.sh - a shell script for compiling. You shouldn't need to modify this; in particular, the compile commands for the micro- programs are very tricky. To compile all the programs, use the command:

```
sh compile-it.sh
```

and to clean up any object and executable files:

```
sh compile-it.sh clean
```

You can compile additional micro-programs with the following command:

```
sh compile-it.sh compile myprog.c
```

q1prog.c, q2prog1.c, q2prog2.c, q3prog.c - "micro-programs" that will run inside the mini-OS you are writing.

vector.s - an assembly language file containing stub functions to invoke mini-OS calls through a jump vector at 0x09002000

You will need to modify homework.c to implement the functions `q1()`, `q2()`, and `q3()` as described in the matching comments in homework.c. The result will be a miniature multi-tasking, multi-user operating system which is able to load and execute programs with user I/O.

Deliverable

The following file from your repository will be examined, tested, and graded:

homework.c

ELF file format handling

In this assignment you will load code from an executable file in the ELF format into memory. For more details on this format and how to read it into memory, please check the last two pages of this assignment.

Question 1 – Program loading, output

The micro-program **q1prog.c** uses the `print` micro-system-call (index 0 in the vector table) to print out "Hello world".

a) complete the print() system call, the skeleton of which may be found in **homework.c .** Since this

function is in **homework.c**, it can make use of standard library functions such as `printf()`.

b) in the `q1()` function, add code to read `q1prog` (not `q1prog.c`, `q1prog.o`, or anything else) into memory starting at 0x09000000. (the code you are provided conveniently sets the pointer 'proc1' to this address) The main() function is the 'entry point' of the executable (see the description of the ELF format at the end of this assignment) and you should assign that address to a function pointer and call it to execute the micro-program.

The test script **q1test.sh** may be used to test your answer to this question.

Question 2 – Input, simple command line

Add two more functions to the vector table, at offsets 1 and 2:

`void readline(char *buf, int len)` - read a line of input into 'buf'
`char *getarg(int i)` - gets the i'th argument (see below)

Note that `readline` should read input and store it into memory starting at `buf`, returning when either (a) a newline character has been received and stored, or (b) `len-1` bytes have been stored. Before returning, append a null character (0) to the end of the string.

Write a simple command line which prints a prompt and reads command lines of the form '`cmd arg1 arg2 ...`'. For each command line:

- Save `arg1`, `arg2`, ... in a location where they can be retrieved by `getarg`
- Load and run the micro-program named by '`cmd`'
- If the command is "`quit`", then exit rather than running anything

Note that this should be a general command line, allowing you to execute arbitrary commands that you may not have written yet.

You should be able to test your question 2 implementation by running `q1prog` and `q2prog`; `q1prog` should print "Hello World" and `q2prog` should behave similarly to the "grep" program, outputting lines of code which match its argument. (instead of end-of-file, it will exit when it sees a blank line)

```
> q1prog
Hello world
> q2prog test
this is a line that doesn't match
this line contains "test"
– this line contains "test"
this line doesn't

>
```

In addition, the file **q2test.sh** will perform additional tests on your answer for question 2.
NOTE - your vector assignments have to match the ones in vector.s - 0 = print, 1 = readline, 2 = getarg

Remember that your command line should be able to load and execute arbitrary micro-program files. You can test this by creating a new program, e.g. `myuprog.c`, which is a modification of e.g. `q1prog1.c`, compiling it (see directions above, under "compile-it.sh"), and then running it from the q2 command line. Note that if opening the file fails (probably because it doesn't exist) you should print an error and continue.

Question 3 – Context switching

Create two processes which switch back and forth. You will need to add another 3 functions to the table, at offsets 3, 4, and 5:

`void yield12(void)` - save process 1, switch to process 2
`void yield21(void)` - save process 2, switch to process 1
`void uexit(void)` - switch to saved parent (i.e. homework.c) stack

The code for this question will load 2 micro-programs, `q3prog1` and `q3prog2`, into memory at locations `proc1` and `proc2`. (0x9000000 and 0x9001000) which are provided and merely consists of interleaved calls to `yield12()` or `yield21()` and `print()`.

To implement `yield12`, `yield21`, and `uexit`, as well as to switch to process 1, you will need to use the following two functions defined in **misc.c**:

`setup_stack(stack_ptr_t stack, void (*function)())`
sets up a stack (growing down from address '`stack`') so that switching to it from '`do_switch`' will call '`function()`'. Returns the resulting stack pointer to be used by `do_switch`.

`do_switch(stack_ptr_t *location_for_old_sp, stack_ptr_t new_stack_ptr)`
saves current stack pointer to (`*location_for_old_sp`), sets stack pointer to '`new_stack_ptr`', and returns. Note that the return takes place on the new stack.

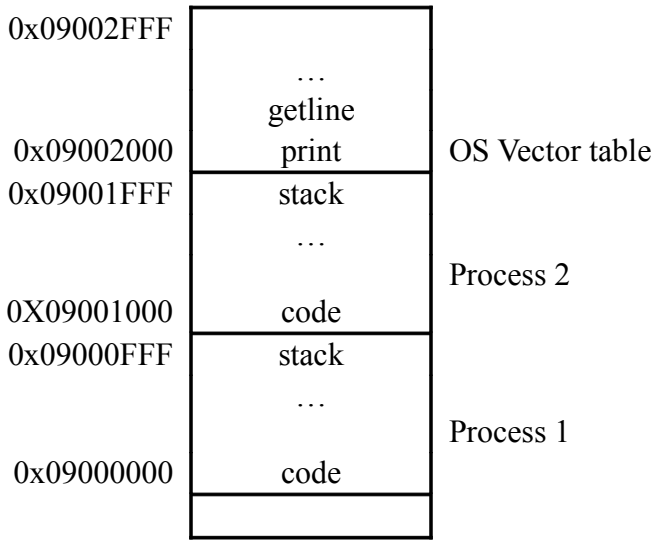
Hints:

- Use `setup_stack()` to set up the stack for each process to `proc1_stack` and `proc2_stack` (0x900FFFC and 0x901FFFC respectively). Note that setup_stack returns a stack pointer value which you can switch to via do_switch.
- You need a global variable for each process to store its context (i.e. stack pointer)
- To start you need to switch to the stack pointer for process 1; you should save the main program stack pointer when you do this, so that `uexit` can return to it.

The file **q3test.sh** may be used to test your answer to this question.

Additional information and hints:

1. Memory layout:
The code I've provided in homework.c creates three 4096-byte memory segments. The memory map is as follows:



2. Don't modify vector.s, compile-it.sh, or **misc.c**

If you want to play with them on your own time you are free to, but for grading the code you submit in **homework.c** will be compiled with standard versions of these files rather than ones that you modified.

3. Pay attention to compiler warnings

Even in C they catch a lot of bugs.

4. Match function prototypes.

If the micro-program thinks that the print function is 'void print(char*)', then the function you pass to setvector() in homework.c had better have the same prototype - the compiler won't catch any mistakes here. In most cases the prototypes in uprog.h will catch this, but not always.

5. Micro-program compilation

The following files represent the steps taken in compiling the micro-program q1prog.c:

`q1prog.o` - linkable object code
`q1prog` - ELF format, lined at 0x09000000 or 0x09001000

Note that although 'q1prog' is in the same format as normal Linux executables, it will crash if you try to run it from the Linux command line (try it – it won't hurt anything), as it expects to be able to access the vector table at 0x09002000..

6. Single-instruction stepping

GDB has a command, 'stepi', which steps by a single instruction. It doesn't display the instruction, however. If you want that, use the following command:

```
display /i $pc
```

After every step, this will tell GDB to display the memory pointed to by the program counter ($pc in gdb syntax), as a decoded instruction (/i).

Other gdb commands which may be useful, besides the 'print' command which lets you print various C expressions, and the commands described in Homework 0:

```
info regs    - show registers
x/10i <addr> - examine 10 instructions at <addr>
x/10x <addr> - examine 10 words in hex at <addr>
backtrace    - what it says
```

7. Symbol files

When you load code and data from 'q1prog' into memory, gdb has no idea that they represent a program. To fix this, tell the debugger to load symbols from 'q1prog':

```
add-symbol-file q1prog 0x9000000
```

(or 0x09001000 for q3prog2)

More debugging:

The 'valgrind' command is often useful in finding bugs - it is a memory checker which detects all sorts of bugs which are all too common in C. To use it just pass the command and arguments to valgrind:

```
valgrind ./homework q1
```

Note that valgrind is not going to be very useful for question 3, as it doesn't like it at all when you switch stacks.

How to read ELF Files

First we'll describe using Unix I/O system calls (open/close/read/write) rather than the Standard I/O Library calls (fopen/fread/fgets/printf) you're more likely to be familiar with. (you're free to read data from the ELF file however you want; however as you'll see below, the low-level Unix I/O system calls "fit" this task very well.)

Unix I/O

The functions we care about:

```
int fd = open(filename, O_RDONLY);
```

The 'open' call takes a filename and a flag indicating whether we're opening it for reading, writing, or both, and it returns an integer called the "file descriptor".

We'll use three other system calls:

```
read(fd, ptr, len);
lseek(fd, byte_offset, SEEK_SET);
close(fd);
```

The 'read' system call reads 'len' bytes from the file into memory starting at memory address 'ptr'. Note that 'read' doesn't specify *where* to read from – it just reads the next 'len' bytes, starting where it last left off. If we want to jump around in the file, which you need to do when reading an ELF file, then we'll need the 'lseek' call. This sets the current position in the file to 'byte_offset' bytes from the beginning, and the next 'read' will start there. Finally, 'close' does what you would expect.

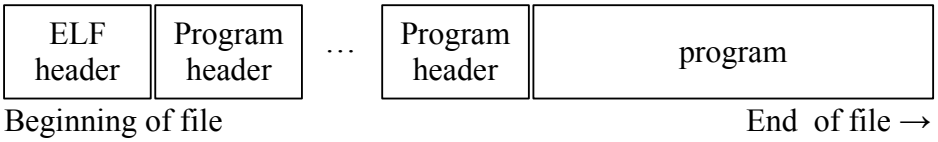
For more information you can run the following commands:

```
man 2 read
man 2 lseek
```

ELF Headers

To interpret the ELF file we'll use the include file "elf32.h", which defines the structures and constants found in the file. (there's a system file, <elf.h>, but it's a lot harder to use)

An ELF file looks like this:



It starts with an ELF header which includes:

- "Magic number", identifying it as an ELF file, plus CPU type and other info
- Address of the program "entry point", where execution should start.
- Location of the program headers, as an offset in bytes from the beginning of the file.

To read the header we can do the following:

```
struct elf32_hdr hdr;
int fd = open(filename, O_RDONLY);
read(fd, &hdr, sizeof(hdr));
```

In the debugger we can now see the ELF header:

```
(gdb) n
65        int fd = open("q1prog", O_RDONLY);
(gdb) n
66        read(fd, &hdr, sizeof(hdr));
(gdb) n
67    }
(gdb) p hdr
$1 = {e_ident = "\177ELF\001\001\001\000\000\000\000\000\000\000\000\000",
  e_type = ET_EXEC, e_machine = EM_386, e_version = 1, e_entry = 0x9000000,
  e_phoff = 52, e_shoff = 4788, e_flags = 0, e_ehsize = 52, e_phentsize = 32,
  e_phnum = 2, e_shentsize = 40, e_shnum = 13, e_shstrndx = 10}
```

So now we know that the entry point for 'q1prog' is at 0x09000000, and that there are 2 program

headers (e_phnum) at offset 52 bytes (e_phoff) from the beginning of the file.

So we can declare an array and read the program headers:

```
int n = hdr.e_phnum;
struct elf32_phdr phdrs[n];
lseek(fd, hdr.e_phoff, SEEK_SET);
read(fd, &phdrs, sizeof(phdrs));
```

and after executing this code we can see the program headers in the debugger:

```
(gdb) p phdrs[0]
$2 = {p_type = PT_LOAD, p_offset = 4096, p_vaddr = 0x9000000,
  p_paddr = 0x9000000, p_filesz = 144, p_memsz = 144, p_flags = 5,
  p_align = 4096}
(gdb) p phdrs[1]
$3 = {p_type = PT_GNU_STACK, p_offset = 0, p_vaddr = 0x0, p_paddr = 0x0,
  p_filesz = 0, p_memsz = 0, p_flags = 7, p_align = 16}
(gdb)
```

The micro-programs are going to only have two program sections, and only one of them will be of type PT_LOAD, which means that it should be loaded into memory. To do this we have to find the offset in the file (p_offset, or 4096 bytes in this case) and the length (p_filesz). If we want to load the program at the address ‘proc1’, we’ll have to search(by looping for i=0 to i<hdr.e_phnum, and when phdrs[i].p_type == PT_LOAD, do the following:

```
lseek(fd, phdrs[i].p_offset, SEEK_SET);
read(fd, proc1, phdrs[i].p_filesz);
```

Once this is done, you can close the file descriptor and you’re ready to execute the program.

“Factoring” common functionality

You are going to need to read an ELF file in the solution to each of the questions in this homework. Rather than write this code in each place that it is needed, and then fix bugs in only some of those copies but not others, you should “factor” this into a function. Note that the function interface is going to need:

- input – file name
- output – success or failure (needed for question 2, in case the file is not found)
- output – entry point address (from the ELF header)

A C function has a single return value, so additional outputs have to be passed by reference:

```
int function(int input1, int *output2) {
     *output2 = 222;
     return 111;
}

int output2;
int output1 = function(input1, &output2);
```

Here we pass in the address of a variable (‘output1’) and the function stores a value in the memory location pointed to by that address – i.e. it sets the value of the variable, so the final values of output1 and output2 are 111 and 222, respectively.
