CS 5600 – Systems and Networks
Homework 1 – Program Loading and Context Switching

Assigned: September 10, 2012
Due: September 27, 2012

This homework you will write a miniature operating system. This mini-OS runs inside of a Linux process, and can load tiny programs from the filesystem and execute them. This document will first describe the code you are being given, and then give the programming questions.

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 will be required to submit a **progress report** partway through the assignment, accompanied by a Git checkin of your work at that point.

Things you can lose points for:

- failure to achieve any progress by the progress report date
- Violations of class C coding standards – uninitialized pointers, unreadable code (e.g. not indented), lack of comments, use of inline assembler.

Note that sharing code across groups or significant differences between checked-in work and your progress report description (i.e. lying about how much work you have completed) will be considered academic dishonesty. A plagiarism checker will be run on all submitted responses, and any suspicious ones will be investigated.

Team partners will be granted read and write permissions to each others' Git repositories for this homework. Please choose one of them for submitting your work, and indicate which one by creating a file named 'SUBMITTED_HERE' in that directory and pushing it to the repository.

Programming Assignment materials and resources

You will download the skeleton code for the assignment from the CS 5600 repository server, xevious.ccs.neu.edu, using the '`git clone`' command:

```
git clone gitolite@xevious.ccs.neu.edu:id-hw1
```

where '*id*' is your Blackboard user ID. (e.g. if your BB id is “smith.p”, then your repository is “smith.p-hw1”) From time to time you will commit checkpoints of your work into your local repository using '`git commit`':

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

and update the repository with your latest work using the '`git push`' command:

```
git push
```

The last version pushed to the repository before the homework deadline will be graded. You will be given access to your team member's Git repository; please choose one of the two for submitting your work, and indicate which one by creating a file named 'SUBMITTED_HERE' in the top level of that repository.

You should be able to complete this homework on any 32-bit x86 Linux system (**NOT 64-bit)**; however, if you have difficulty I would urge you to use one of the departmental Linux workstations in the first-floor lab. A partial list of the machines you can access remotely via SSH is:

breakout blackwidow hi-way bosconian astroblaster eagle gauntlet vanguard bubbles

A more complete list may be found by looking around the first floor lab, or via the following command on any CCIS machine:

```
ldapsearch -x cn=lab-machines
```

Your department home directory is available on all of these machines; please use a non-world-readable directory for your work (e.g. the 'classes' directory) so that others cannot read it.

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 2 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
```

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

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 found at the very beginning of the micro-program; you must invoke it by calling through a function pointer initialized to this address.

The test script **q1-test.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 program 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 **q2-test.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

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.

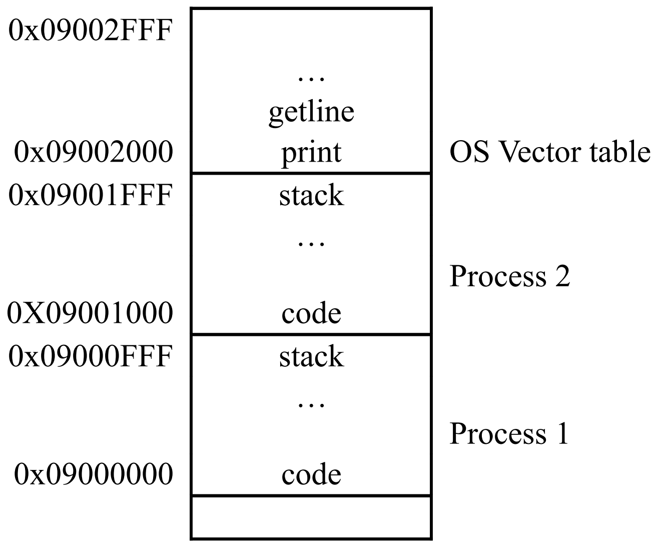
Hints:

- 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 **q3-test.sh** may be helpful in testing 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 q2prog.c:

`q2prog.o` -	linkable object code
`q2prog.elf` -	ELF format - i.e. a "normal" executable
`q2prog` -	binary-only executable, ready to load into your mini-OS

Note that although 'q2prog.elf' is in the same format as normal Linux executables, you can't run it from the Linux command line, as it expects to be able to access the vector table.

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 instruction bytes from e.g. 'q1prog' into memory, gdb has no idea that they represent a program. The original compiled file, 'q1prog.elf' (elf = “extended loader format) contains a large amount of information besides the instruction bytes, including this debug information, and can be loaded into gdb as so:

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

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 q2
```

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.