ASSIGNMENT 2

What's new in SOS1?

- A default exception handler *(exceptions.c)*
- A rudimentary process control block *(kernel_only.h)*
- A DUMB memory manager *(memman.c)*
- API for two system calls – `printf` and `getc` *(lib.h, lib.c, systemcalls.c, kernelservices.c)*
- Ability to run user programs in single tasking mode (this assignment)

Background

The background material for this assignment is in Chapter 3 of "Understanding a Simple Operating System." **You must read it (and the previous chapters) before attempting this assignment.**

The `run` command

(console.c)

SOS1 adds a new command: `run`. This command is used to execute a user program. The format of the command is: `run <start_LBA> <n_sectors>`. The implementation of the command is in *console.c*, the `command_run` function. It does some error checks and then calls the `run` function in *runprogram.c*.

User programs

(/userprogs)

SOS1 uses a raw binary format for user program executables. The *userprogs* folder has a sample program (*test.c*) and an adapted gcc compiler (`gcc2`) to produce executables in this format. Compile *test.c* from inside the *userprogs* directory as follows:

```
$./gcc2 -o test.out test.c
```

Program executables are written to `SOS.dsk` based on the *progs.conf* file. This is done by the `create` script you would use to compile SOS.

Executing user programs

(runprogram.c)

The `void run(uint32_t LBA, uint32_t n_sectors)` function is called from within *console.c*. The function requests `alloc_memory` for a memory area to load the user program (starting location `0x400000` for the DUMB memory manager). We always request for 16KB more than what is needed (12KB to be used as the user-mode stack, and 4KB for use as kernel-mode stack). It then loads sectors `LBA`, `LBA+1`,…, `LBA+n_sectors-1` from the disk to the memory area. The remainder of the function will be implemented by you in this assignment, and will perform the following tasks.

1. Fill in a rudimentary PCB for the user program.
2. Save state information in the console's PCB (so that you can return to it).
3. Call `switch_to_user_process` to start the user program.

Assignment objective

The file *runprogram.c* is filled with lots of **`TODO`** blocks. These **`TODO`** blocks are what you will have to finish. Some easy assembly language coding will also be required; there are some necessary pointers to write them at the end of this document. This is a really straightforward assignment if you understand the basics and do exactly what the **`TODO`** block asks; otherwise debugging can be real tough! Read the comments thoroughly.

Setup

Download the *ASG2.zip* file from the assignment page. Extract the files in it. In SOS1, use `./create` to create the `SOS.dsk` file. Just like you did for SOS0, create a new machine called SOS1 in VirtualBox (inside the XUbuntu environment), and use the *SOS.vmdk* file in the SOS1 folder for the hard drive. Once SOS boots, type the following at the console:

```
% run 1100 2
```

Nothing will happen for now, but once you finish the assignment, recompile everything, and restart the machine, you should see the *test.c* program run inside SOS.

Submission

Submit in Canvas the modified *runprogram.c* file, and a README file containing any information you feel the GTA should know before grading your program. Comment your program well (include your name).

Grading

The assignment is worth **60 point**. 40 of these points will be for the `switch_to_user_process` function. The assignment will be graded within the SOS development environment. It is your responsibility to ensure that the submitted code runs in that environment (irrespective of where you wrote or tested your code). Failure to do so will result in full penalty.

The late policy is available in the course syllabus. You must work alone on this assignment.

All 32-bit operations (registers must be 32-bit and variables must be of `uint32_t` type)

Writing assembly without any parameters

```
asm volatile ("<INSTRUCTION 1>\n" "<INSTRUCTION 2>\n" .... );
```

Examples:

```
asm volatile ("pushl $0x23\n" "popl %ds\n");
asm volatile ("pushfl\n"); //pushes the EFLAGS register's value into the stack
asm volatile ("pushal\n"); //pushes all general purpose register values into the stack
```

Moving a variable's value into a register (EDI, ESI, EAX, EBX, ECX, EDX, ESP or EBP)

```
asm volatile ("movl %0, %%<REGISTER>\n": :"m"(<VARIABLE>));

asm volatile ("movl %0, %%edi\n": :"m"(console.cpu.edi));
```

Moving a register's content into a variable

```
asm volatile ("movl %%<REGISTER>,%0" : "=r"(<VARIABLE>));

asm volatile ("movl %%esp,%0" : "=r"(console.cpu.esp));
```

Pushing a variable into the stack

```
asm volatile ("pushl %0\n": :"m"(<VARIABLE>));

asm volatile ("pushl %0\n": :"m"(p->cpu.ss));
```

Poping a value from the stack into a variable

```
asm volatile ("popl %0\n": "=r"(<VARIABLE>));

asm volatile ("popl %0\n": "=r"(console.cpu.eflags));
```