CMSC 412, Spring 2017

Project 0

Due: Friday, February 3rd, 2017 5:00 pm

- Download Project Source:

```
git clone https://gitlab.cs.umd.edu/cmsc412_sp2017/spring2017.git
```

use your directory id and password

Introduction

In this course we will implement a big project in subsequent phases. In each project we will be adding some new functionality to GeekOS. GeekOS is a tiny operating system kernel for x86 PCs. Its main purpose is to serve as a simple but realistic example of an OS kernel running on real hardware. To run our OS, we need to either use an actual machine (which must be an x86 machine) or the other (simpler) idea is to use an emulator; we will use QEMU.

The purpose of this assignment (project0) is to get you familiar with the GeekOS development environment, including the QEMU x86 emulator and the debugger gdb. The assignment is to modify GeekOS to impose resource limits on GeekOS processes in two simple ways:

- Only permit N user processes from being active at any given time (for some given N). An attempt to Spawn the N+1st process will result in failure.
- Only permit a given user process from performing M system calls (for some given M). When a process attempts to perform the M+1st system call, it should exit (i.e., the M+1st system call should be treated as an Exit system call).

Background

Running GeekOS

To compile GeekOS, untar the project source, change your directory to build, and execute `make`. To run the OS in the emulator execute `make run`.

Once you have started up GeekOS, you will be taken to the GeekOS shell prompt. That is, the first program that GeekOS runs is shell.exe, which takes commands for you to run just like the UNIX shell program. The GeekOS file system is set up so that a number of user programs are installed in the directory `/c`. The source code for these programs is in the directory `src/user` in the distribution. The programs `b.exe`, `null.exe`, and `long.exe` are three such programs. The first program simply prints all of its arguments. The second is an infinite loop. The third is a long, but not infinite loop. You can run the programs as you would expect. For example, to run the `b.exe` program, you could do

```
$ /c/b.exe 1 2 3 4
```

And it would print out the four arguments that you provided. The shell also takes a number of directives. For example, typing `exit` will terminate the shell (and your interaction with the OS). Look at `src/user/shell.c` to understand the other features of the shell.

You can easily add your own user programs by adding a file to `src/user`.

Debugging GeekOS

Start the emulator in debug mode with `make dbgrun`, then in another terminal run gdb with `make dbg`; the debugger will attach to the operating system. The symbols for user programs will not be loaded. We will fix this problem for project 1. In debug mode the system starts paused, so you can enter breakpoints before execution. To begin execution enter `continue` into the debugger.

Now we'll talk about how GeekOS works.

User processes vs. the kernel processes

In writing an operating system, you want to make a distinction between the activities that operating systems code are allowed to do and the activities user programs are allowed to do. The goal is to protect the system from incorrect or malicious code that a user may try to run. Bad things a program could do include:

- Making the operating system and/or other user programs crash
- Looking at data that belong to the system or to other programs
- Circumventing access control
- Using hardware incorrectly, with all the negative consequences that result e.g. machine crash or data corruption

Preventing these sorts of mistakes or attacks is accomplished by controlling the parts of memory that can be accessed when user code is running and limiting the set of machine operations that the user code can execute. The x86 processor provides the operating system with facilities to support these controls. A program that is running in this sort of controlled way is said to be running in *user mode*.

Anatomy of a User Process

In GeekOS, there is a distinction between **Kernel Threads** and **User Threads**. As you would expect, kernel activities (that is, *processes*) run as kernel threads, while user programs (or rather, user processes) run in user threads. A kernel thread is represented by a `Kernel_Thread` structure (in `include/geekos/kthread.h`)

```
struct Kernel_Thread {
    ulong_t esp;                        /* offset 0 */
    volatile ulong_t numTicks;          /* offset 4 */
    int priority;
    DEFINE_LINK(Thread_Queue, Kernel_Thread);
    void* stackPage;
    struct User_Context* userContext;
    struct Kernel_Thread* owner;
    int refCount;
  ...
}
```

The kernel thread contains all of the mechanisms necessary to run and schedule a process. In order to represent user processes, GeekOS includes an extra field in the Kernel_Thread structure that points to a User_Context structure. In a thread representing a kernel process, the User_Context will be NULL. The User_Context is defined in include/geekos/user.h:

```
struct User_Context {
    /* We need one LDT entry each for user code and data segments. */
#define NUM_USER_LDT_ENTRIES 2

    /*
     * Each user context contains a local descriptor table with
     * just enough room for one code and one data segment
     * describing the process's memory.
     */
```

```
    struct Segment_Descriptor ldt[NUM_USER_LDT_ENTRIES];
    struct Segment_Descriptor* ldtDescriptor;

    /* The memory space used by the process. */
    char* memory;
    ulong_t size;

    /* Selector for the LDT's descriptor in the GDT */
    ushort_t ldtSelector;
   ...
};
```

Most of the information in the `User_Context` structure has to do with the memory layout for the process, and you don't need to understand that for now. You will have to modify `User_Context` to contain some bookkeeping information as part of this project.

User threads are created using the routine `Start_User_Thread`. This takes as its first argument the `User_Context` to run within that thread. This context is created by the `Sys_Spawn` *system call*. A system call is like a function call that a user program makes to the OS kernel, to ask it to perform some service.

System Calls

The operating system kernel presents an interface to user processes for the services it will perform on their behalf. These are essentially functions. However, rather than literally performing a function call to access them, user processes have to use a special mechanism, called a system call. System calls are designed to protect the kernel's memory from malicious processes. On Intel's x86 processor, a user process issues a system call via a *trap*, which indicates that a system call is being requested; the identity of the system call routine and/or the arguments to pass to it are stored in the processor registers. This mechanism is hidden from the typical C programmer, since it is typically used only in the standard C library routines.

In GeekOS, when a user process issues a system call, the trap causes the routine `Syscall_Handler` in `geekos/trap.c` to be invoked. This routine examines the current value in the register `eax` and calls the appropriate system routine to handle the syscall request. The value in `eax` is called the **Syscall Number**. The routine that handles the syscall request is passed a copy of the caller's *context state*, so the values in general registers (`ebx, ecx, edx`) can be used by the user program to pass parameters to the handler routine and can be used to return values to the user program. This state is defined in the struct `Interrupt_State`, defined in `geekos/int.h`.

The routines that implement GeekOS system calls are in `geekos/syscall.c`. The `Sys_Spawn` code is implemented here, along with code for other system calls, like `Sys_Exit` for the system call used by a process to terminate itself. If you are curious how system calls are invoked from user processes, take a look at `src/user/shell.c` in your project distribution; you'll be modifying this program in your next project.

Creating User Processes - Sys_Spawn()

In order to create a user process, the `Sys_Spawn` syscall is used (defined in `geekos/syscall.c`). The function calls `Spawn()` (defined in the file `src/geekos/user.c`) that is used to launch user programs. FYI, this `Spawn()` function does the following (see the comments in `user.c` as well)

- `Read_Fully()` function to read an executable file from disk into memory
- `Parse_ELF_Executable()` function to populate an `Exe_Format` data structure.
- `Load_User_Program()` to set up the memory image for the new process and create a `User_Context` with the loaded program
- `Start_User_Thread()` with the new `User_Context`

`Sys_Spawn` returns the process id (pid) of the new thread.

Project Requirements

The purpose of this assignment is to modify GeekOS to impose resource limits on GeekOS processes in two simple ways:

1. Only permit a user process to perform at most *M* system calls (for some *M* determined by the Limit system call).
2. Add a system call LIMIT to GeekOS to set the limit on the number of times system calls can be made by the calling process (0..2*1024^3-1 are the valid ranges). A value of 0 as the limit means there is no limit on the number of systems calls that can be made. The C library entry point to call this system call should be Limit(int resource, int limit); where resource is the resource number (0 is system calls) and limit is the limit.
3. Add a default limit on the number of system calls per process to be 0 which means no limit.

Finally, you need to create a user-level process that spawns some other process a desired number of times. .

Limiting Number of Processes

To limit the number of processes, you should maintain a counter for the *current* number of (active) **user** processes. Then you must determine whether a Spawn is allowed by checking the value of the counter. Follow the code through the `Sys_Spawn` system call (defined in `geekos/syscall.c`) and make modifications as necessary in the kernel. If the limit has been reached then the process should not be created and `Sys_Spawn` should fail by returning -1.

Limiting Number of System Calls

In order to limit the number of system calls, you need to modify the `User_Context` structure (defined in `include/geekos/user.h`) to include a counter for the number of system calls the user process has performed. Then you should modify the kernel to update this counter each time it performs a system call on this process's behalf. For the system call that exceeds the limit (i.e., the N+1st system call), `Sys_Exit` should be called instead to kill the process.

Creating a user process to spawn a specified number of processes

Look at the code provided in `src/user/*` and specifically at `shell.c` to see how to spawn other processes using the system call interface provided. Write a user-level process called `spawn.exe` that takes as its argument the number *P* and then spawns *P* processes. Spawn the provided user-level program **`null.exe`**. Your spawn.exe program should display an error message "Maximum number of processes reached" if spawn fails due to reaching the maximum number of spawned processes.

Grading Criteria

Unpack and compile correctly and we can see that you've made a good effort at the project	20
Limiting number of processes correctly	30
Limiting number of system calls correctly	30
Testing spawn.exe *P* (where *P* is the number of times to spawn null.exe)	20
Total	100