Running User Programs (SOS1)

In this part, we will extend SOS0 so that it can run one user program at a time. However, a few things have to be decided before we can load a user program and start executing it.

GDT Entries

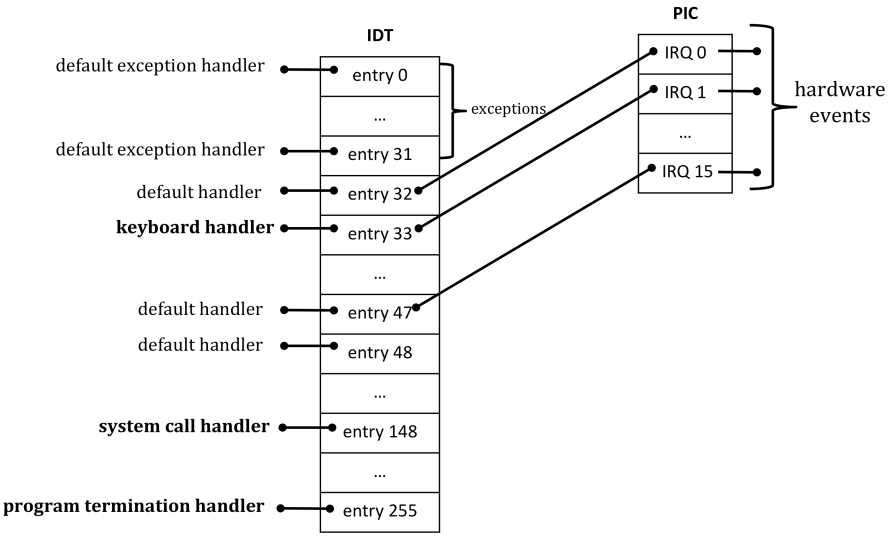
Our GDT at this point contains three entries: the null descriptor (entry 0), and two segments that span all available memory (entries 1 and 2). Entry 1 is setup with DPL = 0 (kernel access only), and will be used whenever the kernel runs its code. Entry 2 is setup with DPL = 0 and will be used whenever the kernel accesses any data in memory. Since no memory addressing is possible without referring to a GDT entry (in protected mode), we will have to setup two more GDT entries so that user programs can use them (DPL = 3). For now we will not set any base, limit or other properties for these entries; the values will depend on the running user program. The GDT will look like this at this point.

GDT entry 0	`0000000000000000`	Null segment (not used)
GDT entry 1	`00cf9a000000ffff`	Base = 0, Limit = 4 GB, Has code, DPL = 0
GDT entry 2	`00cf92000000ffff`	Base = 0, Limit = 4 GB, Has data, DPL = 0
GDT entry 3	`0000000000000000`	Not initialized
GDT entry 4	`0000000000000000`	Not initialized
GDT entry 5	`0000000000000000`	TSS (discussed later)

The GDT is declared in the `startup.S` file in SOS1 (ignore the sixth entry for now). We can access the GDT in our C code by declaring the `gdt` variable with an extern modifier in the C file – `extern GDT_DESCRIPTOR gdt[6]`. So, before a user program is started, we will set up `gdt[3]` and `gdt[4]` appropriately.

Default Exception Handler

Recall that interrupts 0 to 31 are generated for special events (often arising due to software problems). We will change the handler for these interrupts so that we can distinguish between regular interrupts and exceptions. The default exception handler is in `exceptions.c`. Its setup is similar to that of the default interrupt handler. When an exception occurs, the default exception handler will print OUCHH! Fatal exception. on the display, and then attempt to switch to the console. We are assuming here that the source of the exception is a user program, and the kernel code is error free!



User Programs

Users write their programs in one of many languages – C, C++, Java, Python, etc. These high-level language programs are then transformed into machine opcodes, i.e. instructions that the CPU knows how to execute. These are instructions such as ADD, MOV, JMP, CALL, etc. Compilers do this job. The question then is, although the CPU's instruction set is the same irrespective of whether we run Windows or Linux, why doesn't a program compiled in Windows also run in Linux?

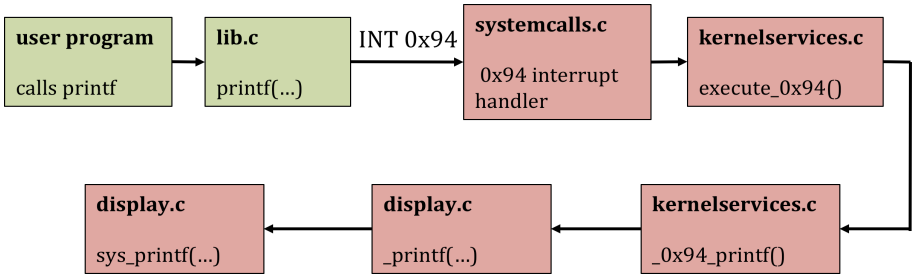
Executable Format

The answer to the aforementioned question is "because the executable produced by the compiler is not simply the machine opcodes, but also includes other metadata (e.g. file size, 32-bit or 64-bit code, location of entry point, sections, etc.), whose organization varies from one OS to another." When machine opcodes are packaged together with the metadata, we have what is called an *executable format*. In Windows, it is the EXE format, and in Linux, it is the ELF format.

In SOS, we do not include metadata in our executable. So, our program executables will consist of only machine opcodes. This is also known as a *raw binary executable*.

System Calls

User programs will not be able to do much if we do not allow them to get OS services. So, we will need to implement at least two system calls, `printf` and `getc`, before a basic user program can be run. The best way to understand a system call is to track its flow from the point where it is called. Lets do that. We will see how a `printf` in our user program will channel down to the `sys_printf` function in `display.c`.



The first component we need is a user side implementation of `printf`. This function is not part of the kernel and can be invoked directly by a user program. We have put it in `lib.c`, which also includes other functions that user programs can use. `printf` in `lib.c` uses the register based method of parameter passing – it loads the arguments (all are addresses) in the EBX and ECX registers, loads a service identifier number into EAX (we will use the number 2 to mean the display printing service; macros are defined in `lib.h`), and then generates interrupt 0x94. What will happen then?

In SOS0, this will print "Unhandled interrupt!" on the display since the handler for interrupt number 0x94 is the default handler. So now we need a different handler for entry 148 (0x94) of the IDT; we need a *system call handler*. The main function puts the address of this handler through `init_system_calls` in `systemcalls.c`. The system call handler begins at the location labeled as `handler_syscall_0X94_entry`. The code here is similar to any other interrupt handler, i.e. save CPU state, handle the interrupt, and then reload the CPU state. Our handler calls `execute_0x94` in `kernelservices.c`, which is where the system call is serviced. `execute_0x94` checks which service is requested (how?) and then calls the appropriate routine, in our case the `_0x94_printf` routine. Note that we are already in kernel mode. So, it is simply a matter of checking the parameters passed in

the EBX and ECX registers, and then calling `sys_printf`. Control flows back the way it came, and the handler will return back to `printf` in `lib.c`. `_0x94_printf` also puts the value 1 (success) or 0 (failure) in the EDX register, which the `printf` in `lib.c` can check to see if the system call was successful. This method can also be used when the kernel needs to pass data back to the user program (see the `getc` system call).

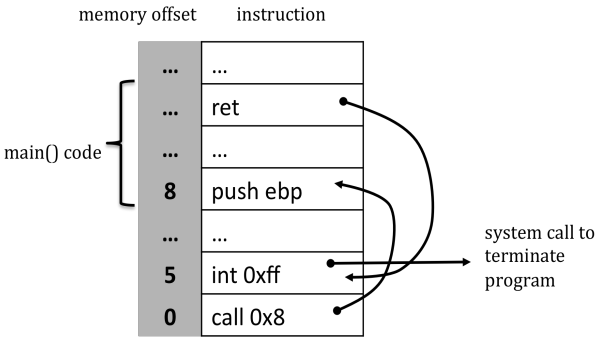
Creating User Programs

User programs can now print to the display and get user input by using the `printf` and `getc` functions in `lib.c`. But, how do we type our programs, and then compile them to a raw binary? Well, we cannot do that in SOS! SOS does not have an editor and a compiler. Hence, we will write our programs outside of SOS, compile it outside of SOS, and then put them in a fixed location in `SOS.dsk`. After all, our objective is to learn how user programs are executed by an operating system. Note that SOS also does not have any file system; so, inside SOS, we cannot refer to the program by a name. Instead, we will have to remember where we put the program executable on the disk, how many bytes it is, and then load it using the `read_disk` function.

We can type a user program in an editor of our choice (e.g. gedit) in the virtual environment. The language will be C, and the program should not use any library function that is not provided in `lib.c`. Similarly, when implementing a library function, we cannot use any standard C function (no stdlib, stdio, etc.). We will put the program in the `userprogs` directory. We should not use the regular gcc or g++ compilers, as they will produce executables in the ELF format. However, different flags can be passed to these compilers that force them to produce a raw binary executable. There is a script called `gcc2` that will do this. To compile your program `test.c`, go to the `userprogs` directory, and run

```
./gcc2 test.c –o test.out
```

`test.out` is then a raw binary executable. `gcc2` also compiles in the code from `lib.c`, so that when our user program calls `printf`, the code will be part of the executable. Note that this is not the way it is done in commercial operating systems. Ideally, it is sufficient to have only one copy of the library in memory, and different user programs should be able to use it. But then, this is SOS!



There is a problem here. Say we write a program that prints “Hello World.” So, the program will make a system call and do the job. Once the printing is done, control will go back to the user program. But, how will SOS know that the program has finished execution? The way to make this happen is to modify all user programs so that the first instruction calls the main function of the user program, and the second instruction issues a specific interrupt (in our case interrupt 0xFF). The following instructions will come from our program. When a program generates interrupt 0xFF, the handler never returns control back to the program. This handler is set up in `systemcalls.c`. The `gcc2` script is coded to make this necessary modification in all user programs. Further, the `create` script takes the executables and places

them on the disk while creating `SOS.dsk`. See the output messages of `create`, and note down the sector number and size (in bytes) of the program on disk.

The run Command

SOS0 does not have a command to run user programs located on the disk. So, lets add one. The `run` command (implemented in `runprogram.c`) takes two arguments – first one specifying the start sector number of the program on the disk, and the second specifying the number of sectors that the program occupies. So, if a program is located at sector 1100 and is 700 bytes in size, the command will be

```
run 1100 2
```

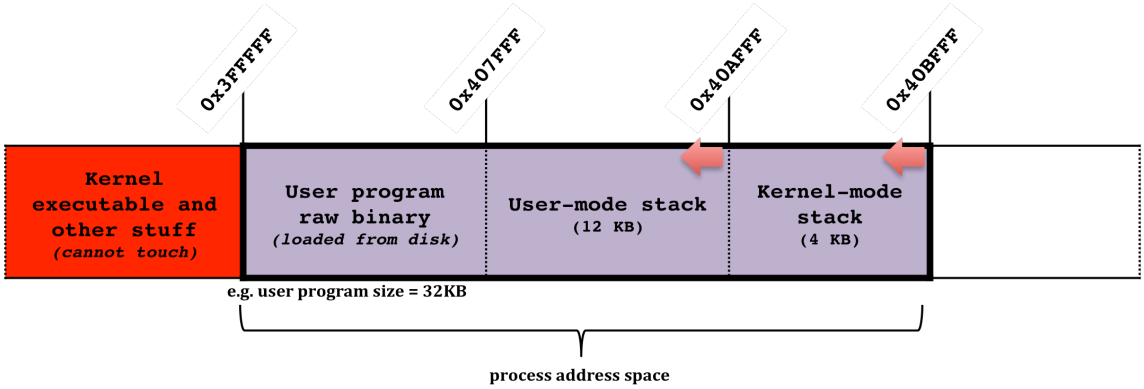
This command allocates memory for the program, transfers the executable from the disk to memory, saves the console's state, sets up a process control block, and then switches control to the user program. Lets look at these steps one by one.

The DUMB Memory Manager

The function of a memory manager is to keep track of occupied memory, and tell where memory is available whenever other pieces of code request for it. Things can get complex here. In SOS1, we will have a rather dumb memory manager. This manager does not really track occupied memory, but instead always says that all of physical memory beginning at 4MB is available for use. `alloc_memory` in `memman.c` implements this, and always returns 0x400000 as the beginning address of the allocated memory. It does check to see that we are not requesting more than what is there : `(total_memory – 4MB)`.

Program Address Space

A program needs, at the very least, a stack to be able to execute. Where will this stack be in memory? Lets design how our program will be laid out in memory.



The `run` command will begin by requesting the (DUMB) memory manager an area of memory large enough to hold the program executable and 16KB more. As a result, `alloc_memory` will return the address 0x400000. So, the user program's executable will always begin from memory address 0x400000. Immediately following the executable, we will use 12KB as the stack space for the user program (also called *user-mode stack*). Note that stacks grow downwards, so the stack actually begins at the end of this 12KB. Immediately following the user-mode stack space, we will use 4KB for another kind of stack, the *kernel-mode stack*. The CPU will use this special stack during system calls; more on this later.

Process Control Block

When a system call is made, the handler needs to save the current CPU state. When the handler is done, it needs to revert back to the CPU state it was in before the interrupt. Where will the CPU state be stored? We need a **process control block (PCB)** for each process – the console process and the user program process. The PCB structure (`PCB struct`) is defined in `kernel_only.h`. Currently, it has variables for the CPU registers, and two other 32-bit variables, `memory_base` and `memory_limit`. `memory_base` will hold the start address of the process address space, and `memory_limit` will be such that `(memory_base + memory_limit)` is the last address of the process address space.

```
typedef struct process_control_block {
         struct {
               uint32_t ss;
               uint32_t cs;
               uint32_t esp;
               uint32_t ebp;
               uint32_t eip;
               uint32_t eflags;
               uint32_t eax;
               uint32_t ebx;
               uint32_t ecx;
               uint32_t edx;
               uint32_t esi;
               uint32_t edi;
         } cpu;

         uint32_t memory_base;
         uint32_t memory_limit;

} __attribute__ ((packed)) PCB;
```

We have defined two global variables – `PCB console` and `PCB user_program` – in `runprogram.c`. In addition, there is also a PCB pointer – `PCB *current_process` – which we will point to `user_program` before switching to it. The system call handler will always save the CPU state in the PCB pointed to by `current_process`. When reloading the state, we will do so from either `console` or `user_program`, depending on which process we are switching to. The keyboard interrupt handler uses the stack to save the CPU state.

Switching to a User Program

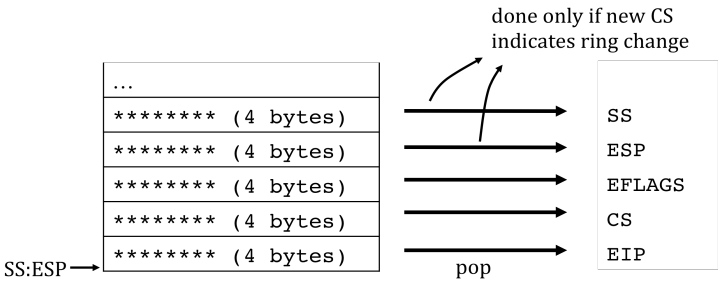
We are almost there. After loading the user program executable from the disk to memory, here is how the `run` command proceeds.

First we need to put values into `console.cpu.esp`, `console.cpu.ebp`, `console.cpu.eflags` and `console.cpu.eip` so that when we want to resume the console (as a result of INT 0xFF), we resume from the point where we switched to the user program. This is what makes SOS1 single tasking – the console does not get control until the user program has issued INT 0xFF. We can simple put the current values of ESP, EBP and EFLAGS into their respective locations in the `console` PCB. As for the EIP, we will put the memory address following the `switch_to_user_process` function call (see `runprogram.c`).

Next, its time to fill in some values into the `user_program` PCB, namely, the `memory_base`, `memory_limit`, `cpu.ss` (stack segment selector; should refer to GDT entry 4), `cpu.esp` (beginning address of stack relative to the segment base), `cpu.cs` (code segment selector; should refer to GDT entry 3), `cpu.eip` (beginning address of instructions relative to the segment base), and `cpu.eflags`. Segment selectors must be formed so that they refer to the correct GDT entry, and have the correct RPL (=3).

With the user program PCB initialized, we call `switch_to_user_process` to do the actual switch. Remember that we are yet to initialize entry 3 and entry 4 of the GDT. The contents of these entries will be used when the user program is running. For example, when the user program is running, and say EIP is 0x10, we are actually referring to memory location (segment base in entry 3 + 0x10). So, we will fill the two entries now. See `kernel_only.h` for a description of the `GDT_DESCRIPTOR struct` and accordingly fill in the values in `gdt[3]` and `gdt[4]`. For example, the segment base and segment limit of these entries should be same as `memory_base` and `memory_limit` in the user program's PCB. Next, we will load the CPU state from the PCB into the actual registers, but hold off on the SS, ESP, CS, EIP and EFLAGS registers.

So how will the actual switch to ring 3 happen? The answer is the IRETL instruction. When this instruction is executed, the CPU performs the following steps.



1. Pop 32 bits from the stack and place it in EIP
2. Pop 32 bits from the stack and place the lower 16 bits in CS
3. Pop 32 bits from the stack and place it in EFLAGS
4. If lower 2 bits of new CS indicates that a ring change will occur, then
    a. Pop 32 bits from the stack and place it in ESP
    b. Pop 32 bits from the stack and place the lower 16 bits in SS

So, in order to switch to ring 3 and start running the user program, we only need to make sure that the values we want to put in the SS, ESP, EFLAGS, CS, and EIP registers are pushed into the stack (in that order) before issuing IRETL. With the right values in the stack, IRETL will cause a ring switch, bring alive our user program and its instructions will start running. Voila! SOS1 can now run programs. Notice that IRETL is also used to resume a process in the keyboard interrupt handler. A part of the `run` command and most of the `switch_to_user_process` function is the next assignment.

Kernel-Mode Stack

We never talked about those extra four kilobytes, the kernel-mode stack. Lets do. When the user program issues a system call, the CPU will come back to ring 0 and execute the system call handler. The handler is just like any other function; it also needs a stack to work. Will it use the same stack that the user program was working with? For security reasons, no! The CPU actually wants us to switch to a different stack. And before we switch from ring 0 to ring 3, we must tell the CPU where this stack is. In fact, modern CPUs can automatically save state (called hardware task switching) when a process switch happens, but operating systems prefer doing it via software instructions. The `TSS_STRUCTURE struct` in `kernel_only.h` is a data type to hold the CPU state. Since we are not using hardware task switching, most of the variables there are irrelevant, except for `esp0` and `ss0`.

Lets see how we specify the kernel-mode stack. First, we define the variable `TSS_STRUCTURE TSS` in `systemcalls.c`. Next we create a sixth GDT entry that describes the memory area that has this variable (see `setup_TSS` in `systemscalls.c`), and then use the LTR instruction to tell the CPU where the `TSS_STRUCTURE` variable is in memory. Whenever the INT instruction is executed, the CPU

will load `TSS.esp0` to ESP and `TSS.ss0` to SS, effectively performing a switch from the user-mode stack to the kernel-mode stack. Thereafter, it will push the values in SS, ESP, EFLAGS, CS, and EIP to the kernel-mode stack, and then run the handler (which will continue to use the kernel-mode stack). Therefore, `switch_to_user_process` should put the right values in `TSS.esp0` and `TSS.ss0` before doing the switch; otherwise the user program will crash when it does a system call.