Understanding a Simple Operating System

SOS is a Simple Operating System designed for the 32-bit x86 architecture. Its purpose is to understand basic concepts of operating system design. These notes are meant to help you recall the class discussions.

Registers in the IA-32 x86 Architecture

As an operating system designer, we must be aware of the registers available in the CPU. The IA-32 x86 architecture first appeared with the 80386 processors. A CPU that complies with this architecture contains the following set of registers.

1. **General purpose registers:** EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP, all 32 bits (4 bytes) in size. Out of these, ESP is used to point to the current location of the stack (within the SS segment).
2. **Segment registers:** CS, DS, SS, ES, FS and GS, all 16 bits in size. These registers are used in protected mode addressing, which assumes that memory is divided into segments. The CS (code segment) register is used to refer to the segment that contains executable code. The SS (stack segment) register is used to refer to the segment that contains the stack.
3. **Instruction pointer:** EIP, 32 bits in size. In protected mode, the CPU uses this register, in conjunction with the CS register, to determine the location of the next instruction to execute.
4. **Program status register:** EFLAGS, 32 bits in size. The bits of this register indicate different status/control states of the CPU.

These are the basic registers available for program execution. In addition, multiple other registers exist to control and monitor various operations of the CPU. For example, **control registers** CR0, CR1, CR2, CR3 and CR4 determine the operating mode of the CPU. **Memory management registers** GDTR, IDTR, LDTR and the Task register hold memory locations of data structures necessary for protected mode operation. Chapter 3, Vol. 1 of the Intel Developer Manual has more details on these and other register sets.

BIOS (Basic Input/Ouput System) Routines

Basic input/output services are provided by the BIOS at startup. In order to use such a service, we will have to know the service type (given by a number, and also called a **BIOS interrupt number**) and a service number. All of these are standardized. For example, BIOS interrupt number 0x10 refers to services related to the display, and the service number 0x0E within this type refers to the service that allows printing characters on the display. When using BIOS services, the service number has to be loaded in the AH register, and the service is invoked by using the INT instruction with the BIOS interrupt number as the operand.

```
MOV $0x0e, %ah
…
INT $0x10
```

However, depending on the service, other registers may also have to be set up before the INT instruction. This set up is required to tell the BIOS about any pertinent parameters related to the service. For example, when printing to the display, one has to specify what character to display. This is done by loading the AL register with the ASCII code of the character, before the INT instruction. For services that return data (e.g. read disk sectors), we will have to know where the BIOS places the return data; it could be in a register, or pre-defined locations in memory. Read the Wiki page on "BIOS interrupt call" to learn about different BIOS interrupt numbers and the services available in them. Few BIOS interrupts that we will use here – 0x10 (video services), 0x13 (disk services) and 0x15 (miscellaneous services).

Real Mode Addressing

Old 8086 and 80186 processors had a 20-bit address bus (an address bus is used by the CPU to specify which memory location it wants to read/write). Therefore, a maximum of 2^{20} bytes (1 MB) could be accessed by the CPU. However, the CPU had only 16-bit registers. Since it was not possible to store a 20-bit number in a 16-bit register, the CPU designers introduced the **real mode addressing** scheme. An address is to be specified using two 16-bit numbers – one called the *segment* number and the other called the *offset*. The format is segment:offset. An address such as 0xF000:0xFFFF will be converted to a 20-bit

address by shifting the segment number 4 bits to the left and then adding the offset part to the resulting number. Therefore, 0xF000:0xFFFF would become 0xFFFFF.

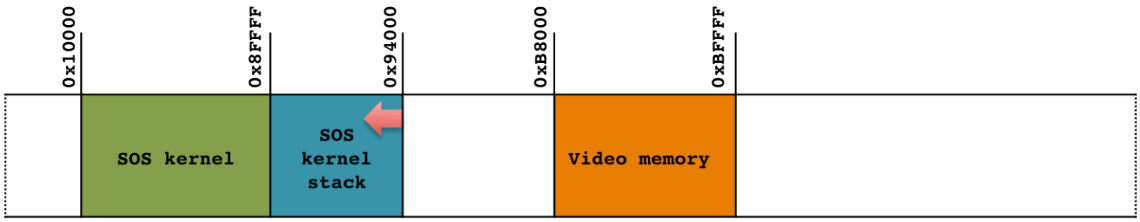
Note that a real mode address such as 0xF800:0x7FFF will also convert to 0xFFFFF. So, the same memory location can be addressed using different segment-offset combinations in real mode. What will 0xF800:0x8000 convert to? Answer: 0x00000. Applying the conversion method to 0xF800:0x8000 actually results in 0x100000; but the CPU only keeps the lower 20 bits, hence only 0x00000 remains. This introduces a wrap-around behavior in memory addressing, and was in fact exploited by some programs in the old days. For the sake of backward compatibility, all modern CPUs start with real mode addressing.

Organization of SOS on Disk and Memory

Before heading forward, we need to decide how the memory and the disk will be laid out in a SOS system. Say we decided on the following disk layout on a 128 MB disk.

Sector 0 of the disk is the **boot sector**; some special code/data will go here. Remember, when the computer boots from this disk, it will load the contents of this sector into memory and start executing it. We will use sectors 1 through 1024 (512 KB in total) to store the operating system executable (the SOS kernel). Remaining sectors will be used to store user programs and files.

Similarly, we need to decide where in memory will the kernel (and user programs later) be loaded. Lets use the following map for now.



We will load the kernel image (a maximum of 512 KB) from the disk and place it at location 0x10000 through 0x8FFFF (512 KB total) in memory. We will use the memory range 0x90000 to 0x94000 (16 KB) as stack for the kernel. Also, the address range 0xB8000 to 0xBFFFF should not be touched since it is used as video memory (more on this later). This setup is sufficient to get us started. But, the question is how will the kernel image move from the disk to memory? That's where the special code in the boot sector comes into play. Since, the CPU is designed to copy sector 0 of the disk to memory, and then execute whatever is there, all we need to do is put some code in sector 0 that reads sectors 1 through 1024 from the disk, loads it to memory at our designated location, and starts executing it (essentially starting up the SOS kernel). Sector 0 is sometimes also called the **MBR (Master Boot Record)**.

Master Boot Record

The MBR is 512 bytes in size. However, the BIOS does not consider any 512 bytes in sector 0 of a disk as a boot record. There are some guidelines we need to follow regarding how these 512 bytes are to be organized. There are many standards; we will use the simplest of all of those – the classical MBR format (see more formats in Wikipedia "Master Boot Record").

The most important part for a sector 0 to be considered a boot sector by the BIOS is the boot signature. The boot signature is the number 0x55 stored in the 511th byte of the sector, followed by the number 0xAA stored in the 512th byte. Besides this, a classical MBR has the following structure.

Byte	Content	Size
0 to 445	Code	446 bytes
446 to 461	Partition table entry 1	16 bytes
462 to 477	Partition table entry 2	16 bytes
478 to 493	Partition table entry 3	16 bytes
494 to 509	Partition table entry 4	16 bytes
510	0x55	1 byte
511	0xAA	1 byte

Therefore, we have a maximum of 446 bytes to write the code necessary to load the SOS kernel from the disk to memory. Partition table entries are used if the disk is divided into partitions. We do not really need to put any meaningful data in them for SOS to work; but we will put some data in entry 1 for the sake of demonstration. A partition table entry is 16 bytes. See the Wikipedia article on "Master Boot Record" to learn what goes in these 16 bytes. The first byte of these 16 bytes should be 0x80 to indicate that the partition is bootable (contains an OS). The fifth byte contains a number indicating which type of file system is present in the partition. All commercial operating systems have an associated number. We will use the number 0x10 to mean SOS, although this number is already is use! We will ignore the other bytes. Therefore, the 512 bytes of sector 0 of the disk will look like the following.

446 bytes of code to load the SOS kernel from disk to memory															
														80	00
00	00	**10**	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00	00	00	**55**	**AA**

Our first programming task is to write a program (in assembly) which when compiled produces 512 bytes of output that resembles the above picture (see `MBR.S` and then `build/MBR.bin` using the gHex utility).

SOS Startup

So far so good. We have managed to load the SOS kernel executable from the disk to memory, and can start executing it. But, what is the kernel executable? Well, it could be any program. For us, it will be the SOS operating system. We will build it progressively, adding more and more features as we go. The kernel executable will begin by switching to protected mode. In SOS0, the entry point of our SOS kernel is in the `startup.S` file. It begins by obtaining the total amount of memory installed in the system and storing it in a memory location that we have named `total_memory`.

A20 Line

Unlike the 8086 processor, the 80286 processor had a 24-bit address bus, and then 80386+ processors had 32-bit or 64-bit address buses. A 32-bit address bus allows us to address up to 2^{32} bytes (4 GB) of memory. As a result of this advancement, wrapping of memory addresses as in a 8086 processor will no longer happen. This may cause problems for some programs that relied on the wrapping behavior. To prevent this from happening, the BIOS always starts the CPU with the 21st line of the address bus (also called the A20 line) disabled. We need to enable the A20 line so that we can use more than just 1 MB of

memory. There are multiple ways of enabling the A20 line; we use the method where we need to send some special commands to the onboard keyboard controller (see OSDev Wiki "A20 Line"). Why isn't there an enable method for A21 – A31 lines?

32-bit Protected Mode Addressing

Intel 80386 introduced the 32-bit **protected mode**. In this mode, the CPU is designed to check for memory access violations, i.e. whether a process is using its part of the memory or trying to access memory locations in restricted areas. Protected mode also requires a different form of addressing. The design involved first separating memory into *segments*, and then specifying where these segments begin and end in a table called the **Global Descriptor Table** (GDT). A memory address is then made up of two numbers – a 16-bit segment selector number (stored in one of the segment registers: CS, DS, ES, FS, GS, or SS), and a 32-bit number for the offset (EIP and ESP are typically used here). The top 13 bits of the 16-bit segment selector tells which entry in the GDT do we want to use. The lowest two bits (bits 0 and 1) specify what is known as the **Requested Privilege Level (RPL)**, which we will discuss a little later. Therefore, an address such as 0x0010:0x100 means we want to access the third (top 13 bits of 0x0010 are 0b0000000000010 = 2) segment as described in the GDT, and in that segment we want the 256th (0x100) byte. So if segment 3 begins from, say, location 0x1000, then we will be referring to memory location 0x1000 + 0x0100 = 0x1100 (the 4353rd byte).

In protected mode, the CPU fetches the next instruction to execute from the CS:EIP address. Therefore, what is stored in CS is important. However, we cannot simply move a value into the CS register. Similarly, the CPU uses SS:ESP as the location of the stack.

Privilege Level

A privilege level (PL) indicates what ring(s) does the CPU need to be in to access a certain resource. If a resource is set to be at PL 0, then it can be accessed only when the CPU is in ring 0 (kernel mode). If PL is 3, then the resource can be used whenever the CPU is in ring 3 (user mode) or lower. Two bits are used to specify a privilege level, with PL = 0 (00 in binary) and PL = 3 (11 in binary) being the most heavily used. The current ring (called the **Current Privilege Level** or **CPL**) is always equal to that given by bits 0 and 1 of the value stored in the CS register. As we will see later, a privilege level appears in other places as well, and it is important to understand how they interplay.

Global Descriptor Table (GDT)

Lets take a deeper look at the GDT. A GDT is a table. Each entry in the table is 64 bits long. Remember that the purpose of one entry is to describe one memory segment (as to be used in protected mode addressing). We will need one GDT entry for each memory segment. The 64 bits of a GDT entry are used in the following manner.

- 32 bits to specify the flat memory address where the segment begins (also called the *segment base*)
- 20 bits to specify the size of the segment (also called the *segment limit*)
- 8 bits (1 byte) to specify properties of the segment – the PL required to access the segment (called the **Descriptor Privilege Level** or **DPL**), is there code or data in the segment? is the segment writable? etc.
- 4 bits to specify additional properties, such as whether the segment size is in 1 byte or 4 KB units.

See `kernel_only.h` (the `GDT_DESCRIPTOR struct`) for a detailed description of a GDT entry. Note that the above information is put into an entry in a rather convoluted manner! For example, the 20 bits of the segment limit is put in bits 0 to 15 (16 bits), and then bits 48 to 51 (4 bits) of the 64 bit entry. To begin, our GDT will have three entries and look like this.

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

Here entry 1 is describing a segment which spans 4 GB of memory, starting at location 0, is used to store instructions, and can only be accessed if the CPU is currently in ring 0. Entry 2 is similar but it is a segment that is used to store data. As we can see, segments may overlap. Entry 1 will be referred to by the CS (Code Segment) register when running kernel code, and entry 2 will be referred to in other segment registers (e.g. the SS register).

Notice that although we have talked about segmenting memory, we have set up one large segment spanning the entire memory. This is fine, since GDT entries 1 and 2 can only be used in ring 0 (by the kernel), and the kernel should be able to access all locations in memory. We will later create more GDT entries for user programs; there we will have to carefully set the base and limit, so that programs cannot access each other's memory locations.

More on Privilege Levels

Privilege levels seem to appear in many places. So, lets list them cleanly and understand how they interact.

1. Descriptor Privilege Level (DPL): This is the privilege level appearing in a GDT entry. It is the privilege level required to access the memory segment described in the GDT entry.

2. Requested Privilege Level (RPL): This is the privilege level we want to switch to. It is part of the 16 bits (bit 0 and bit 1) of the value we load into a segment register. The upper 13 bits of the value states which GDT entry we want to use. An important point to note is that we cannot directly move a value into the CS segment register.

3. Current Privilege Level (CPL): This is the privilege level currently set in the value in the CS register, i.e. bits 0 and 1 of the CS register. Once again, the value in the CS register cannot be changed directly.

Given these three places where a privilege level appear, the CPU allows us to change privilege levels only if **max(CPL,RPL) ≤ DPL**. Hence, a user program (running in ring 3) cannot directly change to kernel mode (ring 0) – why? For example, entry 1 of the GDT is set such that privilege level is 0. So, DPL of entry 1 = 0. The CPL is 3 when the user program is running. Lets say that the user program wants to access a memory location using the address 0x0008:0x1000. This means that the user program is trying to access byte number 0x1000 in the segment described by GDT entry 1 (higher 13 bits of 0x0008 evaluate to 1). Also, it wants to access the location as a ring 0 program (bits 0 and 1 of 0x0008 evaluate to 0, i.e. RPL = 0). However, since max(CPL=3, RPL=0) = 3 is not less than or equal to DPL (=0), the CPU will not allow the access and cause a fault. You can test other possible combinations and see when privilege level changes are allowed.

This is a simplified explanation. Refer to sections 5-5 to 5-8 of the developer's manual for an extended explanation.

The GDT Setup

The GDT can be placed anywhere in memory. But, how will the CPU know where it is? The answer is a special instruction, **LGDT**, which we will use to inform the CPU of the location of the GDT in memory. The LGDT instruction has one operand – a memory address. When the LGDT instruction is executed, the CPU reads 6 bytes from the memory address. The top 4 bytes signify the memory address where the GDT starts in memory, and the remaining 2 bytes signify the size (in bytes) of the GDT. The CPU notes these values in a special register called the GDTR, and uses it whenever it needs to look up the GDT. Note that the GDT must be set up before protected mode is enabled.

Enabling Protected Mode

With the GDT set up, we are now ready to enable protected mode. This is done by enabling a bit in a special register called **CR0** (control register zero). CR0 is a 32-bit register used to enable/disable different features of the CPU. For example, if bit 0 of this register is set (made 1), then the CPU goes to protected mode. Similarly, if bit 2 is set, the CPU assumes that no hardware to do floating-point operations exists in the system. See Wikipedia "Control register" for details on other bits.

Here is a dilemma. As soon as protected mode is enabled, the CPU will start using CS:EIP as the address for the next instruction. So, the value in CS has to be a valid segment selector – the number 0x0008, referring to GDT entry 1 and RPL = 0. Unfortunately, neither is this number guaranteed to be in CS, nor can we simply use a MOV instruction to do so! The solution is to use the LJMP (long jump) instruction, which allows us to specify a segment selector and an offset as its two operands. With an instruction such as

`LJMP $0x0008,` <address of the instruction immediately following this instruction>

we force the CPU to load the CS register with 0x0008.

Calling main()

After protected mode is enabled, we first set up the stack for the kernel to use –

1. load the data/stack segment registers (ES, DS, FS, GS, and SS) with 0x10 (GDT entry 2, RPL=0), and
2. load the stack pointers (ESP and EBP) with 0x94000 (see SOS layout in memory).

And then we call the main() function (in `main.c`), which will do a few more initializations and then start up the command prompt. We can also do most of the programming in C now.

In main()

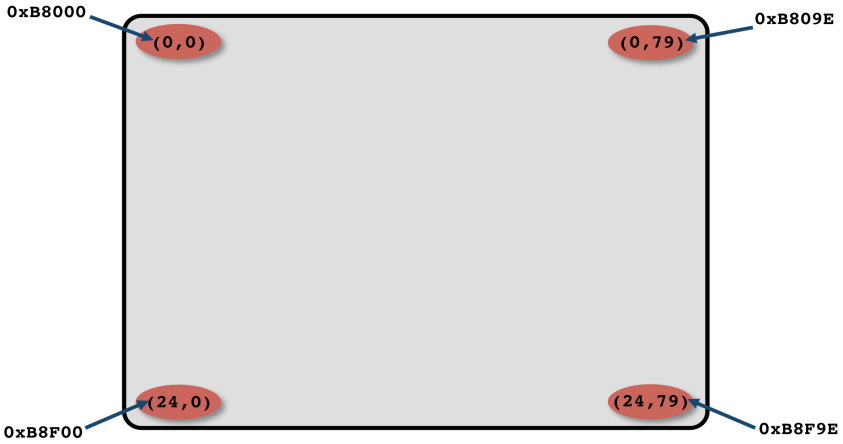
The `main` function in `main.c` begins by finishing the remaining initializations, and then starting the console. Lets look at some of these initializations.

Disk Initialization

The disk initialization routine `init_disk` is in `disk.c`. The function queries the disk to determine the size of the hard drive. This is done using some port I/O. We will postpone discussion on this until we have discussed I/O subsystems (towards the end of the quarter). At this point, the function of importance is the `read_disk` function, which allows us to read a given number of sectors from the disk, provided a sector (LBA) number to start the read from, and a buffer to place the read data.

Display Initialization

The display initialization routine `init_display` is in `display.c`. The memory range 0xA0000 to 0xBFFFF is used to map the display to memory. In other words, writing to these addresses is equivalent to writing to the display. A Color Graphics Adapter (CGA) system uses the 32KB range from 0xB8000 to 0xBFFFF. In an 80 columns by 25 rows text mode display, each location on the display is represented by 2 bytes. Address mapping happens row wise.



The two bytes corresponding to each location stores the foreground and background color (in the first byte), followed by the 8-bit ASCII code of the character to be displayed (in the second byte). The higher 4 bits of the first byte signify the background color and the lower 4 bits signify the foreground color. For example, if we write 0x1F to memory address 0xB8000 and 0x53 to address 0xB8001, we would be writing S to the top-left corner of the screen.

`display.c` defines the global variables `cursor_x` and `cursor_y` to keep track of the current cursor position, and a color variable to store the current color setting. Macros for the colors usable in a 16-color text mode are given in `kernel_only.h`. The current cursor position (the position where the blinking line is displayed) is necessary so that we can keep track of the position where the next character should be displayed (when the user presses a key).

The display initialization function draws an ASCII art of the word 'SOS' using a white-on-blue color, and displays the total memory in the system and the disk size. The cursor is then set to (0,8) and the color is set to light-grey-on-black. We will always keep this *artwork* visible; so our display screen will actually be from row 8 to row 24.

A number of helper functions are available in `display.c`, including one similar to printf (called `sys_printf`). Later we will have to implement a system call so that users can indirectly use this function to display on the screen. Take a look at the helper functions. The most important helper function is the `display_character` function, which writes a character to the appropriate location in video memory (depending on the current cursor position), increments the cursor position, and then displays the cursor at the new position.

Setting Up Interrupt Handlers

In protected mode, the CPU has 256 interrupts that we can program. Recall that when an interrupt occurs, the CPU transfers execution to the corresponding interrupt service routine (also called an *interrupt handler*). Therefore, before interrupts can be used, we need to tell the CPU where the handlers are corresponding to the 256 possible interrupts.

Interrupts 0 to 31 are special interrupts, often called *exceptions*. They are fired when certain special events happen; e.g. interrupt zero is fired when an instruction attempts to perform a division by zero. See OSDev Wiki "Exceptions." We can use the remaining interrupts (32 to 255) in whatever way we like – to transfer control to the OS when user programs need some kernel level service (as in a system call), or when certain hardware events (e.g a key press) occur.

The interrupt number is simply an index into a table, the **Interrupt Descriptor Table (IDT)**. An entry in the IDT encodes the location of the function to execute when an interrupt occurs. The `init_interrupts` function in `interrupts.c` sets up this table and tells the CPU about the location of the table.

Interrupt Descriptor Table (IDT)

Much like the GDT, the IDT is also an array of 8-byte (64 bit) entries describing "gateways" (*gates*, in short) to what has to be done when an interrupt occurs. x86 systems allow for 256 such gates; therefore the IDT has 256 8-byte entries. Gates can be interrupt-gates or trap-gates, the difference being that hardware interrupts are automatically disabled when accessing interrupt-gates. There is also a third type called task-gates (not very popular with modern operating systems). The 64 bits of an IDT entry are used in the following manner.

- 48 bits to specify the protected mode address of the function to execute when the interrupt fires; in other words, a 16-bit segment selector and a 32-bit offset
- 8 bits to specify properties of the interrupt routine – a privilege level specifying the ring(s) in which the CPU must be if this interrupt is invoked using the INT instruction, is it an interrupt/trap gate?, etc.
- 8 bits are reserved and are always zero

See `kernel_only.h` (the `IDT_DESCRIPTOR struct`) for a detailed description of an IDT entry. Again, like a GDT entry, the above information is put into an entry in a convoluted manner! See section 6.10 and 6.11 of the developer's manual for an extended discussion on interrupts.

The IDT is stored as an array in our OS – `IDT_DESCRIPTOR IDT[256]`, defined in `interrupts.c`. The contents of this array are filled so that a default interrupt handler runs on any of these interrupts. The interrupts are set as interrupt-gates and the interrupts cannot be manually invoked from user space (privilege level zero). We will change the handler for some of these interrupts, but a little later.

A Default Interrupt Handler

Lets look at how the default interrupt handler works. This handler is defined in `interrupts.c`. The CPU always pushes the current CS, EIP and EFLAGS values on an interrupt (remember these). The CPU

may also push a few other values depending on which ring was active when the interrupt occurred. If the interrupt is set to be an interrupt-gate (in the IDT), then the CPU will also disable further interrupts.

The default handler begins at the location labeled as `handler_default_entry`. The code first disables further interrupts (the hardware does this automatically if the interrupt is set up as an interrupt-gate), pushes the CPU state (all segment and general-purpose registers) into the stack, and then calls the `default_interrupt_handler` function. This function simply prints the message "Unhandled interrupt!" on the screen. After returning from the function, we need to bring the CPU back to the state it was in before the interrupt. First, we pop off the segment and general-purpose registers, then enable interrupts, and then call the IRETL instruction. This is a special instruction that will pop the CS, EIP and EFLAGS values from the stack (remember, they were pushed by the CPU when the interrupt was fired) and load the values into the appropriate registers. As a result, the CPU will revert back to the state it was in before the interrupt, and continue execution.

Remember, we said that we cannot load values directly into the CS and EIP registers. IRETL is one way to do it – push new values for CS, EIP and EFLAGS into the stack, and then use IRETL.

Load IDT

Okay, so we have a default interrupt handler, and we have the IDT array setup so that the handler for all interrupts is the default one. But, how will the CPU know where the IDT is in memory? The answer is the LIDT instruction, which works similar to the LGDT instruction.

The Programmable Interrupt Controller (PIC)

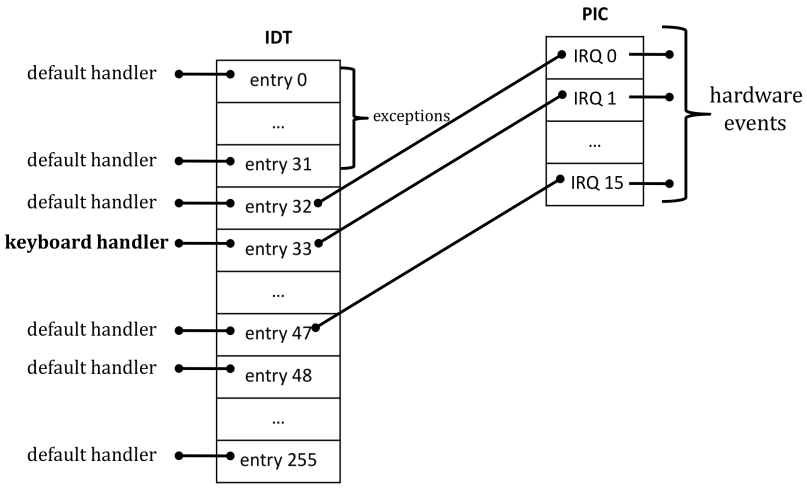
The PIC is a small piece of hardware on the motherboard that is responsible for generating interrupts when certain hardware events occur (see Wikipedia "Intel 8259"). These interrupts are also known as IRQs. The Intel 8259 PIC supports up to 16 IRQs. Some of them are already mapped to certain hardware events by default – e.g. IRQ 1 is fired when a user hits a key on the keyboard (see OSDev Wiki "IRQ"). Our responsibility is to attach them to interrupt numbers (so that an interrupt handler runs when an IRQ fires), and also set up appropriate handlers for those interrupt numbers.

The `setup_PIC` function in `interrupts.c` does this. It initializes the PIC, sets it up so that IRQ numbers 0 to 7 result in interrupt numbers 32 to 39, and IRQ numbers 8 to 15 result in interrupt numbers 40 to 47. After that, it masks (disables) all IRQ lines except for IRQ 1 (the keyboard IRQ). All of this is done using some port I/O, which we will talk about later.

The Keyboard Interrupt Handler

So, IRQ 1 is fired when the user presses a key on the keyboard. And, IRQ 1 will result in firing interrupt 33. At this point, this will not result in anything interesting since the default interrupt handler is set up to run on all interrupts. Lets change this. The keyboard interrupt handler is written in `keyboard.c` and the `init_keyboard` function changes IDT entry 33 so that the keyboard handler is run when a key is pressed on the keyboard. The handler begins similar to the default one, but performs a series of involved operations that read the key pressed and then stores it in the `current_key` variable. The actions here will make more sense after we discuss port I/O. The important function here is the `sys_getc` function, which is used to retrieve user keystrokes. Later, we will implement a system call so that user programs can also obtain keyboard input.

Here is what the global interrupt mapping looks like at this point.



Starting the Console

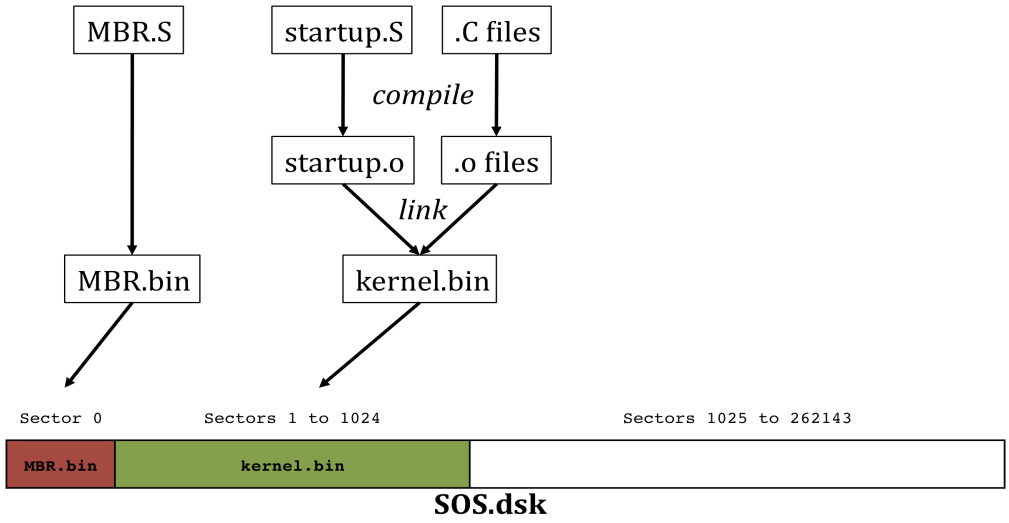
With all of these initializations done, we are now ready to start the console. The console is part of the kernel, so it will run in kernel mode. Therefore, it can print to the display and read keystrokes without any system calls. The infinite loop of the console is in `start_console` in `console.c`. The loop body reads a command, runs it if its valid, and then repeats. This is straightforward C code. The only command that does something at this point is `diskdump`, which we can use to print the contents of one or more sectors of the hard drive (using the `read_disk` function).

There is another file called `io.c` that we have not talked about. The functions in this file allow us to talk to hardware using port I/O, a topic that we will discuss later.

Putting It All Together

Okay! So, we have a bunch of assembly files (`MBR.S` and `startup.S`) and then a bunch of C files. We understand how control begins at `MBR.S`, then transfers to `startup.S`, and then into the C files. But recall that `MBR.S` supposedly loads one big kernel executable from the disk to memory! Where is that kernel executable? Well, we will have to compile `startup.S` and the C files, and create that kernel executable. The `create` script does this for us. Essentially, it compiles each file individually (see the *Makefile* if you are conversant with it) to create the machine level instructions, and then combines them together into one big executable called `kernel.bin`. An important point here is that when we combine the individual machine instruction files, the one coming from `startup.S` must be the first one. This is because the code in `MBR.S` will transfer control to the beginning of the kernel executable, and we want the code in `startup.S` to run before anything else.

Note that `MBR.S` is not part of `kernel.bin`. That is because the MBR is not part of the OS kernel. It is simply 512 bytes of code/data that needs to sit in the first sector of our disk. For testing, we will use a 128 MB file instead of an actual hard drive. This file will be set up just like the way SOS is supposed to sit on a disk – the first 512 bytes will be the MBR code, the kernel executable will be in the next 1024 x 512 bytes, and the rest of the file will be all zeros. The `create` script does this for us. It first creates a 128 MB file called `SOS.dsk`, fills it up with all zeros, copies the compiled code of `MBR.S` into the first 512 bytes, and then copies `kernel.bin` (our SOS kernel executable) immediately after it.



When we tell VirtualBox to use this file as the disk, things roll into action. The code in `MBR.bin` runs first, which loads `kernel.bin` into memory and starts running it. As a result, all initializations are performed, and the console comes alive. Now we are ready to implement deeper OS concepts.

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

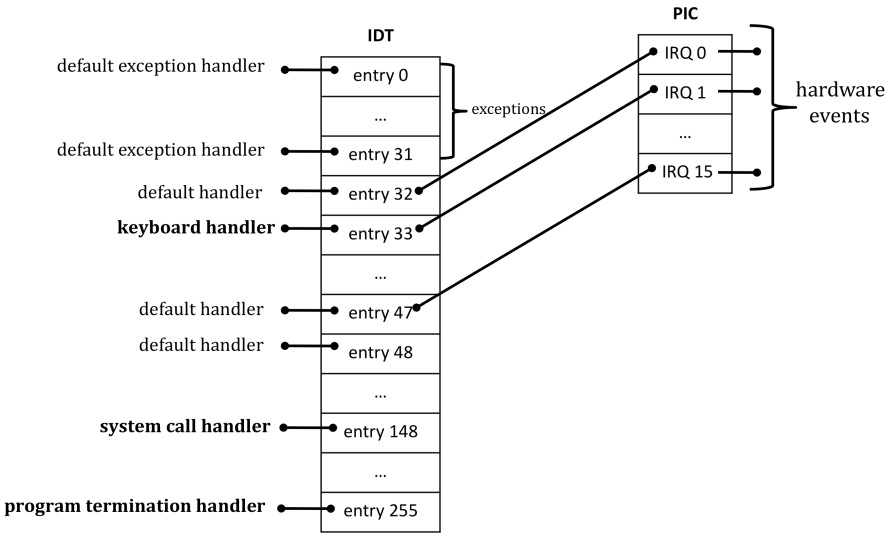
Our GDT at this point contains three entries: the null descriptor (entry 0), and two segments that span all available memory (entries 1 and 2). Entry 1 is setup with DPL = 0 (kernel access only), and will be used whenever the kernel runs its code. Entry 2 is setup with DPL = 0 and will be used whenever 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?

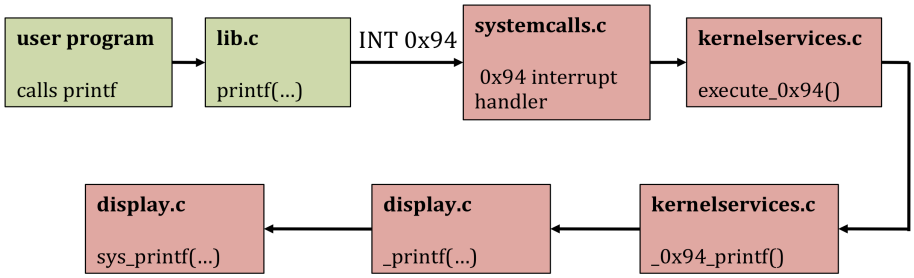
Executable Format

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).

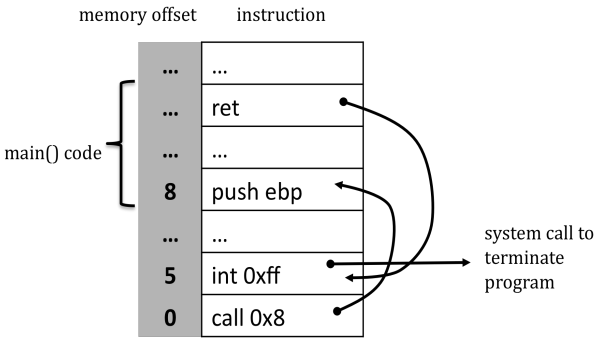
Creating User Programs

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
```

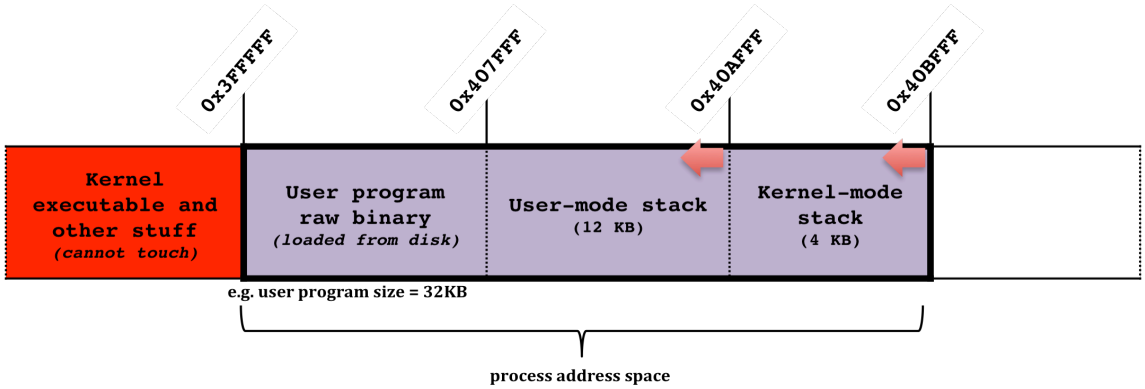
This command allocates memory for the program, transfers the executable from the disk to memory, saves the console's state, sets up a process control block, and then switches control to the user program. Lets look at these steps one by one.

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

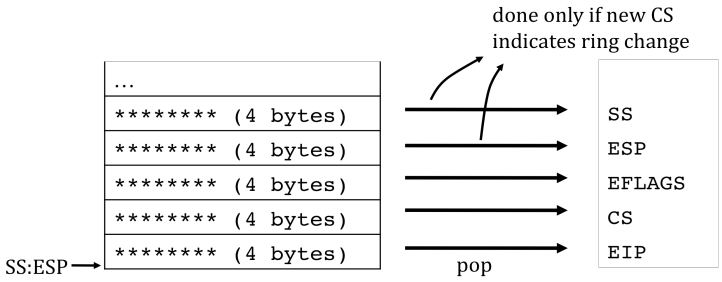
We are almost there. After loading the user program executable from the disk to memory, here is how the `run` command proceeds.

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

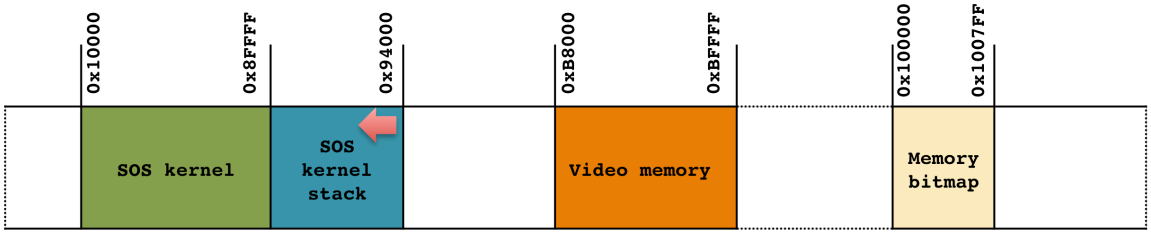
will load `TSS.esp0` to ESP and `TSS.ss0` to SS, effectively performing a switch from the user-mode stack to the kernel-mode stack. Thereafter, it will push the values in SS, ESP, EFLAGS, CS, and EIP to the kernel-mode stack, and then run the handler (which will continue to use the kernel-mode stack). 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.

Running Multiple User Programs (SOS2)

SOS1 is capable of running user programs. While a program is running, SOS1 is blocked, except when handling system calls. The control is transferred back to the program after the system call, or an interrupt, is handled. SOS1 resumes only when the program terminates (INT 0xFF). This model makes SOS1 a single-tasking system. In this part, we will extend SOS1 so that it can run multiple user programs by switching between them periodically.

NAÏVE Memory Manager

Unlike SOS1, SOS2 has a functional memory manager. SOS2's NAÏVE memory manager is implemented in `memman.c`. It views memory in terms of 4KB frames. It keeps track of which frames are free and which are in use through a bitmap vector (`mem_bitmap`). We will put this memory bitmap at memory address 0x100000 (frame 256). SOS supports up to a maximum of 64MB RAM. So, 2KB (0x100000 to 0x1007FF) is sufficient for the bitmap vector. The NAÏVE memory manager does not handle allocation of the first 4MB of memory, and considers it as being unavailable (first 128 bytes of the memory bitmap is set to 0x00).



Memory is requested using the `alloc_memory` function. The number of bytes requested is converted into a whole number of frames. The memory manager then finds the first frame where that many number of frames are contiguously available. This is also called the *first-fit* approach. See the memory manager code to understand how it is implemented.

The NAÏVE memory manager also implements a `dealloc_memory` function. This can be used to return memory back to the manager. During a call to `alloc_memory`, the memory manager stores the number of allocated frames in the first 4 bytes of the first allocated frame. The pointer returned by `alloc_memory` is offset by 4 bytes from the beginning of the first allocated frame. So, whenever `dealloc_memory` is called, the memory manager simply looks at the 4 bytes preceding the pointer, and determines how many frames are being returned.

Programmable Interval Timer (PIT)

Remember the PIC. It is a small piece of hardware on the motherboard that is responsible for generating interrupts when certain hardware events occur (see Wikipedia "Intel 8259"). These interrupts are also known as IRQs. For example, IRQ1 is generated when a keyboard event occurs. We mapped IRQ1 to interrupt 33, and installed a keyboard handler for IDT entry 33.

Similar to the PIC, there is another small piece of hardware on the motherboard called the programmable interval timer (PIT). The Intel 8253 PIT is simply a counter that can be programmed to generate an output signal when it counts down to zero (see Wikipedia "Intel 8253"). The 8253 PIT operates at a theoretical frequency of ~1193182 Hz, i.e. ~1193182 output signals per second. We will set up the 8253 PIT to work at a frequency of ~100 Hz, i.e. generate 100 signals per second, or a signal every 10 milliseconds. This set up is done in the `init_time` function in `timer.c`. The `main` function calls this function before starting the console. Again, the configuration is done using some port I/O. The question is what happens when the output signal is generated? By default, the PIT output signal is connected to IRQ0 of the PIC; and, we have mapped IRQ0 to fire interrupt 32. Therefore, the interrupt handler registered for IDT entry

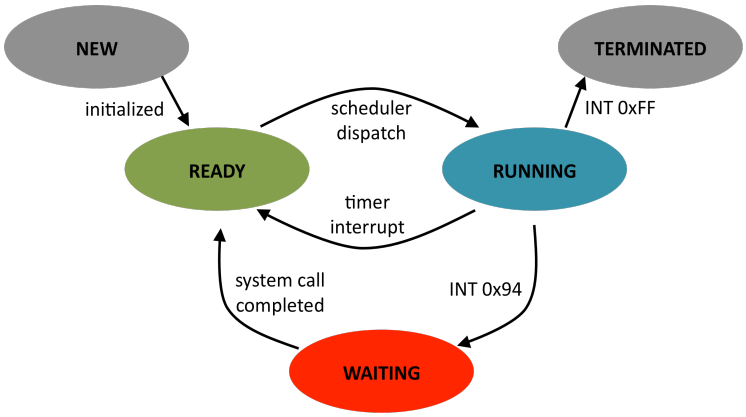
32 will execute once every 10 milliseconds. Of course, we have our default handler sitting there, so our display will simply be flooded with the "Unhandled interrupt!" message. What we need is a timer interrupt handler! Before that, lets introduce process states.

Process States

While multiple user programs may be active in SOS, only one of them can actually run on the CPU. SOS will simply switch between the programs to create the illusion that all programs are running. This requires us to introduce the notion of process states, so that we can track what each process is doing. Processes in SOS2 can be in one of five states:

1. **NEW:** process is created
2. **READY:** process is ready to run
3. **RUNNING:** process is running
4. **WAITING:** process is blocked
5. **TERMINATED:** process has finished

A process begins in the **NEW** state, and stays in this state during its initialization. This state will be more important in SOS3. As soon as the process is initialized, it moves to the **READY** state. When it is dispatched to the CPU, it moves to the **RUNNING** state. A **RUNNING** process will move back to **READY** state when a timer interrupt (interrupt 32) occurs. The timer interrupt handler will do this. When a process issues a system call, it is moved to the **WAITING** state. When the process can resume after a system call, it is moved to the **READY** state. Finally, when the process issues INT 0xFF (as its last instruction), it is moved to the **TERMINATED** state.



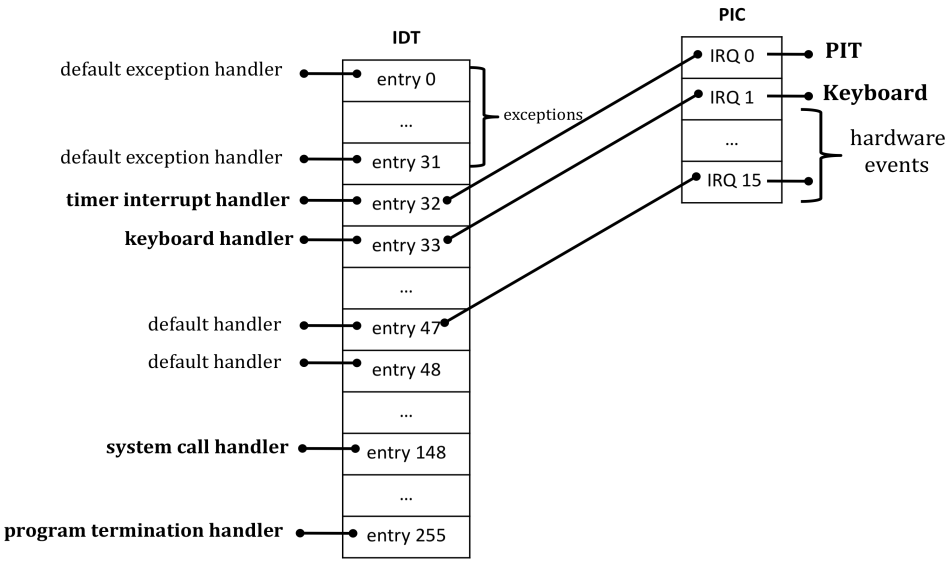
The state of a process is stored in the process' PCB. Therefore, we have added a few more entries to the PCB C struct. The `state` variable will hold the process' current state; `pid` is a process identifier number assigned by SOS; `sleep_end` will be used as an alarm clock to wake the process (coming up); and, `prev_PCB` and `next_PCB` will be used to arrange multiple PCBs into a doubly linked list structure (coming up).

```
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 pid;
        uint32_t memory_base;
        uint32_t memory_limit;
        enum {NEW, READY, RUNNING, WAITING, TERMINATED} state;
        uint32_t sleep_end;
        struct process_control_block *prev_PCB, *next_PCB;
} __attribute__ ((packed)) PCB;
```

Timer Interrupt Handler

We enabled the onboard timer (8253 PIT) and have set it up to generate interrupt 32 every 10 milliseconds. The handler for this interrupt is in `timer.c`, and begins at the `handler_timer_entry` location. We will call every timer interrupt an *epoch*. The handler for the timer interrupt first stores the state of the current process (using the `current_process` pointer) in its PCB, increments an epoch count variable (`elapsed_epoch`), updates the uptime clock on the display, and then invokes the `schedule_something` function in `scheduler.c`. The `scheduler_something` function will implement a scheduler that will decide which process gets to next run on the CPU. Therefore, no matter which user program is running, our scheduler will always get invoked every 10 milliseconds.



Note that the state save in the timer handler is done differently based on whether the current process is the console or not. This is because, when the timer interrupt occurs, the CPU could be running the console, or a user program. Since the console runs in ring 0, the CPU will only push EFLAGS, CS and EIP into the stack before executing the handler; compared to when a user program is running, when the CPU will push SS, ESP, EFLAGS, CS and EIP before running the handler. So the stack will look different in the two cases.

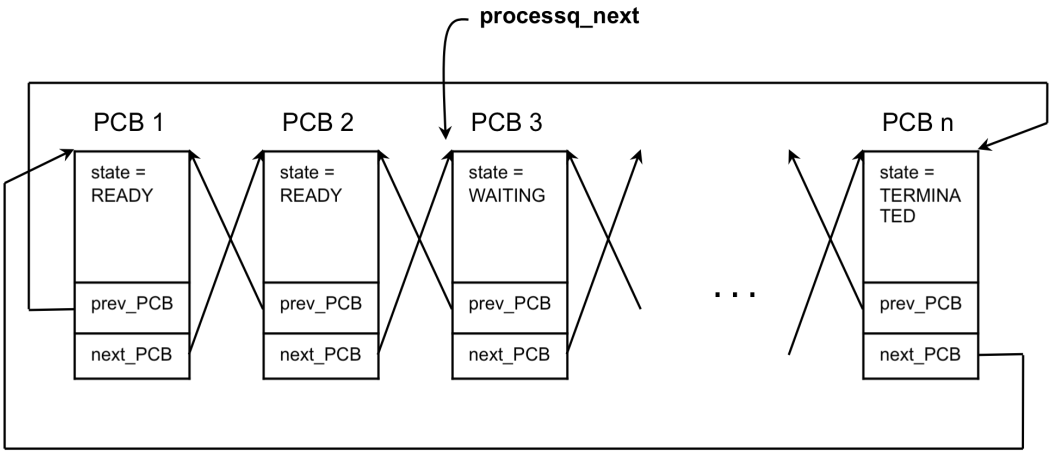
Sleep System Call

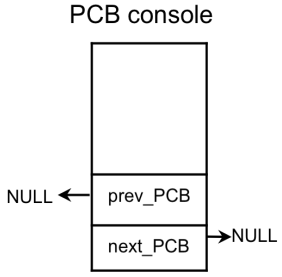
SOS2's system call (INT 0x94) handler has also undergone a change. Unlike SOS1, where transfer goes back to the user program after the system call is serviced, SOS2 calls the scheduler after servicing a system call. The scheduler is left with the decision to whether resume the same process, or pick a different one (we do the latter). The program termination (INT 0xFF) handler and the default exception handler also call the scheduler instead of resuming the console.

SOS2 also has a new system call that user programs can use – `sleep`. Any process calling the `sleep` function (with an argument specifying the number of milliseconds) is moved to the WAITING state. See `kernel_services.c`. The `sleep_end` variable in the process' PCB is set to the epoch when the sleep will finish (the process' state then have to be changed to READY).

The run Command

The run command in SOS2 behaves a little differently than in SOS1. Its syntax is the same, and is implemented in the `run` function in `runprogram.c`. SOS1 had one PCB for the console and one for the user program. We reused the `user_program` PCB since only one user program could be active at a time. However, SOS2 can have multiple processes active, and we do not know how many. Therefore, the run function allocates memory for the PCB of a process using the memory manager. After allocating memory for the PCB, the run function works similar to SOS1 – requests memory to load the program (this time the NAÏVE memory manager makes sure two programs are not given the same memory area), loads the program from disk to memory, sets up the process PCB (with state as NEW and `sleep_end` as 0), and then calls the `add_to_processq` function in `scheduler.c`. As you can see, unlike in SOS1, the run function does not start running the user program immediately, and actually returns back to the console. The program will run when the scheduler picks the process. In other words, SOS2 treats all user programs as background jobs. In fact, we have also disabled the `getc` system call, since background jobs cannot obtain keyboard input. Any user program that attempts to use `getc` will move to the WAITING state forever.





Process Queue

As we saw, the run command does not really run the user program. It simply creates the PCB of the program, loads the program to memory, and calls the `add_to_processq` function in `scheduler.c`. The `add_to_processq` function is what inserts the newly created process in the *process queue*. The process queue in SOS is a circular doubly linked list of the PCBs of all user processes currently in the system. The global variable `PCB *processq_next` should always point to the head of this queue. When a new process is created, `add_to_processq` inserts the process at the tail end of the queue, i.e. before the one pointed to by `processq_next`. We will use the `prev_PCB` and `next_PCB` variables in the PCB structure to create the list. `add_to_processq` changes the state of the process to READY after adding it to the queue. Note that the addition of a process to the queue should not be interrupted; otherwise, we may end up messing up the list. Therefore, interrupts are disabled before modifying the list, and then enabled after the modification is done.

The Scheduler

The SOS2 scheduler is implemented in the `schedule_something` function in `scheduler.c`. This scheduler is a simple round-robin scheduler with 50% share given to the console. Therefore, the scheduler picks READY processes from the process queue one after the other; but every alternate timer interrupt, it runs the console. For example, if there are three READY processes in the queue (say P_1, P_2, and P_5), the order of execution will look like the following.

P_1 → console → P_2 → console → P_5 → console → P_1 → console → P_2 → console → P_5 → console → ...

Note that the scheduler should only choose a process that is in READY state, and not any that is in WAITING or TERMINATED state. It is very possible that the queue has multiple processes, but none of them are in a READY state. In that case, the scheduler has no option but to choose the console again.

Before the scheduler picks a READY process, it cleans up processes that are in the TERMINATED state, and wakes up processes whose `sleep_end` time has reached. Cleaning up of TERMINATED processes involve removing them from the doubly linked list (so that the scheduler do not encounter them again) and freeing the memory used by the process. This is done through the `remove_from_processq` function. Recall that when a program calls the `sleep` system call, the `sleep_end` value in the process' PCB is set to the time epoch when the sleep will end. The system call also sets the process' state to WAITING. The scheduler should change the state of such processes to READY if the current time epoch (can be obtained using the `get_epochs` function in `timer.c`) is larger or equal to the one set in `sleep_end`. This should be done as and when a process in WAITING state is encountered in the queue.

After this cleanup, the scheduler needs to pick a READY process. But, which one? We have processes in a circular list. So, let the scheduler pick the process pointed to by the `processq_next` pointer. Before the scheduler actually switches to this process, it should change `processq_next` to now point to the next process in the list (using the `next_PCB` pointer of the selected process). Otherwise, the same process will get selected every time. Moreover, it should set `current_process` (remember, the system call and timer interrupt handlers use this pointer to save state), and change the state of the selected process to RUNNING. The actual switching is done using the `switch_to_user_process` or `switch_to_kernel_process` functions. Both functions take as argument a PCB – use `switch_to_user_process` to switch over to a user process selected from the process queue, and `switch_to_kernel_process` when switching over to the console process.

The implementation of the `add_to_processq`, `remove_from_processq` and the `schedule_something` functions are left as an exercise. All of these are in `scheduler.c`. There are three pre-defined global variables in `scheduler.c` – `PCB console`, `PCB *current_process`, and `PCB *processq_next`. The source code for parts of SOS1 that was left as an exercise (the `run`

and `switch_to_user_process` functions) is not made visible in SOS2, and provided as a C object file. The `create` script of SOS2 adds in this code so that the functions are available to you for use.

The Complete Picture

When the user starts a program, SOS2 adds it to the process queue. The round-robin scheduler is invoked every 10 milliseconds. As the scheduler picks one process after the other, the programs make progress, creating the impression that all programs are making progress. And we have multi-tasking in SOS!

ps Command

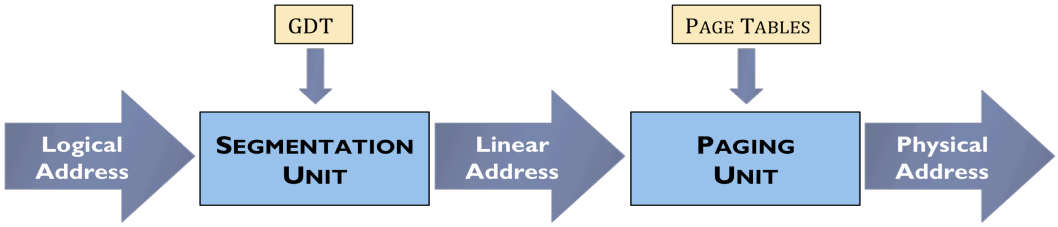
SOS2 has a new console command – `ps`. The command shows a list of active processes (the ones in the process queue), along with their `pid`, `state`, location in memory and size. The implementation of this command is in `console.c`.

Paging (SOS3)

In this section, we will introduce paging in SOS. Paging will allow SOS processes to have a much larger address space than the amount of available physical memory. It will also allow us to store different parts of the process address space in non-contiguous locations in physical memory.

GDT Entries

When paging is enabled, logical addresses in an x86 architecture will go through two translation units. The first is the segmentation unit, and the second is the paging unit.



We have already been using the segmentation unit, which essentially adds the segment base to the (logical) addresses generated in a program. The output of this unit is also called a *linear address*. Till now, the linear addresses have been the same as the physical memory addresses since the paging unit is not enabled. When we turn the paging unit on, a linear address will then be converted to a physical address based on the page tables. This could become complicated and messy! It will be easier if we can somehow disable the segmentation unit. Unfortunately, we cannot do that. Another way around is to set up memory segments that span the entire memory, i.e. all segments have their base at zero (also called flat segmentation). In this way, the linear address is the same as the logical address. This is exactly what we will do in SOS3.

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	`00cffa000000ffff`	Base = 0, Limit = 4 GB, Has code, DPL = 3
GDT entry 4	`00cff2000000ffff`	Base = 0, Limit = 4 GB, Has data, DPL = 3
GDT entry 5	`0000000000000000`	initialized in setup_TSS()

In other words, we will not be setting GDT entries 3 and 4 while switching to user programs. In fact, all programs will use the same segment. The paging unit will take care of the necessary memory protection so that one process cannot touch the memory area of another process. In the remaining discussion, the term “logical address” implies the linear address since they are the same.

Logical Addresses

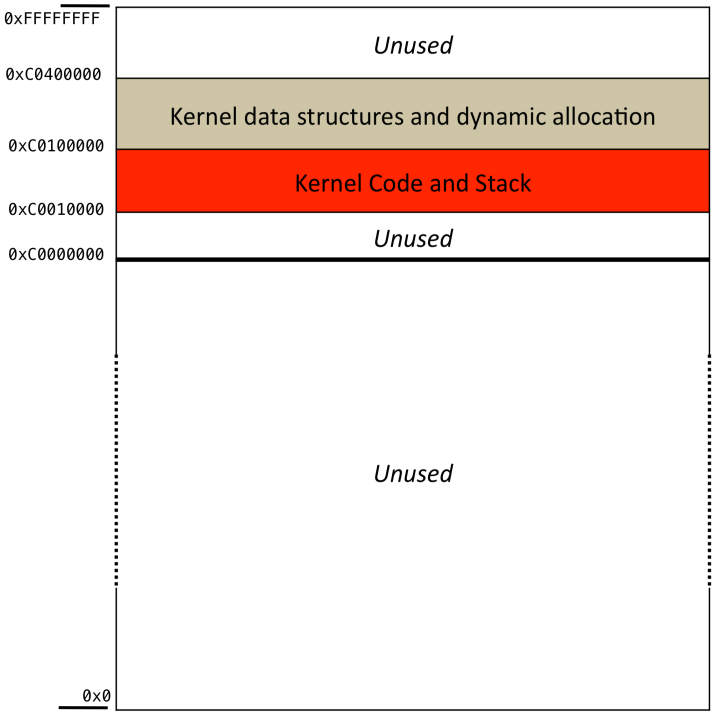
Logical addresses are the addresses generated by programs. Since they are not directly related to physical memory addresses, we can design how a process is organized in the set of all possible logical addresses, and set up the paging unit to do the appropriate translations. For example, when using a 32-bit logical address space, logical addresses can range from zero to $2^{32} - 1$ (0xFFFFFFFF). It is up to us to decide where in this address space will be our process code, where the stack will go, where the heap will start, and so on. We did go through a similar exercise when designing the program address space in SOS1. SOS3 will use a 32-bit address space; so the logical address space of a process will range from 0x00000000 to 0xFFFFFFFF.

NAÏVE Physical Memory Manager (PMM)

Since we have started differentiating between physical memory and logical memory, let us make some minor changes to our memory manager. In fact, we will have two memory managers now, one for physical memory and one for logical memory (we will discuss this more in SOS4).

The NAÏVE physical memory manager is similar to the memory manager in SOS2. The difference is that it distinguishes between kernel memory and user memory. We will define kernel memory to be the first 4 MB of physical memory. The kernel can allocate memory from this region for its internal data structures (e.g. a process PCB). User memory is everything beyond 4 MB. The kernel uses this region for user programs and stacks.

Allocation of frames is done using the `alloc_frames` function in `pmemman.c`. This function takes as argument the number of frames to allocate (each frame is 4 KB) and a mode. The mode is either `KERNEL_ALLOC` (allocation done from first 4 MB) or `USER_ALLOC` (allocation done from 4 MB onwards). It returns the start address of the first allocated frame. The memory manager uses a memory bitmap to keep track of allocated memory. Deallocation is done using the `dealloc_frames` function, which takes as argument the start address of a frame and the number of frames to deallocate. Notice that, unlike in SOS2, the memory manager wants us to explicitly state how many frames to deallocate.



Kernel Logical Address Space

The SOS kernel resides at physical address range 0x10000 to 0x8FFFF, and the kernel stack is at 0x94000. The kernel code uses GDT entry 1, which has a base of zero. Here is a question: say the kernel code refers to memory address 0x100 (a logical address); the physical address then will be 0x100 + segment base (zero) = 0x100; but no portion of the kernel is at physical memory address 0x100; why isn't SOS crashing then? The answer is in the `create` script. The `create` script in SOS0, SOS1, and SOS2 tells the compiler to start counting addresses from 0x10000 onwards when compiling and building the kernel executable (see `kernel.ld.s` in the `build` directory). Therefore, all addresses assigned to variables and functions

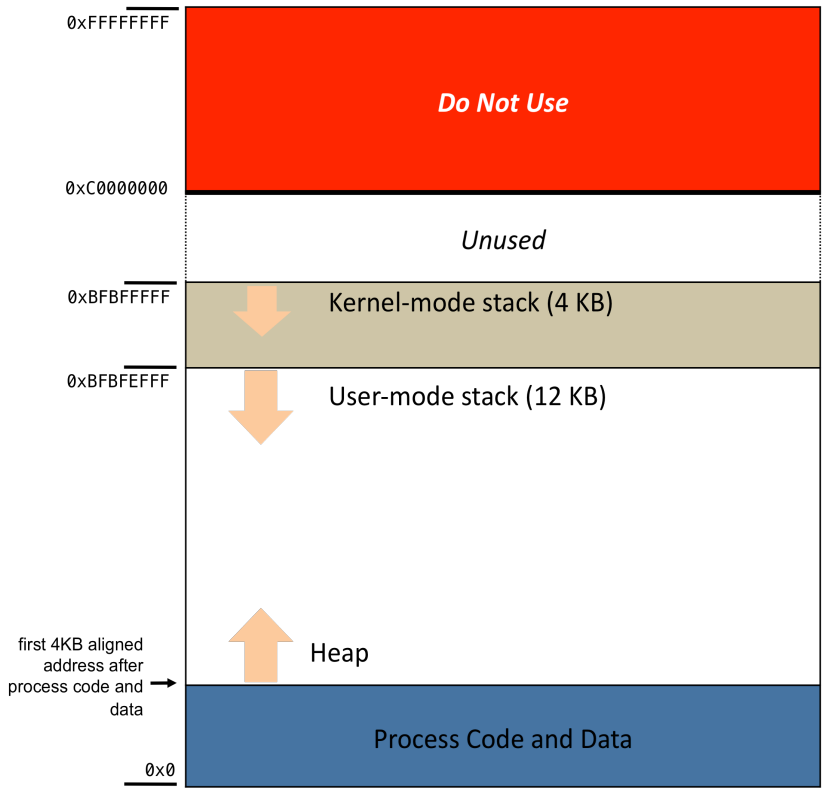
in the kernel code are always more than or equal to 0x10000. In other words, any logical address generated in the kernel code is actually equal to the physical address. We are going to change this now.

The `create` script in SOS3 builds the kernel executable such that all addresses assigned to variables and functions in the kernel code start at 0xC0010000. This is also true for addresses that are hard coded in the code; for example, the `mem_bitmap` pointer points to the address 0x100000 in SOS2, but now points to 0xC0100000 in SOS3. Note that the segmentation unit alone will fail to translate an address correctly in this case. For example, the logical address 0xC0010000 should refer to the first byte of the kernel executable, i.e. physical address 0x10000. However, 0xC0010000 will be translated to the linear address 0xC0010000 by the segmentation unit. If there is no paging (linear = physical), the MMU will then try to access physical address 0xC0010000, which is not correct. Therefore, the paging unit must be set up before the SOS3 kernel executable can run. We will do the actual setup a little later. In effect, what we have done is added a number (lets call it the `KERNEL_BASE` = 0xC0000000) to all logical addresses generated/used in the kernel in previous versions of SOS. The layout of the logical address space of the kernel is shown in the previous page.

In summary, our SOS kernel only uses its logical address range 0xC0010000 to 0xC03FFFFF.

Process Logical Address Space

Similar to the kernel, every process that runs in SOS will also have a layout of where its different parts will be in the logical address space. Lets say we decided to use the following layout.



Therefore, a process will have the executable instructions and data beginning at logical address zero. The first 4 KB aligned address (i.e. an address that is a multiple of 4096) after the process code and data will be the beginning of the heap. The heap is used when memory is dynamically allocated to the process. We

do not have this capability yet; but it is not difficult to implement. The user-mode stack for the process will begin at logical address 0xBFBFEFFF, and the kernel-mode stack will begin at logical address 0xBFBFFFFF. Note that while the kernel-mode stack can still grow up to 4 KB (like in SOS1 and SOS2), the user-mode stack now has much more room to grow. The 12 KB in the layout diagram only means that 12 KB of physical memory is allocated for the user-mode stack when the process is started. Later, we may have to map other objects in this process address space; the large space between the heap and the user-mode stack will be a good place then.

The logical address range from 0xC0000000 to 0xFFFFFFFF is not going to be used by a process. It is not an absolute necessity, but gives us some performance advantage if we do so. The reason will become clear after we set up page tables.

Process Control Block

In order to keep track of different logical addresses, our `PCB` struct has some new members. We will have to assign values to these members before the process is loaded into memory. The new members are collected in the sub-structs called `disk` and `mem`.

```
typedef struct process_control_block {
         struct {
               . . .
         } cpu;
         uint32_t pid;
         ~~uint32_t memory_base;~~
         ~~uint32_t memory_limit;~~
         . . .

         struct {
               uint32_t start_code;
               uint32_t end_code;
               uint32_t start_brk;
               uint32_t brk;
               uint32_t start_stack;
               PDE *page_directory;
         } mem;

         struct {
               uint32_t LBA;
               uint32_t n_sectors;
         } disk;
} __attribute__ ((packed)) PCB;
```

Since we are using flat segmentation now, `memory_base` and `memory_limit` are no longer needed. The `mem` and `disk` sub-structs should be assigned the following values.

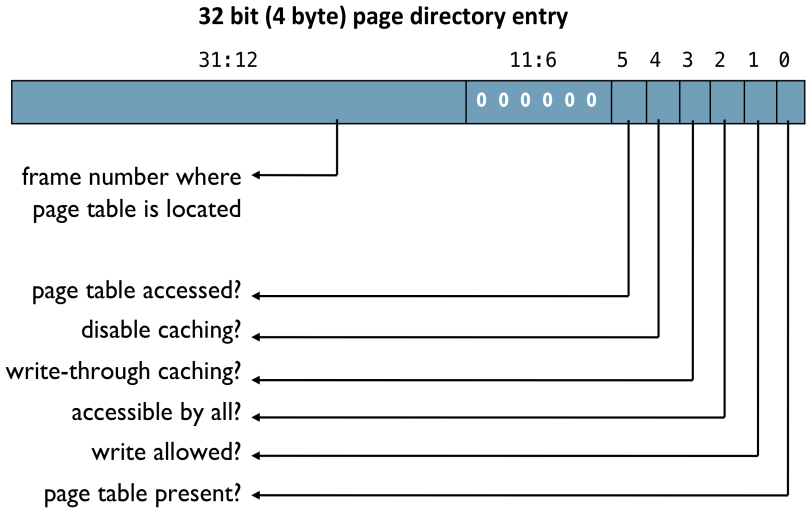
- `start_code`: logical address where the program executable begins
- `end_code`: logical address where the program executable ends
- `start_brk`: logical address where the heap begins
- `brk`: logical address where the heap currently ends (same as `start_brk` for now)
- `start_stack`: logical address where the user-mode stack begins
- `page_directory`: physical address where the process' page directory begins
- `LBA`: start sector of program in disk
- `n_sectors`: number of sectors occupied by program in disk

The assignments to the `mem` struct members are done in the `init_logical_memory` function in `lmemman.c` (coming up). Most of the values can be obtained by looking at the process logical address space layout.

Kernel Page Tables

As mentioned earlier, the paging unit must be set up correctly so that logical addresses get translated into the correct physical addresses. The translation is automatically done by the hardware; we just have to provide the right page tables to the MMU. We will use two-level paging in SOS3, with 4 KB pages. Lets first see how the page directory and the page tables for the kernel need to be set up.

Two-level paging uses one page directory, and one or more page tables. The page directory is an array containing 1024 entries; each entry is 4 bytes. Each page directory entry (PDE) helps translate a 4 MB region of the logical address space. Entry 0 is used to translate logical address range 0 to 0x3FFFFF, entry 1 is used to translate 0x400000 to 0x7FFFFF, …, and entry 1023 is used to translate 0xFFC00000 to 0xFFFFFFFF. Each entry tells us the location of a page table, which further divides the 4 MB region corresponding to the entry into 4 KB regions (pages). The SOS3 kernel uses logical addresses 0xC0010000 to 0xC03FFFFF. Which page directory entry (PDE) covers this address range? The top 10 bits of any address in this range is 0b1100000000 = 768. This means PDE 768 is the only entry that our kernel will ever use. The 4 bytes we put in a PDE has the following structure.

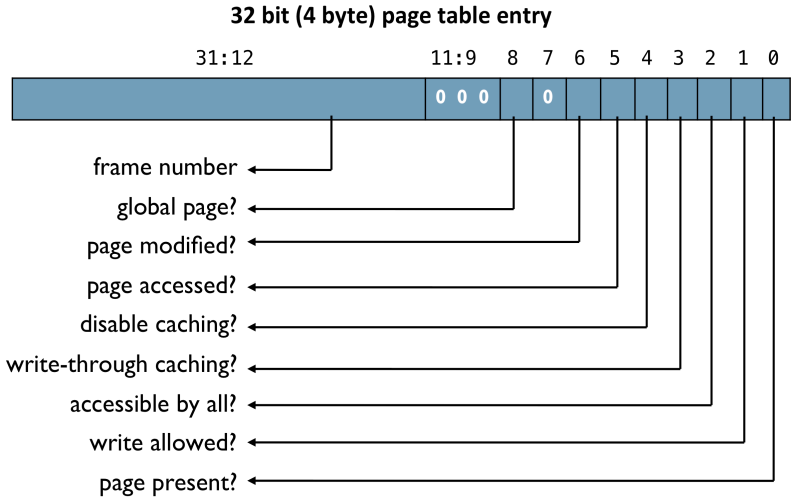


See `kernel_only.h` for a description of what the different flags mean (also see OSDev Wiki "Paging"). The crucial parts are here:

- the frame number tells us the physical memory address of the page table corresponding to this entry; if x is the value in a PDE, then x & 0xFFFFF000 is the physical address of the corresponding page table, which is the same as (frame number << 12)
- bit 0 (*present bit*) should be 1 if we have set up a page table corresponding to the entry; otherwise it should be zero
- bit 1 (*read/write bit*) should be 1 if we want to allow the kernel to read and write to the address range corresponding to this entry; setting it to zero means read only
- bit 2 (*user/supervisor bit*) should be 1 if we want to allow user processes to be able to access the corresponding address range; setting it to zero will restrict access to ring 0 only

`kernel_only.h` has pre-defined macros that you can use to set these bits in a PDE. For the SOS3 kernel, all but PDE 768 will be set to zero (bit 0 is zero, meaning page tables for those regions are not present). We will put the page directory for the kernel at physical memory addresses 0x101000 to 0x101FFF (4 KB). The global variable `PDE *k_page_directory` defined in `lmemman.c` points to this memory address. We have defined the data type `PDE` especially for use with page directory entries; behind the scenes it is simply a `uint32_t` or an unsigned 32-bit number.

The next question is what value do we put in page directory entry number 768. To do that, we will first have to set up a page table. Just like a page directory entry is used to translate a 4 MB logical address range (entry 768 maps address range 0xC0000000 to 0xC03FFFFF), entries in a page table further divide that range into 4 KB regions. Therefore, entries in the page table corresponding to PDE 768 will divide the address range as follows: entry 0 will be used to translate addresses 0xC0000000 to 0xC0000FFF, entry 1 is used for 0xC0001000 to 0xC0001FFF, entry 2 for 0xC0002000 to 0xC0002FFF, and so on. The page table will help translate a 4 MB region; so, there will be 1024 entries in a page table (1024 x 4 KB = 4 MB). The structure of a page table entry (PTE) is similar to that of a page directory entry, except for some additional flags.



Again, see `kernel_only.h` for a description of what the different flags mean (also see OSDev Wiki "Paging"). The crucial parts are here:

- the frame number tells us the physical memory address where the 4 KB region corresponding to this entry starts; if x is the value in a PTE, then x & 0xFFFFF000 is the physical address of the start of the region
- bit 0 (*present bit*) should be 1 if there is a physical memory region corresponding to the logical address range; otherwise it should be zero
- bit 1 (*read/write bit*) should be 1 if we want the kernel to be able to read and write to this region; setting it to zero makes the region read only
- bit 2 (*user/supervisor bit*) should be 1 if we want to allow user processes to be able to access this region; setting it to zero will restrict access to ring 0 only
- bit 8 (*global bit*) should be 1 if we want information on this entry to be permanently present in the TLB

`kernel_only.h` has pre-defined macros that you can use to set these bits in a PTE. We will put the page table corresponding to PDE 768 at physical memory addresses 0x102000 to 0x102FFF (4 KB). The global variable `PTE *pages_768` defined in `lmemman.c` points to this memory address. Similar to the `PDE` data type, we have defined the data type `PTE` for use with page table entries.

The `init_kernel_pages` function in `lmemman.c` initializes both `k_page_directory` and `pages_768`. This function is called by the `main` function before starting the console. The page table for PDE 768 is located at physical address 0x102000. Therefore, the number that goes into `k_page_directory[768]` is `(0x102000 | PDE_PRESENT | PDE_READ_WRITE)`, which is equal to 0x102003. This implies that the page table is at 0x102000 (= 0x102003 & 0xFFFFF000), that it is present (bit 0 of 0x102003 is 1), and that the address range has both read and write access (bit 1 of 0x102003 is 1).

To set up the page table (`pages_768`), we need to decide where does the logical address ranges corresponding to each page table entry map to in physical memory. This is where the location of the kernel executable in physical memory comes into play. Since our kernel is actually located in addresses 0x10000 onwards, a logical address such as 0xC0010000 should translate to physical address 0x10000. Similarly, a logical address such as 0xC0010001 should translate to physical address 0x10001. We do not have to specify this translation for every possible logical address in use; it is sufficient to specify where a particular 4 KB region starts in physical memory. The hardware will add the correct offset extracted from the logical address (the least significant 12 bits). So, the numbers that go into the page table entries will be as follows.

```
pages_768[0]    = 0x00000000|PTE_PRESENT|PTE_READ_WRITE|PTE_GLOBAL
pages_768[1]    = 0x00001000|PTE_PRESENT|PTE_READ_WRITE|PTE_GLOBAL
...
pages_768[16]   = 0x00010000|PTE_PRESENT|PTE_READ_WRITE|PTE_GLOBAL
pages_768[17]   = 0x00011000|PTE_PRESENT|PTE_READ_WRITE|PTE_GLOBAL
pages_768[18]   = 0x00012000|PTE_PRESENT|PTE_READ_WRITE|PTE_GLOBAL
...
pages_768[1023] = 0x003FF000|PTE_PRESENT|PTE_READ_WRITE|PTE_GLOBAL
```

Each entry specifies the beginning of the corresponding 4 KB region in physical memory, that the entry has a valid mapping (bit 0 of each entry is 1), that the regions can be modified (bit 1 of each entry is 1), and that each entry should be permanently stored in the TLB (bit 8 is 1). We have not set bit 2 in any of the entries, meaning a user process cannot access these regions.

The final question is how do we tell the MMU to use this page directory and page table. It is rather simple. The MMU always looks at the CR3 control register to determine the location of the page directory in physical memory. So, we only need to load CR3 with the value 0x101000 (the physical memory address where `k_page_directory` begins), and the MMU will start using our paging setup.

As a summary to this long section, here is what happens in hardware when the kernel attempts to access logical address 0xC0012345. The hardware first figures where the page directory is located by looking at the value stored in the CR3 register (= 0x101000). Next it determines which PDE should be used for 0xC0012345; that will be PDE 768 (0xC0012345 >> 22 = 768). It checks to see if bit 0 of PDE 768 is 1 (yes it is). It will also perform some other checks, such as whether the access is for reading or writing, and if it is allowed, and whether the current mode is allowed access to the entry. After all checks go through, the hardware reads the value in the entry and determines the start location of the page table corresponding to this entry (= 0x102000). Next it determines which PTE should be used for 0xC0012345; that will be PTE (0xC0012345 >> 12) & 0x000003FF = 18. The hardware does similar checks as it did for the page directory. When all checks go through, the hardware determines the start physical address of the region mapped by the page table entry (= `pages_768[18]` & 0xFFFFF000). In our case, this is 0x12000. Lastly, the offset from the logical address (0xC0012345 & 0x00000FFF = 0x345) is added to this start address, 0x12000 + 0x345 = 0x12345, thereby generating the physical address. Errors such as access violation or the present bit is zero will result in an exception (more on this later). As you can see, the translation is automatically done by the hardware, but it works only because we have set up `k_page_directory` and `pages_768` correctly.

Process Page Tables

Each process has its own logical address space. While the structure is the same across processes, the actual logical addresses of certain contents will differ from process to process. For example, program

executables can be of different size; the amount of logical address space they occupy will also be different. As a result, the logical address where the heap starts can also be different. Most importantly, the same logical address will map to different physical addresses in different processes. Therefore, each process will have to maintain its own page directory and page tables.

Unlike the kernel which only needed one page table (and the page directory), a process will require more than just one page table. This is because the logical addresses that a process will be using correspond to different page directory entries. So, the number of page tables that we will have to set up will depend on how many different 4 MB regions does the process address space touch. As an example, the code/data part of the process will definitely need PDE 0 to be set up. But, if the program is larger than 4 MB, then PDE 1 will also have to be set up. Another PDE needs to be set up for the user-mode and kernel-mode stack part. Of course, there will be many page directory entries that can simply be set to zero (not present).

Similarly, the page table corresponding to a PDE may not have to be filled in its entirety. Think about the page table corresponding to the user-mode and kernel-mode stack area. Our user-mode stack is 12 KB and the kernel-mode stack is 4KB, giving a total of 16 KB. Everything surrounding these stacks is unused. So the page table will have 4 non-zero entries (4 x 4 KB = 16 KB); everything else will be zero (not present). The user-mode stack can be allowed to grow later, in which case some of the zero entries will have to be properly set. Lets not worry about that for now.

The page directory and page tables set up for a process is done in the `init_logical_memory` function in `lmemman.c`. This function will first determine how many physical frames (each frame is 4 KB) will be required to store the program executable, the user-mode stack and the kernel-mode stack. It will then have to allocate those many frames from user memory using the `alloc_frames(...,USER_ALLOC)` function in our NAÏVE PMM. After this step, we will know the start physical addresses of each part of the process.

Next, we will have to set up the page directory and page tables to map the logical addresses to the now known physical addresses. The first step is to compute how many page tables will be needed. Each page table (and the page directory) needs 4 KB. We will have to allocate physical frames accordingly, but this time from kernel memory using `alloc_frames(...,KERNEL_ALLOC).` And then comes the fun part of filling in the entries in the page directory and the page tables, along with the appropriate flag bits. The page directory and page tables setup for a process is left as an exercise.

Keep in mind that the page directory entries and page table entries always store physical addresses. The physical address of the process' page directory will be stored in `mem.page_directory` of the process' PCB. When a switch is made to a user process, the `switch_to_user_process` function will load CR3 with the value stored in `mem.page_directory` of the process' PCB. This will bring alive the paging structure of the process and logical addresses will get correctly translated to physical addresses.

Higher Half Kernel

When a process is running regular instructions, its paging structures are providing the necessary logical to physical address translations. Now lets say the program makes a system call. As a result, we will switch to kernel code. What happens? System Crash! This is because the page directory currently in use has all zero entries corresponding to the range of logical addresses that the kernel uses (0xC0010000 onwards). Therefore, anytime a switch is made to kernel code (during system calls or interrupt handling), the kernel's paging structures must be loaded before any code is allowed to run. This is a performance hit since system calls and interrupts will occur quite frequently. The solution is to copy over the kernel's paging data from its page directory to that of every process' page directory. Since, the kernel uses logical addresses beyond 0xC0000000, and we designed a process to never use that range, there will be no conflict. In SOS3, we only need to copy PDE 768 from the kernel's page directory to that of a process.

Once this is done, no reloading is necessary when temporarily running kernel code during system calls and interrupts. All the paging data corresponding to the kernel code is present in the currently loaded process' page directory. Of course, a different page directory will have to be loaded when a process switch is performed; but that process' page directory will also have the kernel page mappings. This copying of PDE 768 from the kernel's page directory is performed in the `init_logical_memory` function.

It is important that kernel paging entries do not have the user/supervisor bit turned on. Otherwise, user processes will also be able to access that address range, which effectively means, they will be able to change kernel code. Not pretty!

The run Command

The `run` command in SOS3 is similar to that in SOS2. The only difference is that it calls `init_logical_memory` after allocating kernel memory for the process' PCB. `init_logical_memory` does all the necessary work of allocating memory for the program code/data and stacks, and setting up page tables. The other difference is that it does not load the program from the disk to memory. It simply marks the state of the process to NEW and calls `add_to_processq` to insert the process into the process queue.

Process Loading

The actual loading of the process code from disk to memory happens in the `schedule_something` function in `scheduler.c`. If the next process to run (`processq_next`) is in the NEW state, the `schedule_something` function loads the page directory of the process (using `mem.page_directory` in the process' PCB), and then calls `load_disk_to_memory` to load the program. The loading is done by using the `disk.LBA` and `disk.n_sectors` members of the process' PCB. These values are set by the `run` function. If the loading is successful, we simply change the state of the process to READY, and continue as before.

Enabling Paging

We have discussed everything necessary to make paging work in SOS. But, how do we tell the CPU to start using paging, i.e. turn the paging unit on. Remember CR0. We used it to enable protected mode (by enabling bit 0). See Wikipedia "Control register" for details on other bits. Paging is enabled when bit 31 of CR0 is set (1). This is done in `startup.S`, before even the `main` function of the kernel runs.

However, it is not sufficient to simply turn on bit 31 in CR0. Page directory/tables must be set up before we do that, and the CR3 register should also be loaded with the physical address of the kernel's page directory. But, the kernel's page directory (`k_page_directory`) is not set up until we move into `main` and call `init_kernel_pages`. Hmmm! To resolve this, `startup.S` sets up a temporary page directory and page table before enabling paging. All of this is done in assembly code. The contents of this temporary page directory/table are same as that of `k_page_directory` and `pages_768`. Later when `init_kernel_pages` is called, we switch to `k_page_directory`. PDE 0 of the temporary page directory is also setup to do identity mapping (logical address = physical address). Think why that is necessary!

Page Fault Exception Handler

Exception number 14 is generated if the kernel or a user program tries to access a logical address for which there is no mapping present in the page directory/tables. This is also known as a **page fault**. It will be useful to know which process generated the page fault. SOS3 replaces the default exception handler for IDT entry 14 with a page fault exception handler (implemented in `exceptions.c`). If the page fault was caused by the kernel code, this handler lets you know about it and halts the system. If a user program

caused the page fault, the handler will print the `pid` and `disk` data from the PCB, along with the logical address that caused the page fault. After that, the state of the user process is changed to TERMINATED. In the real world, the page fault handler is where you will swap pages in and out of the swap space in the disk.

Implementation Detail

Look in the `init_kernel_pages` function in `lmemman.c`. While we have been saying that `k_page_directory` points to 0x101000 and `pages_768` points to 0x102000, the pointers in the function actually point to 0xC0101000 and 0xC0102000 respectively.

```
PDE *k_page_directory = (PDE *)(0xC0101000);
PTE *pages_768 = (PTE *)(0xC0102000);
```

This is an implementation tweak since we want to be able to work with these pointers in kernel code, and our kernel is using logical addresses. Paging is already enabled when `init_kernel_pages` run and the temporary page directory and page table is in place at this point. Therefore, `k_page_directory` and `pages_768` must be logical addresses so that when we dereference them (in a statement such as `k_page_directory[768] = ...`), they get translated to the correct physical address to which they actually refer. In general, for any variable in the kernel code, the physical address is easily obtained by subtracting the `KERNEL_BASE` (0xC0000000) from the logical address of the variable.

Process Synchronization (SOS4)

In this section, we will add process synchronization capabilities to SOS. More specifically, we will allow user programs to create and use mutex locks and semaphores. We will also provide support for shared memory. Using these three concepts, a user program will be able to share data with other user programs, and also synchronize access to the shared data.

NAÏVE Logical Memory Manager

Let us first talk about the logical memory manager in SOS4 (this also existed in SOS3, but was rarely used). The NAÏVE logical memory manager is responsible for allocating and deallocating pages. SOS uses the physical memory manager whenever it needs frames; similarly, SOS uses the logical memory manager whenever it needs to map an area from the logical address space of a process to physical frames. By mapping (allocating) previously unused pages of the logical address space, SOS makes it possible for a process to use the addresses in those pages.

The NAÏVE logical memory manager has two functions to allocate pages – `alloc_kernel_pages` and `alloc_user_pages`. These functions are part of `lmemman.c`.

`alloc_kernel_pages` allocates pages in the kernel's logical address space. This function takes as argument the number of pages to allocate, and then allocates that many frames from kernel memory using `alloc_frames(..., KERNEL_ALLOC)`. Since, kernel memory is always from the first 4 MB of physical memory, and the paging structures of the kernel are already set up to map 0xC0000000–0xC03FFFFF to the first 4 MB of physical memory, the function simply adds `KERNEL_BASE` to the frame address to generate the corresponding logical address.

`alloc_user_pages` is more involved. Besides the number of pages to allocate, this function also requires the start logical address of these pages (called the `base` in the function arguments). For example, lets say we want to allocate 3 pages starting at logical address 0x80000000 for a process. To perform this allocation, we will first need to allocate 3 frames, and then update the paging structures of the process so that addresses 0x80000000 to 0x80003000 map to these three frames. `alloc_user_pages` does exactly this. In addition, the function also uses a `mode` argument to specify whether the new pages are read only, or read/write enabled. The function returns the start logical address of the allocated pages, which is the same as `base`.

There is also a `dealloc_page` function that unmaps a page, meaning it removes the page table entry corresponding to the page and deallocates the associated frame.

Shared Memory

Providing shared memory becomes straightforward after paging is enabled. We only need to map pages belonging to the logical address space of different processes to the same physical frames. However, a few questions must be answered for that.

1. Which logical addresses will be used for shared memory?
2. How will the OS keep track of shared memory pages created by a process?
3. How will a process tell the OS that it is creating shared memory pages?
4. How will a process ask the OS to map some of its pages to the shared memory area created by another process?

Let us first see what data structures and functions are involved in the implementation of shared memory in SOS4. After that we will see how SOS answers the four questions.

SOS4 defines a `SHMEM` data type in `kernel_only.h`. This structure contains three unsigned 32-bit variables – `refs` to keep track of the number of processes that has access to a shared memory area; `base` to store the physical start frame of a shared memory area; and, `size` to store the size of the shared memory area (maximum is 4 MB).

```
typedef struct {
        uint32_t refs;
        uint32_t base;
        uint32_t size;
} SHMEM;
```

The `PCB` struct now also includes a variable called `shared_memory`. This struct variable has two members – a boolean variable called `created` to indicate that a process has created a shared memory area, and an unsigned 8-bit number called `key` (more on this coming up). SOS4 does not allow a process to work with more than one shared memory area at a time.

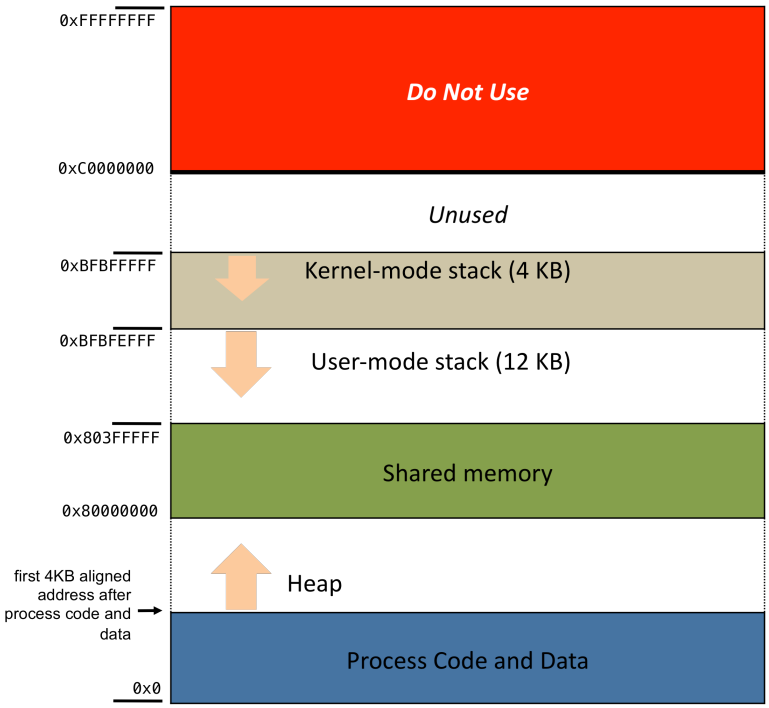
```
typedef struct process_control_block {
          struct {
                . . .
          } cpu;

          . . .

          struct {
                bool created;
                uint8_t key;
          } shared_memory;

          struct {
                int wait_on;
                uint32_t queue_index;
          } mutex;

          struct {
                int wait_on;
                uint32_t queue_index;
          } semaphore;

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

The following table summarizes the shared memory related system calls and the corresponding kernel functions that get executed as a result of a system call.

System Call *(in lib.c)*	**Kernel function** *(in shared_memory.c)*
`void *smcreate(uint8_t key, uint32_t size);`	`void *shm_create(uint8_t key, uint32_t size, PCB *p);`
`void *smattach(uint8_t key, uint32_t mode);`	`void *shm_attach(uint8_t key, uint32_t mode, PCB *p);`
`void smdetach();`	`void shm_detach(PCB *p);`

Shared Memory in Logical Address Space

SOS4 always begins shared memory pages of a process at logical address 0x80000000 (the macro `SHM_BEGIN` defined in `kernel_only.h`). A process can ask SOS4 to provide shared memory regions up to a size of 4 MB. This means the largest logical address belonging to a shared memory area will be 0x803FFFFF. This answers the first question we raised earlier.



Shared Memory Objects

When a process is started, the shared memory area is not mapped (page directory and page table entries are zero). In order to create the shared memory, a process uses the `smcreate` system call, passing to it the size of the shared memory area to create (maximum is 4 MB). At this point, the kernel can do the frame allocation and update the page tables of the process to map the shared memory area addresses to the allocated frames. This is easily done by calling `alloc_user_pages(n_pages, SHM_BEGIN, ..., ...)`. Upon return from the system call, the process can start to use the memory area. However, the purpose of shared memory is to allow multiple processes to access these same frames. To do this, when a process requests SOS to create the shared memory area in its logical address space, SOS will have to remember the memory frames corresponding to this shared area. SOS can then map the shared memory addresses of another process to these same frames, effectively making both processes read and write from the same physical memory area. Note that this will require two processes to somehow tell SOS that they want to share memory.

To keep track of shared memory areas created by processes, SOS4 uses an array of `SHMEM` type variables, `SHMEM shm[SHMEM_MAXNUMBER]` declared in `shared_memory.c`. We will call them shared memory objects. `SHMEM_MAXNUMBER` is the maximum number of shared memory objects that will be tracked by SOS. In SOS4, this cannot be more than 256. An argument to the `smcreate` system call is an unsigned 8-bit number called `key`. Therefore, the key is a number between 0 and 255 (both inclusive). `shm_create` uses the value in `key` as an index into the `shm` array. After creating the shared memory area for the process, `shm_create` stores the address of the first frame (mapped to the shared memory area) and the size of the area in `shm[key].base` and `shm[key].size` respectively. It also sets the `shared_memory.created` variable in the process's PCB, and stores `key` in `shared_memory.key`. This answers the second and third question.

Once a process creates a shared memory area using a certain key, other processes can attach to it using the same key value. This is what is done using the `smattach` system call. `shm_attach` works very similar to `shm_create`, but instead of allocating new frames, `shm_attach` uses the frame address

stored in `shm[key].base`. As you can see, the key value is what allows multiple processes to share the same physical frames (answer to the fourth question). Cooperating processes will have to decide on a key to use first; one of the processes will have to then create the shared memory area using the key; other processes can then attach to the memory area by using the same key. There is a second argument in `smattach` called `mode`, which should be either `SM_READ_ONLY` or `SM_READ_WRITE` (defined in `lib.h`). `shm_attach` also sets the `shared_memory.created` variable in the process's PCB, and stores `key` in `shared_memory.key`. In the following example, process A creates an integer `p` in shared memory using the key 123. Processes B and C attach to the shared memory area created by A using the same key 123. In effect, the variables `p` (in process A), `q` (in process B), and `r` (in process C), all mean the same memory location, although they are being used in three different processes. Process B can only read from the memory location since it has attached to it in read only mode (`SM_READ_ONLY`). Process C can both read and write since it has attached to it in read/write mode (`SM_READ_WRITE`). The creating process, process A, can read and write by default.

```
// in process A (creator)
#define MY_KEY 123
int *p = (int *)smcreate(MY_KEY,sizeof(int));

// in process B
#define MY_KEY 123
int *q = (int *)smattach(MY_KEY, SM_READ_ONLY);

// in process C
#define MY_KEY 123
int *r = (int *)smattach(MY_KEY, SM_READ_WRITE);
```

Since SOS4 does not allow a process to create or attach to more than one shared memory area at a time, it should be possible to detach from the area once a process is done using it. Ideally, any process that creates or attaches to a shared memory area (as identified by a key) should detach from it. This is done using the `smdetach` system call, which finally ends up executing `shm_detach`. This function frees up the process's logical address space corresponding to the shared memory area, i.e. it cleans up the paging structures corresponding to the logical address range 0x80000000 to 0x803FFFFF. However, it does not necessarily deallocate the corresponding frames. Whenever a process creates or attaches to a particular shared memory area using a certain key, the reference count for that area is incremented by one (`shm[key].refs++`). When detaching, the reference count is decremented by one. The physical frames corresponding to the shared memory area are deallocated only if the reference count becomes zero. Processes can create shared memory areas with a certain key only if the reference count of the corresponding shared memory object (`shm[key].refs`) is zero (meaning that the object is not in use). `shm_detach` is automatically called for a process when it terminates.

See `shared_memory.c` for details on the implementation and the return values of the system calls. In the SOS4 implementation, shared memory areas are identified by keys. An alternative to this is to use an alphanumeric string. Think of the string as a secret known only to the cooperating processes. It is much better than having a number between 0 and 255. With strings as keys, SOS can use any available shared memory object (or even allocate it dynamically using `alloc_kernel_pages`); of course, it will have to remember which object was assigned to which string. Other processes can attach to the object using the same string. SOS will have to search through the shared memory objects and determine which object was assigned to the string. This is a little bit more programming, and is the way Linux does it.

Queue

We need a queue implementation for mutex and semaphore wait queues. Ideally, a linked list implementation is desired; but SOS4 has a simple implementation using arrays in `queue.c`. A queue in SOS4 is defined in the following struct.

```
typedef struct {
```

```
        uint32_t head;
        uint32_t count;
        uint32_t *data;
} QUEUE;
```

Here, `head` will store the head of the queue, `count` will be the number of items currently in the queue, and `data` is a dynamically allocated array holding the queue items. As you can see, the items in a queue are unsigned 32-bit numbers. They will in fact be addresses of process PCBs. All queues must be initialized by using the `init_queue` function. When a `QUEUE` variable is created, and initialized, there is really no place to insert items into the queue (`data` is `NULL`). We allocate the `data` array only when the first item is inserted into the queue. The `data` array is one page (4 KB) large, allocated using `alloc_kernel_pages`. This means it can hold a maximum of 4096/4 = 1024 items, but we can make it smaller by changing the `Q_MAXSIZE` macro in `kernel_only.h`. The default value is 256. The queue is implemented as a circular array; so, trying to write more than `Q_MAXSIZE` items will result in some items being overwritten.

Items (PCB addresses) can be inserted into a queue using the `enqueue(QUEUE *q, PCB *p)` function. The function inserts the address stored in `p` to the tail end of the queue `q`, i.e. `(q->head + q->count)%Q_MAXSIZE`. It returns the index in the array `data` where the item is inserted. Items can be removed from the queue using the `dequeue` function. This function will return the PCB address stored at the head end of the queue; `NULL` if the queue is empty. There is also a function called `remove_queue_item` that allows us to delete an item from the middle of a queue. The implementation is rather straightforward. Take a look at it before using any of these functions.

Mutex

A `MUTEX` datatype is defined in `kernel_only.h`. SOS4 predefines an array of `MUTEX` variables – `MUTEX mx[MUTEX_MAXNUMBER]` – in `mutex.c`. `MUTEX_MAXNUMBER` is defined in `kernel_only.h` and set to 256.

```
typedef struct {
        bool available;
        uint32_t creator;
        PCB *lock_with;
        QUEUE waitq;
} MUTEX;
```

The `MUTEX` struct contains the variables necessary to keep track of a mutex variable. The boolean member `available` is set to `FALSE` if the mutex variable is in use by user programs, the `creator` member holds the `pid` of the process that created the mutex variable, the `lock_with` member has the address of the PCB belonging to the process that currently has the mutex lock, and the `waitq` member is the waiting queue corresponding to the mutex variable. Note that in a typical mutex implementation (such as the ones you will read in a book), a boolean variable (say `M`) is used to track the state of the mutex; `M=1` means mutex is locked, `M=0` means mutex is unlocked. We do it in a slightly different manner, where `lock_with` not only tells us the state of the mutex, but also which process has the lock. Effectively, the mutex is unlocked if `lock_with` is `NULL`, otherwise it is locked. Keep reading to understand their exact use.

Various functions in `mutex.c` manipulate the `mx` mutex variable array. The first of these, `init_mutexes`, is called by the `main` function in SOS4. `init_mutexes` makes `available=TRUE` for the mutex variables `mx[1]` to `mx[255]` (`mx[0]` is not used), initializes the queues associated with these variables, and marks the mutexes as being in unlocked state (`lock_with` is `NULL`). The following table lists the other functions in `mutex.c` and the corresponding system calls that finally execute these functions. Some of these functions use a `mutex_t` data type, which is simply a different name for an `unsigned char` (see `lib.h`).

System Call *(in lib.c)*	**Kernel function** *(in mutex.c)*
`mutex_t mcreate();`	`mutex_t mutex_create(PCB *p);`
`void mlock(mutex_t key);`	`bool mutex_lock(mutex_t key, PCB *p);`
`bool munlock(mutex_t key);`	`bool mutex_unlock(mutex_t key, PCB *p);`
`void mdestroy(mutex_t key);`	`void mutex_destroy(mutex_t key, PCB *p);`

Every process also has a variable called `mutex` in its PCB. This struct variable has two members – an integer variable called `wait_on`, and an unsigned integer called `queue_index`. The use of these members will become clear as we discuss the mutex implementation.

`mutex_t mutex_create(PCB *p)`

A program begins by creating a mutex using the `mcreate` system call. The return value of this system call is a number between 1 and 255. Think of this number as an identifier for the mutex. Subsequent functions to use the mutex must pass this identifier to the kernel. The identifier is like the key used in shared memory, with the difference that the identifier is provided by the kernel and not decided in the user program. The kernel function `mutex_create` gets executed as a result of the `mcreate` system call. `mutex_create` first finds a mutex variable that is available for use in the `mx` array (`available` is `TRUE`); if none is available, then it returns zero. For the mutex variable that it finds, it then sets `available` (in use now), `creator` (`pid` of process that made this call), `lock_with` (lock is with no process at this point), and resets the waiting queue associated with the mutex (`waitq.head` and `waitq.count`). The function then returns the array index.

`bool mutex_lock(mutex_t key, PCB *p)`

A program calls `mlock` when it wants to obtain a lock on a mutex. The mutex is specified using a `mutex_t` value, called the `key` here. This is the identifier that is returned from an earlier call to `mcreate`. The kernel function `mutex_lock` gets executed as a result of the `mlock` system call. `mutex_lock` returns a boolean. It performs the regular mutex locking operations – if the lock is with no process, give it to the calling process (change `lock_with`) and return `TRUE`; otherwise, insert the calling process into the waiting queue of the mutex, and return `FALSE`. If a process is going to be inserted into the queue of a mutex, the `mutex.wait_on` and `mutex.queue_index` members of the process's PCB are also updated. `wait_on` is set to the key of the mutex on which the process is going to wait, and `queue_index` is the index in the waiting queue where this process is inserted. `wait_on` should be -1 for a process not in the waiting queue of a mutex. These two members are used for cleanup if a process waiting in a mutex queue abnormally terminates (see `free_mutex_locks`).

Notice that the boolean value returned by `mutex_lock` is not returned to the user program (`mlock` has a return type `void`). In fact, it is used by `_0x94_mutex_lock` in `kernelservices.c` that calls `mutex_lock`. A return value of `TRUE` means that the process's state can be changed to READY, otherwise the state should be WAITING (process has been blocked).

`bool mutex_unlock(mutex_t key, PCB *p)`

A program calls `munlock` to release an already obtained mutex lock. The kernel function `mutex_unlock` gets executed as a result of the `munlock` system call. `mutex_unlock` returns a boolean, indicating if the unlocking was successful or not. An unlock operation is not successful if the calling process does not own the lock on the mutex. After a mutex is unlocked (`lock_with=NULL`), `mutex_unlock` gives the lock to the process at the head of the waiting queue, if any. Accordingly, we will have to set `lock_with`, `mutex.wait_on` in the process PCB, and wake up the process.

`void mutex_destroy(mutex_t key, PCB *p)`

A process that created a mutex can destroy it. Destroying a mutex simply means that the variable in the `mx` array is now available again.

`mutex_lock`, `mutex_unlock` and `mutex_destroy` should simply return (with the appropriate return value, if any) if the mutex variable identified by `key` has not been created. The implementation of the above four functions is left as an exercise.

It is important to know the exact semantics of operation whenever working with process synchronization primitives. So, here they are for SOS4 mutex locks. We must ensure that our implementation enforces these semantics.

- If a process tries to lock or unlock a mutex that has not been created, then the behavior is undefined (do not know how the user program will behave).
- Only a process that has a lock on the mutex can unlock it.
- Locks are non-recursive, meaning if the process holding the lock tries to obtain the lock again (before unlocking it), it will cause a deadlock.
- Only the process that created the mutex can destroy it.
- A mutex is automatically destroyed when the creator process terminates.
- Mutex locks are not automatically released when the process holding the lock terminates. The process must call `munlock` to release the lock.
- A process in the waiting queue of a mutex is automatically removed from the queue if it terminates (abnormally) while waiting.

Semaphore

The SOS4 semaphore implementation follows the same structure as mutex variables. A `SEMAPHORE` datatype is defined in `kernel_only.h`. SOS4 predefines an array of `SEMAPHORE` variables – `SEMAPHORE sem[SEM_MAXNUMBER]` – in `semaphore.c`. `SEM_MAXNUMBER` is defined in `kernel_only.h` and set to 256.

```
typedef struct {
        bool available;
        uint32_t creator;
        int value;
        QUEUE waitq;
} SEMAPHORE;
```

The `SEMAPHORE` struct contains the variables necessary to keep track of a semaphore variable. The boolean member `available` is set to `FALSE` if the semaphore variable is in use by user programs, the `creator` member holds the `pid` of the process that created the semaphore variable, the `value` member has the current value of the semapahore, and the `waitq` member is the waiting queue corresponding to the semaphore variable.

Various functions in `semaphore.c` manipulate the `sem` semaphore variable array. The first of these, `init_semaphores`, is called by the `main` function in SOS4. `init_semaphores` makes `available=TRUE` for the semaphore variables `sem[1]` to `sem[255]` (`sem[0]` is not used), initializes the queues associated with these variables, and initializes their values to zero. The following table lists the other functions in `semaphore.c` and the corresponding system calls that finally execute these functions. Some of these functions use a `sem_t` data type, which is simply a different name for an `unsigned char` (see `lib.h`).

System Call *(in lib.c)*	**Kernel function** *(in semaphore.c)*
`sem_t screate(uint8_t init_value);`	`sem_t semaphore_create(uint8_t init_value, PCB *p);`
`void sdown(sem_t key);`	`bool semaphore_down(sem_t key, PCB *p);`
`void sup(sem_t key);`	`void semaphore_up(sem_t key, PCB *p);`
`void sdestroy(sem_t key);`	`void semaphore_destroy(sem_t key, PCB *p);`

Every process also has a variable called `sempahore` in its PCB. This struct variable has two members – an integer variable called `wait_on`, and an unsigned integer called `queue_index`. These members have the same function as in the mutex implementation.

`sem_t semaphore_create(uint8_t init_value, PCB *p)`

A program begins by creating a semaphore using the `screate` system call, and passes to it the initial value to set for the semaphore. The return value of this system call is a number between 1 and 255. Similar to mutex variables, this number is an identifier for the semaphore. Subsequent functions to use the semaphore must pass this identifier to the kernel. The kernel function `semaphore_create` gets executed as a result of the `screate` system call. `semaphore_create` first finds a semaphore variable that is available for use in the `sem` array (`available` is `TRUE`); if none is available, then it returns zero. For the semaphore variable that it finds, it then sets `available` (in use now), `creator` (`pid` of process that made this call), `value` (set to `init_value`), and resets the waiting queue associated with the semaphore (`waitq.head` and `waitq.count`). The function then returns the array index.

`bool semaphore_down(sem_t key, PCB *p)`

A program calls `sdown` when it wants to perform the *down* operation on a semaphore. The semaphore is specified using a `sem_t` value, called the `key` here. This is the identifier that is returned from an earlier call to `screate`. The kernel function `semaphore_down` gets executed as a result of the `sdown` system call. `semaphore_down` returns a boolean. It performs the regular semaphore down operations – if the semaphore's value is not zero, decrement it and return `TRUE`; otherwise, insert the calling process into the waiting queue of the semaphore, and return `FALSE`. Just like in the mutex implementation, `semaphore.wait_on` and `semaphore.queue_index` members of the process's PCB are also updated.

`void semaphore_up(sem_t key, PCB *p)`

A program calls `sup` to perform an *up* operation on a semaphore. The kernel function `semaphore_up` gets executed as a result of the `sup` system call. `semaphore_up` first increments the value of the semaphore. It then attempts to wake up a process, if any, which was blocked as a result of a down operation. Accordingly, we will have to decrement the semaphore again, update `semaphore.wait_on` in the process PCB, and wake up the process. Recall, that `wait_on` should be -1 if a process is not waiting in a queue.

`void semaphore_destroy(sem_t key, PCB *p)`

A process that created a semaphore can destroy it. Destroying a semaphore simply means that the variable in the `sem` array is now available again.

`semaphore_down`, `semaphore_up` and `semaphore_destroy` should simply return (with the appropriate return value, if any) if the semaphore variable identified by `key` has not been created. The implementation of the above four functions is left as an exercise.

The SOS4 semaphore semantics are as follows.

- If a process tries to perform up or down operations on a semaphore that has not been created, then the behavior is undefined (do not know how the user program will behave).
- Only the process that created the semaphore can destroy it.
- A semaphore is automatically destroyed when the creator process terminates.
- A process in the waiting queue of a semaphore is automatically removed from the queue if it terminates (abnormally) while waiting.

mem Command

SOS4 has a new console command called `mem`. It displays the amount of free memory in bytes (hex notation).

Pitfalls

The mutex and semaphore implementation in SOS4 are simple. As such, it has some downsides. A malicious process can obtain locks for all 255 mutex variables and prevent other cooperating processes from making progress. To resolve this, the `key` used in the lock/unlock and up/down operations should never be used directly in the kernel code. Instead, the kernel should provide large random identifiers when the variables are created. There should be a mapping present inside the kernel from the identifier to the actual mutex/semaphore variable. The same should also be done for shared memory areas. The variables themselves can be created dynamically and maintained in a linked list, instead of pre-allocating them in a fixed size array. Another option is to allow the user processes to create the variable in its shared logical address space, and pass a reference to it during the system calls. The malicious process will have to attach to the shared memory to do any harm. This can be prevented by having secret alphanumeric strings (only known to the cooperating processes) as keys to create/attach to a shared memory area.

The queue implementation is another issue. If an array is used to hold the queued items, then the array better be sufficiently large. Otherwise, there should be provisions for the queue to grow in size when it becomes full. SOS4 can hold up to 1024 items in its mutex and semaphore queues. Hopefully, this is large enough!

Sample User Programs

Given below are two SOS4 user programs that make use of the above synchronization primitives. The first of these is a program that reads one character at a time from a static string and writes it to a shared buffer of a smaller size. It also creates some shared memory area to hold the buffer, two variables called `in` and `out` to track the next write and next read positions in the buffer, two `mutex_t` variables and three `sem_t` variables. Mutex `mx_buffer` and `mx_consumer_count` provide mutual exclusion to the `slot` array and the `n_consumers` variable respectively. Semaphores `sem_empty` and `sem_full` are for synchronized access to `slot`. Binary semaphore `sem_done` is to synchronize the termination of the producer and consumer processes. This is a typical producer program; however, it uses additional synchronization to make sure that all consumer processes terminate when there are no more items to produce.

```
/************************************
* PRODUCER
************************************/
#include "lib.h"

#define SM_KEY        36
#define BUFFER_SIZE   5

typedef struct {
         char slot[BUFFER_SIZE];
         int in, out;
         int n_consumers;
         mutex_t mx_buffer;
         mutex_t mx_consumer_count;
         sem_t sem_empty;
         sem_t sem_full;
         sem_t sem_done;
} SHARED_DATA;

void main() {
         char str[] = "It looked like a good thing: but wait till I tell you. We were
```

```
down South, in Alabama--Bill Driscoll and myself-when this kidnapping idea struck us. It
was, as Bill afterward expressed it, \"during a moment of temporary mental apparition\";
but we didn't find that out till later.\n";

        int i = 0;
        SHARED_DATA *b = (SHARED_DATA *)smcreate(SM_KEY, sizeof(SHARED_DATA));

        if (b==NULL) {
              printf("Unable to create shared memory area.\n");
              return;
        }

        b->sem_empty = screate(BUFFER_SIZE);
        b->sem_full = screate(0);
        b->sem_done = screate(0);

        if (b->sem_empty == 0 || b->sem_full == 0 || b->sem_done == 0) {
              smdetach();
              printf("Unable to create semaphore objects.\n");
              return;
        }

        b->mx_buffer = mcreate();
        b->mx_consumer_count = mcreate();

        if (b->mx_buffer == 0 || b->mx_consumer_count == 0) {
              smdetach();
              printf("Unable to create mutex objects.\n");
              return;
        }

        b->in = 0; b->out = 0; b->n_consumers = 0;
        printf("Producing items...consumers can run now.\n");

        while(str[i]!=0) {
              sleep(50); // simulation: producer producing next item
              sdown(b->sem_empty);
              // begin critical section
              b->slot[b->in] = str[i];
              b->in = (b->in+1)%BUFFER_SIZE;
              // end critical section
              sup(b->sem_full);
              i++;
        }

        printf("\nDone producing...waiting for consumers to end.\n");

        mlock(b->mx_consumer_count);
        int alive;

        do {
              mlock(b->mx_buffer);
              alive = b->n_consumers;
              munlock(b->mx_buffer);

              if (alive > 0) { // consumers are alive
                      sdown(b->sem_empty);
                      b->slot[b->in] = 0; // send END signal to consumer
                      b->in = (b->in+1)%BUFFER_SIZE;
                      sup(b->sem_full);
                      sdown(b->sem_done);
              }
        } while (alive>0);

        printf("\nShutters down!\n");

        sdestroy(b->sem_full);
        sdestroy(b->sem_empty);
        sdestroy(b->sem_done);
```

```
        mdestroy(b->mx_buffer);
        mdestroy(b->mx_consumer_count);
        smdetach();
}
```

The second program is an implementation for a consumer which reads characters from the shared buffer and prints it to the display. There can be more than one consumer program running concurrently. All consumer processes terminate gracefully.

```
/***********************************
* CONSUMER
***********************************/
#include "lib.h"

#define SM_KEY        36
#define BUFFER_SIZE   5

typedef struct {
        char slot[BUFFER_SIZE];
        int in, out;
        int n_consumers;
        mutex_t mx_buffer;
        mutex_t mx_consumer_count;
        sem_t sem_empty;
        sem_t sem_full;
        sem_t sem_done;
} SHARED_DATA;

void main() {
        int i;
        char c;

        SHARED_DATA *b = (SHARED_DATA *)smattach(SM_KEY, SM_READ_WRITE);
        if (b==NULL) {
              printf("No memory area to attach to.\n");
              return;
        }

        mlock(b->mx_consumer_count);
        b->n_consumers++;
        munlock(b->mx_consumer_count);

        do {
              sdown(b->sem_full);
              mlock(b->mx_buffer);

              // begin critical section
              c = b->slot[b->out];
              b->out = (b->out+1)%BUFFER_SIZE;
              if (c==0) {
                      b->n_consumers--;
                      munlock(b->mx_buffer);
                      sup(b->sem_empty);
                      break;
              }
              // end critical section

              munlock(b->mx_buffer);
              sup(b->sem_empty);

              printf("%c",c);
              sleep(300); // simulation: consumer using item
        } while (TRUE);

        sup(b->sem_done);
        smdetach();
  }
```