System and Device Programming - Lab 5

Professor P. Laface

Tamer Saadeh (222201) <tamer@tamersaadeh.com>

Politecnico di Torino - 28th April, 2015

Part I.

pack.sh

The following contains a commented version of the `pack.sh` script, which is a script that helps with compiling and running a custom UNIX-like OS.

```bash
#!/bin/bash

# check if parameters provided
if [ $# -ne 1 ]
then
echo "Usage: $0  number of k-bytes"
exit 1
fi

kbytes=$1
# clean log files
rm pack1.txt 1>/dev/null 2>/dev/null
rm pack2.txt 1>/dev/null 2>/dev/null

# save these to chmod files back
var_user=laface
var_group=laface

# Creates an empty file of size $kbytes*1024*512 with filled with zeros
# of: output file / if: input file / bs: block size / count: how many
# blocks / seek: skips forward into file
# The file created will have the following structure:
# [000..........000][000...000]
#  ^^^^^^^^^^^^^^^^  ^^^^^^^^
#  $kbytes*1024*512     512     bytes
dd if=/dev/zero of=hd.img bs=512 count=1 seek=$((kbytes*1024))

# mounts the image as a loop device in /dev/loop1
losetup /dev/loop1 hd.img

# The following will generate a file fdisk.input so that we can script
# the formating of disk.
# The meaning of the content of the file:
# x: use "expert" mode
# h: set the number of heads to 16
# s: set the number of sectors/tracks in the disk to 63
# c: set the number of cylinders to $kbytes
# r: return to main menu, ie exit "expert" mode
# n: creates a new physical partition numbered 1
# The next two lines use the default values for the new partion for the
# first cylinder location (ie 1) and the last cylinder (ie 256)
# a: set boot flag for the first partition
# w: write modifications to disk file (ie /dev/loop1 or hd.img)

cat <<EOF > fdisk.input
x
h
16
s
63
```

```
c
EOF

echo $kbytes >> fdisk.input

cat <<EOF >> fdisk.input
r
n
p
1


a
1
w
EOF

# actually run fdisk with the previous commands
fdisk hd.img < fdisk.input 1>>pack1.txt 2>>pack2.txt

# list partition tables in the disk we just created along with the size
# in sectors rather than cylinders
fdisk -l -u /dev/loop1 1>>pack1.txt 2>>pack2.txt

# extract the number of the blocks from the fdisk command
# tail -1 : keeps the last line
# tr -s " " : squeezes the spaces
# cut -d " " -f 5 : splits by space and gets the 5th field, ie the blocks
#                   number with the "+" sign at the end
# cut -d "+" -f 1 : get rid of the "+" sign so that we have a pure number
blocks=$(fdisk -u -l /dev/loop1 | tail -1 | tr -s " " | \
                cut -d " " -f 5 | cut -d "+" -f 1)

# detach the loop-ed disk image
losetup -d /dev/loop1

# remount the disk image with an offset as 63*512 so to skip the MBR
# data since these data are stored in the first track and limiting the
# size of the disk to the actual size (ie $blocks*1024). This is needed
# so that we know where to stop when creating the filesystem.
losetup -o $((63*512)) --sizelimit $(($blocks*1024)) /dev/loop0 hd.img \
        1>>pack1.txt 2>>pack2.txt

# creates an ext2 filesystem on the disk
mkfs.ext2 /dev/loop0

# create the grub boot menu item
# we set the title of te grub menu item to "minimal linux like kernel"
# we tell grub where to find the kernel we want to start and on which disk.
# Mount it as read-only, be quiet and display a splash screen for the kernel
# (if available). Finally, be quiet and don't print anything to the screen
# while loading the kernel.
cat <<EOF > menu.lst
```

```
title  minimal linux like kernel
kernel /boot/minimal_linux_like_kernel root=/dev/hda ro quiet splash
quiet

EOF

# mount /dev/loop0 to mount_point
mkdir mount_point 1>>pack1.txt 2>>pack2.txt
mount /dev/loop0 mount_point 1>>pack1.txt 2>>pack2.txt

# copy grub files from local hard drive
mkdir -p mount_point/boot/grub
cp ../grub/stage1 ../grub/stage2 menu.lst mount_point/boot/grub \
        1>>pack1.txt 2>>pack2.txt

# copy kernel
cp -v src/kernel mount_point/boot/minimal_linux_like_kernel \
        1>>pack1.txt 2>>pack2.txt

# umount mount_point
umount mount_point 1>>pack1.txt 2>>pack2.txt
rm -r mount_point 1>>pack1.txt 2>>pack2.txt

# detach /dev/loop0
losetup -d /dev/loop0  1>>pack1.txt 2>>pack2.txt

# creates the grub.input file for grub
# The following runs in GRUB shell, which means:
# device: specifices the device and the image to install grub on
# geometry: prints information about the device we specified with
#           the given cylinders, head and sector sizes
# root: sets the first partion as the root partition for the 2nd
#       stage of booting
# setup: installs grub to the MBR of the device
cat <<EOF > grub.input
device (hd0,0) hd.img
EOF
echo geometry \(hd0\) $kbytes 16 63 >> grub.input
cat <<EOF >> grub.input
root (hd0,0)
setup (hd0)
quit
EOF

../grub/grub.bin --device-map=/dev/null < grub.input \
         1>>pack1.txt 2>>pack2.txt

# Set permissions to our user to the logs and image
chown ${var_user}:${var_group} pack1.txt
chown ${var_user}:${var_group} pack2.txt
chown ${var_user}:${var_group} hd.img
chown ${var_user}:${var_group} menu.lst
```

Part II.

Custom Kernel for the x86 Architecture

1. Booting a custom kernel

To boot a kernel in the x86 architecture the image must provide a global descriptor table and handle interrupt service routines adhering to certain arrangements.

1.1. Global Descriptor Table (GDT)

The x86 architecture has 6 different types of memory segments: code (`cs`), data (`ds`), stack (`ss`), extra (`es`), and two general purpose segments (`fs` and `gs`). For the limit purpose of this simple custom kernel, all segments except the `cs` will be set to point to the same portion of memory (ie 0x10), which is further explained in 1.1.3 below.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Base address (bits 0-15)																Segment limit (bits 0-15)															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Base address (bits 24-31)								G	D	0	A	Limit (bits 16-19)				P	DPL		DT	T				Base address (bits 16-23)							

Table 1.1.: GDT memory structure

The above memory structure can be implemented using the following C code structure:

```c
struct gdt_entry_struct {
        u16int limit_low;
        u16int base_low;
        u8int  base_middle;
        u8int  access;
        u8int  granularity;
        u8int  base_high;
} __attribute__((packed));
```

Code snippet 1.1: GDT entry structure

Notice that since elements that are less than 8 bits wide, one has to combine the bits into a single variable and manage setting them manually using bitwise operations. The `__attribute__((packed))` tells the compiler (GCC in this case) not to rearrange the fields in the structure. This is important as we want the exact arrangement in memory as that's what the x86 architecture is expecting.

1.1.1. Granularity (bits 52-55)

Bit 52 is available for OS usage. Bit 53 is reserved to be zero always. Bit 54 describes the size of the segment limit whether it is 16-bit (when set to 0) or 32-bit (when set to 1). Bit 55, the granularity (G) bit, defines the multiplier of the limit register (segment limit); if set to 1 the multiplier is set to 4 Kilobytes (4096 bytes), otherwise the multiplier is 1 (not modified).

1.1.2. Access (bits 40-47)

The structure of the access flag byte is a bit signifying the presence (P, bit 47) or otherwise of a segment in memory, which is useful for virtual memory management. The next 2 bits for defining the privilege level of the descriptor (DPL, bits 46-47), which can take values between 0 and 3; this flag is sometimes called the ring level. When the ring level is set to 0 the process is the most privileged and runs in kernel-mode; on the other hand, ring level 3 is the least privileged, which is typically how most processes run in user-mode. This is very important as it enforces protection of certain CPU instructions from being run in higher ring levels. The following bit is the descriptor type bit, which if set means the the segment is a code/data segment, if not then it is a system segment. The next 4 bits represent the type of a segment (T); the first bit (bit 43) describes whether the segment is data or code, if set to 1, then the segment is a code segment. Bit 41 sets whether the segment is read only or read/write (for data segments) and execute only or execute/read (for code segments). This is important as it provides protections from process being possibly being exploited.

1.1.3. Code

```
gdt_set_gate(0, 0, 0, 0, 0);
gdt_set_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF);
gdt_set_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF);
gdt_set_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF);
gdt_set_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF);

// assembly function call
gdt_flush((u32int)&gdt_ptr);
```

Code snippet 1.2: GDT segment creation

The `gdt_set_gate()` function is a helper function that simplifies loading 5 code and data segments into memory using the x86 GDT structure presented earlier. It manipulates the parameters and places them in an array (`gdt_entries`). 0xFFFFFFFF is hexadecimal for about 4 billion (or 2^{32}), which is the maximum memory size an x86 system can has due to the base address being 32 bits. However, the first GDT entry must be made of all zeros this is called the null segment; a system will not boot without this segment.

The “access” parameter is explained below for each segment (after converting from hexadecimal to binary):

P	DPL		DT	T			
47	46	45	44	43	42	41	40
1	0	0	1	1	0	1	0

Table 1.2.: Kernel-mode code segment (0x9A=1001 1010_2)

P	DPL		DT	T			
47	46	45	44	43	42	41	40
1	0	0	1	0	0	1	0

Table 1.3.: Kernel-mode data segment (0x92=1001 0010_2)

P	DPL		DT	T			
47	46	45	44	43	42	41	40
1	1	1	1	1	0	1	0

Table 1.4.: User-mode code segment (0xFA=1111 1010_2)

P	DPL		DT	T			
47	46	45	44	43	42	41	40
1	1	1	1	0	0	1	0

Table 1.5.: User-mode data segment (0xF2=1111 0010_2)

As can be seen in the above tables bit 43 is changed to differentiate between data and code segments. While bits 45 and 46, are used here to separate the kernel from the user memory segments.

P	D		A
55	54	53	52
1	1	0	0

Table 1.6.: Granularity (0xCF & 0xF0=1100 0000_2)

This means that the segments are 32-bit wide and the multiplier of the limit register is 4K.

```
[GLOBAL gdt_flush]   ; Allows the C code to call gdt_flush()

gdt_flush:
   mov eax, [esp+4]
   lgdt [eax]        ; Load the new GDT pointer

   mov ax, 0x10      ; 0x10 is the offset in the GDT to data segment
   mov ds, ax        ; Load all data segment selectors
   mov es, ax
   mov fs, ax
   mov gs, ax
   mov ss, ax
   jmp 0x08:.flush   ; 0x08 is the offset to code segment
.flush:
   ret
```

Code snippet 1.3: GDT memory loading

To load these segments one must tell the CPU where to find the GDT entries table in memory. This can be done using the `lgdt` instruction to load a pointer of the GDT to memory. Since each GDT entry is 8 bytes and the kernel-mode data segment is the third, its offset is 8 (null segment) + 8 (code segment) = 16 bytes, which is 0x10 in hexadecimal; so we set all our segments to point to it, except for the kernel-mode segment. Finally, we need to start executing our kernel-mode code segment, to do so we need to goto second segment we loaded (after 8 bytes; 0x08), using the `jmp` instruction.

1.2. Interrupt Descriptor Table (IDT)

Interrupts in the x86 architecture can be generated by the CPU when a fault or an exception occurs (eg division by zero) or by software using the `int` instruction. There are 256 possible values of interrupt signals, and all of these must be defined even if they are just set to null. If some of these are not defined (nor null), then the CPU will panic and reset, as it expects them to be available.

Exception/interrupt value		Exception/interrupt meaning
decimal	hexadecimal	
0	0x00	Division by zero
1	0x01	Debugger
2	0x02	Non maskable interrupt
3	0x03	Breakpoint
4	0x04	Overflow
5	0x05	Out of bounds
6	0x06	Invalid opcode
7	0x07	No coprocessor
8	0x08	Double fault
9	0x09	Coprocessor segment overrun
10	0x0A	Invalid task state segment (TSS)
11	0x0B	Segment not present
12	0x0C	Stack fault
13	0x0D	General protection fault
14	0x0E	Page fault
15	0x0F	Unknown interrupt
16	0x10	Coprocessor fault
17	0x11	Alignment check
18	0x12	Machine check
19-31	0x13-0x1F	Reserved

Table 1.7.: Interrupt values and their meanings, as defined by the x86 architecture

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Kernel segment selector																Interrupt address (bits 0-15)															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Interrupt address (bits 16-31)																P	DPL		Type					0	0	0	0	0	0	0	0

Table 1.8.: IDT memory structure

The above memory structure can be implemented using the following C code structure:

```c
struct idt_entry_struct {
        u16int base_lo
        u16int sel;
        u8int  always0;
        u8int  flags;
        u16int base_hi;
} __attribute__((packed));
```

Code snippet 1.4: IDT entry structure

1.2.1. Constants and Type (bits 32-44)

Bits 32-39 must always be set to zero. Bits 40-43 (Type) is the type of the interrupt and is typically set to 6 (0110_2) or 14 (1110_2), which mean they are of type interrupt gate entry. Bit 44 should be zero if bits 40-43 are of type interrupt gate. These constants are represented in the memory structure above.

1.2.2. DPL and P (bits 45-47)

The DPL bits define the minimum required privilege to be able to handle the interrupt. This makes sure certain interrupts are protected from user-space. On the other hand the P bit defines whether the interrupt is present (ie whether the interrupt is handled or not). If this bit is set to zero for a certain interrupt signal, then if this interrupt occurs the CPU generates an “interrupt not handled” exception.

1.2.3. Selector (bits 16-32)

These bits are the relative address in the GDT table of the kernel-mode code segment, which is 0x08 (as explained above 1.1.3).

1.2.4. Code

```
idt_set_gate(0, (u32int) isr0, 0x08, 0x8E);

// assembly function call
idt_flush((u32int)&idt_ptr);
```

Code snippet 1.5: IDT handler memory locations

The `idt_set_gate()` function is a helper function that simplifies loading of interrupt handlers into using the x86 IDT structure presented earlier. It manipulates the parameters and places them in an array (`idt_entries`). The first parameter is the index of the interrupt in the array; in other words the exception value that will be handled by the next parameter. The next parameter is a pointer to the interrupt service routine handler we want to run upon receiving a certain interrupt. The following parameter is the kernel segment selector. Finally the last parameter is the “`flags`”, which is represented by bits 40-47. These bits are inspected further below:

P	DPL		Type				
47	46	45	44	43	42	41	40
1	0	0	0	1	1	1	0

Table 1.9.: IDT flags (0x8E=1000 1110_2)

Bit 47 (P) is set because we are handling this exception. The DPL bits (45-46) are set to zero, which means to handle these exception a segment must have a privilege ring level 0 (ie only kernel-mode segments can handle these exceptions). Finally the type is set to 14 (1110_2), which as described earlier means this is an interrupt gate handler.

```
[GLOBAL idt_flush]   ; Allows the C code to call idt_flush()

idt_flush:
   mov eax, [esp+4]
   lidt [eax]        ; Load the IDT pointer
   ret
```

Code snippet 1.6: IDT memory loading

To load these interrupt service routines one must tell the CPU where to find the IDT entries table in memory. This can be done using the `lidt` instruction to load a pointer of the IDT to memory. Afterwards, the CPU knows that exceptions from 0 to 31 are handled and should call each handler respectively.

2. Monitor

GRUB boots the kernel in text mode, which means the kernel can directly access the framebuffer of the screen, as a set of characters directly mapped to memory. The screen size provided by GRUB is 80 characters wide by 24 characters long. It can be accessed by writing directly to the 0xB8000 address in memory that is mapped to the video memory of the VGA controller, which is not necessarily the RAM. The VGA has two important registers the control (0x3D4) and data (0x3D5) registers that can be used to update the location of the cursor. The framebuffer is an array of 16-bit elements using the following structure for each element:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Background color				Foreground color				ASCII character code							

Table 2.1.: Framebuffer element structure

The color values are 8 bits wide each, as seen in the table above. which means that only 16 colors can be displayed. This leaves another 8 bits for the character code, as such only ASCII characters can be displayed using this method.

Each character on the screen can be accessed using the following formula:

$$character\ address\ (x, y) = B8000_{16} + (80_{10} \cdot y + x)$$

For example, if we want to write a character at column 2 (y = 3) and row 4 (x = 5); we will have to write the 16-bit character element to address 0xB80F5$_{16}$.

3. Debugging the kernel

The easiest method of debugging this minimal kernel is by using tools such as gdb. As GRUB hands control to the system to our kernel, it is useful to set breakpoints at `gdt_flush` and `idt_flush` because that way one could inspect the values and contents of the GDT and the IDT elements being loaded into memory. After initialization of the GDT and IDT, setting breakpoints at `monitor_write` and `isr_handler`. Setting a breakpoint at `monitor_write` helps in knowing what is being printed to the screen and when. Moreover, setting a breakpoint at `isr_handler`, helps in reading the interrupt values being received and that will be printed using the monitor set of functions.