General theme for the next two weeks

• In a complex system …

    App
    Operating System
    Microkernel
    Hardware

• Question: how can we protect individual programs from interference with themselves, or with one another, either directly or by subverting lower layers?

• General approach: leverage programmable hardware features!

Diagrams and Code

• There are a lot of diagrams on these slides
  
  • Many of these are taken directly from the “Intel® 64 and IA-32 Architectures Software Developer’s Manual”, particularly Volume 3
  
  • There is a link to the full pdf file in the Reference section
  
  • There is also a lot of code on these slides
  
  • Remember that you can study these more carefully later if you need to!

Taking stock: Code samples … so far

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vram.tar.gz</td>
<td>video RAM simulation</td>
</tr>
<tr>
<td>hello.tar.gz</td>
<td>boot and say hello on bare metal, via GRUB</td>
</tr>
<tr>
<td>simpleio.tar.gz</td>
<td>a simple library for video RAM I/O</td>
</tr>
<tr>
<td>bootinfo.tar.gz</td>
<td>display basic boot information from GRUB</td>
</tr>
<tr>
<td>mimg.tar.gz</td>
<td>memory image bootloader &amp; make tool</td>
</tr>
<tr>
<td>example-mimg.tar.gz</td>
<td>display basic boot information from mimgload</td>
</tr>
<tr>
<td>example-gdt.tar.gz</td>
<td>basic demo using protected mode segments (via a Global Descriptor Table)</td>
</tr>
<tr>
<td>example-idt.tar.gz</td>
<td>context switching to user mode (via an Interrupt Descriptor Table)</td>
</tr>
<tr>
<td>baremetal.tar.gz</td>
<td>display basic boot information from baremetalloader</td>
</tr>
<tr>
<td>prot.tar.gz</td>
<td>basic demo using protected mode</td>
</tr>
</tbody>
</table>

Segmentation

(or: where do “seg faults” come from?)
Breaking the 64KB barrier …

- The 8086 and 8088 CPUs in the original IBM PCs were 16 bit processors: in principle, they could only address 64KB
- Intel used segmentation to increase the amount of addressable memory from 64KB to 1MB:

We can relocate these segments to different physical memory:

Programs can be organized to use multiple segments:
- For example:
  - One segment for the stack
  - One segment for code
  - One segment for data
  - We can relocate these segments to different physical addresses, just by adjusting the segment registers

How are segments chosen

- The default choice of segment register is determined by the specific kind of address that is being used:

<table>
<thead>
<tr>
<th>Reference Type</th>
<th>Register Used</th>
<th>Segment Used</th>
<th>Default Selection Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>CS</td>
<td>Code Segment</td>
<td>All instruction fetches.</td>
</tr>
<tr>
<td>Stack</td>
<td>DS</td>
<td>Stack Segment</td>
<td>All stack pushes and pops. Any memory reference which uses the ESP or EBP register as a base.</td>
</tr>
<tr>
<td>Local Data</td>
<td>DS</td>
<td>Data Segment</td>
<td>All data references, except when relative to stack or string destination.</td>
</tr>
<tr>
<td>Destination String</td>
<td>ES</td>
<td>Data Segment (pointed to with the ES register)</td>
<td>Destination of string instructions.</td>
</tr>
</tbody>
</table>

- If a different segment register is required, a single byte “segment prefix” can be attached to the start of the instruction

Back to breaking the 64KB barrier …

- Programs can be organized to use multiple segments:
  - For example:
    - One segment for the stack
    - One segment for code
    - One segment for data
    - We can relocate these segments to different physical addresses, just by adjusting the segment registers

Variations on the theme

- Programs can have multiple code and data segments
  - Programmers could use a standard “memory model”
  - Or use custom approaches to suit a specific application
- The machine provides special “far call” and “far jump” instructions that change CS and EIP simultaneously, allowing control transfers between distinct code segments
- There are six segment registers, so programs can have up to 6 active segments at a time (and more by loading new values into the segment registers)
- Segments do not have to be exactly 64KB
- If segments do not overlap, then a stack overflow will not corrupt the contents of other segments - protection!
Accommodating multiple programs

640KB User Memory  
BIOS, Video RAM, etc...

- Now we can have multiple programs in memory at the same time, each with distinct code, data, and stack segments
- But what is to stop the code for one program from accessing and/or changing the data for another?
- Nothing!
- We would like to “protect” programs for interfering with one another, either by accident or design …

Control registers

The current mode

- The current mode is saved in the two least significant bits of the CS register
- The value in CS can only be changed by a limited set of instructions (e.g., it cannot be the target of a movw), each of which performs a privilege check, if necessary, triggering a CPU exception if a violation occurs
- End result: user mode code cannot change its own privilege level to move out of Ring 3!

Segments in protected mode

Protection!

- Ring 0 is sometimes called “supervisor” or “kernel mode”
- Ring 3 is often called “user mode”
Segment registers hold segment selectors

```
<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>Base Address</th>
<th>0</th>
<th>Limit</th>
<th>0</th>
<th>Access Information</th>
</tr>
</thead>
</table>
```

The descriptor cache

```
<table>
<thead>
<tr>
<th>Segment Selector</th>
<th>Base Address</th>
<th>Limit</th>
<th>Access Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS</td>
<td>8</td>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>4</td>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td>2</td>
<td>ES</td>
<td></td>
</tr>
</tbody>
</table>
```

Global and local descriptor tables

```
<table>
<thead>
<tr>
<th>Global Descriptor Table (GDT)</th>
<th>Local Descriptor Table (LDT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDTTR Register</td>
<td>LDTR Register</td>
</tr>
</tbody>
</table>
```

Segments and capabilities

- The GDT and LDT for a given user mode program determine precisely which regions of memory that program can access.
- As such, these entries are an example of a `capability` mechanism.
- The user mode program refers to segments by their index in one of these tables, but it has no access to the table itself:
  - It cannot, in general, determine which regions of physical memory they are accessing.
  - It cannot “fake” access to other regions of memory.
- **The principle of least privilege**: limit access to the minimal set of resources that are required to perform a task.

What if we don’t want to use segments?

- Segmentation cannot be disabled in protected mode.
- But we can come pretty close by using segments with:
  - base address 0
  - length = 4GB
- A common GDT structure:
  - (e.g., in Linux, etc., with no LDT)
Deja vu?

```
movl base, teax
movl limit, tebx
mov teax, tebx
shl $16, teax
mov tdx, teax
mov teax, low
shr $16, teax
mov tdx, tecx
andl $0xff, tecx
xorl tecx, tdx
shr tecx, 4
orl tecx, teax
andl $0x503200, tebx
orl tebx, tdx
movl tebx, high
```

```
Or:

```
lgr dt ptr
ljmp $KERN_CS, $1f
```

## Initializing the GDT entries

```
initGDT: # Kernel code segment:
gdtset name=KERN_CS, slot=4, dpl=0, type=GDT_CODE, \base=0, 1limit=0xffffffff, gran=1

# Kernel data segment:
gdtset name=KERN_DS, slot=5, dpl=0, type=GDT_DATA, \base=0, 1limit=0xffffffff, gran=1

# User code segment:
gdtset name=USER_CS, slot=6, dpl=3, type=GDT_CODE, \base=0, 1limit=0xffffffff, gran=1

# User data segment:
gdtset name=USER_DS, slot=7, dpl=3, type=GDT_DATA, \base=0, 1limit=0xffffffff, gran=1

# TSS

gdtset name=TSS, slot=3, dpl=0, type=GDT_TSS12, \base=tss, limit=tss_len-1, gran=8
```

```
Activating the GDT

```
lgr dt ptr
ljmp $KERN_CS, $1f
```

```
# load code segment
mov $KERN_DS, hax
mov $KERN_CS, hax
mov ax, hax
mov ax, hax
mov ax, hax
mov ax, hax
mov $TSS, hax
ltr hax
ret
```

## Calculating GDT descriptors

```
# macro assembler
```

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
The Task State Segment

<table>
<thead>
<tr>
<th>Segment Address</th>
<th>Description</th>
<th>Type</th>
<th>Flags</th>
<th>Spare</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x00000001</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x00000002</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x00000003</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x00000004</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x00000005</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x00000006</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x00000007</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x00000008</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x00000009</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x0000000A</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x0000000B</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x0000000C</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x0000000D</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x0000000E</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
<tr>
<td>0x0000000F</td>
<td>Code Limit</td>
<td>Fault</td>
<td>No</td>
<td>Code</td>
</tr>
</tbody>
</table>

Implementing the TSS

Exceptions

- What happens if the program you run on a conventional desktop computer attempts:
  - division by zero?
  - to use an invalid segment selector?
  - to reference memory beyond the limits of a segment?
  - etc...
- What happens when there is no operating system to catch you?

Hardware and software interrupts

- **Hardware**: devices often generate interrupt signals to inform the kernel that a certain event has occurred:
  - a timer has fired
  - a key has been pressed
  - a buffer of data has been transferred
  - ...
- **Software**: User programs often request services from an underlying operating system:
  - read data from a file
  - terminate this program
  - send a message
  - ...
- These can all be handled in the same way ...
The interrupt vector

Interrupt Vector

Segment Selector

Interrupt or Trap Gate

GDT or LDT

Interrupt Descriptor

Base Address

Offset

Interrupt Procedure

Destination Code Segment

IDT

Interrupt

Figure 6-3. Interrupt Procedure Call

Storage for the IDT

.set IDT_ENTRIES, 256  # Allow for all poss. interrupts
.set IDT_SIZE, 8*IDT_ENTRIES  # Eight bytes for each idt descr.
.set IDT_INTR, 0x000  # Type for interrupt gate
.set IDT_TRAP, 0x100  # Type for trap gate

.data
.space IDT_SIZE, 0

.idptr: .short IDT_SIZE-1
.long idt

.ready to begin?

lidi idptr $idt

Calculating IDT descriptors

.macro idtcalc desc, slot, dpl=0, type=IDT_INTR, seg=KERN_CS
    # type = 0x000 (IDT_INTR) -> interrupt gate
    # type = 0x100 (IDT_TRAP) -> trap gate
    # The following comments use # for concatenation of bitfields
    mov $seg, %ax  # eax = #    # seg
    shl 1, %dx    # eax = seg #    # slo
    movi 0x8000 | (dpl<13) | (type), %edx
    movi DPL, %dx  # edx = DPL # slo
    .endm

Initializing and activating the IDT

initIDT: # Fill in IDT entries
    # Add descriptors for protected mode exceptions:
    .iret num, 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,16,17,18,19
    idtcalc exclnum, slot=num
    .endr
    # Add descriptors for hardware irqs:
    # ... except there aren't any here (yet)
    # Add descriptors for system calls:
    # These are the only idt entries that we will allow to be
    # called from user mode without generating a general
    # protection fault, so they are tagged with dpl=3.
    idtcalc handler<kpu:ct, slot=0x80, dpl=3
    # Install the new IDT:
    lidi idptr
    ret

Transferring control to a handler

Stack Usage with Privilege-Level Change

Interrupted Procedure's Stack

ESR Before Transfer to Handler

EFLAGS

Stack Usage with No Privilege-Level Change

Interrupted Procedure's Stack

ESR Before Transfer to Handler

EFLAGS

Handler's Stack

Stack Usage on Transfers to Interrupt and Exception-Handling Routines

Interrupted Procedure's Stack

ESR Before Transfer to Handler

EFLAGS

Handler's Stack

Error Code

Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines
## Contexts

```
void *esp
```
Defining a family of (non) handlers

```
.nmacro handler num, func, errorcode=0
    .ex
    .num
    .if errorcode=0
    sub $4, %esp
    # Save segments
    add $4, %esp
    # Save registers
    push %gs
    movl %gs, %eax
    call %func
    call func(struct Context *esp)
    .endif
.endm
```

Some exceptions do not generate an error code ...

```
1: bhi
jmp lb
```

```
ret
exected:
.asciz "Exception 0x%x, frame=0x%x"
```

Defining a family of (non) handlers

```
nohandler:  # dummy interrupt handler
    movl 4(%esp), %ebx
    call %func
    call func(struct Context *esp)
```

```
extern void initContext(struct Context *ctxt, unsigned eip, unsigned esp)
```

```
void initContext(struct Context* ctxt, unsigned eip, unsigned esp) {
    extern char USER_DS[];
    extern char USER_CS[];
    printf("user code segment is 0x%lx", (unsigned)USER_DS);
    printf("user data segment is 0x%lx", (unsigned)USER_CS);
    ctxt->segs.ds = (unsigned)USER_DS;
    ctxt->segs.fs = (unsigned)USER_DS;
    ctxt->segs.gs = (unsigned)USER_DS;
    ctxt->iret.es = (unsigned)USER_CS;
    ctxt->iret.esp = esp;
    ctxt->iret.esp = edx;
    ctxt->iret.eflags = INIT_USER_FLAGS;
}
```

Initializing a context

```
#define CONTEXT_SIZE 72
```

```
extern int switchToUser(struct Context* ctxt);
```

```
.switchToUser:
    movl 4(%esp), %eax
    # Load address of the user context
    movl %eax, %esp
    # Reset stack to base of user context
    addl $CONTEXT_SIZE, %esp
    # Set stack address for kernel reentry
    pop %ax
    pop %ds
    # Restore segments
    pop %fs
    pop %gs
    addl $4, %esp
    # Skip error code
    i ret
```

Switching to a user program

From C:
```
extern int switchToUser(struct Context* ctxt);
```

To Assembly:
```
    .sect CONTEXT_SIZE, 72
    .globl switchToUser
switchToUser:
    movl 4(%esp), %eax  # Load address of the user context
    movl %eax, %esp    # Reset stack to base of user context
    addl $CONTEXT_SIZE, %esp  # Set stack address for kernel reentry
    popa
    pop %edx
    pop %es
    pop %fs
    addl $4, %esp  # Skip error code
    iret
```

Initializing the flags

```
#define INIT_USER_FLAGS (3<<12 | 1<<9 | 1<<1)
```

```
    ID — Identification Flag
    VIF — Virtual Interrupt Flag
    AC — Alignment Check
    VM — Virtual-8086 Mode
    RF — Resume Flag
    NT — Nested Task Flag
    IOPL — I/O Privilege Level
    IF — Interrupt Enable Flag
    TF — Trap Flag
```

```
    Reserved (set to 0)
```

```
Figure 2-5. System Flags in the EFLAGS Register
```
Entering a system call (kernel view)

Initialize IDT entry:
```
    idtcalc handler=kputc, slot=0x80, dpl=3
```

Define a stub to handle the interrupt:
```
.text

kputc:
    subl $4, %esp    # Fake an error code
    push %gs        # Save segments
    push %fs
    push %es
    push %ds
    pusha           # Save registers
    leal stack, %esp # Switch to kernel stack
    jmp kputc_imp
```

Provide a handler implementation:
```
void kputc_imp() {
    /* A trivial system call */
    putchar(user.regs.eax);
    switchToUser(&user);
}
```

Why is this line so important?

A recipe for adding a new system call

- Pick an unused interrupt number.
- Add code to initialize the corresponding IDT entry.
- Write and assembly code stub that saves the user program context and jumps to the handler code.
- Write the implementation of the handler. Be sure to use switchToUser (or equivalent) when the handler is done.
- Add user-level code to access the new system call. This often requires an assembly code fragment using the int instruction, and a declaration/prototype in the C code.
- Color key for example-idt:
  ```
  kernel/init.s  kernel/kernel.c  user/userlib.s  user/user.c
  ```

Reflections

- Bare Metal
  - Segmentation, protection, exceptions and interrupts
- Programming/Languages
  - Representation transparency, facilitates interlanguage interoperability
- Memory areas
  - Vendor-defined layout: GDT, GDTTR, TSS, IDT, IDTR, IRet, Registers, …
  - Self-defined: Context, …
- “Bitdata”
  - Segment and interrupt descriptors, eflags, cr0, …
- Does the need for a “recipe” suggest a language weakness?

Let’s see how all the pieces fit together …