ICS143A: Principles of Operating Systems

Midterm recap, sample questions

Anton Burtsev
November, 2017
Describe the x86 address translation pipeline (draw figure), explain stages.
Address translation

CPU -> Selector -> Offset -> Logical Address -> Segment Translation -> Linear Address -> Page Translation -> Physical Address -> RAM
Logical Address (or Far Pointer) → Segment Selector

Global Descriptor Table (GDT) → Segment Descriptor

Segment Base Address → Linear Address Space → Segment

Segment → Page Directory

Page Directory → Page Table

Page Table → Page

Page → Physical Address Space

Segmentation → Paging
What is the linear address? What address is in the registers, e.g., in %eax?
Logical and linear addresses

- Segment selector (16 bit) + offset (32 bit)
What segments do the following instructions use? push, jump, mov
Programming model

- Segments for: code, data, stack, “extra”
  - A program can have up to 6 total segments
  - Segments identified by registers: cs, ds, ss, es, fs, gs

- Prefix all memory accesses with desired segment:
  - `mov eax, ds:0x80` (load offset 0x80 from data into eax)
  - `jmp cs:0xab8` (jump execution to code offset 0xab8)
  - `mov ss:0x40, ecx` (move ecx to stack offset 0x40)
Segmented programming (not real)

```c
static int x = 1;
int y; // stack
if (x) {
    y = 1;
    printf ("Boo");
} else {
    y = 0;
}
```

```c
ds:x = 1; // data
ss:y; // stack
if (ds:x) {
    ss:y = 1;
    cs:printf(ds:"Boo");
} else {
    ss:y = 0;
}
```
Describe the linear to physical address translation with the paging mechanism (use provided diagram, mark and explain the steps).
Page translation

Linear Address

31  22  21  12  11  0
Directory  Table  Offset

Page Directory

10
PDE with PS=0

10
Page Table

12
4-KByte Page

12
Physical Address

20
PTE

32
CR3
Page translation

Linear Address

31 22 21 12 11 0
Directory Table Offset

4-KByte Page

12

Page Directory

Page Table

PTE

Physical Address

PDE with PS=0

CR3
Page directory entry (PDE)

| 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 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Address of page table | Ignored | 0 | I g n | A | P | C | D | P | W | U | S | R | W | 1 | PDE: page table |

- 20 bit address of the page table
  - Pages 4KB each, we need 1M to cover 4GB
- R/W – writes allowed?
  - To a 4MB region controlled by this entry
- U/S – user/supervisor
  - If 0 – user-mode access is not allowed
- A – accessed
Page translation

- Linear Address
  - Directory
  - Table
  - Offset

- 4-KByte Page
- Physical Address

- Page Directory
  - PDE with PS=0
  - CR3

- Page Table
  - PTE
# Page table entry (PTE)

| Address of 4KB page frame | Ignored | G | P | A | T | D | A | P | C | D | P | W | U | S | R | I | W | 1 |
|---------------------------|---------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB
- R/W – writes allowed?
  - To a 4KB page
- U/S – user/supervisor
  - If 0 user-mode access is not allowed
- A – accessed
- D – dirty – software has written to this page
Page translation

Linear Address

31  22  21  12  11  0
Directory  Table  Offset

Page Directory

PDE with PS=0

CR3

Page Table

PTE

Physical Address

4-KByte Page
Consider the following 32-bit x86 page table setup. %cr3 holds 0x00001000.
The Page Directory Page at physical address 0x00001000:
PDE 0: PPN=0x00002, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00003, PTE_P, PTE_U, PTE_W
PDE 2: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
The Page Table Page at physical address 0x00002000 (which is PPN 0x00002):
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00006, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
The Page Table Page at physical address 0x00003000:
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00005, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
List all virtual addresses that map to physical address 0x00005555
Consider the following 32-bit x86 page table setup. %cr3 holds 0x00001000.

The Page Directory Page at physical address 0x00001000:
PDE 0: PPN=0x00002, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00003, PTE_P, PTE_U, PTE_W
PDE 2: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero

The Page Table Page at physical address 0x00002000 (which is PPN 0x00002):
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00006, PTE_P, PTE_U, PTE_W
... all other PTEs are zero

The Page Table Page at physical address 0x00003000:
PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00005, PTE_P, PTE_U, PTE_W
... all other PTEs are zero

List all virtual addresses that map to physical address 0x00005555
Answer: 0x00000555, 0x00400555, 0x00401555, 0x00800555
What's on the stack? Describe layout of a stack and how it changes during function invocation?
### Example stack

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>[ebp + 16]</td>
<td>(3rd function argument)</td>
</tr>
<tr>
<td>5</td>
<td>[ebp + 12]</td>
<td>(2nd argument)</td>
</tr>
<tr>
<td>2</td>
<td>[ebp + 8]</td>
<td>(1st argument)</td>
</tr>
<tr>
<td>RA</td>
<td>[ebp + 4]</td>
<td>(return address)</td>
</tr>
<tr>
<td>FP</td>
<td>[ebp]</td>
<td>(old ebp value)</td>
</tr>
<tr>
<td></td>
<td>[ebp - 4]</td>
<td>(1st local variable)</td>
</tr>
<tr>
<td></td>
<td>[ebp - X]</td>
<td>(esp - the current stack pointer)</td>
</tr>
</tbody>
</table>
Describe the steps and data structures involved into a user to kernel transition (draw diagrams)
What segment is specified in the interrupt descriptor? Why?
Interrupt descriptor

Interrupt Gate

31  16  15  14  13  12  8  7  5  4  0

Offset 31..16

P  D  PL  0  D  1  1  0  0  0  0  4

Segment Selector

Offset 15..0

0
• Interrupt gate disables interrupts
  • Clears the IF flag in EFLAGS register
• Trap gate doesn't
  • IF flag is unchanged
Which stack is used for execution of an interrupt handler? How does hardware find it?
Interrupt path

Process

User stack of a process (can grow up to 2GBs)

User state (saved by hardware)

EBP

Last stack frame

SS

ESPN

EFLAGS

CS

EIP

Kernel Stack of a process (4K)

Interrupt Vector #

Timer: IRQ0 -> vector 32

Interrupt code

kernel code

Page table

Level 1

Level 2

0 - 4MB
4 - 8MB

2GB - 2GB + 4MB

GDT

IDT

NULL: 0x0
KCODE: 0 - 4GB
KDATA: 0 - 4GB
K_CPU: 4 bytes
CODE: 0 - 4GB
DATA: 0 - 4GB
TSS: sizeof(tss)

CS : HANDLER ADDR

TSS

ss:

esp:

sps:

CR3: pt

Argument 1

Argument 2

Calling EIP ++

Old EBP

Saved local values, e.g. push EAX, etc
Describe organization of the memory allocator in xv6?
Physical page allocator

Virtual

0x80000000 2GB

end (0x801126fc)

kmem.freelist

CS : 0x8    EIP: main
SS : 0x10   ESP: stack
GDT: 0x7c78 TSS: 0x0
IDT: 0x0    CR3: entrypgdir

Protected Mode

2GB + 4MB
0x80400000
Where did free memory came from?
swtch in xv6 doesn’t explicitly save and restore all fields of struct context. Why is it okay that swtch doesn’t contain any code that saves %eip?
swtch:

movl 4(%esp), %eax
movl 8(%esp), %edx

# Save old callee-save registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stacksh
movl %esp, (%eax)
movl %edx, %esp

# Load new callee-save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
popl %ebp
ret

swtch()

struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
Stack inside `swtch()`

User state (saved by hardware):
- SS
- ESP
- EFLAGS
- CS
- CS
- CS
- 0
- 32
- DS
- ES
- FS
- GS
- All registers
- ESP

Call stack:
- vector32()
- alltraps()
- trap()
- yield()
- sched()
- switch(&proc->context, cpu->scheduler)

Kernel Stack of a process (4K)
Trap frame

Context:
- EIP (line: 2479)
- EBP
- EBX
- ESI
- EDI

Proc
Context
&proc->context
cpu->scheduler
EIP (sched)
Suppose you wanted to change the system call interface in xv6 so that, instead of returning the system call result in EAX, the kernel pushed the result on to the user space stack. Fill in the code below to implement this. For the purposes of this question, you can assume that the user stack pointer points to valid memory.
void syscall(void) {
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n", proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
void syscall(void)
{
    int num;

    num = proc->tf->eax;
    if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
        // proc->tf->eax = syscalls[num]();
        proc->tf->esp -= 4;
        *(int*)ptoc->tf->esp = syscalls[num]();
    } else {
        cprintf("%d %s: unknown sys call %d\n",
                proc->pid, proc->name, num);
        // proc->tf->eax = -1;
        proc->tf->esp -= 4;
        *(int*)ptoc->tf->esp = -1;
    }
}
Thank you!