ICS143A: Principles of Operating Systems

Midterm recap, sample questions

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

PDE with PS=0

CR3

Page Table

PTE

Physical Address

4-KByte Page

12

10

10

20

20
Page translation

```
  Linear Address
     31  22  21  12  11  0
   Directory  Table  Offset

  12
   4-KByte Page

  10
   Page Table

  10
   Page Directory

  32
   CR3

  20
   PDE with PS=0

  20
   PTE

  20
   Physical Address
```
Page directory entry (PDE)

- 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

Page Directory

PDE with PS=0

Page Table

PTE

Physical Address

4-KByte Page

CR3
Page table entry (PTE)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of 4KB page frame</td>
<td>Ignored</td>
<td>G</td>
<td>P</td>
<td>A</td>
<td>T</td>
<td>D</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>U</td>
<td>S</td>
<td>R</td>
<td>I</td>
<td>W</td>
<td>1</td>
<td>PTE: 4KB page</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

Page Table

PTE

Physical Address

4-KByte Page

CR3
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

| 10  | [ebp + 16]  (3rd function argument) |
|  5  | [ebp + 12]  (2nd argument)          |
|  2  | [ebp + 8]   (1st argument)          |
| RA  | [ebp + 4]   (return address)        |
| FP  | [ebp]       (old ebp value)         |
|     | [ebp - 4]   (1st local variable)    |
|     | [ebp - X]   (esp - the current stack pointer) |
Describe the steps and data structures involved into a user to kernel transition (draw diagrams)
Interrupt path

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

Interrupt Vector #

Timer: IRQ0 -> vector 32

Kernel code

Page table
Level 1
0 - 4KB
4 - 4KB

Level 2
2GB - 2GB + 4MB

0 - 4K
4K - 8K

(4MB-4K) - 4MB

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

IDT
CS : HANDLER ADDR
....
....

TSS
SS:
SS9:
ESPR:
....

Kernel Stack of a process (4K)

Last stack frame

User state (saved by hardware)

EBP →

Process

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc

ESP →

SS
ESP
EFLAGS
CS
EIP

interrupt code

CS : #1
SS: #2
GDT: gdt
TSS: tss
IDT: idt
CR3: pt

vector32
What segment is specified in the interrupt descriptor? Why?
Interrupt descriptor

Interrupt Gate

<table>
<thead>
<tr>
<th>31</th>
<th>16 15 14 13 12</th>
<th>8 7 5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Offset 31..16</td>
<td>P DPL</td>
<td>0 D110 000</td>
</tr>
<tr>
<td>31</td>
<td>16 15</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Segment Selector</td>
<td>Offset 15..0</td>
<td></td>
</tr>
</tbody>
</table>
- 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)

Code, data, heap

Timer: IRQ0 -> vector 32

Interrupt Vector #

Kernel Stack of a process (4K)

Kernel code

User state (saved by hardware)

EBP →

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc

Last stack frame

ESP →

SS
ESP
EFLAGS
CS
EIP

Kernel Stack

Page table

Level 1

Level 2

GDT

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

IDT

CS: HANDLER ADDR
... 
...

TSS

SS:
SS9:
ESP0:
...

CPU:

CS : #1
SS : #2
GDT: gdt
TSS: tss
IDT: idt
CR3: pt

Interrupt code

EBP
Describe organization of the memory allocator in xv6?
Where did free memory came from?
Describe how a per-CPU variables are stored in xv6?
Tiny segment (8 bytes), two pointers

struct cpu cpus[MAX_CPU]

GDT
- NULL: 0x0
- KCODE: 0 - 4GB
- KDATA: 0 - 4GB
- K_CPU: 8 bytes
- CODE: 0 - 4GB
- DATA: 0 - 4GB
- TSS: sizeof(ts)

GDT
- NULL: 0x0
- KCODE: 0 - 4GB
- KDATA: 0 - 4GB
- K_CPU: 8 bytes
- CODE: 0 - 4GB
- DATA: 0 - 4GB
- TSS: sizeof(ts)

GDT: GS:
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():

```assembly
2958 swtch:
2959 movl 4(%esp), %eax
2960 movl 8(%esp), %edx

2962 # Save old callee-save registers
2963 pushl %ebp
2964 pushl %ebx
2965 pushl %esi
2966 pushl %edi

2968 # Switch stacksh
2969 movl %esp, (%eax)
2970 movl %edx, %esp

2972 # Load new callee-save registers
2973 popl %edi
2974 popl %esi
2975 popl %ebx
2976 popl %ebp
2977 ret
```

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

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

User state (saved by hardware):
- SS
- ESP
- EFLAGS
- CS
- CS
- 0
- 32
- DS
- ES
- FS
- GS
- All registers
- ESP
- EIP (alltraps)
  ...
- EIP (trap)
- ...
- EIP (yield)
  ...
- &proc->context
- cpu->scheduler
- EIP (sched)

Kernel Stack of a process (4K)

Trap frame

Proc

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