Operating System Organization

원유집

KAIST EE
Operating system organization

- The three requirements of operating system
  - multiplexing
  - isolation
  - interaction
Register(x86-32)

- Program Counter: eip (extended instruction pointer)
- Control register: cr0, cr1, cr2, cr3
- Segment Register: cs, ds, es, fs, gs, ss
- GDT/LDT register: gdtr, ldtr
- General Purpose Register: eax, ebx, ecx, edx, edi, esi, ebp, esp
- 80-bit floating-point register: fp0, fp1, fp2, fpr3, fp4, fp5, fp6, fp7
- Debug register: dr0, dr1, dr2, dr3
multiplexing
isolation

Isolation

\[ \text{Chrome} \]
\[ \text{gcc} \]
\[ \text{email} \]

\[ \text{int } i; \]
\[ \text{int } i; \]
\[ \text{int } i; \]

\[ \text{405} \]
\[ \text{405} \]
\[ \text{405} \]
interaction
Abstracting physical resources

Applications can directly interact with hardware resources and use those resources in the best way for the application (e.g., to achieve high or predictable performance).

Some operating systems for embedded devices or real-time systems are organized in this way.

The downside of this library approach: the applications must be well-behaved.

- For example, each application must periodically give up the processor so that other applications can run. Such a cooperative time-sharing scheme may be OK if all applications trust each other and have no bugs.
- Applications do not trust each other, and have bugs. One often wants stronger isolation than a cooperative scheme provides.
Abstraction layer
Abstraction layer (Cont.)

OS: Abstraction Layer

Word

Spreadsheet

write

write

Word

Spreadsheet

write

write

OG
Abstraction layer: file system

- Open
- Close
- Read
- Write

- Block allocation
- Location of a file
- Path name resolution
Abstraction layer: memory management
User mode / Kernel mode

- Strong isolation requires a hard boundary between applications and operating systems. If the application makes a mistake, we do not want the operating system to fail.

- Processors provide hardware support for strong isolation.
  - the x86 processor, has two modes in which the processor can execute instructions: kernel mode & user mode.
  - In kernel mode, the processor is allowed to execute privileged instructions.
  - An application can execute only user-mode instructions (e.g., adding numbers, etc.) and is said to be running in user space, while the software in kernel mode can also execute privileged instructions and is said to be running in kernel space.

- The software running in kernel space (or in kernel mode) is called the kernel.
User mode vs. Kernel mode

- **User Mode**: access only user space
- **Kernel Mode**: access any address.

```c
int i;
*i = 0;
*i = 0;
```

**Seg. fault**

Mode switch

```
int i;
*i = 0;
*i = 0;
```

**But, can be worse!**
System call

Before starting a system call, validate Argument!
**int instruction**

- **Mode switch**
  - An application that wants to read or write a file on disk must transition to the kernel.

- Once the processor has switched to kernel mode, the kernel can then validate the arguments of the system call, decide whether the application is allowed to perform the requested operation, and then deny it or execute it.
System Call (Cont.)

```
j ADDR ; // change eip to ADDR.
```

`function call`

`entry:
ADDR
```
System Call (Cont.)

System Call

\[ J \text{ OxooooFAC00} \]

// check argument

OxooooFAB00

OxooooFAC00 add

Kernel Entry Point shouldn’t be known.

Starting address of the system call.
What part of the operating system should run in kernel mode?

- **Monolithic kernel**
  - All system calls run in kernel mode.
  - The entire operating system runs with full hardware privilege.
  - This organization is convenient because the OS designer doesn’t have to decide which part of the operating system doesn’t need full hardware privilege.

- **Micro kernel**:
  - Executing the bulk of the operating system in user mode.
  - The kernel interface consists of a few low-level functions for starting applications, sending messages, accessing device hardware, etc.
  - The kernel is relatively simple, as most of the operating system resides in user-level servers.

- **Xv6 is implemented as a monolithic kernel.**
Process
Process

- Unit of Isolation
- Address Space, Execution Mode, CPU usage state
Address Space (user)

Objects in the address space:
- Instructions
- Local variables (function call return address)
- Dynamic memory: malloc()
- Global and static variables

Stack:
- HEAP
- DATA
- TEXT

Address Space (User)
Address Space (user) (Cont.)

From lecture 1

But, Where is kernel?
Virtual address space
Process: `struct proc`

- **Page Table**
  - `p->pgdir`
- **Kernel Stack**
  - `p->kstack`
- **Run State**
  - `p->state`

**Process: Unit of Isolation**
- **Page Table**

**Thread: Unit of Execution**
- **Registers**
- **Stacks**
  - Local variables
  - Function call return address
/ *Per-process state*

```c
struct proc {
    uint sz;                     // Size of process memory (bytes)
    pde_t* pgdir;                // Page table
    char *kstack;                // Bottom of kernel stack for this process
    enum procstate state;        // Process state
    int pid;                     // Process ID
    struct proc *parent;         // Parent process
    struct trapframe *tf;        // Trap frame for current syscall
    struct context *context;     // swtch() here to run process
    void *chan;                  // If non-zero, sleeping on chan
    int killed;                  // If non-zero, have been killed
    struct file *ofile[NOFILE];  // Open files
    struct inode *cwd;           // Current directory
    char name[16];               // Process name (debugging)
};
```
struct ptable

- The xv6 kernel manages these `proc` structure as an `array` called `ptable`.
- `ptable` has 64 entries.

```c
struct {
    struct spinlock lock;
    struct proc proc[NPROC];
} ptable;
```

Quiz: how many processes can there be in xv6?
Stack

- Used to save the followings in the function call
  - Function arguments
  - Return address
  - local variables
stack frame for function call

- **ebp** (base pointer register) the address of the beginning of the stack frame, remains unchanged while the function executes.

- **esp** (stack pointer) address of stack top, keeps changing while the function executes.
User stack vs. kernel stack

Executing in User Mode
- User stack

Executing in the kernel
- Use kernel stack

- System Call
  - Entering the kernel
    - Switch from user stack to kernel stack
    - Raise privilege level

<table>
<thead>
<tr>
<th>User mode</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>OX 0000000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel mode</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>OX 80000000 (2GB)</td>
<td></td>
</tr>
</tbody>
</table>

- Interrupt
- Exception
- System call

Kernel stack
- Text and Code
- BIOS
- Stack

Kernel space

User space
Booting
Sample quiz

- In the course of booting, the program counter starts to execute in the lower address space and then at some point switches to the kernel region. Locate the instruction where $eip$ is changed to the kernel address space.
- `entry()` sets up the initial page table with 4MByte page size before it calls `main()`. It initializes the first entry and 512\(^{\text{th}}\) entry. What happens if `entry()` does not initialize the first entry in the page table? Modify the source code. Try to boot it. Examine what happens.
- Please provide the cause about what happens to booting.
1. Turn on the computer.
2. Load the operating system image from the disk to memory.
3. Run the operating system.
4. Create the first address space.
5. Create the first process.
6. Run the first process.
int main(void){
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // detect other processors
    lapicinit(); // interrupt controller
    seginit(); // segment descriptors
    picinit(); // disable pic
    ioapicinit(); // another interrupt controller
    consoleinit(); // console hardware
    uartinit(); // serial port
    pinit(); // process table
    tvinit(); // trap vectors
    binit(); // buffer cache
    fileinit(); // file table
    ideinit(); // disk
    startothers(); // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
    userinit(); // first user process
    mpmain(); // finish this processor's setup
}

This is all you need to understand in this class, but don’t try to understand it now.
Booting 1

ROM: BIOS

MBR: Master Boot Record (512B)

0x0000 7000 (512B)

Boot Loader

Physical memory

Kernel

0x0000 7000 (512B)

Boot Loader

Physical memory
Loading bootloader

- BIOS reads the first sector of the disk and load the boot loader (bootasm.S).
- Jump to bootmain().

Top physical memory ->

Max 0x00400000 (4MB) ->

0x00100000 (1MB) ->

0x00007e00 ->

0x00007c00 ->

0x00000000 ->

Free memory

Memory For device

Boot loader

Stack of Bootloader

```
#include "asm.h"
#include "memlayout.h"
#include "mmu.h"

# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.code16
.globl start
start:
    cli       # BIOS enabled interrupts; disable

    # Set up the stack pointer and call into C.
    movl $start, %esp
call bootmain
```
Booting 2

From Bootloader, jump to entry point of the kernel.

bootmain.c: read the kernel into memory. Jump to the beginning of kernel.

entry.s

entry:

Initial mapping
### bootmain(): Loading the kernel

- The boot loader loads the kernel into physical address 0x100000.
- Jump to entry();

<table>
<thead>
<tr>
<th>Top physical memory</th>
<th>Free memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 - 0x00100000 (1MB)</td>
<td>Kernel</td>
</tr>
<tr>
<td>0x00000000 - 0x00007c00 (512KB)</td>
<td>Memory For device</td>
</tr>
<tr>
<td>0x00007c00 - 0x00007e00</td>
<td></td>
</tr>
<tr>
<td>0x00007e00 - 0x00010000 (1MB)</td>
<td>Boot loader</td>
</tr>
<tr>
<td>0x00010000 - 0x00400000 (4MB)</td>
<td>Stack of Bootloader</td>
</tr>
</tbody>
</table>

#### Code Snippet

```c
#include "elf.h"
#include "x86.h"
#include "memlayout.h"
#define SECTSIZE 512

void readseg(uchar*, uint, uint);

void bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);

    void* elfp = (void*)((char*)elf - elfhdr->e_phoff);
    elfp += elfhdr->e_poffset;
    elf = (struct elfhdr*)elfp;
    eph = elf->e_ehdr;
    entry = (void*)(eph->e_phoff);
    entry(elf->elfhdr.Core.phoff);
}
```

< Read kernel image from disk to memory. >
entry()

create the first virtual address space and jump to main().

entry:
1045 # Turn on page size extension
   # for 4Mbyte pages
1046 movl %cr4, %eax
1047 orl $(CR4_PSE), %eax
1048 movl %eax, %cr4

1049 # Set page directory
1050 movl $(V2P_WO(entrypgdir)), %eax
1051 movl %eax, %cr3

1052 # Turn on paging.
1053 movl %cr0, %eax
1054 orl $(CR0_PG|CR0_WP), %eax
1055 movl %eax, %cr0

1057 # Set up the stack pointer to point high address.
1058 movl $(stack + KSTACKSIZE), %esp

1063 # jump to main.
1064 mov $main, %eax
1065 jmp *%eax
After mapping

- Set up a page table that maps virtual addresses starting at 0x80000000 (called KERNBASE) to physical addresses starting at 0x0.

- Entry 0: virtual addresses 0:0x400000 → physical addresses 0:0x400000.

- Entry 512: virtual addresses KERNBASE:KERNBASE+0x400000 → physical addresses 0:0x400000.

*Figure 1-3. Layout of a virtual address space*
Creating the first process

1. Allocate struct proc
2. Allocate pid
3. Allocate kernel stack
4. Initialize process state

main.c

main()

userinit.c

mpmain.c

main.C
Creating the first process (Cont.)
Creating the first process

5 Details of ‘5’

Initialize the kernel stack space.

sp → esp → esp → esp → esp → esp → esp

Context

trapret
forkret
trapframe

⇒

⇒

⇒

⇒

⇒

⇒
Creating the first process (Cont.)

```c
main.c
{
    ...
    ...
    ...
    (user main);
    mp main();
}
main.c
```

5. Initialize Page Table
   - map kernel space
   - map user space

4. Initialize Page Table
   - map kernel space

5. Initialize "main"
Creating the first process (Cont.)

Creating the first process!

```c
main() {
    ...
    ...
    usermain();
    main();
}
```

Now, we are ready to run the first process.

1. Copy `initcode.s` binary to the user address space.
2. Fill up the values into the trap frame.
3. Set the state of the process **RUNNABLE**.
Questionnaire

- Explain the difference between ⑤ and ⑧. Both of them sets the trap frame.

- Who allocates the page table? Where are these tables?
  x86 (x86) uses two level paging.
Running the first process!

- How to switch page table
- How to switch context
- How to get out of the kernel and start executing the user process.

```
main.c {
    ...
    ...
    usermain();
    mpmain();
}
main.c
```

1. `schedule();`
2. `exec("init.s");`
3. `init.s call exec("sh");`
Running the first process (Cont.)

Running the first process!

```c
main() {
    ...
    ...
    ...
    usermain();
    mpmain();
}
main.c
```

- Look for a process with **RUNNABLE**.
- There is only one **initcode**.
- Change page table (switchvm).
- Change state to **RUNNING**.
- Change context (switch).

⇒ Kernel starts executing ‘forkret’.

Think why!
Running the first process (Cont.)

Running the first process.

Kernel stack of initcode.

initcode is located at VA '0'.

'sp' → switch to 'initcode'.

'sp' → sp → trapret

'ep' = 0

'sp' → sp → trapret

return to user-space and start executing at ep(=0) in the user space.
**Questionnaire**

1. Read code: switchvm, swtch
2. Switchvm switches the page table. How come can the kernel continue executing even after switching the address space?
void scheduler(void) {
    ...
    for(;;){
        ...
        for(p = ptable.proc;
            p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;
        ...
        c->proc = p;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&(c->scheduler), p->context);
        ...
    }
    release(&ptable.lock);
}
}
context switch: \texttt{swtch()}\hfill

\begin{verbatim}
swtch:
    movl 4(esp), %eax
    movl 8(esp), %edx

    # Save old registers
    pushl %ebp
    pushl %ebx
    pushl %esi
    pushl %edi

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

    # Load new registers
    popl %edi
    popl %esi
    popl %ebx
    popl %ebp
    ret
\end{verbatim}
What does `initcode` do?

It executes `/init` binary.

```
# Initial process execs /init.
# This code runs in user space.
#include "syscall.h"
#include "traps.h"

# exec(init, argv)
.globl start
start:
    pushl $argv
    pushl $init
    pushl $0 // where caller pc would be
    movl $SYS_exec, %eax
    int $T_SYSCALL

# for(;;) exit();
exi:
    movl $SYS_exit, %eax
    int $T_SYSCALL
    jmp exit

# char init[] = "/init\0";
init:
    .string "/init\0"

# char *argv[] = { init, 0 };
.align 2
argv:
    .long init
    .long 0
```

`/init` is mother of all processes in Unix!
What does `initcode` do?

The first system call: `exec`
- page table: done
- process structure: done
- Now: let's `exec`
  - replaces memory and registers
  - keep process id, proc structure parent
  - process and file descriptors
Executing `/init`

1. Push parameters
   - $argv$
   - $init$
   - $0$
   *goes to user stack*

2. Setup system call #.
   - `%eax < STSCALL-EXEC`

3. Trap
   - `syscall`

`exec` never returns!
In `init`,

- open console device.
- open fd 0, 1, and 2.
- fork and exec 'shell'.

Now, the system is up!!!
Appendix

- C data types
- Creating libraries
- Finding the address of the function entry point and variables
- Chapter1 code files
- Assembler directives
## C Data Types

<table>
<thead>
<tr>
<th>C Type</th>
<th>stdint.h type</th>
<th>Bits</th>
<th>Sign</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>uint8_t</td>
<td>8</td>
<td>Unsigned</td>
<td>0 ~ 255</td>
</tr>
<tr>
<td>signed char</td>
<td>int8_t</td>
<td>8</td>
<td>Signed</td>
<td>-128 ~ 127</td>
</tr>
<tr>
<td>unsigned short</td>
<td>uint16_t</td>
<td>16</td>
<td>Unsigned</td>
<td>0 ~ 65,535</td>
</tr>
<tr>
<td>short</td>
<td>int16_t</td>
<td>16</td>
<td>Signed</td>
<td>-32,768 ~ 32,767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>uint32_t</td>
<td>32</td>
<td>Unsigned</td>
<td>0 ~ 2^{32}-1</td>
</tr>
<tr>
<td>int</td>
<td>int32_t</td>
<td>32</td>
<td>Signed</td>
<td>-2^{31} ~ 2^{31}-1</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>uint64_t</td>
<td>64</td>
<td>Unsigned</td>
<td>0 ~ 2^{64}-1</td>
</tr>
<tr>
<td>long long</td>
<td>int64_t</td>
<td>64</td>
<td>Signed</td>
<td>-2^{63} ~ 2^{63} - 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C Type</th>
<th>IEE754 Name</th>
<th>Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>Single Precision</td>
<td>32</td>
<td>-3.4^{38} ~ 3.4^{38}</td>
</tr>
<tr>
<td>double</td>
<td>Double Precision</td>
<td>64</td>
<td>-1.7^{308} ~ 1.7^{308}</td>
</tr>
</tbody>
</table>
Assembler directives

- `.code16` create 16 bit code from now on.
- `.code32` create 32 bit code from now on.
- `.global [symbol]` make the [symbol] visible outside.
- `.word [value]` create a memory space for [word] and set its value to [value]. For 16 bit address, 16 bit space is allocated. For 32 bit address, 32 bit address is created.
- `.long [value]` create 32 bit data.
- `.string [str]` create character string.
- `.text` the following segment belongs to text section.
- `.data` the following segment belongs to data section.
- `.comm [symbol], [length]` In bss section, create the space with [length] size and [symbol] name.
what is function entry point?

dump of test.c

```c
#include <stdio.h>

int add(int p1, int p2)
{
    return p1 + p2;
}

int main(int argc, char *argv[])
{
    printf("Please type two numbers\n");

    int a, b;
    scanf("%d %d", &a, &b);

    int res = add(a, b);
    printf("Result of adding : %d\n", res);
}
```
compile the file for debugging purpose

$ gcc -g -o test test.c

run GDB

$ gdb test

(gdb) disas main

(gdb) disas add
finding the address of a variable

- run GDB
  - $ gdb test
  - (gdb) b add
  - (gdb) run
  - (gdb) p &p1
codes to read (chapter 1)

bio.c
bootmain.c
cat.c
console.c
echo.c
exec.c
file.c
forktest.c
fs.c
grep.c
ide.c
init.c
ioapic.c
kalloc.c
kbd.c
kill.c
lapic.c
ln.c
log.c
ls.c
main.c
memide.c
mkdir.c
mkfs.c
mp.c
picirq.c
pipe.c
printf.c
proc.c
rm.c
sh.c
sleeplock.c
spinlock.c
stressfs.c
string.c
syscall.c
sysfile.c
sysproc.c
trap.c
uart.c
ulib.c
umalloc.c
usertests.c
vm.c
wc.c
zombie.c
bootasm.S
entry.S
entryother.S
initcode.S
swtch.S
trapsasm.S
usys.S
vectors.S
asm.h
buf.h
date.h
defs.h
desc.h
fcntl.h
file.h
fs.h
kbd.h
memlayout.h
mmu.h
mp.h
param.h
proc.h
sleeplock.h
spinlock.h
stat.h
syscall.h
traps.h
types.h
user.h
x86.h
Summary

- overview of booting
- creating the first address space
- creating the first process
- running the first process
- do not forget: preview and review