Operating System Organization

Youjip Won

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
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

- Word
- Spreadsheet

Write

OS
Abstraction layer: file system

OS: Abstraction Layer: filesystem

- block allocation
- location of a file
- path name resolution
Abstraction layer: memory management

Youjip Won
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.

Int *i;
*i = 0;
*i = 0;

Seg. fault

mode switch

Int *i;
*i = 0;
*i = 0;

But, can be worse!
Before starting a system call, validate Argument!
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.)

\[ j \text{ ADDR}; \quad // \text{change eip to ADDR.} \]

\[ \text{function call} \]

\[ j \text{ entry;} \]

\[ \text{entry: } \]

\[ \text{ADDR} \]

\[ = \]
System Call

System Call

J Ox000FAC00

? bypass checking!

Ox000FAB00
Ox000FAC00 add

Syscall:
// check argument
;
;

Kernel Entry Point shouldn't be knowns.

Starting address of the system call.
Kernel Organization

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:
- : 
- 4
- HEAP
- DATA
- TEXT

Address Space (User)
Address Space (user) (Cont.)
Virtual address space
**Process**: struct proc

- **Page table**: 
  - pgdir;
- **Kernel stack**: 
  - kstack;
- **Run state**: 
  - proc->state;

**Process**: unit of isolation

- **Page table**

**Thread**: unit of execution

- **Registers**
  - Registers

- **Stacks**
  - Stacks
    - Local variables
    - Function call return address
// Per-process state
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)
};
The xv6 kernel manages these `proc` structure as an `array` called `ptable`.

`ptable` has 64 entries.

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

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

- Use to save the followings in the function call
  - Function arguments
  - Return addresses
  - Local variables of callee
stack frame for function call

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

- **esp (stack pointer)**
  - Address of stack top, keep changing while the function executes.

Stack grows.

- Parameters
- Return addr
- Local variables

Stack frame

- Parameters
- Return addr
- Local variables

Stack grows.

ebp

esp
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
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
}

That’s all you need to understand, but don’t try to understand it now.
Booting 1

ROM BIOS

MBR: Master Boot Record (512B)

Boot Loader

Physical memory

0x0000 7C00 (512B)

Boot Loader

Physical memory

Kernel

0x0010 0000

0x0000 7C00 (512B)
The boot loader loads the xv6 kernel into memory at physical address 0x100000.

```
Top physical memory ->
Max 0x00400000 (4MB) ->
0x00100000 (1MB) ->
0x00007e00 ->
0x00007c00 ->
0x00000000 ->
```

- Free memory
- Xv6 kernel
- Memory For device
- Boot loader
- Stack of Bootloader
Booting 2

Booting 2

- From bootloader, jump to entry point of the kernel.

  bootmain.c: read the kernel into memory.

  jump to the beginning of kernel.

\[\text{bootmain.c} \]

<table>
<thead>
<tr>
<th>\text{entry(0)}</th>
</tr>
</thead>
</table>

\[\Rightarrow\]

<table>
<thead>
<tr>
<th>\text{entry:}</th>
</tr>
</thead>
</table>

| initial mapping |
Setup page table (in entry.s)

4MB

0x8000 0000

4MB

0x0000 0000

4KB page table

Where is it?

physical memory

= Then, call main!
After mapping

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

- Entry 0 maps virtual addresses 0:0x400000 to physical addresses 0:0x400000.

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

![Figure 1-3. Layout of a virtual address space](image)
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
Creating the first process

main() {
  ...
  user_init();
  main();
}

main.c

allocate struct proc

allocate pid

allocate kernel stack

initialize process state.

Allocate kernel stack

Embro
Creating the first process (Cont.)

Creating the first process!

```c
main.c {
    ...
    ...
    
    USER.m() {
        ...
    }
}

main.c
```

5

Kernel Stack

Initialize Kernel Stack

SP

SP

ESP

trap

SP

ESP

SP

ESP

SP

ESP

SP

ESP
Creating the first process (Cont.)

5) Details of '5'

Initialize the kernel stack space.

sp →  

sp → esp  

sp →  

sp →
Creating the first process (Cont.)

main.c

main.c

main.c

Creating the first process!

5

Process

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

kernel stack

Page Table

kernel

user

page table

bios

2GB
Creating the first process (Cont.)

Creating the first process! ☀️

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

```
main.c
```

Now, we are ready to run the first process.

- Copy `initcode.S` binary to the user address space.
- Fill up the values into the trap frame.
- 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?
  · x64 (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.

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

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

Running the first process!

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

main.c

- Schedule();
  - 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.

```
  \[\text{switch to } \text{initcode}\]  \[\Rightarrow \text{sp} \rightarrow \text{initcode}\]

Kernel stack of \text{initcode}.

\text{initcode is located at VA '0'.}

\Rightarrow \text{Start executing at esp(=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 swapping 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: swtch()**

```assembly
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
```
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();
exit:
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
/init (Cont.)

Executing `/init`

1. Push parameters
   - $arg0$
   - $r7$init
   - $0$

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

3. Trap
   - T- 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>-32768 ~ 32767</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

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.code16</td>
<td>create 16 bit code from now on.</td>
</tr>
<tr>
<td>.code32</td>
<td>create 32 bit code from now on.</td>
</tr>
<tr>
<td>.global [symbol]</td>
<td>make the [symbol] visible outside.</td>
</tr>
<tr>
<td>.word [value]</td>
<td>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, 32bit address is created.</td>
</tr>
<tr>
<td>.long [value]</td>
<td>create 32bit data.</td>
</tr>
<tr>
<td>.string [str]</td>
<td>create character string.</td>
</tr>
<tr>
<td>.text</td>
<td>the following segment belongs to text section.</td>
</tr>
<tr>
<td>.data</td>
<td>the following segment belongs to data section.</td>
</tr>
<tr>
<td>.comm [symbol], [length]</td>
<td>In bss section, create the space with [length] size and [symbol] name.</td>
</tr>
</tbody>
</table>
what is function entry point?

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);
}
```
entry point

- 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

```plaintext
5
return p1+p2;

(gdb) p &p1
$l = (int *) 0xfffffffffe3ac
(gdb) p &p2
$s2 = (int *) 0xfffffffffe3a8
```
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
def.h
cfcntl.h
date.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