Architectural Support for OS

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Computer System Organization
Issue #1

• How to perform I/Os efficiently?
  – I/O devices and CPU can execute concurrently
  – Each device controller is in charge of a particular device type
  – Each device controller has a local buffer
  – CPU issues specific commands to I/O devices
  – CPU moves data from/to main memory to/from local buffers

  – CPU is a precious resource; it should be freed from time-consuming tasks:
    • Checking whether the issue command has been completed or not
    • Moving data between main memory and device buffers
Interrupts

• How does the kernel notice an I/O has finished?
  – Polling
  – Hardware interrupt
Interrupt Handling

• Preserves the state of the CPU
  – In a fixed location
  – In a location indexed by the device number
  – On the system stack

• Determines the type
  – Polling
  – Vectored interrupt system

• Transfers control to the interrupt service routine (ISR) or interrupt handler
xv6: IRQ Vector and Gate Setup

• x86: 256 entries in IRQ vector

```c
// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[]; // in vectors.S: array of 256 entry pointers
struct spinlock tickslock;
uint ticks;

void tvinit(void)
{
    int i;

    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
    initlock(&tickslock, "time");
}
```

```plaintext
pushl $0
pushl $254
jmp alltraps
.globl vector255
vector255:
    pushl $0
    pushl $255
    jmp alltraps
```

# vector table
.data
.globl vectors
vectors:
    .long vector0
    .long vector1
vectors.S
```
xv6: Interrupt Handling

```assembly
3302    # vectors.S sends all traps here.
3303    .globl alltraps
3304    alltraps:
3305    # Build trap frame.
3306    pushl %ds
3307    pushl %es
3308    pushl %fs
3309    pushl %gs
3310    pushal
3311
3312    # Set up data segments.
3313    movw $(SEG_KDATA<<3), %ax
3314    movw %ax, %ds
3315    movw %ax, %es
3316    movw %ax, %es
3317    # Call trap(tf), where tf=%esp
3318    pushl %esp
3319    call trap
3320    addl $4, %esp
3321
3322    # Return falls through to trapret...
3323    .globl trapret
3324    trapret:
3325    popal
3326    popl %gs
3327    popl %fs
3328    popl %es
3329    popl %ds
3330    addl $0x8, %esp  # trapno and errcode
3331    iret
```

Hardware-pushed contexts

don't present on privilege change
3302    # vectors.S sends all traps here.
3303 .globl alltraps
3304 alltraps:
3305    # Build trap frame.
3306    pushl %ds
3307    pushl %es
3308    pushl %fs
3309    pushl %gs
3310    pushal
3311
3312    # Set up data segments.
3313    movw $(SEG_KDATA<<3), %ax
3314    movw %ax, %ds
3315    movw %ax, %es
3316
3317    # Call trap(tf), where tf=%esp
3318    pushl %esp
3319    call trap
3320    addl $4, %esp
3321
3322    # Return falls through to trapret...
3323 .globl trapret
3324 trapret:
3325    popal
3326    popl %gs
3327    popl %fs
3328    popl %es
3329    popl %ds
3330    addl $0x8, %esp  # trapno and errcode
3331    iret

---

**xv6: Interrupt Handling**

```
sp from task segment
```

<table>
<thead>
<tr>
<th>ss</th>
<th>esp</th>
<th>eflags</th>
</tr>
</thead>
<tbody>
<tr>
<td>cs</td>
<td>eip</td>
<td></td>
</tr>
</tbody>
</table>

only present on privilege change

```
error code
```

<table>
<thead>
<tr>
<th>trapno</th>
<th>ds</th>
<th>es</th>
</tr>
</thead>
<tbody>
<tr>
<td>fs</td>
<td>gs</td>
<td>eax</td>
</tr>
<tr>
<td>ecx</td>
<td>edx</td>
<td>ebx</td>
</tr>
<tr>
<td>oesp</td>
<td>ebp</td>
<td>esi</td>
</tr>
<tr>
<td>edi</td>
<td></td>
<td>(empty)</td>
</tr>
</tbody>
</table>

```
p->kstack
```

OS-pushed contexts

---

SSE3044: Operating Systems, Spring 2019, Jinkyu Jeong (jinkyu@skku.edu)
Data Transfer Modes

• Programmed I/O (PIO)
  – CPU is involved in moving data between I/O devices and memory
  – By special I/O instructions vs. by memory-mapped I/O

• DMA (Direct Memory Access)
  – Used for high-speed I/O devices to transmit information at close to memory speeds
  – Device controller transfers blocks of data from the local buffer directly to main memory without CPU intervention
  – Only an interrupt is generated per block
DMA Example

1. CPU programs the DMA controller
2. DMA requests transfer to memory
3. Data transferred
4. Ack

Interrupt when done
Address
Count
Control
Buffer
Drive
Main memory
CPU
DMA controller
Disk controller
Bus
Issue #2

• How to prevent user applications from harming the system?
  – What if an application accesses disk drives directly?
  – What if an application executes the HLT instruction?

<table>
<thead>
<tr>
<th>HLT—Halt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
</tr>
<tr>
<td>F4</td>
</tr>
</tbody>
</table>

Description

 Stops instruction execution and places the processor in a HALT state.
Protected Instructions

• Protected or privileged instructions
  – The ability to perform certain tasks that cannot be done from user mode
  – Direct I/O access
    • e.g. IN / OUT instructions in IA-32
  – Accessing system registers
    • Control registers
    • System table locations (e.g. interrupt handler table)
    • Setting special “mode bits”, etc.
  – Memory state management
    • Page table updates, page table pointers, TLB loads, etc.
  – HLT instruction IA-32
CPU Modes of Operation

• Kernel mode vs. user mode
  – How does the CPU know if a protected instruction can be executed?
  – The architecture must support at least two modes of operation: kernel and user mode
  • 4 privilege levels in IA-32: Ring 0 > 1 > 2 > 3
  • 2 privilege levels in ARM: User vs. Supervisor
  – Mode is set by a status bit in a protected register
    • IA-32: Current Privilege Level (CPL) in CS register
    • ARM: Mode field in CPSR register
  – Protected instructions can only be executed in the privileged level (kernel mode)
Issue #3

• How to ask services to the OS?
  – How can an application read a file if it cannot access disk drives?
  – Even a “printf()” call requires hardware access
  – User programs must ask the OS to do something privileged
System Calls

• OS defines a set of system calls
  – Programming interface to the services provided by OS
  – OS protects the system by rejecting illegal requests
  – OS may impose a quota on a certain resource
  – OS may consider fairness while sharing a resource

• A system call is a protected procedure call
  – System call routines are in the OS code
  – Executed in the kernel mode
  – On entry, user mode $\rightarrow$ kernel mode switch
  – On exit, CPU mode is changed back to the user mode
System Calls Example

• POSIX vs. Win32

<table>
<thead>
<tr>
<th>Category</th>
<th>POSIX</th>
<th>Win32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Management</td>
<td>fork</td>
<td>CreateProcess</td>
<td>Create a new process</td>
</tr>
<tr>
<td></td>
<td>waitpid</td>
<td>WaitForSingleObject</td>
<td>Wait for a process to exit</td>
</tr>
<tr>
<td></td>
<td>execve</td>
<td>(none)</td>
<td>CreateProcess = fork + exec</td>
</tr>
<tr>
<td></td>
<td>exit</td>
<td>ExitProcess</td>
<td>Terminate execution</td>
</tr>
<tr>
<td></td>
<td>kill</td>
<td>(none)</td>
<td>Send a signal</td>
</tr>
<tr>
<td>File Management</td>
<td>open</td>
<td>CreateFile</td>
<td>Create a file or open an existing file</td>
</tr>
<tr>
<td></td>
<td>close</td>
<td>CloseHandle</td>
<td>Close a file</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>ReadFile</td>
<td>Read data from a file</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>WriteFile</td>
<td>Write data to a file</td>
</tr>
<tr>
<td></td>
<td>lseek</td>
<td>SetFilePointer</td>
<td>Move the file pointer</td>
</tr>
<tr>
<td></td>
<td>stat</td>
<td>GetFileAttributesEx</td>
<td>Get various file attributes</td>
</tr>
<tr>
<td></td>
<td>chmod</td>
<td>(none)</td>
<td>Change the file access permission</td>
</tr>
<tr>
<td>File System Management</td>
<td>mkdir</td>
<td>CreateDirectory</td>
<td>Create a new directory</td>
</tr>
<tr>
<td></td>
<td>rmdir</td>
<td>RemoveDirectory</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td></td>
<td>link</td>
<td>(none)</td>
<td>Make a link to a file</td>
</tr>
<tr>
<td></td>
<td>unlink</td>
<td>DeleteFile</td>
<td>Destroy an existing file</td>
</tr>
<tr>
<td></td>
<td>chdir</td>
<td>SetCurrentDirectory</td>
<td>Change the current working directory</td>
</tr>
<tr>
<td></td>
<td>mount</td>
<td>(none)</td>
<td>Mount a file system</td>
</tr>
</tbody>
</table>
Exceptions

• Interrupts
  – Generated by hardware devices
    • Triggered by a signal in INTR or NMI pins (IA-32)
  – Asynchronous

• Exceptions
  – Generated by software executing instructions
    • Divide-by-zero
    • INT instruction in IA-32
  – Synchronous
  – Exception handling is same as interrupt handling
Exceptions in IA-32

• Traps
  – Intentional
  – System call traps, breakpoint traps, special instructions, …
  – Return control to “next” instruction

• Faults
  – Unintentional but possibly recoverable
  – Page faults (recoverable), protection faults (unrecoverable), …
  – Either re-execute faulting (“current”) instruction or abort

• Aborts
  – Unintentional and unrecoverable
  – Parity error, machine check, …
  – Abort the current program
OS Trap

• There must be a special “trap” instruction that:
  – Causes an exception, which invokes a kernel handler
  – Passes a parameter indicating which system call to invoke
  – Saves caller’s state (registers, mode bits)
  – Returns to user mode when done with restoring its state
  – OS must verify caller’s parameters (e.g. pointers)

Examples:
- INT instruction (IA-32)
- SVC instruction (ARM)
Implementing System Calls

```
count = read (fd, buffer, nbytes);
```
xv6: System Call Handling

- int 64 invokes system call

```c
#include <asm/um.h

#define DPL_USER 0x3 // User DPL

for (i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);

SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

initlock(&tickslock, "time");
```

```c
void trap(struct trapframe *tf)
{
    if (tf->trapno == T_SYSCALL) {
        if (myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if (myproc()->killed)
            exit();
        return;
    }

    switch (tf->trapno) {
    case T_IRQ0 + IRQ_TIMER:
        if (cpuid() == 0) {
            acquire(&tickslock);
            ticks++;
            wakeup(&ticks);
            release(&tickslock);
        }
        lapiceoi();
        break;
    }
```

```c
alltraps:
    # Build trap frame.
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    pushal
    # Set up data segments.
    movw $(SEG_KDATA<<3), %ax
    movw %ax, %ds
    movw %ax, %es
    # Call trap(tf), where tf=%esp
    pushl %esp
    call trap
```
xv6: System Call Handling

- `%eax` stores system call number

```c
3700 void
3701 syscall(void)
3702 {
3703     int num;
3704     struct proc *curproc = myproc();
3705     num = curproc->tf->eax;
3706     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
3707         curproc->tf->eax = syscalls[num]();
3708     } else {
3709         printf("%d %s: unknown sys call %d\n",
3710             curproc->pid, curproc->name, num);
3711         curproc->tf->eax = -1;
3712     }
3713 }
```

```c
3672 static int (*syscalls[])(void) = {
3673     [SYS_fork] sys_fork,
3674     [SYS_exit] sys_exit,
3675     [SYS_wait] sys_wait,
3676     [SYS_pipe] sys_pipe,
3677     [SYS_read] sys_read,
3678     [SYS_kill] sys_kill,
3679     [SYS_exec] sys_exec,
3680     [SYS_fstat] sys_fstat,
3681     [SYS_chdir] sys_chdir,
3682     [SYS_dup] sys_dup,
3683     [SYS_getpid] sys_getpid,
3684     [SYS_brk] sys_brk,
3685     [SYS_sleep] sys_sleep,
3686     [SYS_utime] sys_utime,
3687     [SYS_open] sys_open,
3688     [SYS_write] sys_write,
3689     [SYS_mknod] sys_mknod,
3690     [SYS_unlink] sys_unlink,
3691     [SYS_link] sys_link,
3692     [SYS_mkdir] sys_mkdir,
3693     [SYS_close] sys_close,
3694     };
```

xv6 currently supports 21 system calls
xv6: System Call Handling

- User-level codes invoking a system call

```c
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) 
.globl name; 
    name: 
    movl $SYS_## name, %eax; 
    int $T_SYSCALL; 
ret
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
```

Question 1:
How parameters are passed from user to kernel?

Question 2:
Find and explain the code snippets accessing parameters in the read system call (sys_read()) in xv6.
SSE3044: Operating Systems, Spring 2019, Jinkyu Jeong (jinkyu@skku.edu)

**xv6: System Call Summary**

Your application

C Library

User mode

Kernel mode

Kernel image

---

... read(fd, ...)
... pushl fd
... call read
...

a.c

Your application

a.S

C Library

syscall.S

kernel.S

... read:
... movl $4 %eax
... int 64
... ret
...

syscall()

C Library

Your application

... syscall()

... num = eax;
... eax = syscall[num]() /* sys_read() */
...

... syscall()

... if ( tf->trapno == 64)
... syscall();
...

... syscall()

... alltraps:
... push registers
... call trap
...
Typical OS Structure

- **Kernel**
  - Arch-dependent kernel code
  - System Call Interface
  - C Library (libc)

- **Hardware Platform**

- **User Application**

- **User space**

- **Kernel space**
Issue #4

• How to take the control of the CPU back from the running program?
  – Cooperative approach
    • Each application periodically transfers the control of the CPU to OS by calling various system calls
    • A special system call can be used just to release the CPU (e.g. `yield()`)
    • Can be used when OS trusts user applications

  – What if a process ends up in an infinite loop? (due to a bug or with a malicious intent)
Timers

• A non-cooperative approach
  – User a hardware timer that generates a periodic interrupt
  – The timer interrupt transfers control back to OS

• The OS preloads the timer with a time to interrupt
  – 10ms for Linux 2.4, 1ms for Linux 2.6, 4ms for Linux 4.1
  – 10ms for xv6

• The timer is privileged
  – Only the OS can load it

  Question 3:
  Find and explain the code snippets enabling and handling timer interrupts
Issue #5

• **How can we protect memory?**
  
  – Unlike the other hardware resources, we allow applications to access memory directly without OS intervention. Why?

  – From malicious users:
    OS must protect user applications from each other

  – For integrity and security:
    OS must also protect itself from user applications
Simplest Memory Protection

- Use base and limit registers
- Base and limit registers are loaded by OS before starting an application
Virtual Memory

• Modern CPUs are equipped with memory management hardware
  – MMU (Memory Management Unit)

• MMU provides more sophisticated memory protection mechanisms
  – Virtual memory
  – Paging: page table pointers, page protection, TLBs
  – Segmentation: segment table pointers, segment protection

• Manipulation of MMU is a privileged operation
Issue #6

• How to coordinate concurrent activities?
  – What if multiple concurrent streams access the shared data?
  – Interrupt can occur at any time and may interfere with the interrupted code

```
LOAD R1 ← Mem[X]
ADD R1 ← R1, #1
STORE R1 → Mem[X]
```

```
LOAD R1 ← Mem[X]
ADD R1 ← R1, #1
STORE R1 → Mem[X]
```
Synchronization

• Turn off/on interrupts

• Use a special atomic instruction
  – Read-Modify-Write (e.g. INC, DEC)
  – Test-and-Set
  – Compare-and-Swap
  – LOCK prefix in IA-32
  – LL (Load Locked) & SC (Store Conditional) in MIPS
OS and Architecture

• The functionality of an OS is limited by architectural features
  – Multiprocessing on MS-DOS/8086?

• The structure of an OS can be simplified by architectural support
  – Interrupt, DMA, atomic instructions, etc.

• Most proprietary OSes were developed with the certain architecture in mind
  – SunOS/Solaris for SPARC
  – IBM AIX for Power/PowerPC
  – HP-UX for PA-RISC