COS 318: Operating Systems

OS Structures and System Calls

Kai Li
Computer Science Department
Princeton University

(http://www.cs.princeton.edu/courses/cos318/)
Outline

- Protection mechanisms
- OS structures
- System and library calls
Protection Issues

- **CPU**
  - Kernel has the ability to take CPU away from users to prevent a user from using the CPU forever
  - Users should not have such an ability

- **Memory**
  - Prevent a user from accessing others’ data
  - Prevent users from modifying kernel code and data structures

- **I/O**
  - Prevent users from performing “illegal” I/Os

- **Question**
  - What’s the difference between protection and security?
Architecture Support: Privileged Mode

An interrupt or exception (INT)

User mode
- Regular instructions
- Access user memory

Kernel (privileged) mode
- Regular instructions
- Privileged instructions
- Access user memory
- Access kernel memory

A special instruction (IRET)
Privileged Instruction Examples

- Memory address mapping
- Flush or invalidate data cache
- Invalidate TLB entries
- Load and read system registers
- Change processor modes from kernel to user
- Change the voltage and frequency of processor
- Halt a processor
- Reset a processor
- Perform I/O operations
x86 Protection Rings

Privileged instructions
Can be executed only
When current privileged
Level (CPR) is 0

Operating system
Kernel

Operating system
Services

Applications

Level 0
Level 1
Level 2
Level 3
Layered Structure

- Hiding information at each layer
- Layered dependency
- Examples
  - THE (6 layers)
  - MS-DOS (4 layers)
- Pros
  - Layered abstraction
- Cons
  - Inefficiency
  - Inflexible
Monolithic

- All kernel routines are together
- A system call interface
- Examples:
  - Linux
  - BSD Unix
  - Windows
- Pros
  - Shared kernel space
  - Good performance
- Cons
  - Instability
  - Inflexible
Microkernel

- Services are implemented as regular process
- Micro-kernel get services on behalf of users by messaging with the service processes
- Examples:
  - Mach, Taos, L4, OS-X
- Pros?
  - Flexibility
  - Fault isolation
- Cons?
  - Inefficient (Lots of boundary crossings)
  - Insufficient protection
  - Inconvenient to share data between kernel and services
Virtual Machine

- Virtual machine monitor
  - Virtualize hardware
  - Run several OSes
  - Examples
    - IBM VM/370
    - Java VM
    - VMWare, Xen

- What would you use virtual machine for?
- Does virtual machine need more than two modes?
Two Popular Ways to Implement VMM

VMM runs on hardware

VMM as an application

(A special lecture later in the semester)
System Call Mechanism

- **Assumptions**
  - User code can be arbitrary
  - User code cannot modify kernel memory

- **Design Issues**
  - User makes a system call with parameters
  - The call mechanism switches code to kernel mode
  - Execute system call
  - Return with results
Interrupt and Exceptions

- **Interrupt Sources**
  - Hardware (by external devices)
  - Software: INT n

- **Exceptions**
  - Program error: faults, traps, and aborts
  - Software generated: INT 3
  - Machine-check exceptions

- See Intel document volume 3 for details
## Interrupt and Exceptions (1)

<table>
<thead>
<tr>
<th>Vector #</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide error (by zero)</td>
<td>Fault</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug</td>
<td>Fault/trap</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>NMI interrupt</td>
<td>Interrupt</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>Trap</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>Trap</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND range exceeded</td>
<td>Trap</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid opcode</td>
<td>Fault</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device not available</td>
<td>Fault</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double fault</td>
<td>Abort</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Coprocessor segment overrun</td>
<td>Fault</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td></td>
</tr>
</tbody>
</table>
## Interrupt and Exceptions (2)

<table>
<thead>
<tr>
<th>Vector #</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment not present</td>
<td>Fault</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack-segment fault</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General protection</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Reserved</td>
<td>Fault</td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>Floating-point error (math fault)</td>
<td>Fault</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment check</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>19-31</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td></td>
<td>User defined</td>
<td>Interrupt</td>
</tr>
</tbody>
</table>
System Calls

- Operating system API
  - Interface between an application and the operating system kernel

- Categories
  - Process management
  - Memory management
  - File management
  - Device management
  - Communication
How many system calls?

- 6th Edition Unix: ~45
- POSIX: ~130
- FreeBSD: ~130
- Linux: ~250 ("fewer than most")
- Windows: 400? 1000? 1M?
V6/usr/sys/ken/sysent.c

/*
  /*
  * This table is the switch used to transfer
  * to the appropriate routine for processing a system call.
  * Each row contains the number of arguments expected
  * and a pointer to the routine.
  */

int syscall[] =
  { 0, &nullsys, 0, &exit, 0, &fork, 2, &read, 2, &write, 2, &open, 0, &close, 0, &wait, 2, &creat, 2, &link, 1, &unlink, 2, &exec, 1, &chdir, 0, &time, 3, &mkod, 2, &chmod, 2, &chown, 1, &sbreak, 2, &stat, 2, &seek, 0, &getpid, /* 0 = indir */ /* 1 = exit */ /* 2 = fork */ /* 3 = read */ /* 4 = write */ /* 5 = open */ /* 6 = close */ /* 7 = wait */ /* 8 = creat */ /* 9 = link */ /* 10 = unlink */ /* 11 = exec */ /* 12 = chdir */ /* 13 = time */ /* 14 = mkdir */ /* 15 = chmod */ /* 16 = chown */ /* 17 = break */ /* 18 = stat */ /* 19 = seek */ /* 20 = getpid */ /* 21 = mount */ /* 22 = umount */ /* 23 = setuid */ /* 24 = getuid */ /* 25 = stime */ /* 26 = ptrace */ /* 27 = x */ /* 28 = fstat */ /* 29 = x */ /* 30 = strftime; inoperative */ /* 31 = stty */ /* 32 = tty */ /* 33 = x */ /* 34 = nice */ /* 35 = sleep */ /* 36 = sync */ /* 37 = kill */ /* 38 = swtch */ /* 39 = x */ /* 40 = x */ /* 41 = dup */ /* 42 = pipe */ /* 43 = times */ /* 44 = prof */ /* 45 = tiu */ /* 46 = setgid */ /* 47 = getgid */ /* 48 = sig */
OS Kernel: Trap Handler

- HW Device Interrupt
- System Call
- HW exceptions
- SW exceptions
- Virtual address exceptions

HW implementation of the boundary

- Syscall table
- System Service dispatcher
- System services
- Exception dispatcher
- VM manager’s pager
- Exception handlers
- Interrupt service routines
Passing Parameters

- **Pass by registers**
  - # of registers
  - # of usable registers
  - # of parameters in system call
  - Spill/fill code in compiler

- **Pass by a memory vector (list)**
  - Single register for starting address
  - Vector in user’s memory

- **Pass by stack**
  - Similar to the memory vector
  - Procedure call convention
Library Stubs for System Calls

Example:
```c
int read( int fd, char * buf, int size) {
    move fd, buf, size to R₁, R₂, R₃
    move READ to R₀
    int $0x80
    move result to R_{result}
}
```

Linux: 80
NT: 2E

User program

Kernel in protected memory
**System Call Entry Point**

**EntryPoint:**
- switch to kernel stack
- save context
- check $R_0$
- call the real code pointed by $R_0$
- place result in $R_{result}$
- restore context
- switch to user stack
- iret (change to user mode and return)

(Assume passing parameters in registers)
A simple system call (6th Edition chdir)

- "call the real code pointed by $R_0$
  place result in $R_{\text{result}}"

```
chdir()
{
    register *ip;
    extern uchar;
    ip = namei(&uchar, 0);
    if(ip == NULL)
        return;
    if((ip->i_mode&IFMT) != IFDIR) {
        u.u_error = ENOTDIR;
        bad:
        iput(ip);
        return;
    }
    if(access(ip, IEXEC))
        goto bad;
    iput(u.u_cdir);
    u.u_cdir = ip;
    prele(ip);
}
```
Design Issues

- **System calls**
  - There is one result register; what about more results?
  - How do we pass errors back to the caller?
  - Can user code lie?
  - How would you perform QA on system calls?

- **System calls vs. library calls**
  - What should be system calls?
  - What should be library calls?
Syscall or library?

/*
 * open system call
 */
open()
{
    register *ip;
    extern uchar;

    ip = namei(&uchar, 0);
    if(ip == NULL)
        return;
    u.u_arg[1]++;
    open1(ip, u.u_arg[1], 0);
}

/*
 * creat system call
 */
creat()
{
    register *ip;
    extern uchar;

    ip = namei(&uchar, 1);
    if(ip == NULL) {
        if(u.u_error)
            return;
        ip = maknode(u.u_arg[1]&07777&(~ISVTX));
        if (ip==NULL)
            return;
        open1(ip, FWRITE, 2);
    } else
        open1(ip, FWRITE, 1);

    /* common code for open and creat.
     * check permissions, allocate an open file structure,
     * and call the device open routine if any.
     */
    open1(ip, mode, trf)
    int *ip;
    {
        register struct file *fp;
        register *rip, m;
        int i;

        rip = ip;
        m = mode;
        if(trf != 2) {
            if(m&FREAD)
                access(rip, IREAD);
            if(m&FWRITE)
                access(rip, IWRITE);
            if((rip->i_mode&IFMT) == IFDIR)
                u.u_error = EISDIR;
        }
        if(u.u_error)
            goto out;
        if(trf)
            itrunc(rip);
        prele(rip);
        if ((fp = alloc()) == NULL)
            goto out;
        fp->f_flag = m&(FREAD|FWRITE);
        fp->f_inode = rip;
        i = u.u_arO[R0];
        open1(rip, m&FWRITE);
        if(u.u_error == 0)
            return;
        u.u_ofile[i] = NULL;
        fp->f_count--;
        out: iput(rip);
    }
}
Backwards compatibility...

NAME

open - open a file

SYNOPSIS

```c
#include <sys/stat.h>

#include <fcntl.h>

int open(const char *path, int oflag, ...);
```

The use of `open()` to create a regular file is preferable to the use of `creat()`, because the latter is redundant and included only for historical reasons.
Division of Labors (or Separation Of Concerns)

Memory management example

Kernel
- Allocates “pages” with hardware protection
- Allocates a big chunk (many pages) to library
- Does not care about small allocs

Library
- Provides malloc/free for allocation and deallocation
- Application use these calls to manage memory at fine granularity
- When reaching the end, library asks the kernel for more
Feedback To The Program

- Applications view system calls and library calls as procedure calls
- What about OS to apps?
  - Various exceptional conditions
  - General information, like screen resize
- What mechanism would OS use for this?
Summary

- Protection mechanism
  - Architecture support: two modes
  - Software traps (exceptions)

- OS structures
  - Monolithic, layered, microkernel and virtual machine

- System calls
  - Implementation
  - Design issues
  - Tradeoffs with library calls