CS 134:
Operating Systems
System Calls

CS34

2013-05-19

CS 134:
Operating Systems
System Calls

Overview

Overview
The Processor Status Word
Protection
Types of Protection
Memory Protection
System Calls
Next Assignment

Processor Status Words

Every processor, even a microcontroller, has a status word (often called PSW). Common contents are:

- Protection control
- Interrupt control
- Single-step flag
- Condition codes

Processor Status Words

Every processor, even a microcontroller, has a status word (often called PSW). Common contents are:

- Protection control
- Interrupt control
- Single-step flag
- Condition codes

MIPS Status

MIPS keeps a STATUS word in control register 12:

- Various cache-control bits
- “Boot flag” for booting from ROM
- Five hardware interrupt enables
- Two software interrupt enables
- Three bit pairs called old/previous/current:
  - Kernel/user mode
  - Global interrupt enable

MIPS Status

MIPS keeps a STATUS word in control register 12:
- Various cache-control bits
- “Boot flag” for booting from ROM
- Five hardware interrupt enables
- Two software interrupt enables
- Three bit pairs called old/previous/current:
  - Kernel/user mode
  - Global interrupt enable

How MIPS Interrupts Work

MIPS works like most machines:

- Finish currently executing instructions
- Drain pipeline
- Disable interrupts
- Switch to kernel mode
- Start execution at known location

Minor MIPS detail: in STATUS, old/previous/current is shifted left and current is set to 0 (kernel mode, no interrupts)

How MIPS Interrupts Work
MIPS works like most machines:
Finish currently executing instructions
Drain pipeline
Disable interrupts
Switch to kernel mode
Start execution at known location
Minor MIPS detail: in STATUS, old/previous/current is shifted left and current is set to 0 (kernel mode, no interrupts)

Protection

Processes need to be insulated from each other.

What needs protection?

What do we want from hardware to provide protection?

Protection

Processes need to be insulated from each other.

What needs protection?

What do we want from hardware to provide protection?

Stop here to discuss.

User & Kernel Mode

Two states:

- *User mode*—Processes
- *Kernel mode*—OS code to support processes

The *hardware* usually knows what state we're in. (Why?)

What happens when we change state?

User & Kernel Mode
Two states:
User mode—Processes
Kernel mode—OS code to support processes
The hardware usually knows what state we're in. (Why?)
What happens when we change state?

CPU Protection

If a program hangs, it shouldn't hang the machine

Use a timer interrupt!

- Decremented every clock tick
- Zero $\Rightarrow$ Interrupt

CPU Protection

If a program hangs, it shouldn't hang the machine

Use a timer interrupt!

- Decremented every clock tick
- Zero ⇒ Interrupt

I/O Protection

Protect I/O devices from errant programs

Solution: *I/O Protection*

- Only kernel may interact with I/O hardware
- I/O instructions are privileged
- Interrupt jumps to kernel, sets kernel mode

2013-05-19

CS34
└ Protection
  └ Types of Protection
    └ I/O Protection

I/O Protection

Protect I/O devices from errant programs

Solution: *I/O Protection*

- Only kernel may interact with I/O hardware
- I/O instructions are privileged
- Interrupt jumps to kernel, sets kernel mode

Memory Protection

Protecting I/O devices also requires that we protect

- Interrupt vector
- Interrupt service routines (and rest of kernel)
- Operating system data structures

from modification by errant or malicious programs

Solution: *Memory Protection*

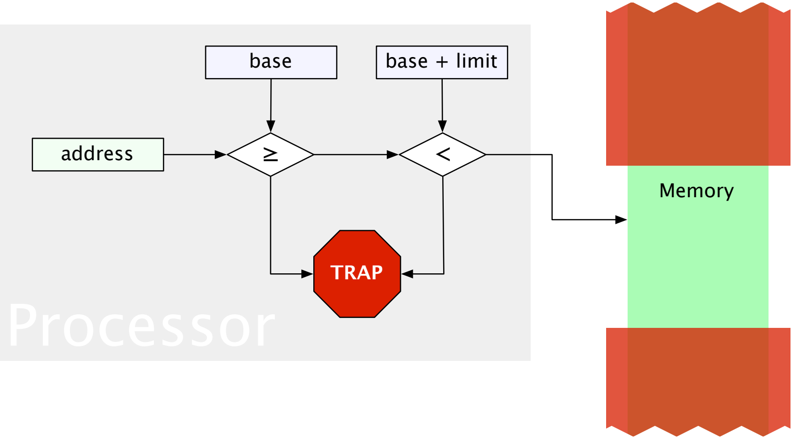
Class Exercise

What's the *simplest* solution we could ask from hardware makers to solve problem of ensuring that a program doesn't access outside its own chunk of physical memory?

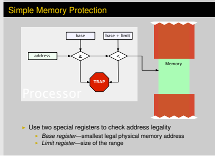
Memory Protection
Protecting I/O devices also requires that we protect
Interrupt vector
Interrupt service routines (and rest of kernel)
Operating system data structures
from modification by errant or malicious programs
Solution: *Memory Protection*
Class Exercise
What's the *simplest* solution we could ask from hardware makers to solve problem of ensuring that a program doesn't access outside its own chunk of physical memory?

Here, we're looking for base/limit registers.

Simple Memory Protection

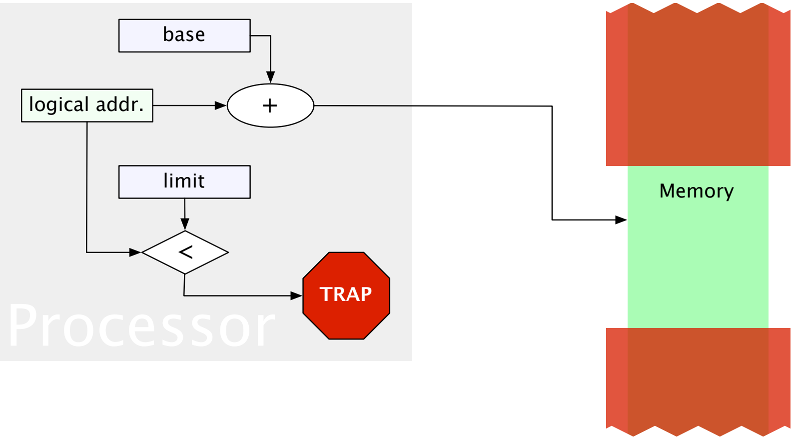


- Use two special registers to check address legality
  - *Base register*—smallest legal physical memory address
  - *Limit register*—size of the range

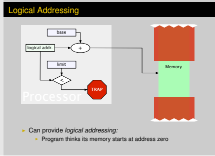


- Memory outside designated range can't be accessed by user-mode code
- In kernel mode, process has unrestricted access to all memory
- Load instructions for base and limit registers are privileged
- Checks can proceed in parallel

Logical Addressing



- Can provide *logical addressing:*
  - Program thinks its memory starts at address zero



Class Question

Given that I/O instructions are privileged. . . and that misusing a modern I/O device can destroy it

> *How does a user-mode program perform I/O?*
>
> *(or do anything else it is "forbidden" to do directly)*

Class Question

Given that I/O instructions are privileged. . . and that misusing a modern I/O device can destroy it
How does a user-mode program perform I/O?
(or do anything else it is "forbidden" to do directly)

System Calls

System Call: A method used by a process to request action by the operating system

Implemented as either

- Software interrupt (aka Trap)
- Special `syscall` instruction

Usually works just like hardware interrupt—control passes through interrupt vector to a service routine in the OS, mode bit is set to kernel

Class Question

What things do we need to do in the kernel part of a syscall?

System Calls
System Call: A method used by a process to request action by the operating system
Implemented as either
- Software interrupt (aka Trap)
- Special `syscall` instruction
Usually works just like hardware interrupt—control passes through interrupt vector to a service routine in the OS, mode bit is set to kernel
Class Question
What things do we need to do in the kernel part of a syscall?

The kernel must first save status. Then it needs to figure out which syscall is being made (including verification of legality). Any parameters must be recovered from user space; then the implementing function is called. Finally, results are returned to the user, status is restored, and user mode is resumed.

Most system calls re-enable interrupts during their execution.

MIPS System Call Example

Example code from `libc` on OS/161

```
reboot:
   addiu v0, $0, SYS_reboot /* load syscall no. */
   syscall                  /* make system call */
   beq a3, $0, 1f           /* a3= 0 =>call succeeded */
   nop                      /* delay slot */
   sw v0, errno             /* failure: store errno */
   li v1, -1                /* and force return to -1 */
   li v0, -1
1:
   j ra                     /* return */
   nop                      /* delay slot */
```


MIPS System Call Example

Example code from libc on OS/161

reboot:
addiu v0, $0, SYS_reboot /* load syscall no. */
syscall /* make system call */
beq a3, $0, 1f /* a3= 0 =>call succeeded */
nop /* delay slot */
sw v0, errno /* failure: store errno */
li v1, -1 /* and force return to -1 */
li v0, -1
1:
j ra /* return */
nop /* delay slot */

X86 System Call Example

Hello World on Linux

```
    .section .rodata
greeting:
    .string "Hello World\n"
    .text
_start:
    mov   $12,%edx        /* write(1, "Hello World\n", 12) */
    mov   $greeting,%ecx
    mov   $1,%ebx
    mov   $4,%eax         /* write is syscall 4 */
    int   $0x80

    xorl  %ebx, %ebx      /* Set exit status and exit */
    mov   $0xfc,%eax
    int   $0x80

    hlt                   /* Just in case... */
```


X86 System Call Example
Hello World on Linux

Functionality

What functionality should be implemented as system calls?

Functionality

What functionality should be implemented as system calls?

Let class brainstorm, them make list on board.

Some POSIX System Calls

Call	Description
`pid = fork()`	Create child process
`pid = waitpid(pid, &statloc, options)`	Wait for child to terminate
`s = execve(name, argv, environp)`	Replace process's image
`exit(status)`	Terminate process
`fd = open(file, how, ...)`	Open file for read/write
`s = close(fd)`	Close open file
`n = read(fd, buffer, nbytes)`	Read data from file into buffer
`n = write(fd, buffer, nbytes)`	Write data from buffer to file
`pos = lseek(fd, offset, whence)`	Move file pointer
`s = stat(name, &buf)`	Get file's status information
`s = mkdir(name, mode)`	Create new directory
`s = rmdir(name)`	Remove empty directory
`s = link(name1, name2)`	Create link to file
`s = unlink(name)`	Remove directory entry
`s = mount(special, name, flag)`	Mount file system
`s = umount(special)`	Unmount file system
`s = chdir(dirname)`	Change working directory
`s = chmod(name, mode)`	Change file's protection bits
`s = kill(pid, signal)`	Send signal to a process
`secs = time(&seconds)`	Get elapsed time since 1/1/70

Beyond System Calls—Library Interfaces

System calls tend to be minimal and low-level

Programmers prefer to use higher-level routines

Class Exercise

What is the key difference between system calls and library calls?

Beyond System Calls—Library Interfaces
System calls tend to be minimal and low-level
Programmers prefer to use higher-level routines
Class Exercise
What is the key difference between system calls and library calls?

The Next Assignment

In the next assignment, you must implement

- `open`, `read`, `write`, `lseek`, `close`, `dup2`
- `fork`, `_exit`
- `chdir`, `getcwd`
- `getpid`
- `execv`, `waitpid`

What are the data structures you'll need? Initialization? How/when is data changed or copied?

In general, how should it all work?

The Next Assignment
In the next assignment, you must implement
- open, read, write, lseek, close, dup2
- fork, _exit
- chdir, getcwd
- getpid
- execv, waitpid
What are the data structures you'll need? Initialization? How/when is data changed or copied?
In general, how should it all work?