Protection and System Calls

Otto J. Anshus

Recall the Protection Issues

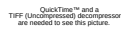
- I/O protection
  - Prevent users from performing illegal I/O's
- Memory protection
  - Prevent users from modifying kernel code and data structures
  - …and each others code and data
- CPU protection
  - Prevent a user from using the CPU for too long
    - Throughput of jobs, and response time to events (incl. user interactive response time)

Architecture Support: Privileged Mode

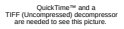
Interrupts and Exceptions

- Interrupt sources
  - HW (from external devices)
  - SW: **int n**
- Exceptions
  - Program errors: faults, traps, aborts
  - SW generated: **int 3**
  - Machine-check exceptions
- See Intel doc Vol. 3 for details

Interrupts and Exceptions



Interrupts and Exceptions



Privileged Instruction Examples

- Memory address mapping
- Cache flush or invalidation
- Invalidating TLB entries
- Loading and reading system registers
- Changing processor mode from kernel to user
- Changing the voltage and frequency of the processor
- Halting a processor
- I/O operations

Table 2-2. Summary of System Instructions

Instruction	Description	Useful to Application?	Protected from Application?
LLDT	Load LDT Register	No	Yes
SLDT	Store LDT Register	No	No
LGDT	Load GDT Register	No	Yes
SGDT	Store GDT Register	No	No
LTR	Load Task Register	No	Yes
STR	Store Task Register	No	No
LIDT	Load IDT Register	No	Yes
SIDT	Store IDT Register	No	No
MOV CR*n*	Load and store control registers	Yes	Yes (load only)
SMSW	Store MSW	Yes	No
LMSW	Load MSW	No	Yes
CLTS	Clear TS flag in CR0	No	Yes
ARPL	Adjust RPL	Yes[1]	No
LAR	Load Access Rights	Yes	No
LSL	Load Segment Limit	Yes	No

Table 2-2. Summary of System Instructions (Contd.)

Instruction	Description	Useful to Application?	Protected from Application?
VERR	Verify for Reading	Yes	No
VERW	Verify for Writing	Yes	No
MOV DB*n*	Load and store debug registers	No	Yes
INVD	Invalidate cache, no writeback	No	Yes
WBINVD	Invalidate cache, with writeback	No	Yes
INVLPG	Invalidate TLB entry	No	Yes
HLT	Halt Processor	No	Yes
LOCK (Prefix)	Bus Lock	Yes	No
RSM	Return from system management mode	No	Yes
RDMSR[3]	Read Model-Specific Registers	No	Yes
WRMSR[3]	Write Model-Specific Registers	No	Yes
RDPMC[4]	Read Performance-Monitoring Counter	Yes	Yes[2]
RDTSC[5]	Read Time-Stamp Counter	Yes	Yes[2]

IA32 Protection Rings

I/O

2^16=0-FFFFh
8-bit ports
2*8=16 bit port
4*16=32 bit port

- I/O ports:
  - created in system HW for com. w/peripheral devices
  - Examples
    - connects to a serial device
    - connects to control registers of a disk controller
- I/O address space
  - I/O instructions
    - in, out: between ports and registers
    - ins, outs: between ports and memory locations
  - I/O protection mechanism
    - I/O Privilege Level (IOPL): I/O instr. only from Ring Level 0 or 1 (typical)
    - I/O permission bit map: Gives selective control of individual ports

Will look at this and memory mapped I/O later

System Calls

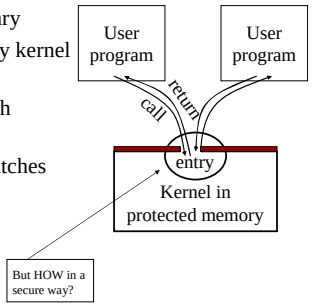
- Operating System API
  - Interface between a process and OS kernel
- Categories
  - Process management
  - Memory management
  - File management
  - Device management
  - Communication

System Calls

- Process management
  - end, abort , load, execute, create, terminate, set, wait
- Memory management
  - mmap & munmap, mprotect, mremap, msync, swapon & off,
- File management
  - create, delete, open, close, R, W, seek
- Device management
  - res, rel, R, W, seek, get & set atrib., mount, unmount
- Communication
  - get ID's, open, close, send, receive

System Call Mechanism

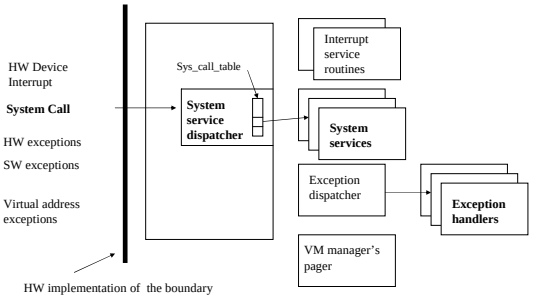
- User code can be arbitrary
- User code cannot modify kernel memory
- Makes a system call with parameters
- The call mechanism switches code to kernel mode
- Execute system call
- Return with results



System Call Implementation

- Use an "interrupt"
  - Hardware devices (keyboard, serial port, timer, disk,…) and **software** can request service using interrupts
  - The CPU is interrupted
  - ...and a service handler routine is run
  - …when finished the CPU resumes from where it was interrupted (or somewhere else determined by the OS kernel)

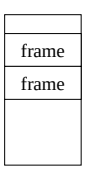
OS Kernel: Trap Handler



Passing Parameters

- Pass by registers
  - Simple, but limited
    - #registers
    - #usable registers
    - #parameters in syscall
- Pass by memory vector
  - A register holds the address of a location in users memory
- Pass by stack
  - Push: done by library
  - Pop: done by Kernel

Kernel has access to callers address space, but not vice versa



The Stack

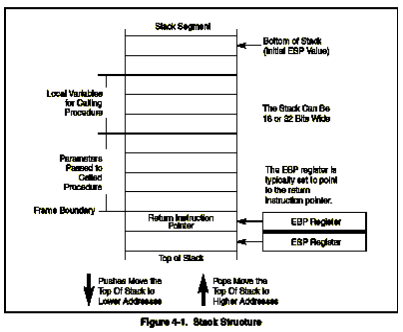


Figure 4-1. Stack Structure

- Many stacks possible, but only one is "current": the one in the segment referenced by the SS register
- Max size 4 gigabytes
- PUSH: write (--ESP);
- POP: read(ESP++);
- When setting up a stack remember to align the stack pointer on 16 bit word or 32 bit double-word boundaries

Library Stubs for System Calls

- **User** process: read(fd, buf, size)

```
int read( int fd, char * buf, int size)
{
    move READ to R0
    move fd, buf, size to R1, R2, R3
    int $0x80
    load result code from Rresult
}
```

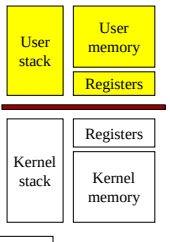
32-255 available to user

Returns here when work is done

Could be an error code

Win NT: 2E

Linux: 80



System Call Entry Point

int 0x80

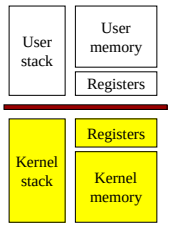
SW interrupt

- Assume passing parameters in registers

OS Kernel EntryPoint:

switch to kernel stack;
save all registers;
if legal(R_0) call sys_call_table[R_0];
restore user registers;
switch to user stack;
iret;

Kernel Mode: Total control. All interrupts are disabled



System Call Entry Point

int 0x80

SW interrupt

- Assume passing parameters in registers

EntryPoint:

switch to kernel stack;
save user context;
if legal(R_0) **call** service;
restore user context;
switch to user stack;
iret;

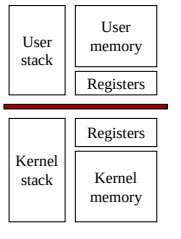
Kernel Mode: Total control. All interrupts are disabled

Change to user mode and return

Has put results into buf

Or: User stack

Or: some register



System Call Entry Point

int 0x80

SW interrupt

- Assume passing parameters in registers

EntryPoint:

switch to kernel stack;
save all registers;
if legal(R_0) call sys_call_table[R_0];
restore user registers;
switch to user stack;
iret;

Save/Restore Context?

If this code takes a long time: should ENABLE interrupts

READ returns with result and handler must return them to user

Or SCHEDULE to run another

Polling instead of Interrupt?

- OS kernel could check a request queue instead of using an interrupt?
  - Waste CPU cycles checking
  - All have to wait while the checks are being done
  - When to check?
    - Non-predictable
    - Pulse every 10-100ms?
      - » too long time
- Same valid for HW Interrupts vs. Polling
- However, spinning can give good performance (more later)

But used for Servers

Design Issues for Syscall

- We used only one result reg, what if more results?
- In kernel and in called service: Use caller's stack or a special stack?
  - Use a special stack
- Quality assurance
  - Use a single entry or multiple entries?
    - Simple is good?
      - Then a single entry is simpler, easier to make robust
- Can kernel code call system calls?
  - Yes, but should avoid the entry point mechanism

System calls vs. Library calls

- Division of labor (a.k.a. Separation of Concerns)
- Memory management example
  - Kernel
    - Allocates “pages” (w/HW protection)
    - Allocates many “pages” to library
    - Big chunks, no “small” allocations
  - Library
    - Provides malloc/free for allocation and deallocation of memory
    - Application use malloc/free to manage its own memory at **fine** granularity
    - When no more memory, library asks kernel for a new chunk of pages

User process vs. kernel

- User process -> kernel
  - syscalls
- Kernel -> user process
  - Kernel is all powerful
    - Can write into user memory
    - Can terminate, block and activate user processes