Protection and System Calls

Tore Brox-Larsen, Spring 2019

Including Original Slides and Adaptations by
Kai Li, Princeton University and
Otto J. Anshus, University of Tromsø

Some HW Platforms

- Low-End Mobile Phone
  - One processor: Single or multi core
- Other
  - One or more processors, multiple or many single- or multiple threaded cores
- *Our projects address the case of a single single-threaded processor!*

Environment

- Many concurrently running “processes”
  - Each assuming ”exclusive” access to the whole computer (”Mine, all mine”)
  - But typically also:
    - Shielded from some HW-specific peculiarities
    - Access to services provided by OS
    - Protected from stumbling onto other processes and OS
    - Ability to communicate and coordinate with other processes
- Explicit and implicit sharing and protection of computer resources

Processes

- "A program under execution"
  - Contains one or more threads
    - *Thread—Flow of execution, control flow—Unit of execution*
  - Single-threaded: Contains one control thread representing the execution of a sequential program
  - Multi-threaded: Contains one or more control threads representing a parallel program
- Unit of resource allocation
  - Threads within any process share all resources except processor registers

OS Provides Protection

- CPU protection
  - Preventing users from using the CPU for too long
- Memory protection
  - Prevent users from modifying kernel code and data structures
- I/O protection
  - Prevent users from performing illegal I/O's
- File system protection
  - Preventing unauthorized use of file

Basic HW Protection Mechanisms (i)

- Two (or more) privilege levels
  - Highest privilege level
    - ”Anything is allowed”
  - Lowest privilege level
    - Only what can be safely done by anyone is available
- Privilege level switch mechanisms?
- How are privilege levels applied?

User/Kernel Mode

- User mode
  - Regular Instructions
  - Access user-mode memory
  - Illegal attempts causes faults/exceptions
- Kernel (supervisor, privileged) mode
  - Regular instructions
  - Privileged instructions
  - Access both user- and kernel-mode memory
  - An instruction to change to user mode

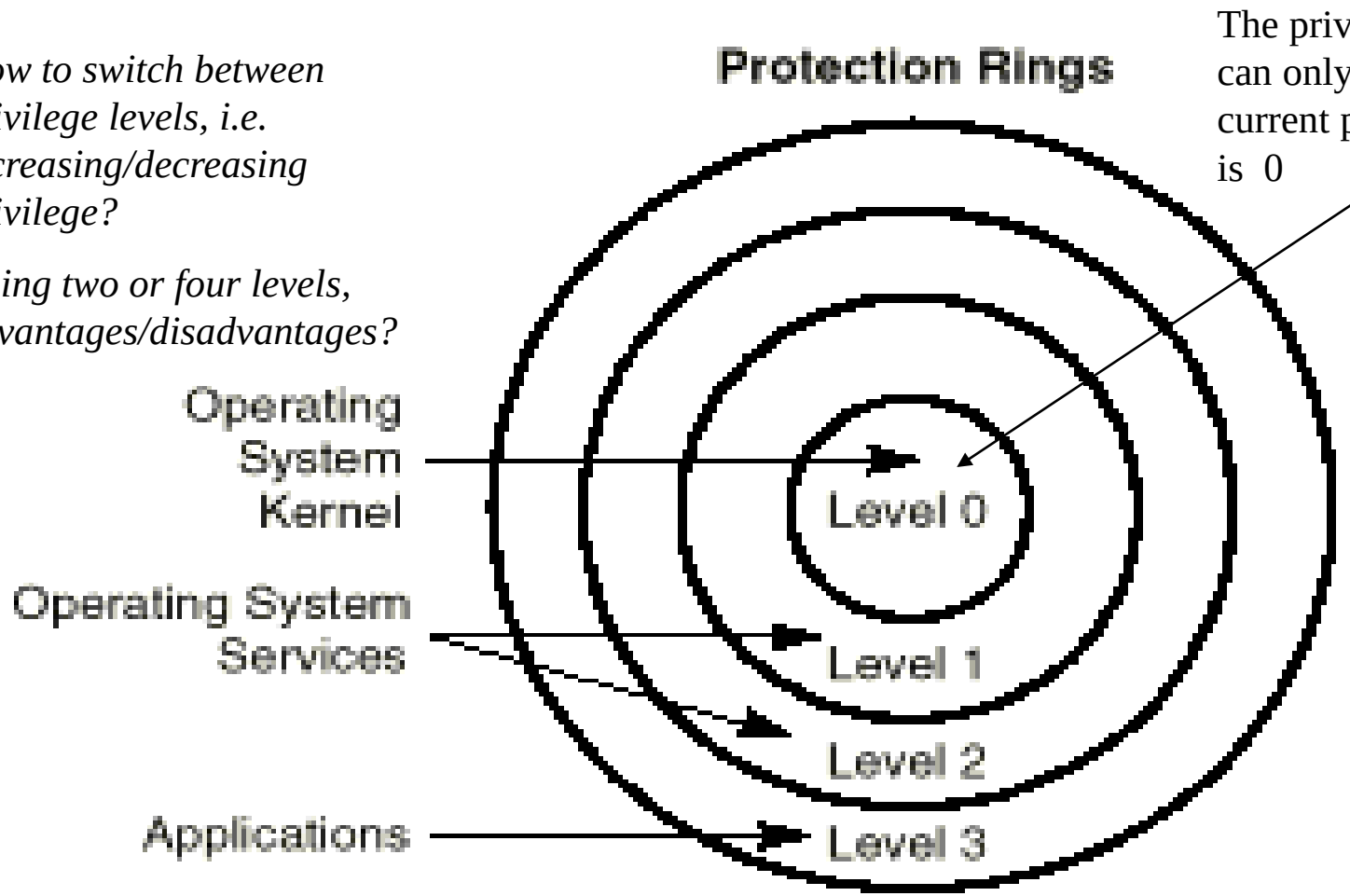
Examples of Privileged Instructions

- Setting up memory address mappings
- Flushing cache, invalidating cache
- Invalidating TLB
- Setting system registers
- I/O operations
  - *Memory mapped I/O uses unprotected move instructions. Protection is provided through the mechanisms for memory protection*

Possible Use of Intel Privilege Levels

How to switch between privilege levels, i.e. increasing/decreasing privilege?

Using two or four levels, advantages/disadvantages?



The privileged instructions can only be executed when current privilege level (CPL) is 0

I/O

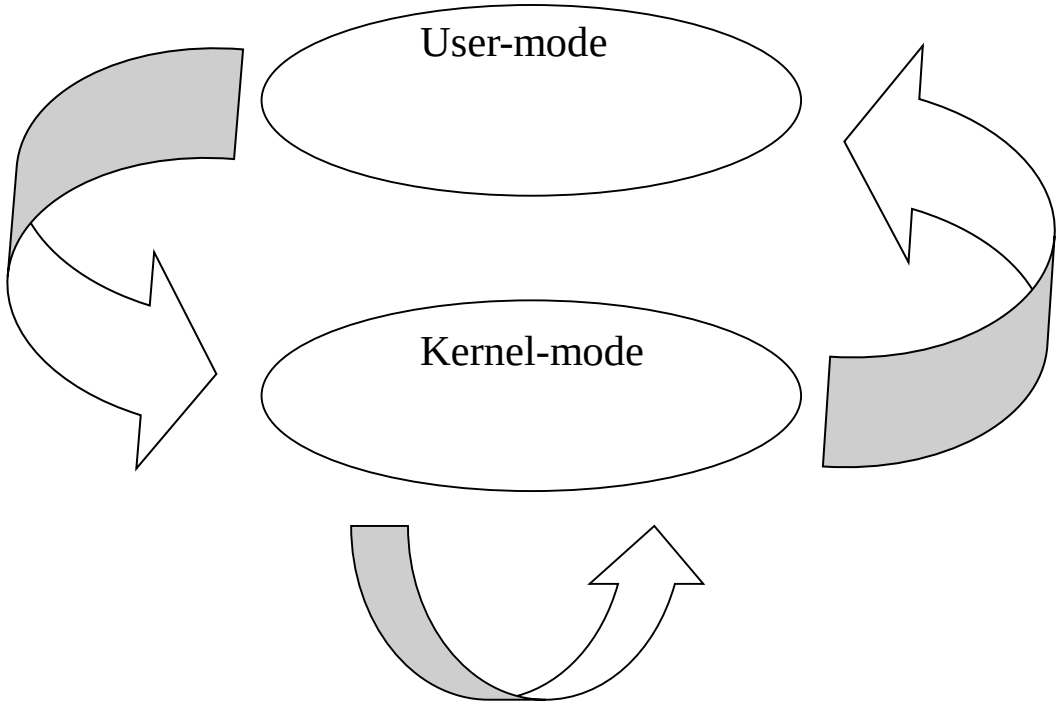
2^16=0-FFFFh
8-bit ports
2*8=16 bit port
4*16=32 bit port

- 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

Interrupts are imporant

- Interrupt,
- User system call,
- User-mode trap



Return to user-mode

- Interrupt,
- Kernel system call
- Kernel-mode trap

Basic HW Protection Mechanisms (ii)

- Memory protection
  - Provided by a ”memory management unit (MMU),” conceptually a level of logic between the processor and memory. Privileged instructions set restrictions on how regions in memory address space may be accessed. MMU traps when instructions attempt to break the restrictions – The trap invokes the operating system
    +new support for not execute
- Atomic ReadWrite
  - Uninterrupted read/write of memory address
  - Supports cooperative coordination

Some Currently Ditched HW Protection Mechanisms

- Tagged architectures
- Capability based architectures
- Object oriented architectures
- Support for fine-grained memory segmentation
- *May or may not remain ditched. May or may not be reintroduced in original or modified form*

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

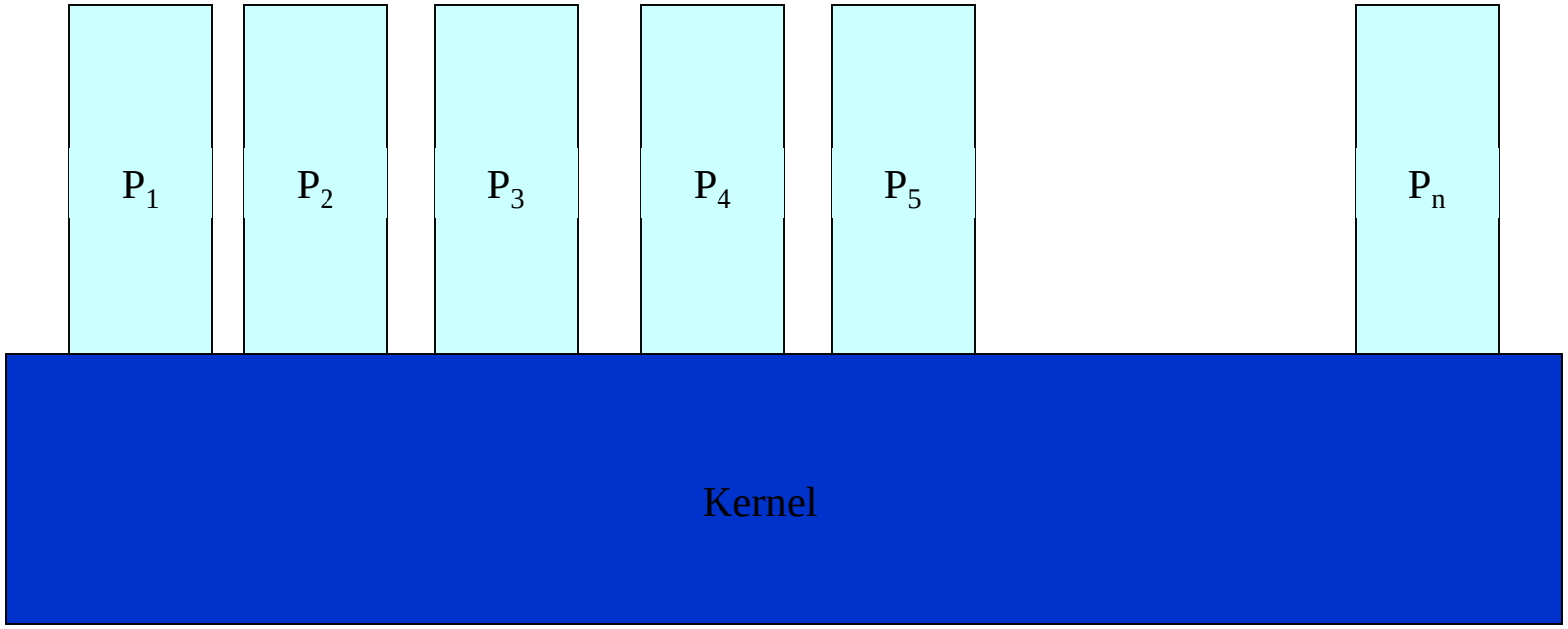
Table 2-2. Summary of System Instructions (Contd.)

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[3]	Read Time-Stamp Counter	Yes	Yes[2]

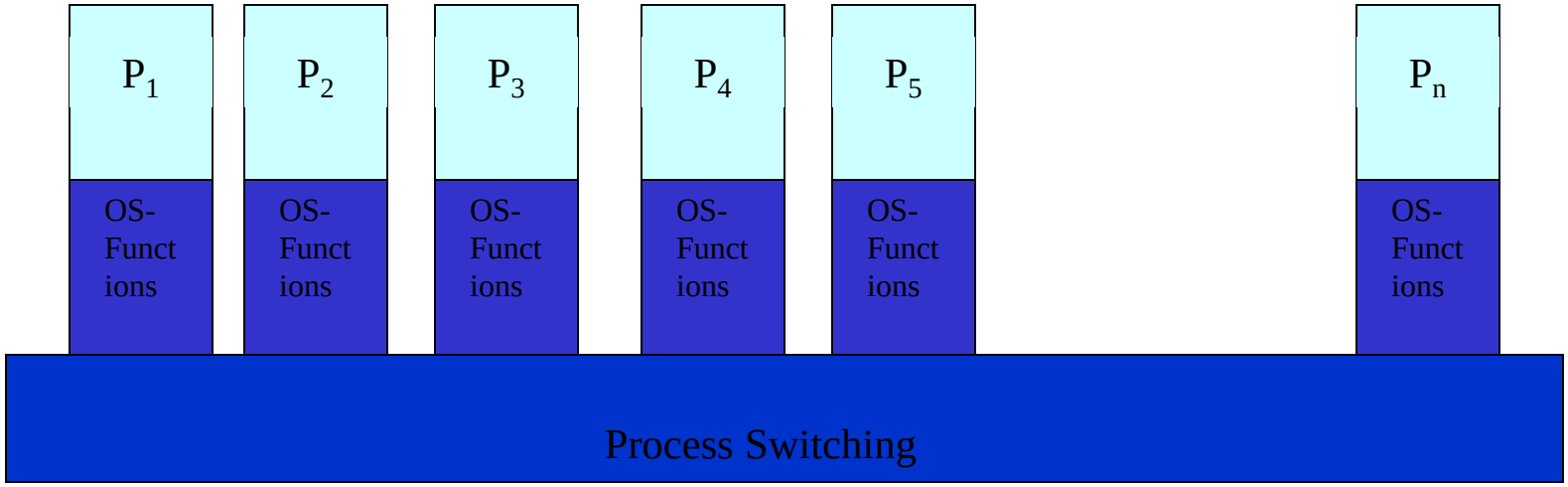
Intel Pentium Protection Checks

- Limit checks
- Type checks
- Restriction of addressable domain
- Restriction of procedure entry-points
- Restriction of instruction set
- *Violations cause exceptions!*

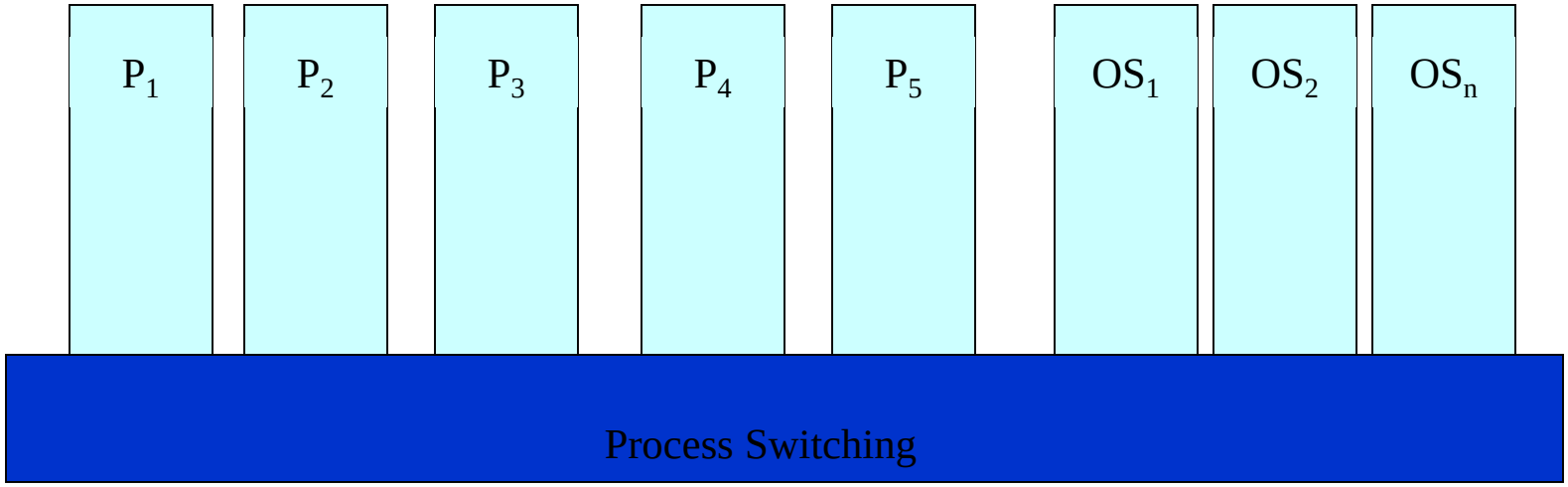
Processes & OS: Where is the OS executing: Separate Kernel



OS-Functions Executing within Processes

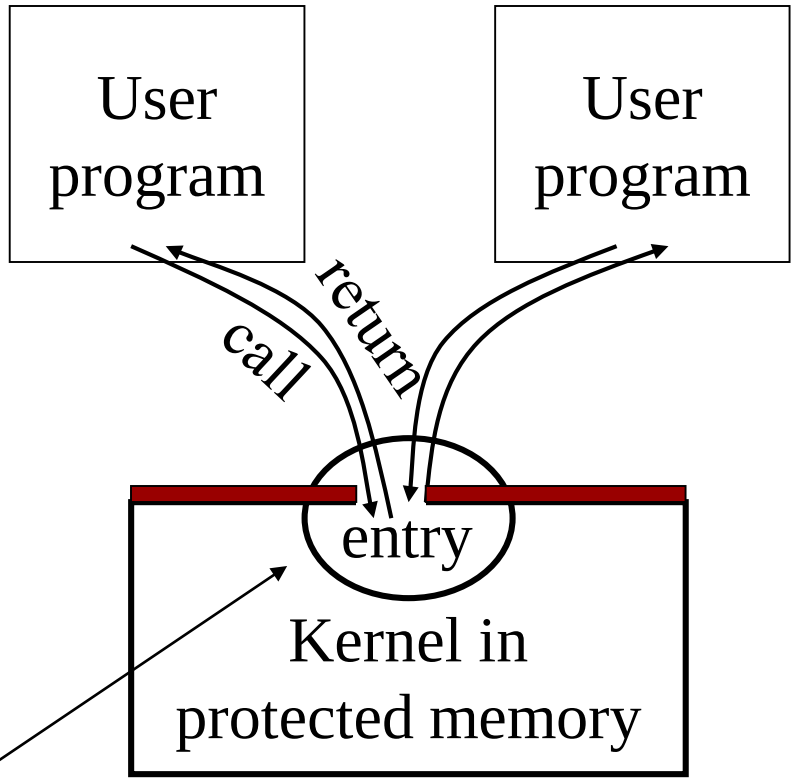


OS-Functions Executing in Separate Processes



System Call Mechanism

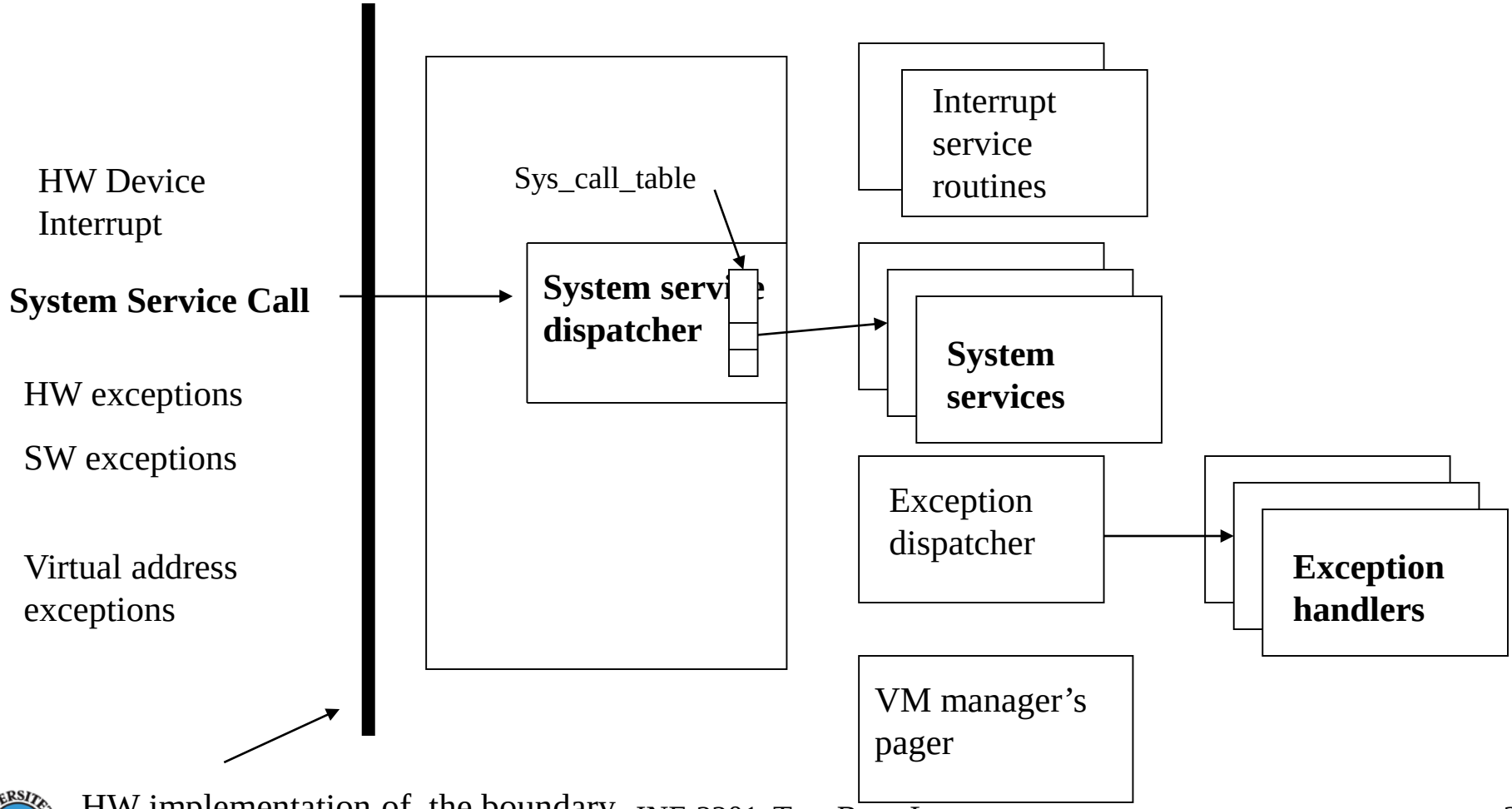
- User code can be arbitrary
- User code cannot modify kernel memory
- Makes a system call with parameters
- The call mechanism switches code to kernel mode
- Execute system call
- Return with results



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)

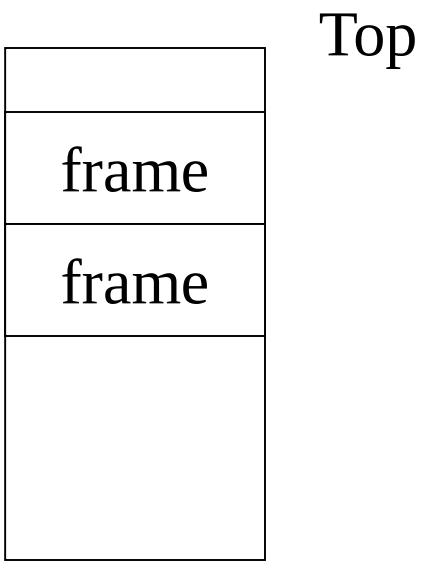
OS Kernel: Trap Handler



Passing Parameters

- Passing in registers
  - Simplest but limited
- Passing in a vector
  - A register holds the address of the vector
- Passing on the stack
  - Push: library
  - Pop: System

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



The Stack

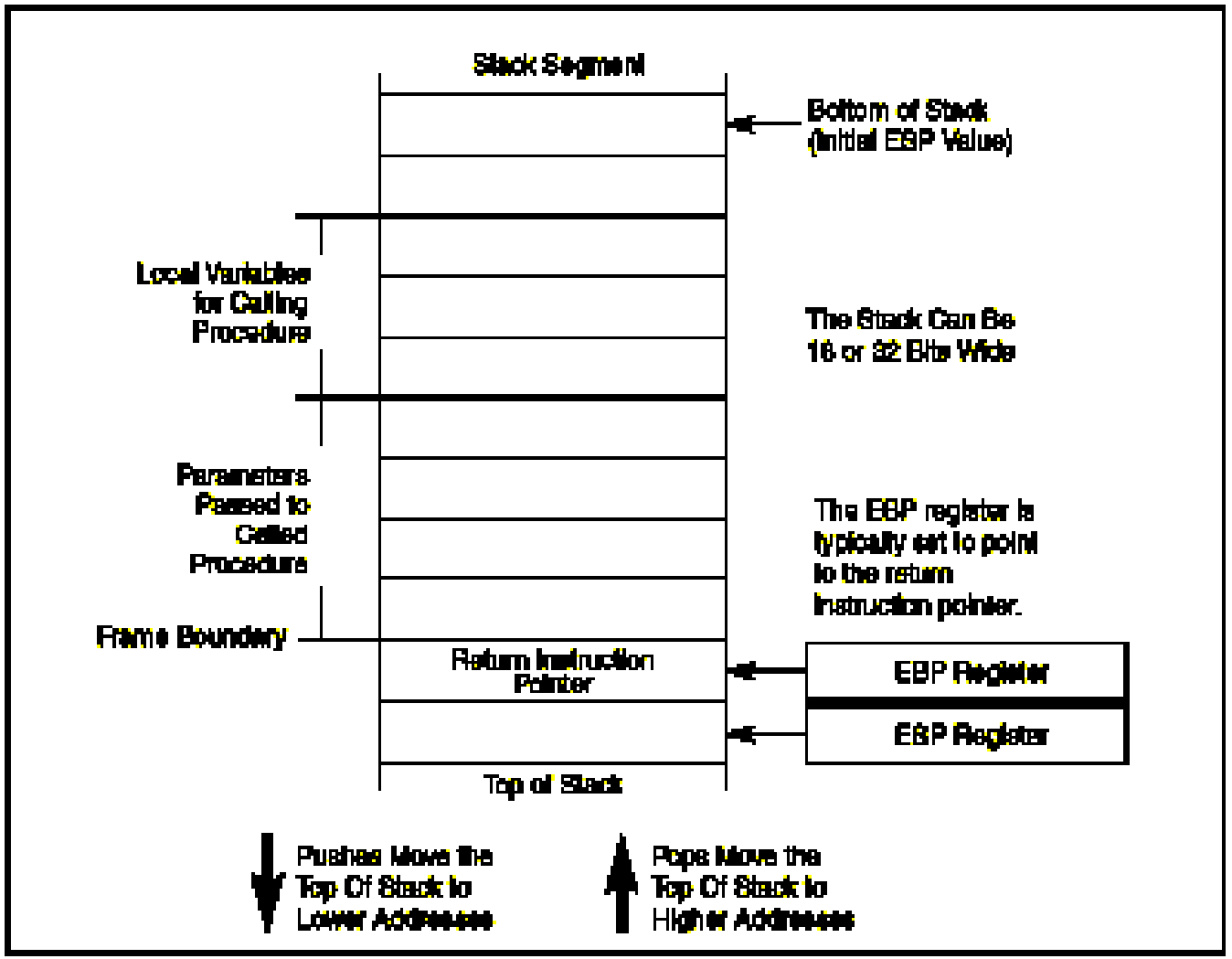


Figure 4-1. Stack Structure

- 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

Issues in System Call Mechanism

- Use caller's stack or a special stack?
  - Use a special stack
- Use a single entry or multiple entries
  - A single entry is simpler
- System calls with 1, 2, 3, … N arguments
  - Group system calls by # of args
- Can kernel code call system calls?
  - Yes and should avoid the entry

Library Stubs for System Calls

- 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
}
```

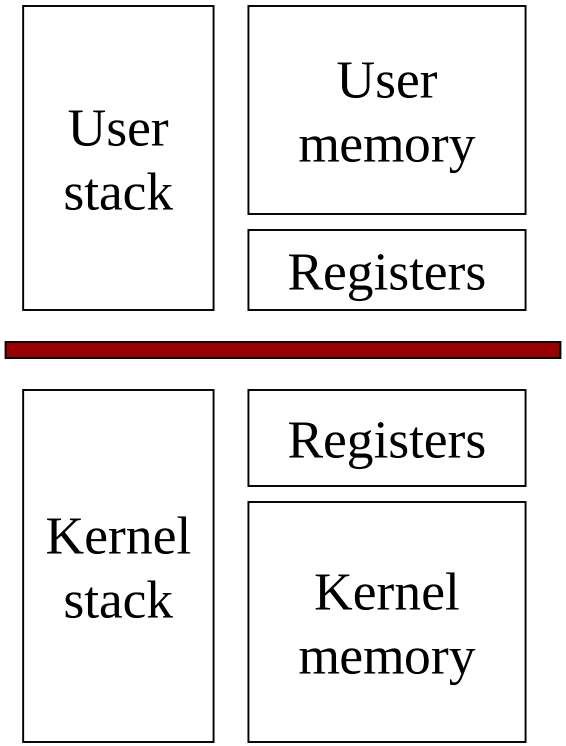
32-255 available to user

Return when work is done

Could be an error code

Win NT: 2E

Linux: 80



System Call Entry Point

int 0x80

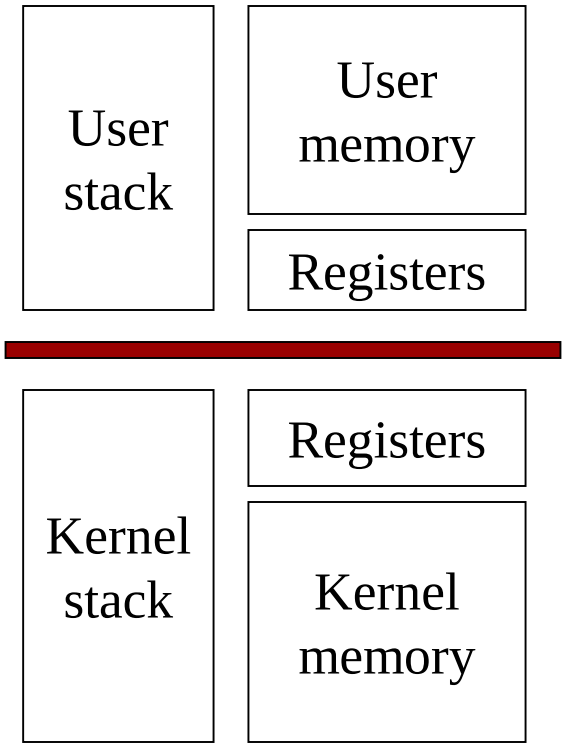
SW interrupt

- Assume passing parameters in registers

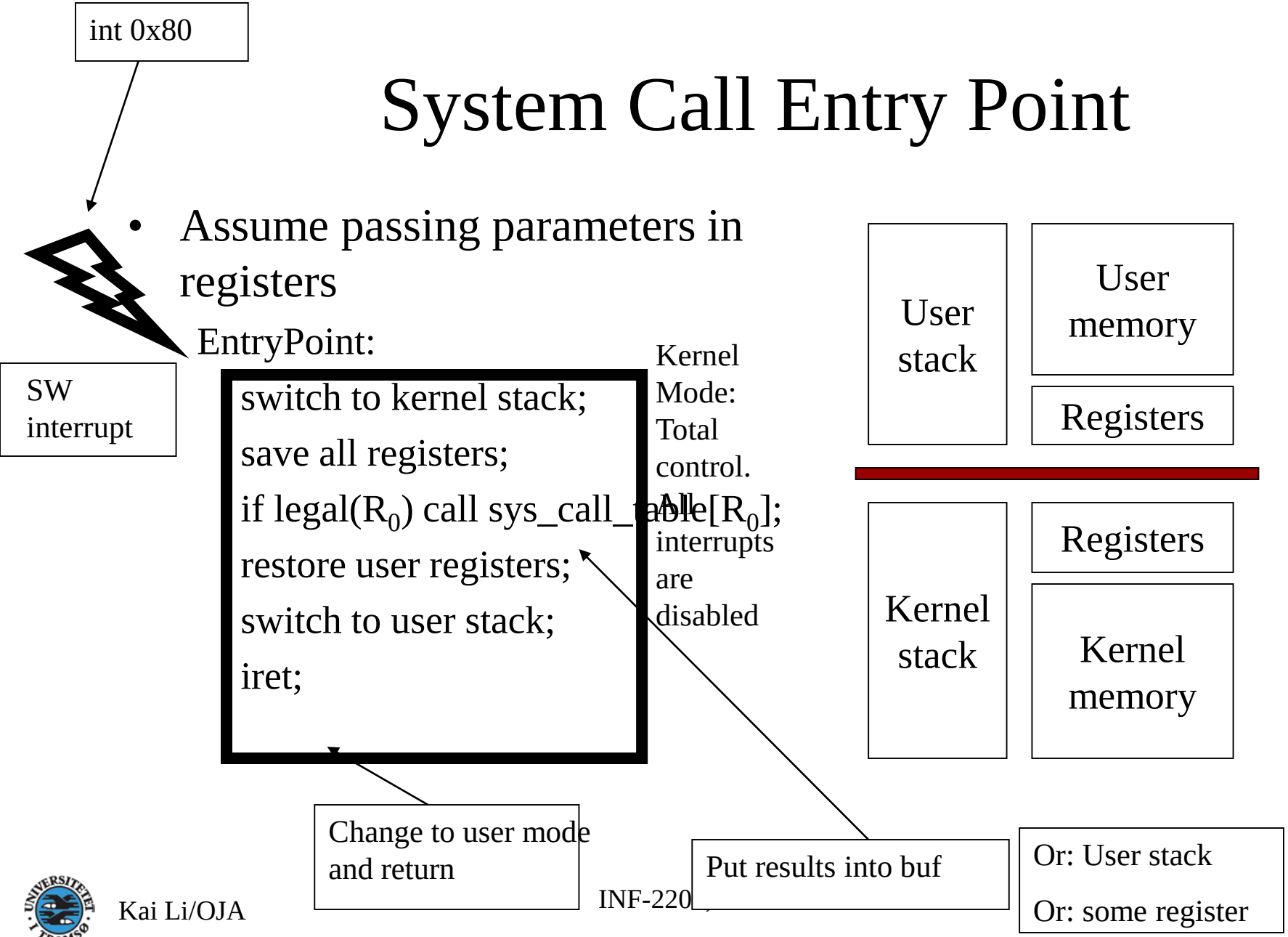
EntryPoint:

```
switch to kernel stack;
save all registers;
if legal(R0) call sys_call_table[R0];
restore user registers;
switch to user stack;
iret;
```

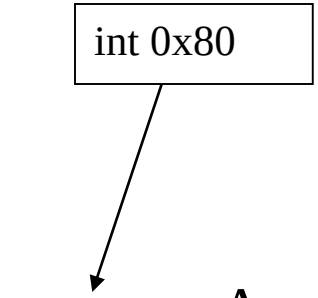
Kernel Mode: Total control. All interrupts are disabled



System Call Entry Point



System Call Entry Point



- Assume passing parameters in registers

EntryPoint:

```
switch to kernel stack;
save all registers;
if legal(R0) call sys_call_table[R0];
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?

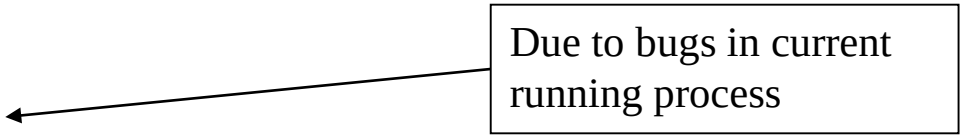
- OS kernel could check a request queue instead of using an interrupt?
  - Waste CPU cycles checking
  - All have to wait while the checks are being done
  - When to check?
    - Non-predictable
    - Pulse every 10-100ms?
      - » too long time

But used for Servers

- Same valid for HW Interrupts vs. Polling

Interrupts and Exceptions

- Processor exceptions
  - MMU address faults, divide by zero, etc
  - 386: the first 32 “interrupt descriptor table” entries are special descriptors, trap gates, mapping exceptions to handler code
- Interrupts from hardware
  - slow: int ON, usual, timer
  - fast: int OFF, less complex, keyboard
- Interrupts from software: sys calls



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