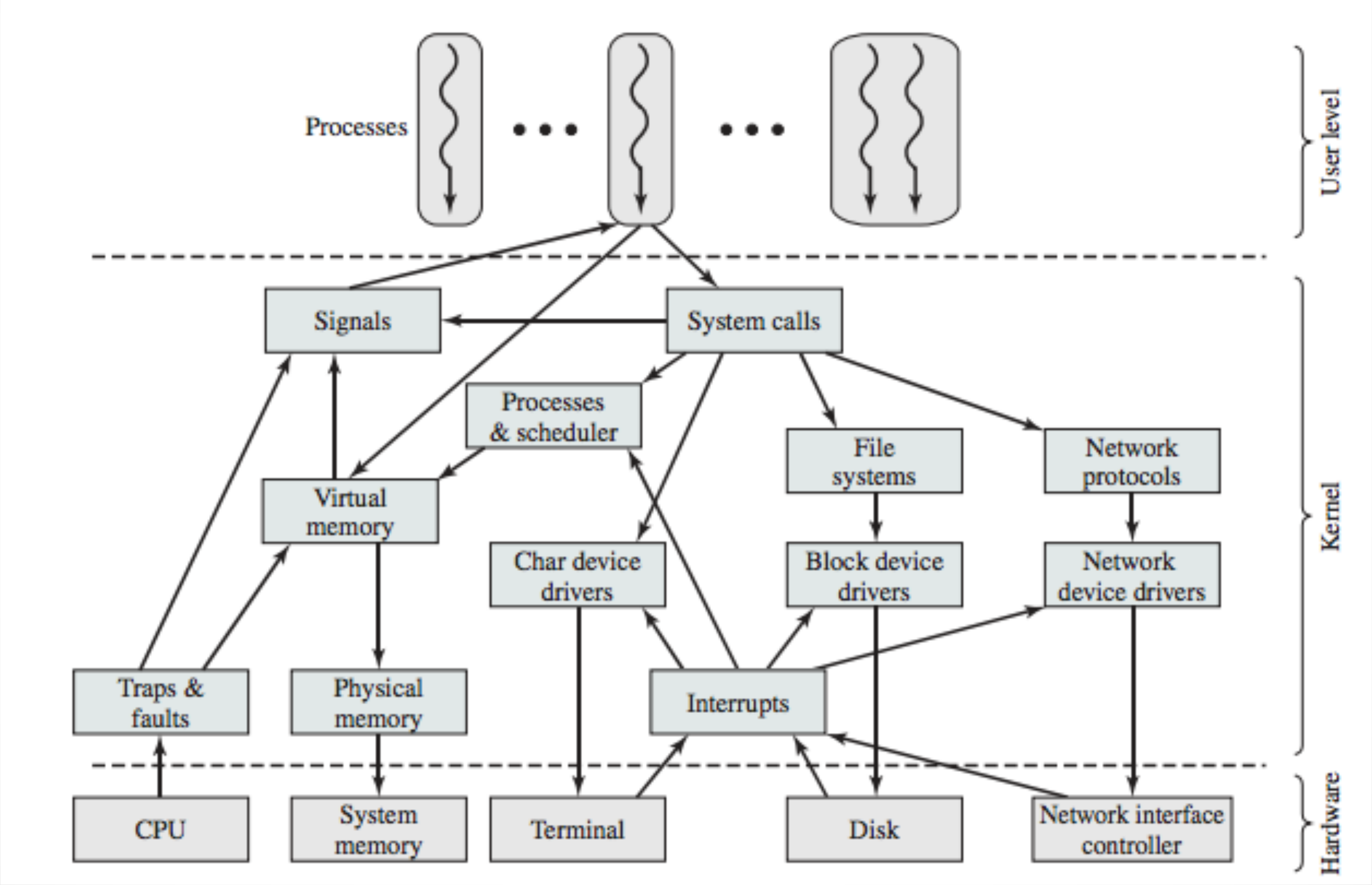
# Operating Systems Workings

Operating Systems Design

# Linux kernel components



# Contents

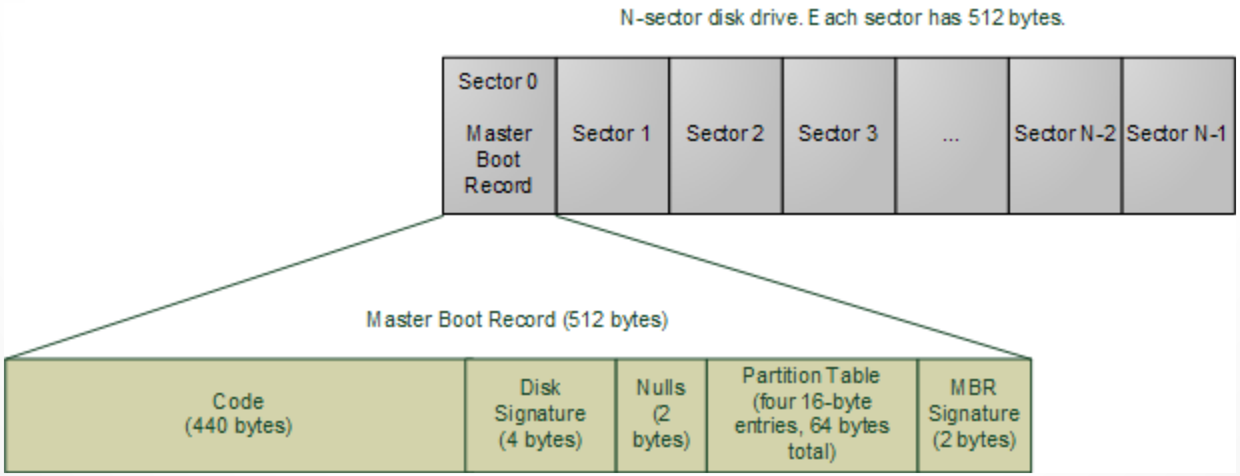
- **Operating system booting process**
- Operating system execution
- Operating system events
  - Hardware interrupts
  - Exceptions
  - System calls
  - Software interrupts
- Kernel processes

# Booting process

- Power the motherboard
  - Initializes its firmware
  - Get CPUs running
  - One CPU chosen bootstrap processor
    - runs Basic Input Output System (BIOS) and kernel init code in real mode (paging disabled)
    - Other CPUs halted
    - EIP: a jump to the memory location mapped to BIOS entry point (Intel: reset vector 0xFFFFFFF0)
- BIOS code
  - Init some hardware in the machine
  - Power-on Self Test tests some components
  - Reads Master Boot Record (MBR), typically from sector 0 of the hard disk
    - OS- specific bootstrapping program (starts to execute it)
    - Partition table

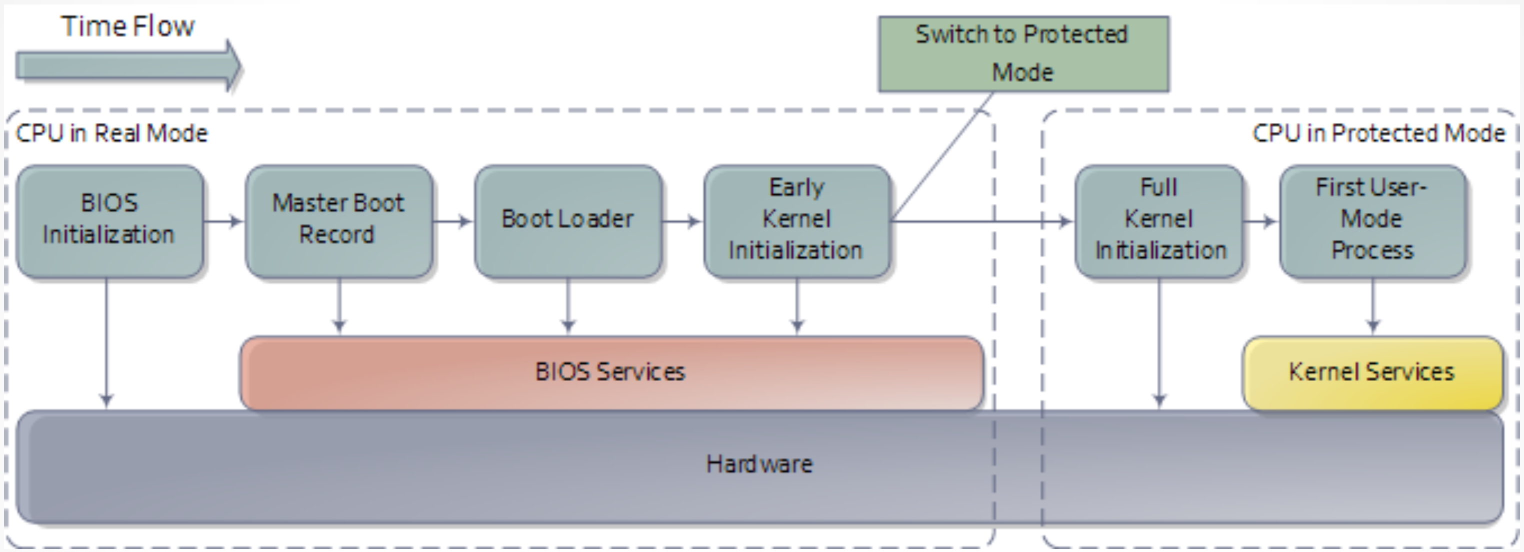
# Booting process

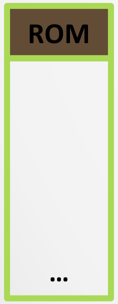
- MBR
  - Loader
    - Windows MBR: loads the only active partition from partition table and runs the boot sector
    - LILO or GRUB: loads a disk sector containing the bootstraper offering various boot options
  - Partition table:64 byte area with 4 16-byte entries
- Start OS kernel



# Boot sequence

PC

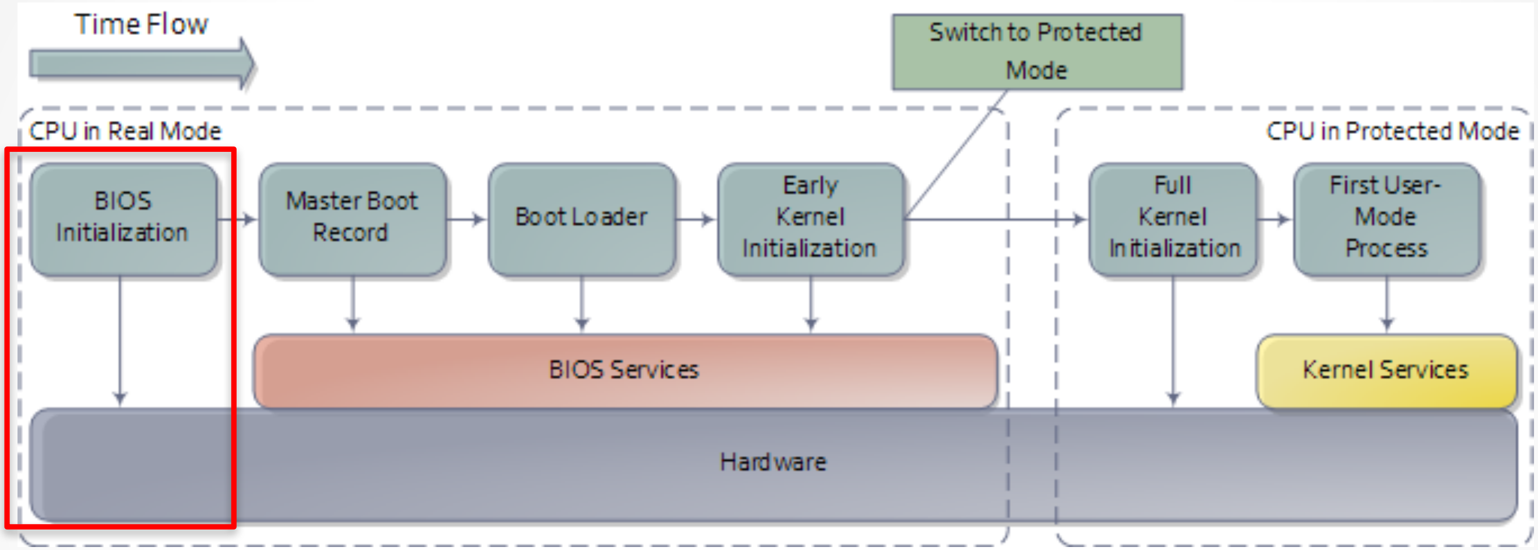


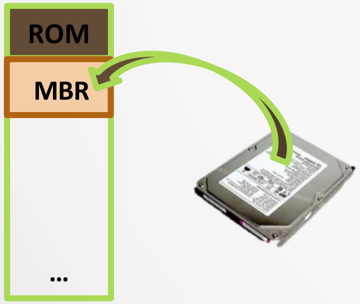




- The *Reset* loads the predefined values on CPU registers
  - PC ← Boot address of the ROM loader (FFFF:0000)

# Boot sequence

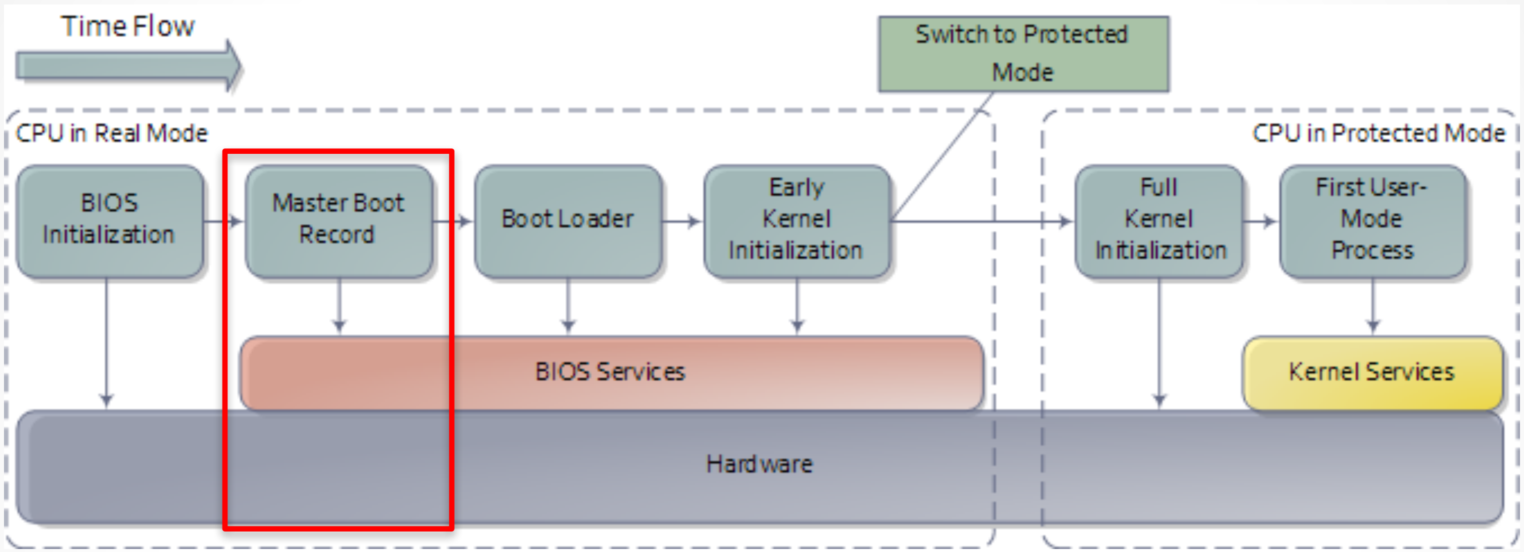
PC

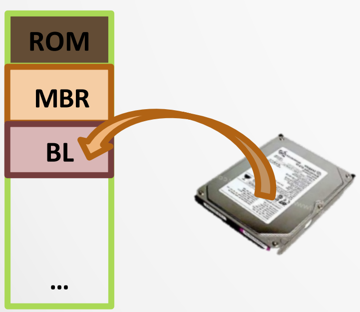



- The ROM loader is executed
  - *Power-On Self Test* (POST)
  - Load into memory (0000:7C00) the Master Boot Record

# Boot sequence
PC

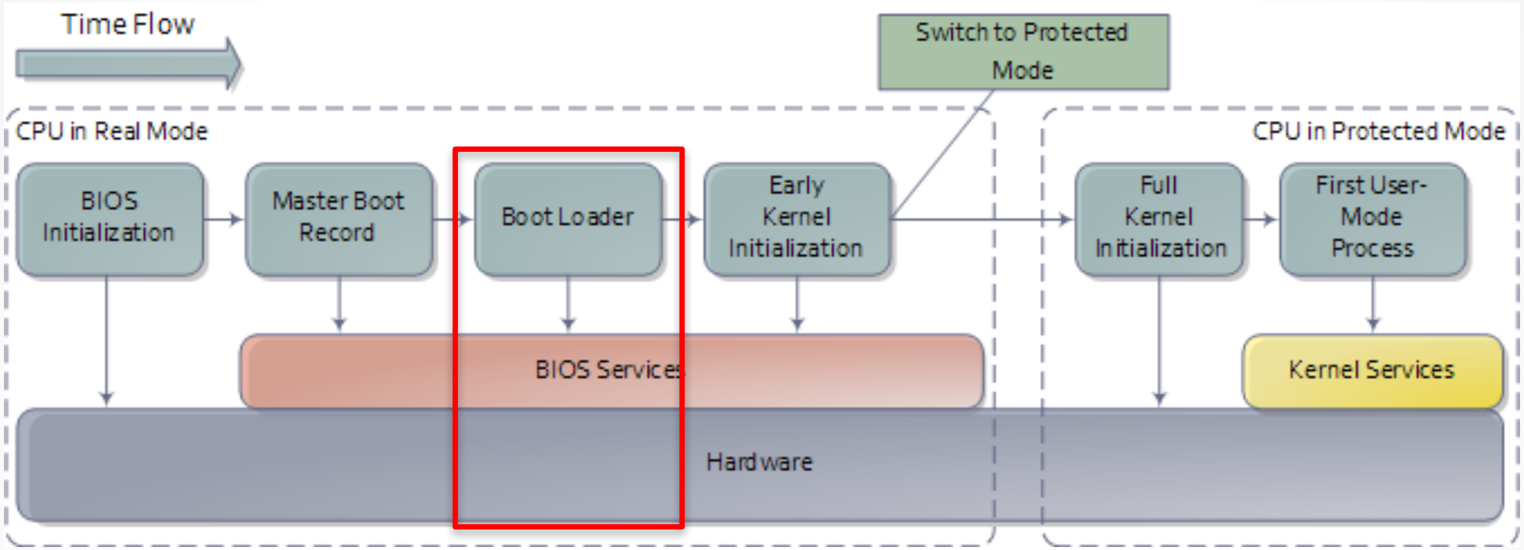


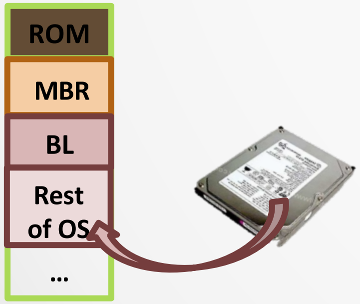




- The Master Boot Record is executed
  - (It is the first part of the OS Loader)
  - Scan the partition table for an active partition.
  - Load the Boot Record from this partition into memory.

# Boot sequence

PC

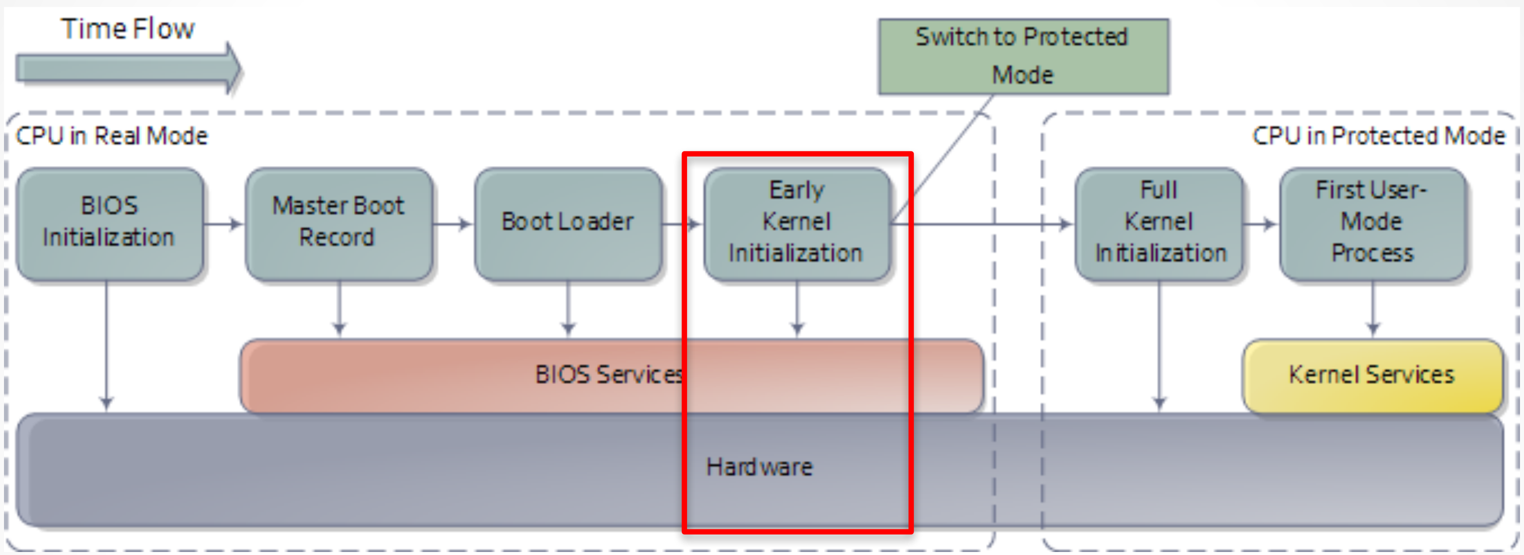


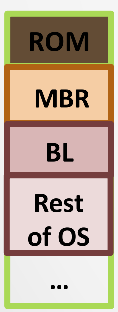




- The Boot Loader is executed
  - (It is the second part of the OS Loader)
  - The boot loader brings up into memory the stay-resident part of the operating system (kernel and modules)
  - It might list several booting options (debugging, single, etc.)

# Boot sequence

**PC**

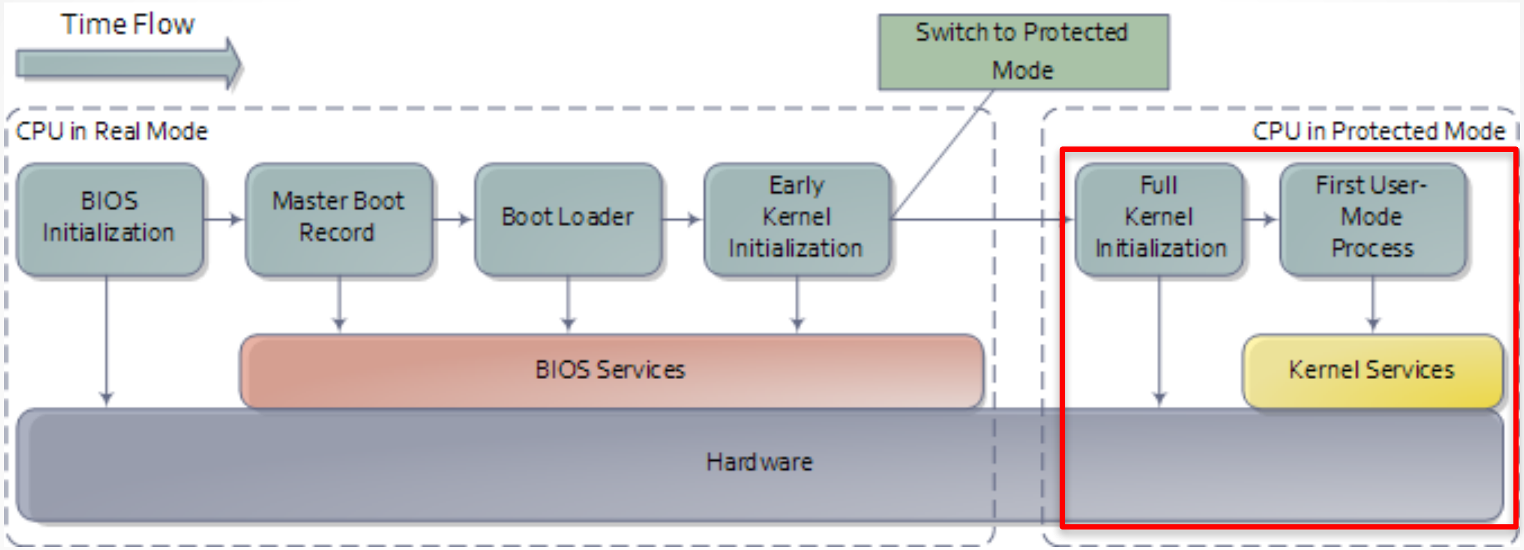


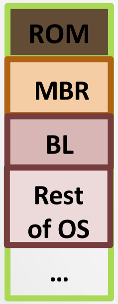




- The Kernel Initialization is executed
  - Hardware initialization
  - It checks file systems for errors
  - It sets the initial internal structures for the operating system
  - Switch to protect mode…

# Boot sequence

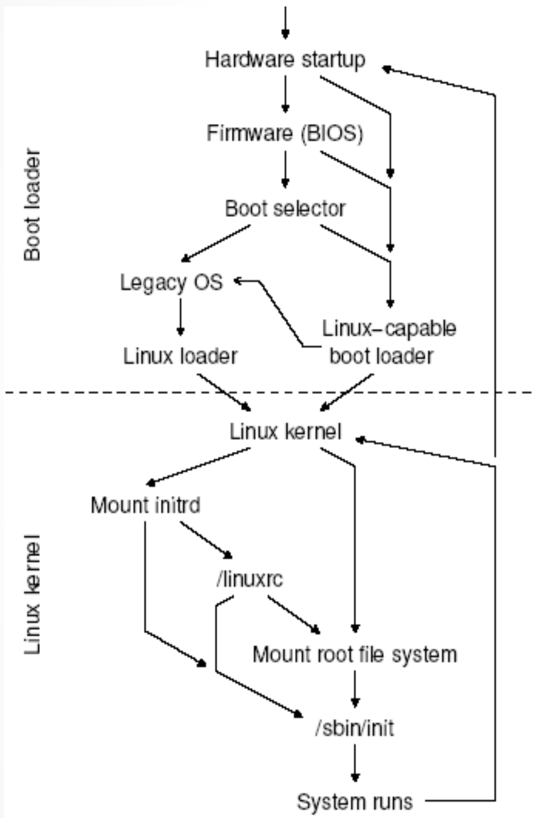
PC



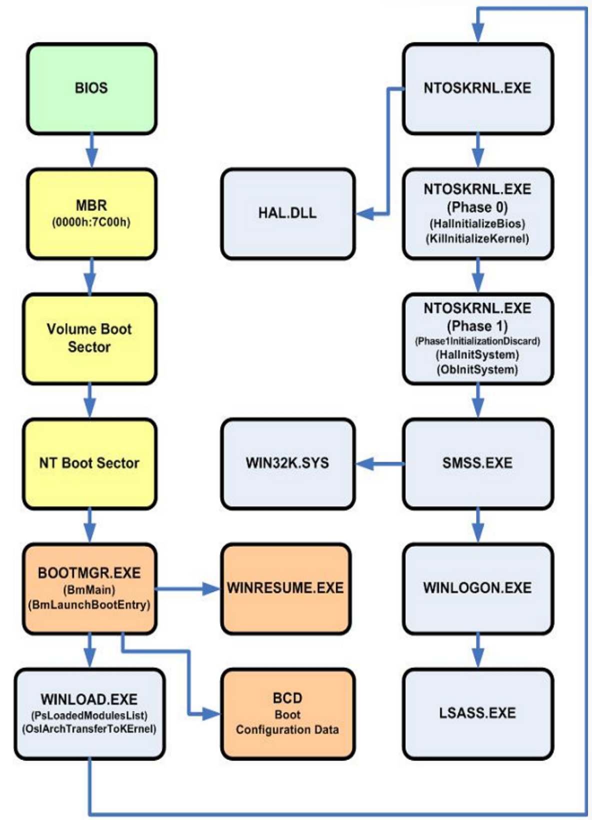




- The Kernel Initialization is executed
  - It sets the rest of structures and tasks in protected mode
  - It builds the initials processes
    - Kernel processes, System services, and login

# Examples of boot sequences



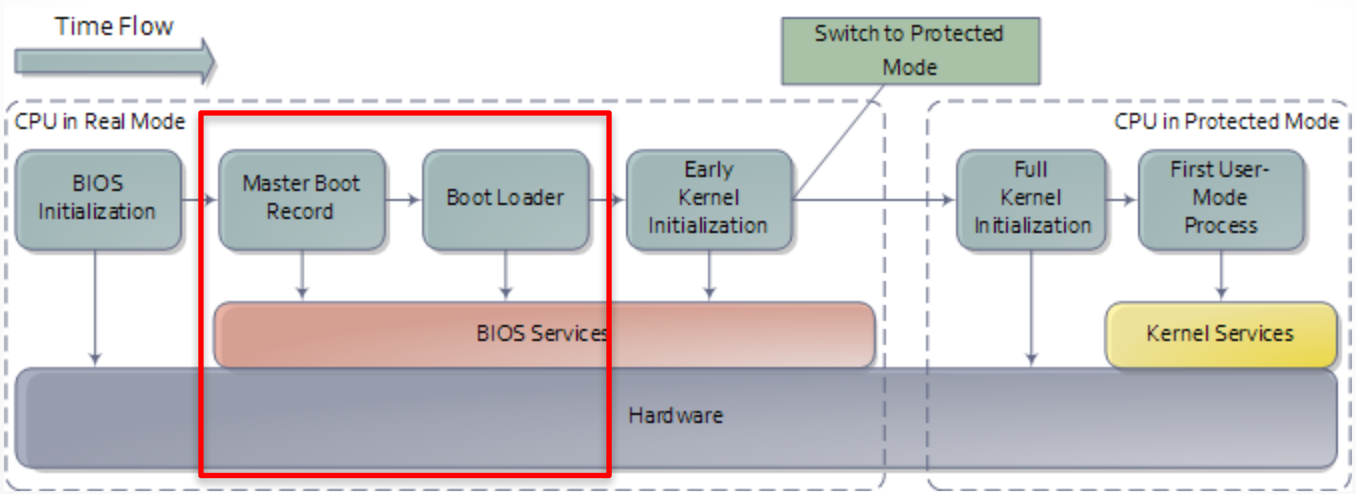

- GNU-Linux




- MS Windows

# GNU-Linux PC (1)

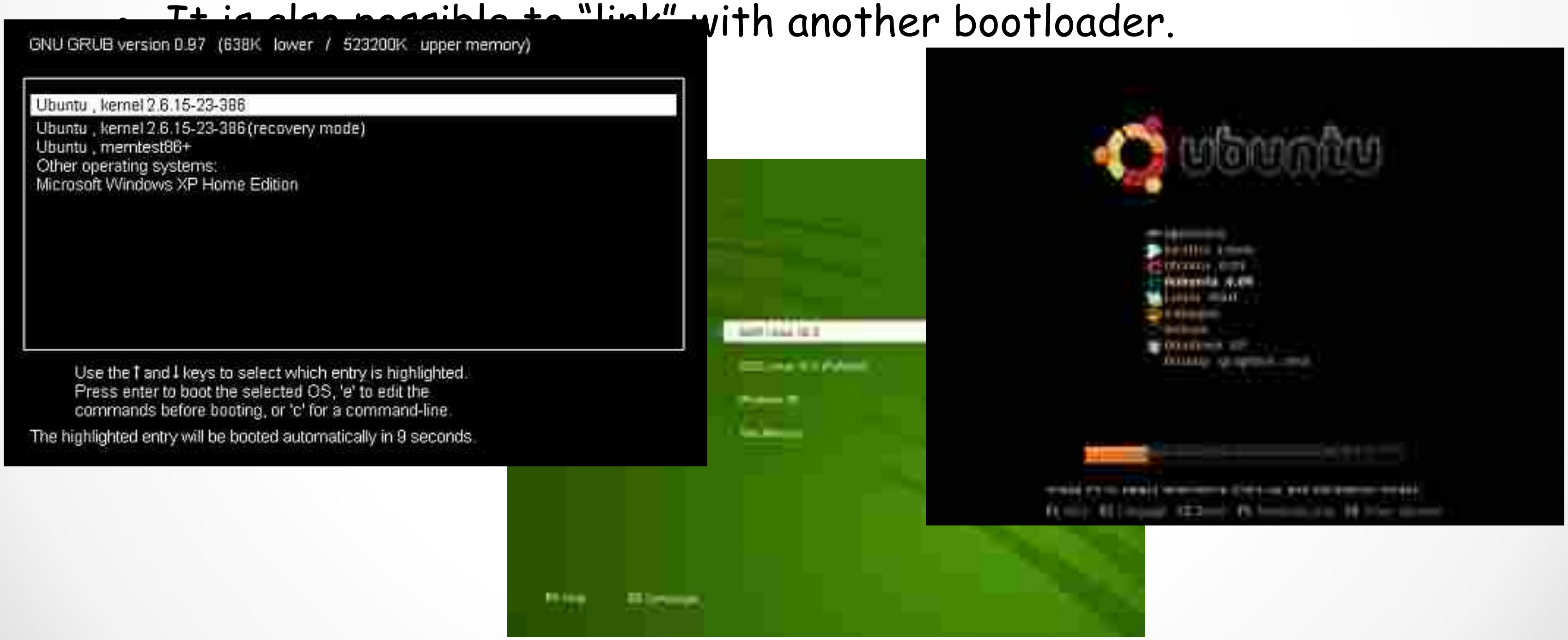
- The LILO (*Linux Loader*) or GRUB (*Grand Unified Bootloader*) is executed
  - LILO or GRUB are the combination of both, first and second phase loader.
  - The kernel image (vmlinuz) is loaded into memory, and this image is executed with the predefined parameters/options.




http://www.ibm.com/developerworks/linux/library/l-linuxboot/

# GNU-Linux PC (1)

- The LILO (*Linux Loader*) or GRUB (*Grand Unified Bootloader*) is executed
  - In the second phase, GRUB shows a menu with several possible configuration (kernel with associated parameters) taken from /etc/grub.conf
  - It is also possible to "link" with another bootloader.



# GNU-Linux (1)

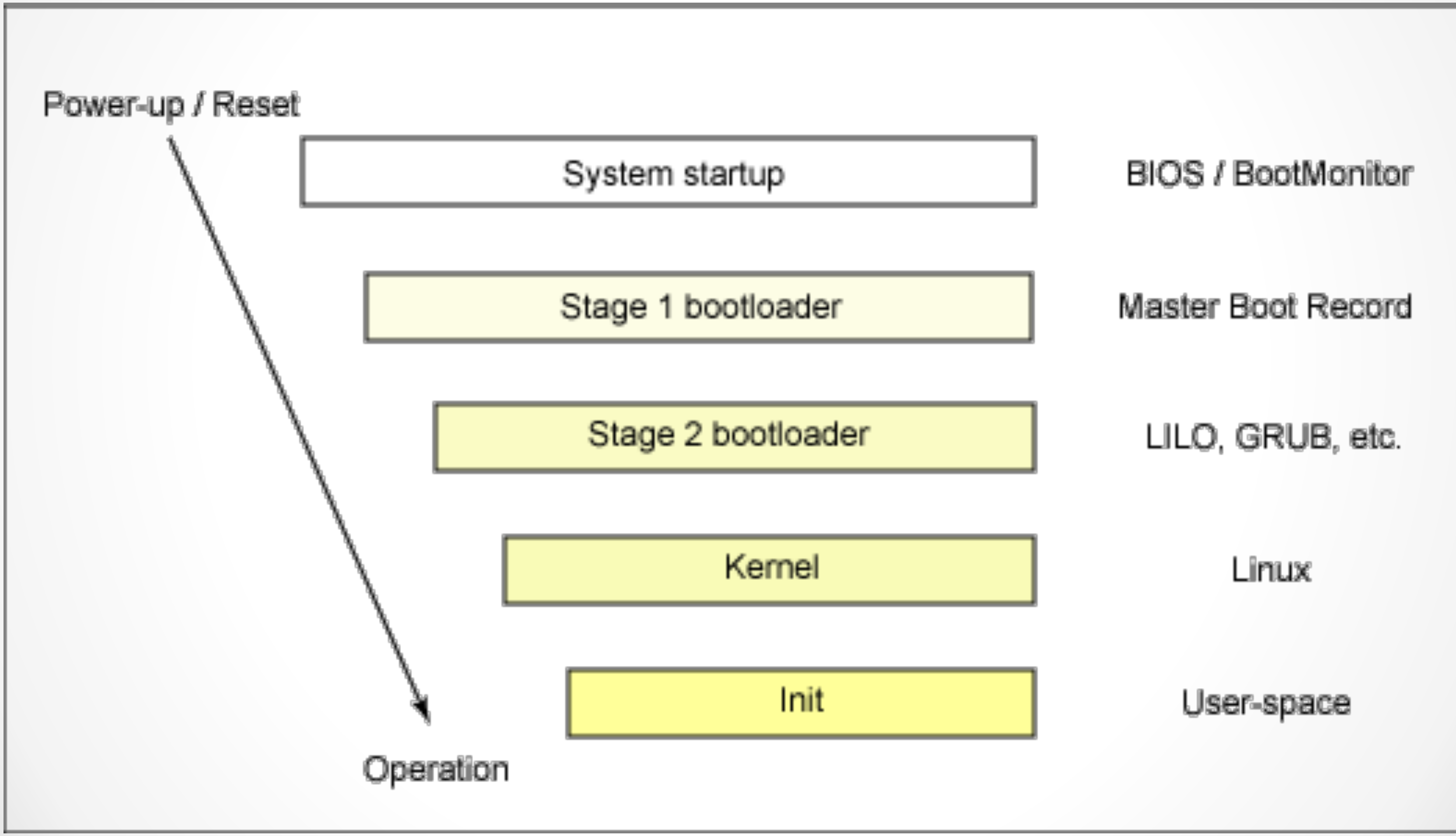
- The LILO (*Linux Loader*) or GRUB (*Grand Unified Bootloader*) is executed
  - Nowadays the bootloader has an elaborated graphic user interface, but can be used through a normal command line interface.
  - It is possible to modify the boot options (for this specific OS boot) in a simple way.

```
grub> kernel /bzImage-2.6.14.2
  [Linux-bzImage, setup=0x1400, size=0x29672e]

grub> initrd /initrd-2.6.14.2.img
  [Linux-initrd @ 0x5f13000, 0xcc199 bytes]

grub> boot
```

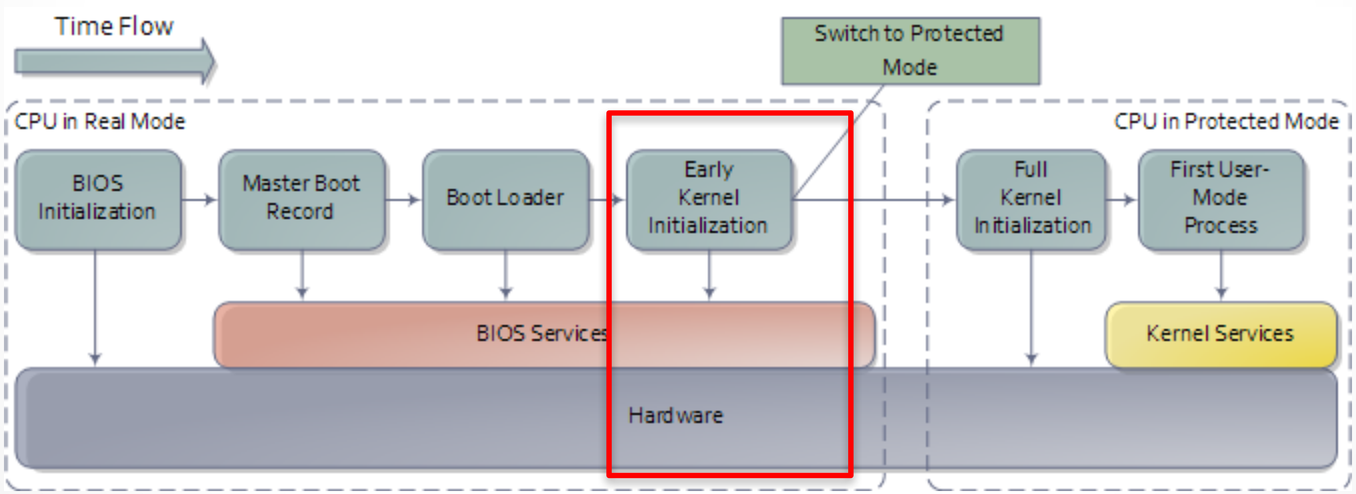
# Linux boot process




http://www.ibm.com/developerworks/linux/library/l-linuxboot/

# GNU-Linux (2)

- The CPU executes the initial kernel image (vmlinuz)
  - Hardware initialization (and change to protect mode)
  - The initial RAM disk is loaded (initrd).

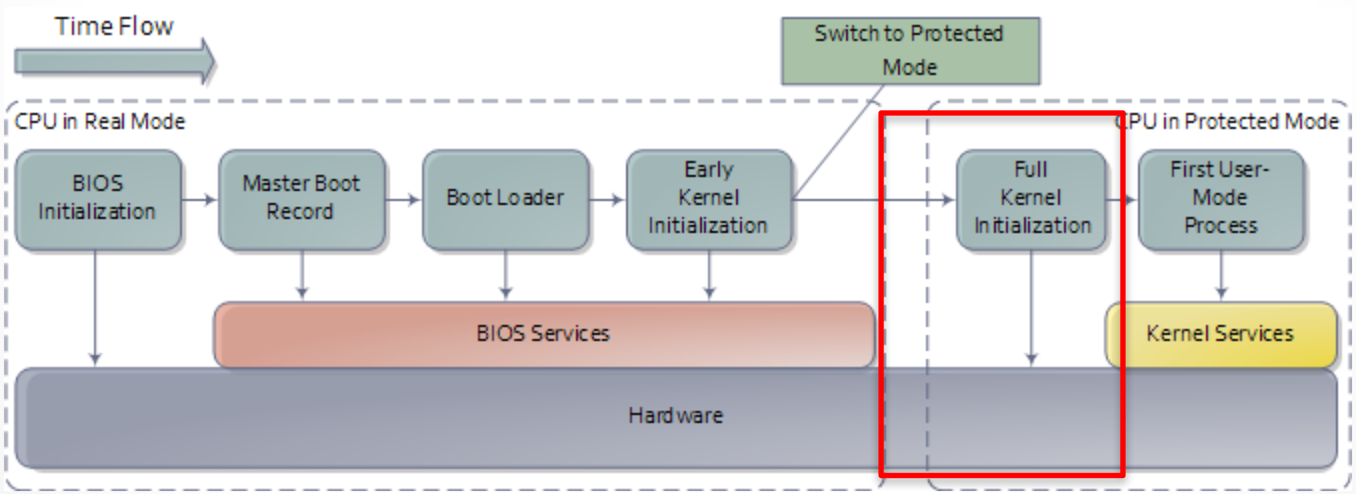


# GNU-Linux (2)

- The CPU executes the initial kernel image (vmlinuz)
  - The hardware is detected and for those hardware elements having its drivers compiled in the initial kernel, they are initialized.
  - The swapper (process 0) creates the in-memory kernel data structures.

# GNU-Linux (3)

- The CPU executes the rest of the kernel image (initrd)
  - The initrd is the initial root file, and it contains the additional drivers needed to keep booting the system.
  - The init process (process 1) is loaded and is executed (if it is necessary).





http://www.ibm.com/developerworks/linux/library/l-linuxboot/

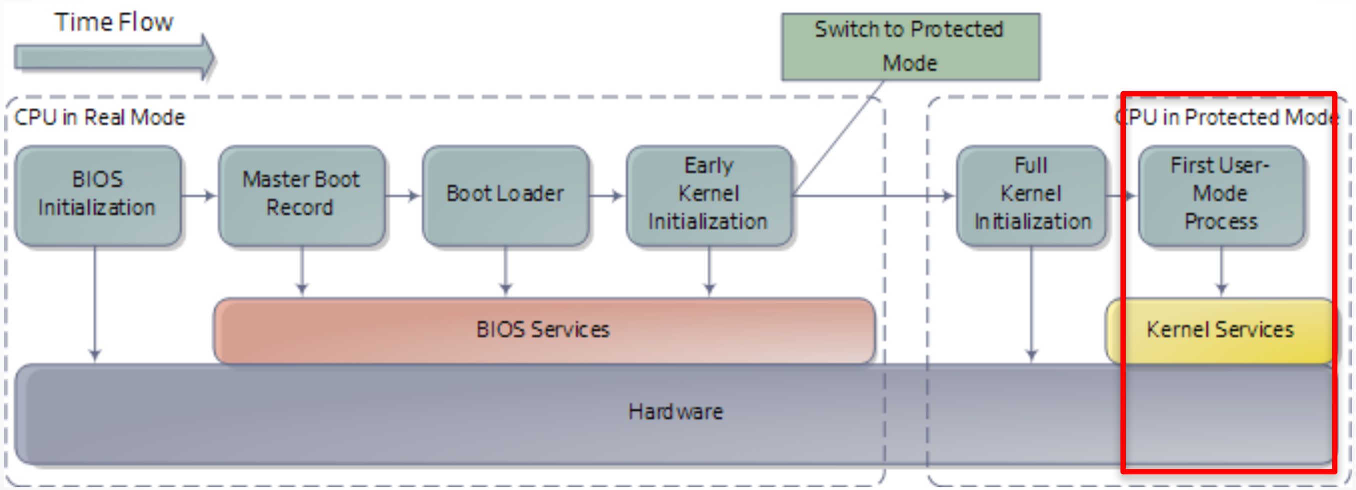
# GNU-Linux (3)

- The CPU executes the rest of the kernel image (initrd)
  - The initrd will 'pivot' to the end-root file system.
    - It might be itself (embedded systems), hard disk partition, NFS, etc.
  - From there, it is executed the init process that will end the boot process…

```
Initializing basic system settings ...
Updating shared libraries
Setting hostname: engpc23.murdoch.edu.au
INIT: Entering runlevel: 4
rc.M ==> Going multiuser...
Starting system logger ...                                   [  OK  ]
Initialising advanced hardware
Setting up modules ...                                       [  OK  ]
Initialising network
Setting up localhost ...                                     [  OK  ]
Setting up inet1 ...                                         [  OK  ]
Setting up route ...                                         [  OK  ]
Setting up fancy console and GUI
Loading fc-cache ...                                         [  OK  ]
rc.vlinit ==> Going to runlevel 4
Starting services of runlevel 4
Starting dnsmasq ...                                         [  OK  ]
==> rc.X Going to multiuser GUI mode ...
XFree86 Display Manager
Framebuffer /dev/fb0 is 307200 bytes.
Grabbing 640x480 ...
```

# GNU-Linux (4)

- The CPU executes the init process
  - The init process starts the system processes…
  - … and the terminals (*login* o *xlogin*) for user autentication.
  - Then, it will sleep for events to come (cpu_idle)

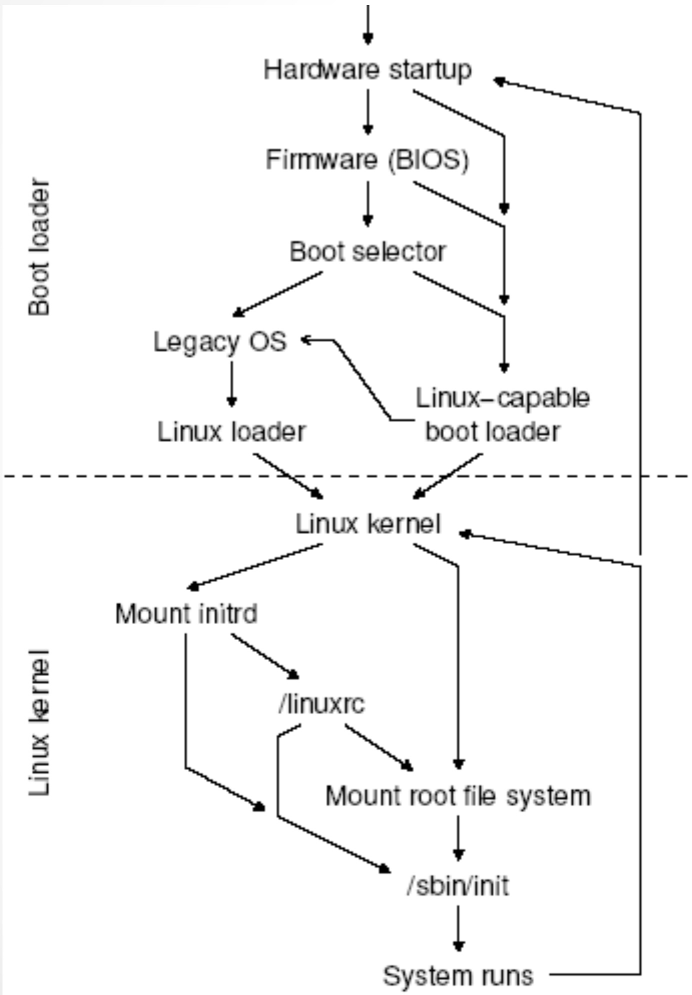


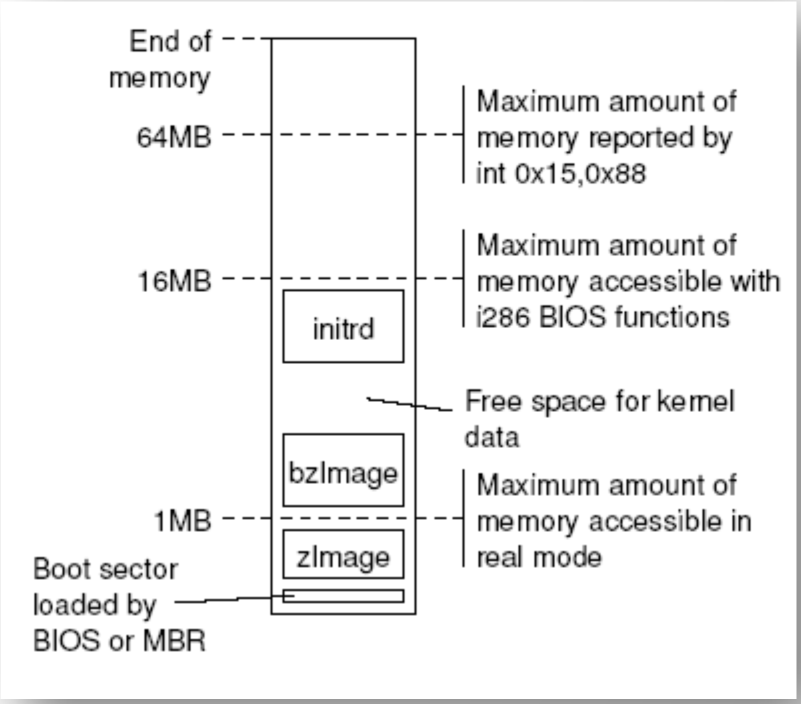
# GNU-Linux (4)

- The CPU executes the init process
  - The init process starts the system processes.
    - Following the inittab configuration.
    - The process to be executed are ordered in several boot directories
      - One directory for the initial boot (rcS.d)
      - Five for the typical system boot profiles (rc[1-5].d)
      - One for shutdown (rc0.d) and another for reboot (rc6.d)
    - The init process will stay around all the time (orphans, stopping, etc.)
  - During the initial boot (rcS.d) usually:
    - The rest of drivers are loaded.
    - The root file system is checked (if needed) and is remounted for read-write.
    - The rest of file systems are checked and mounted.
  - During the the typical system boot (rc1.d, rc2.d, …)
    - The services (daemons) are started, an the login/xlogin is executed.

http://www.ibm.com/developerworks/linux/library/l-linuxboot/

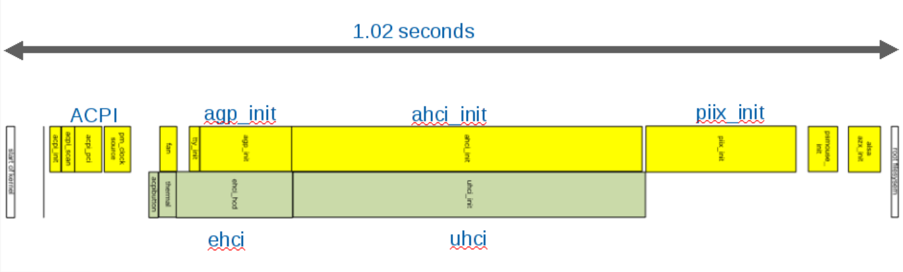
# GNU-Linux boot process



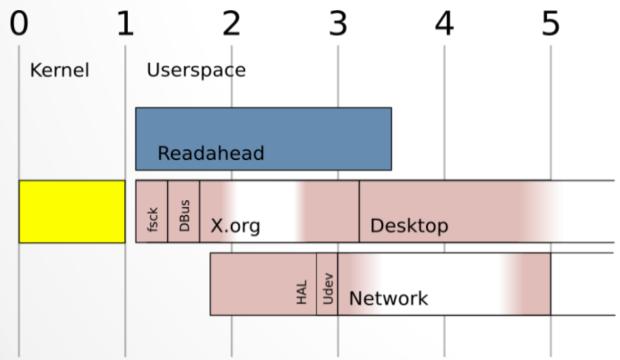



# Speeding Linux Boot

- Hardware asynchronous initiation



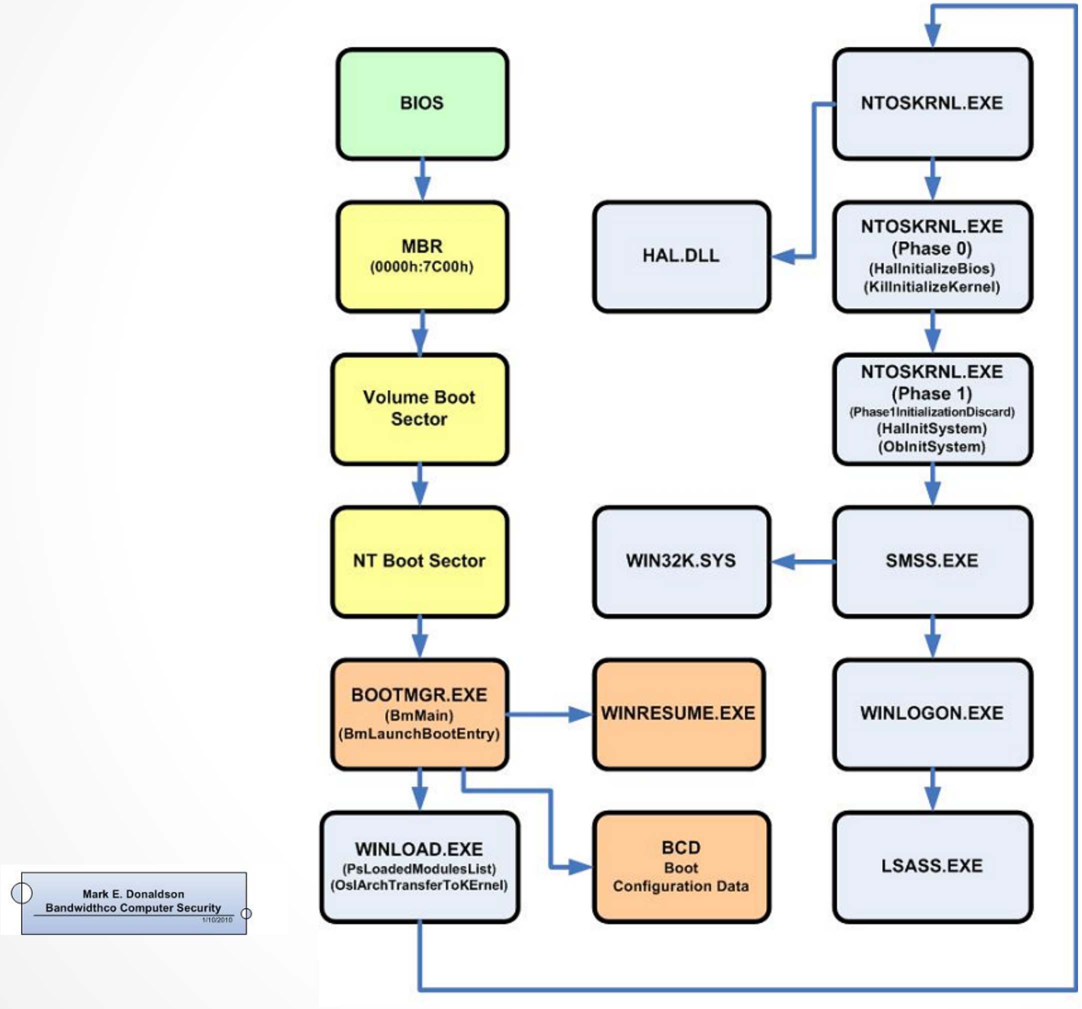

- Services asynchronous initiation



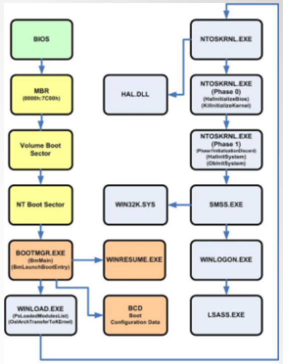

Bootchart


http://www.digitaltrendshttp://lwn.net/Articles/299483/

# Windows 7

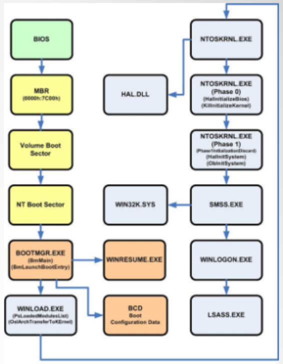

BIOS
MBR
(0000h:7C00h)
Volume Boot Sector
NT Boot Sector
BOOTMGR.EXE
(BmMain)
(BmLaunchBootEntry)
WINLOAD.EXE
(PsLoadedModulesList)
(OslArchTransferToKErnel)
HAL.DLL
WIN32K.SYS
WINRESUME.EXE
BCD
Boot
Configuration Data
NTOSKRNL.EXE
NTOSKRNL.EXE
(Phase 0)
(HalInitializeBios)
(KiInitializeKernel)
NTOSKRNL.EXE
(Phase 1)
(Phase1InitializationDiscard)
(HalInitSystem)
(ObInitSystem)
SMSS.EXE
WINLOGON.EXE
LSASS.EXE
Mark E. Donaldson
Bandwidthco Computer Security

# Windows 7 (1)



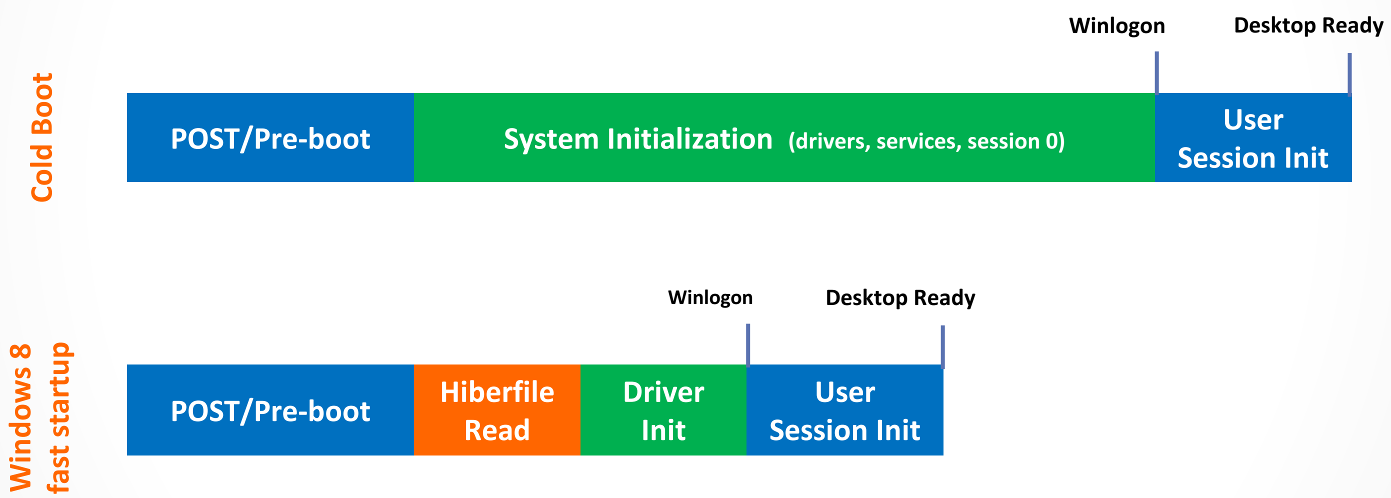

- The CPU executes the MBR
  - It finds and loads in memory the volume boot sector and the NT boot sector (8 KB, it understands FAT32 and NTFS)
- The CPU executes the NT boot sector (NTBS)
  - It finds and loads in memory the BOOTMGR.EXE
- The CPU executes the BOOTMGR.EXE
  - It checks for hibernation image. If available then executes WINRESUME.EXE
  - It mounts and extract the basic information of the BCD (*Boot Configuration Data*)
  - It shows a user menu with several boot options.
  - It moves to 64 bits mode (if available) and loads WINLOAD.EXE in memory.
- The CPU executes the WINLOAD.EXE
  - It loads NTOSKRNL.EXE, HALL.DLL, and drivers in memory for booting, with the SYSTEM branch of the registry tree.

# Windows 7 (2)




- The CPU executes the NTOSKRNL.EXE, that initialize the system in two phases:
  - Phase 0: It initializes the kernel itself.
    - HAL Initialization, starts the screen driver, boots the debugger.
  - Phase 1: It initializes the system.
    - It loads the needed drivers and stops the debugger.
    - At the end of the phase, it loads the first user process (smss.exe).
- The CPU executes the SMSS.EXE
  - The session manager loads the rest of the registry.
  - It configures the environment to execute the Win32 subsystem (WIN32K.SYS)
  - It loads in memory the WINLOGON.EXE process.
  - It loads the rest of services and non-essential drivers (to early show the desktop)
  - It loads the security subsystem LSASS.EXE

# Fast startup on Windows 8




http://www.digitaltrends.com/computing/windows-8-boot-time-scaled-down-to-eight-seconds/

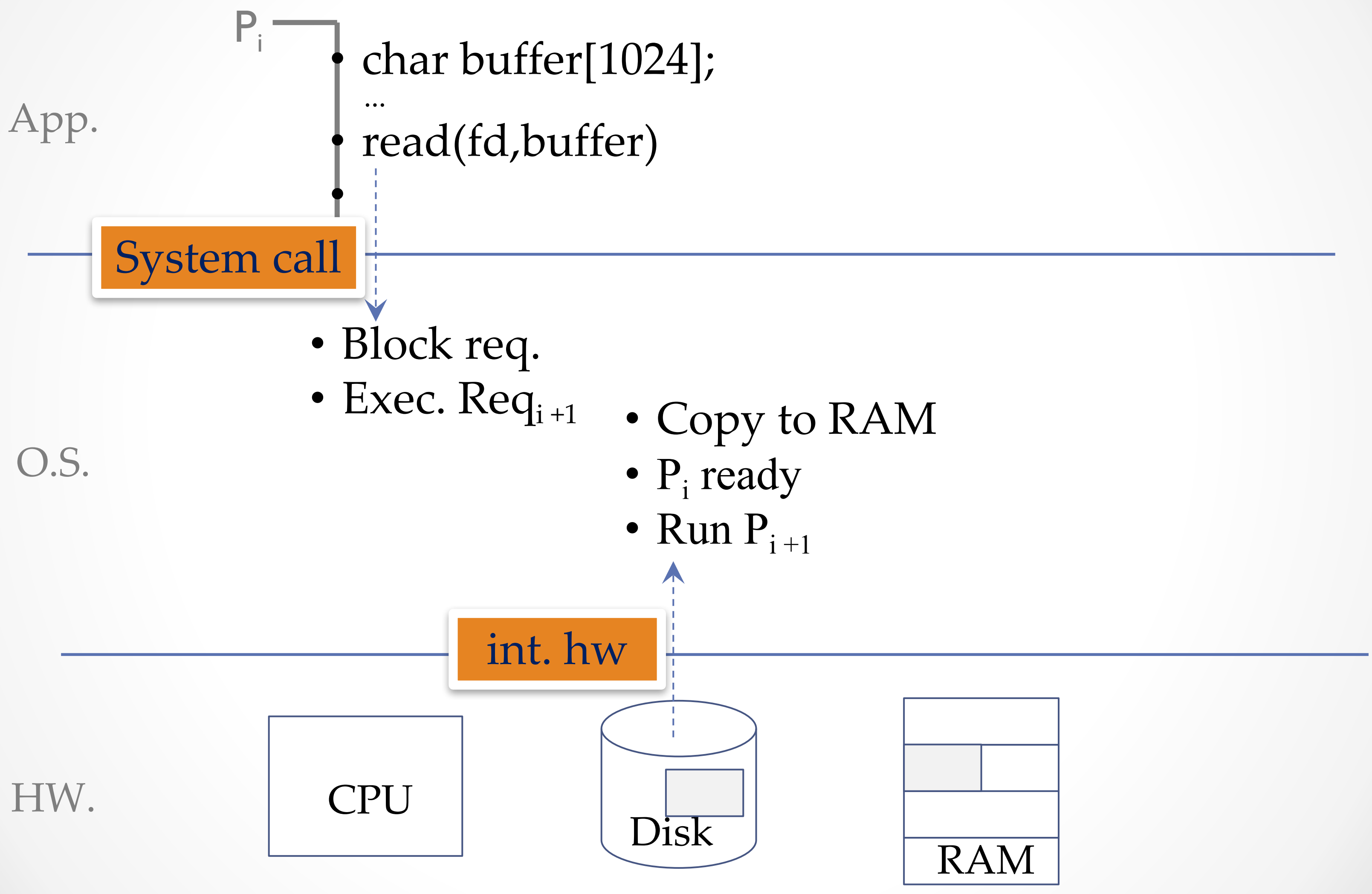
# Contents

- Operating system booting process
- **Operating system execution**
- Operating system events
  - Hardware interrupts
  - Exceptions
  - System calls
  - Software interrupts
- Kernel processes

# Operating System Execution

- Kernel utilization:
  - As an executable program
    - Only for booting
  - As library
    - OS: passive entity
    - Processes: active entities using the kernel as a library
    - Access to OS services through syscalls
    - Interrupts and exceptions
    - There is always one process executing (or idle process)
  - As kernel processes
    - Only special tasks, eg. swapping

# Example as a library

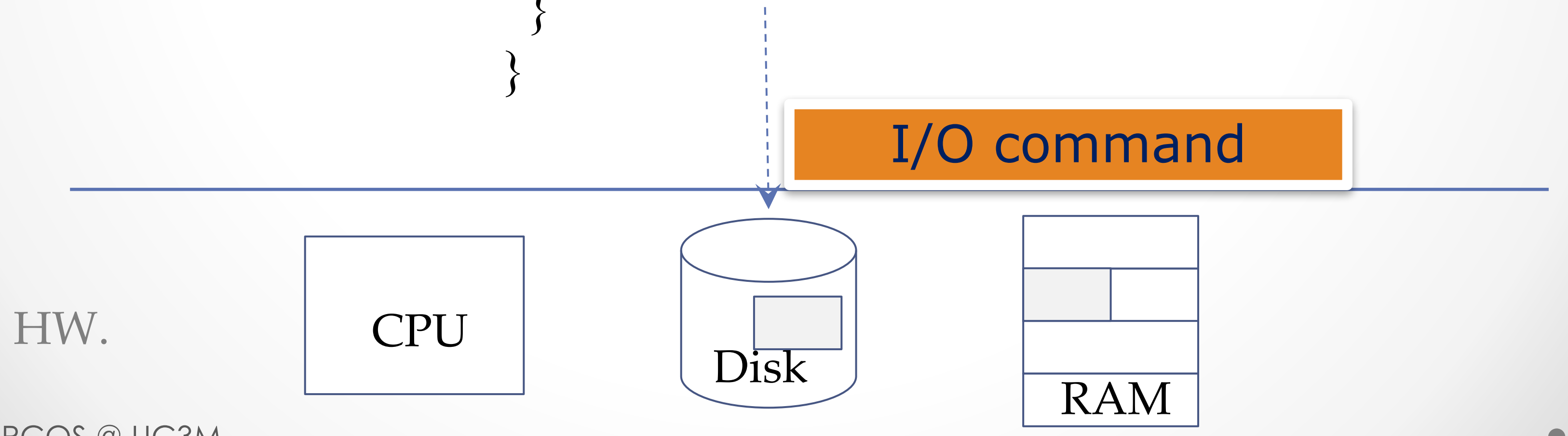


# Kernel process example

- idle_disk initially TRUE
- A system call for a disk data sets idle_disk to FALSE
- A disk interrupt serving the last pending request sets idle_disk to TRUE
- This example performs busy waiting

O.S.

```
while (1) {
  while (idle_disk) {
      • Sleep(1);
      • idle_time ++;
      • If (idle_time > 30 s)
            Sleep disk
  }
}
```



# Quiz

When an event occurs:

a) The mode is changed to kernel mode and a context switch is executed
b) The mode is changed to kernel mode and there is no context switch
c) The mode is changed to kernel mode and a context switch is performed only if a different process is waiting for the event to happen.
d) The mode is not changed and there is no context switch until a handler is loaded.

# Kernel and user mode

- Two execution modes:
  - Kernel mode (privileged)
    - Access to the whole memory space
    - Executes all CPU instructions
  - User mode (unprivileged)
    - Access only to the memory space of one process
    - Can not execute all instructions and cannot access all CPU registers
- When an event occurs:
  - The mode is changed to kernel mode
  - But there is no context switch:
    - The event is handled with in the context of the executing process
    - The memory map belongs to the currently executing process, even though it has no relationship with the event.

# Quiz

In what order does the event manager execute?

a) Save CPU state to stack, Run event handler routine, Return to previous state
b) Switch to Kernel mode, Run event handler routine, Return to User mode
c) Save CPU state to stack, Switch to Kernel mode, Run event handler routine, Return to previous state
d) Disable other interrupts, Run event handler routine, Enable other interrupts

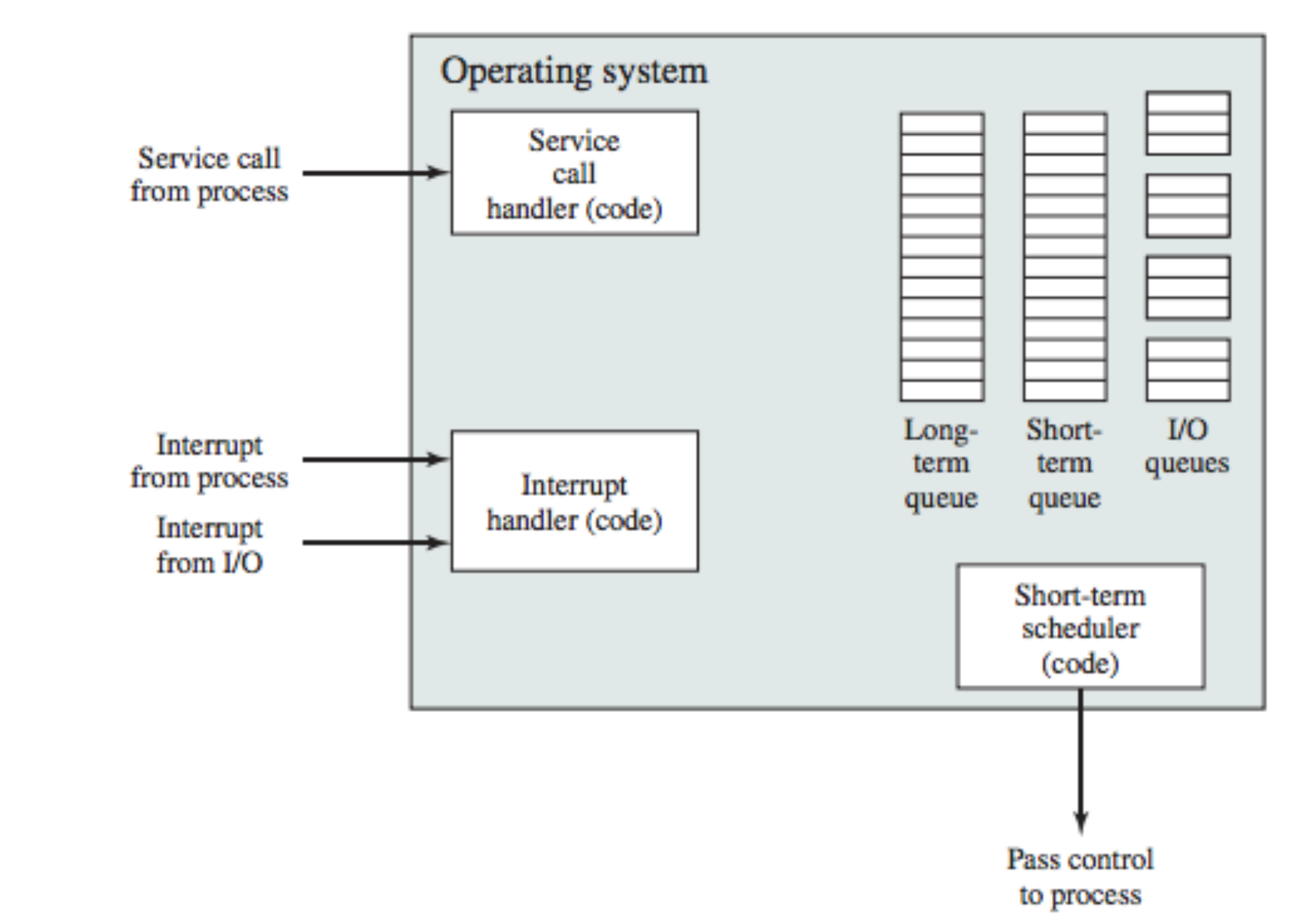
# Mode change

- When the event occurs:
  - Save partially the stack state
    - Typically PC and state registers (OS saves the remainder if necessary)
  - CPU switches to kernel mode and jumps to the associated event handler
    - Nested events can occur.
  - Event handler does its job and finishes with RETI (return from interrupt instruction)
    - Returns to previous mode
    - Goes to the interrupted instruction
    - Restores the stack state
- Details:
  - The system uses two stacks:
    - User stack
    - Kernel stack
  - No events when booting:
    - System mode, inhibited interrupts, inactive MMU

# Contents

- Operating system booting process
- Operating system execution
- **Operating system events**
    - Hardware interrupts
    - Exceptions
    - System calls
    - Software interrupts
- Kernel processes

# Key elements for OS events



# Quiz

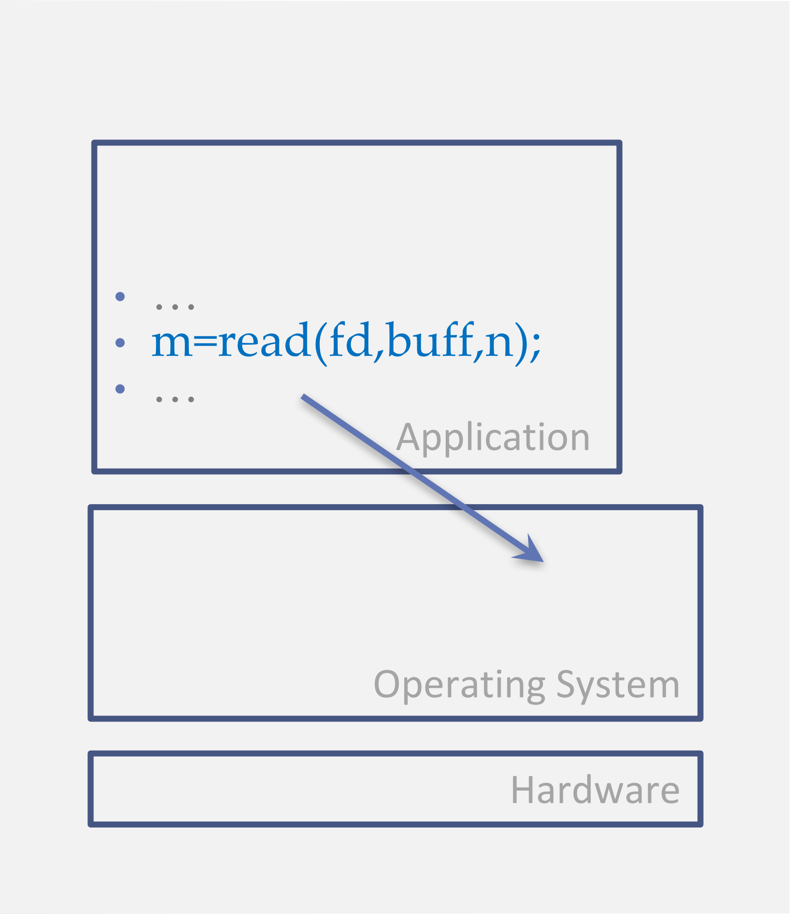
Which of the following is NOT an OS event?

a) HW/SW interrupt.
b) Signal.
c) System call.
d) Exception.

# Event types

- System calls
  - Request of OS service
- Exceptions
  - Exceptional events occurring when executing an instruction
  - Eg.: Arithmetic exception, Segmentation violation, Page Fault
- Hardware interrupts
  - Events generated by the hardware
  - Asynchronous with respect to execution
  - Eg.: When the disk finishes an operation
- Software interrupts
  - Interrupts caused by software, usually programs in user mode
  - Unlike hardware interrupts they are not handled immediately, only at certain points (eg. after a hardware interrupt or syscall)
  - Example: alarm

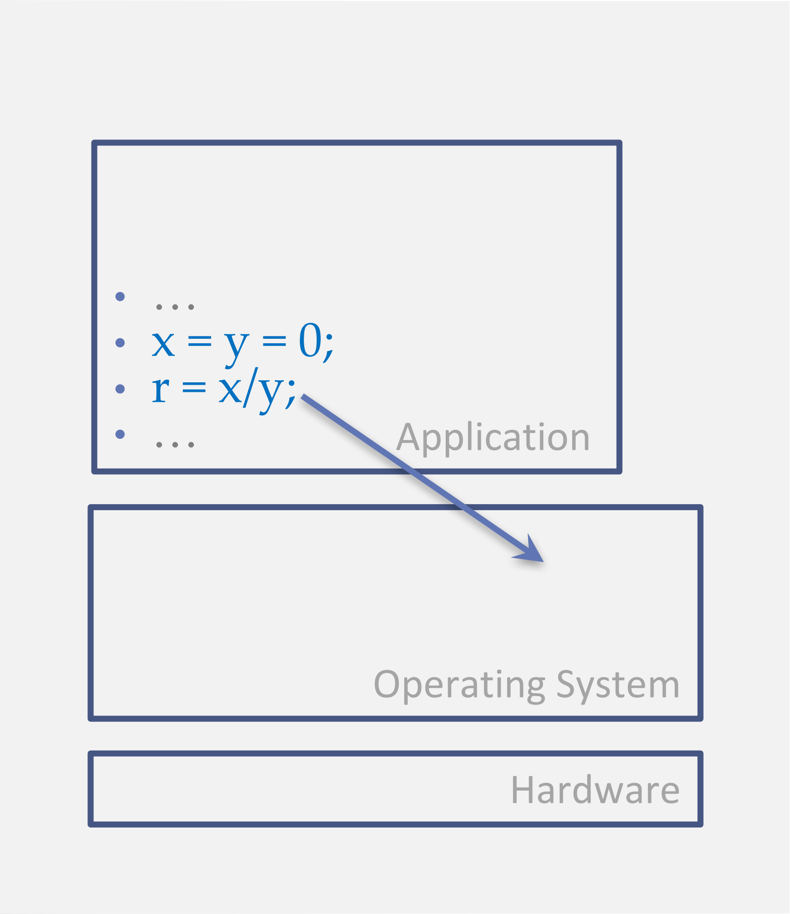
# System calls




- Operating system's service request event.

- The user applications access to the operating system services through system calls.
- The programmers see them as functions to be called within his/her applications.

# Exceptions

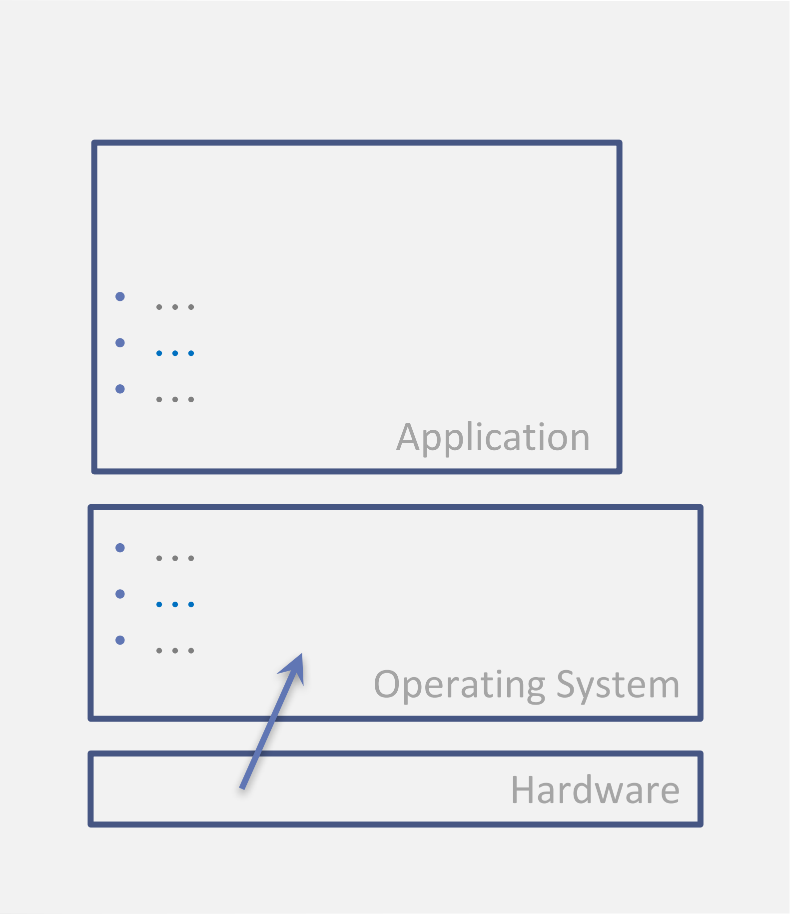



- Exceptional events that occur while executing an instruction.

- It could be a problem (zero division, illegal instruction, segmentation fault, etc.) or an advice (page fault, etc.)
  - ~ Hardware interrupts created by the CPU itself.

- It will be needed a subroutine associated to each exception that might happen.

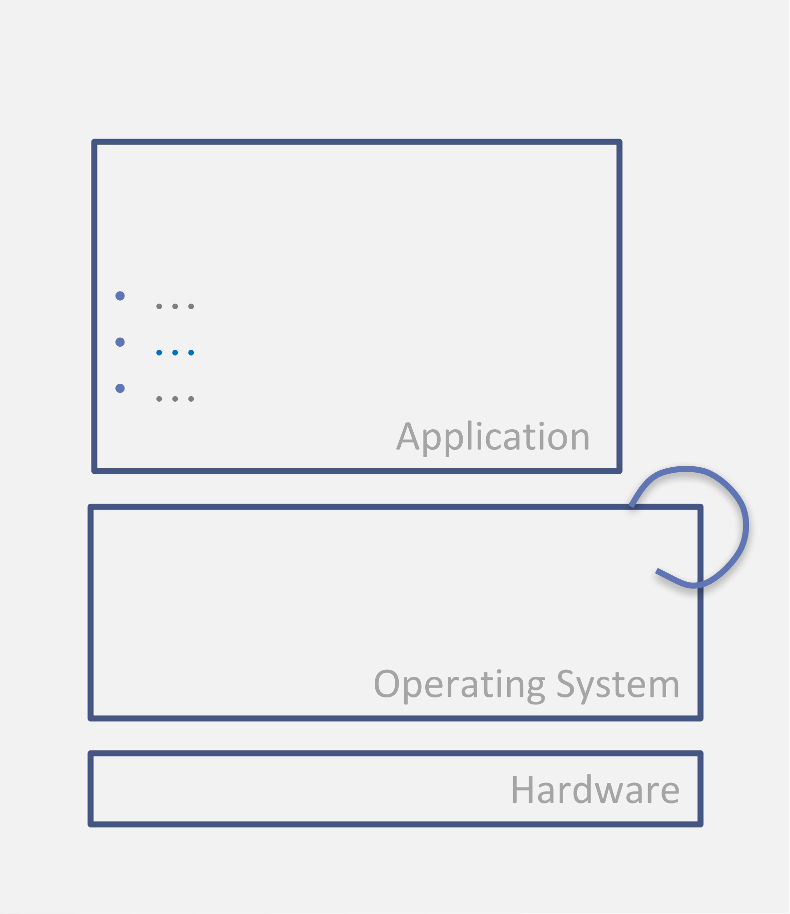
# Hardware interrupts




- Events from hardware.

- The operating system has to handle something that the hardware needs (incoming data, exceptional situation, etc.)
- It will need a subroutine associated to each hardware event that can occur.

# Software interrupts




- Delayed event from a pending part of other event.

- The operating system has to attend something that was postponed in other event.
  - ~ Fast-food behavior
- If nothing with more priority is available, all pending event handler (tasks) are executed.

# Event characterization

- Synchrony
  - Synchronous events
    - Predictable activation
    - Executed on behalf of the currently executing process
  - Asynchronous events
    - Unpredictable activation
    - May refer to any process
    - Executed in the context of an unrelated process
- Software or hardware-generated:
  - Hardware-generated
    - Received from a device
  - Software-generated
    - Generated by an assembly instruction

# Quiz

Which would be an example of an asynchronous hardware event and a synchronous software event?

a) Exceptions; Hardware Interrupts
b) System Call; Software Interrupts
c) Exceptions; Software Interrupts
d) Hardware Interrupts; System Call

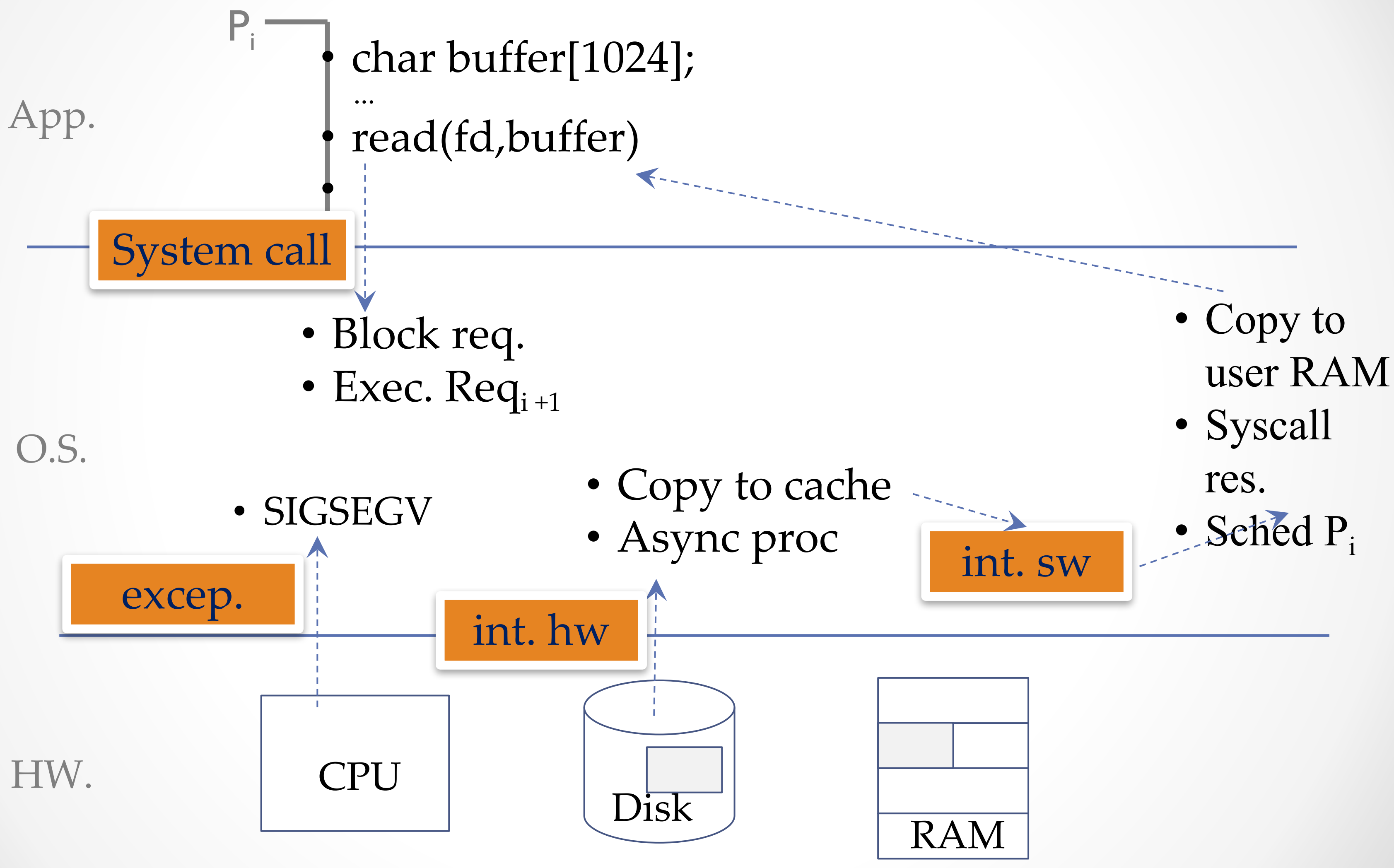
# Event classification

| | Synchronous | Asynchronous |
|---|---|---|
| Hardware | Exceptions | Hardware interrupts |
| Software | System call | Software interrupts |

# Origin and execution

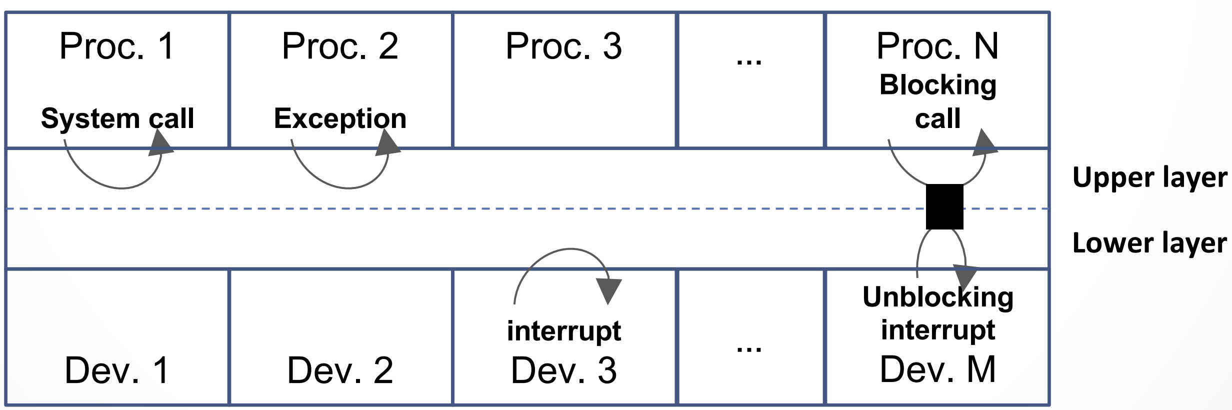
| | Exec mode | Origin |
|---|---|---|
| **Hardware int.** | • **User** or **system**<br>• **NO** different treatment | • I/O devices<br>• interrupt among CPUs (IPI) |
| **Exceptions** | • User or system<br>• **YES,** different treatment | • CPU (int. hw. of CPU)<br>• Usually program errors (div by 0, segment violation, ..)<br>• Not always (page fault, debugging, etc.) |
| **system calls** | • Always **user** | • Applications |
| **Software int** | • Always **system** | • Processing of any of the former events:<br>used for non-critical parts |

# Example: call as a library



# Relationship among events

- Components that handle synchronous events
  - More related with processes
- Components that handle asynchronous events
  - More related with devices
- Both types of events are related
  - e.g. disk access (read system call + disk interrupt)

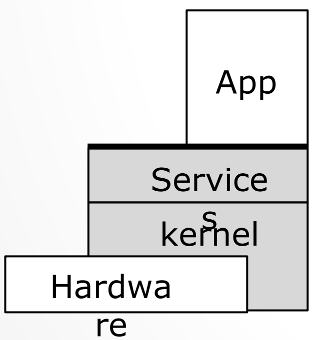


# Event management

- The O.S. use to be generic and independent of the hw. architecture
  - Linux without priority (SPARC support it) and Windows with priority (Intel doesn't support it)
- All events are treated in a similar way (like hardware interrupts)
  - We already know the major part in the event management
    - First save part of the CPU state on the system stack
      - Usually the PC and status registers (the OS will save the rest if it is necessary)
    - The CPU switch into kernel mode and jump to the associated event handler
      - Other event can be fired (and treated) while treating other one.
    - The event handler routine treats the event
    - The event handler routine ends:
      - The previous state is restored, switch back to the previous mode, continue execution with RETI

# Event management

- Type 1 > no events on system boot
  - System mode, disabled interrupts, inactive MMU
- Type 2 > when a event occurs, the operating system comes in to handle it:
  - It will change the mode (to kernel mode),
  - but NOT necessarily a context switch:



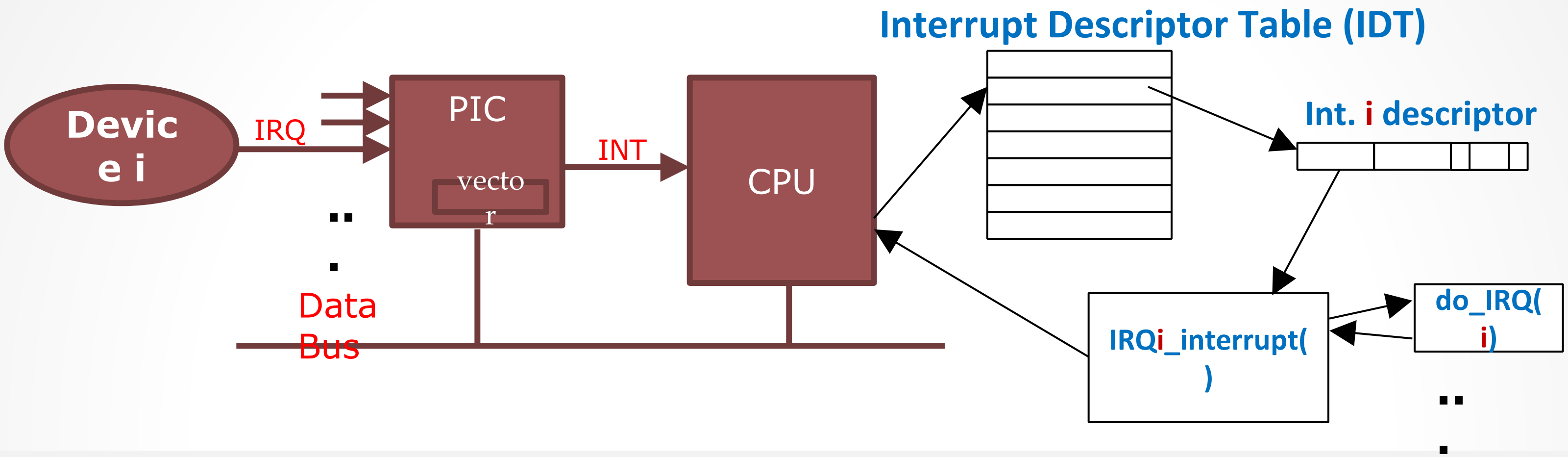

- The event is handled in the context of the active process
- The active memory map is the one associated with the current process, even though there is no relationship between it and the event.
- The system uses two independent stacks:
  - User stack: for the user mode
  - System stack: for the system mode

# Hardware interrupt

## characteristics

- Asynchronous events from hardware
- Previous execution mode:
  - It could be users or system (NOT will affect on how is handled)
- Fired by:
  - I/O devices
  - System critical conditions (e.g. power cut)
  - Intra-Processor interrupt (IPI)

# Hardware interrupts

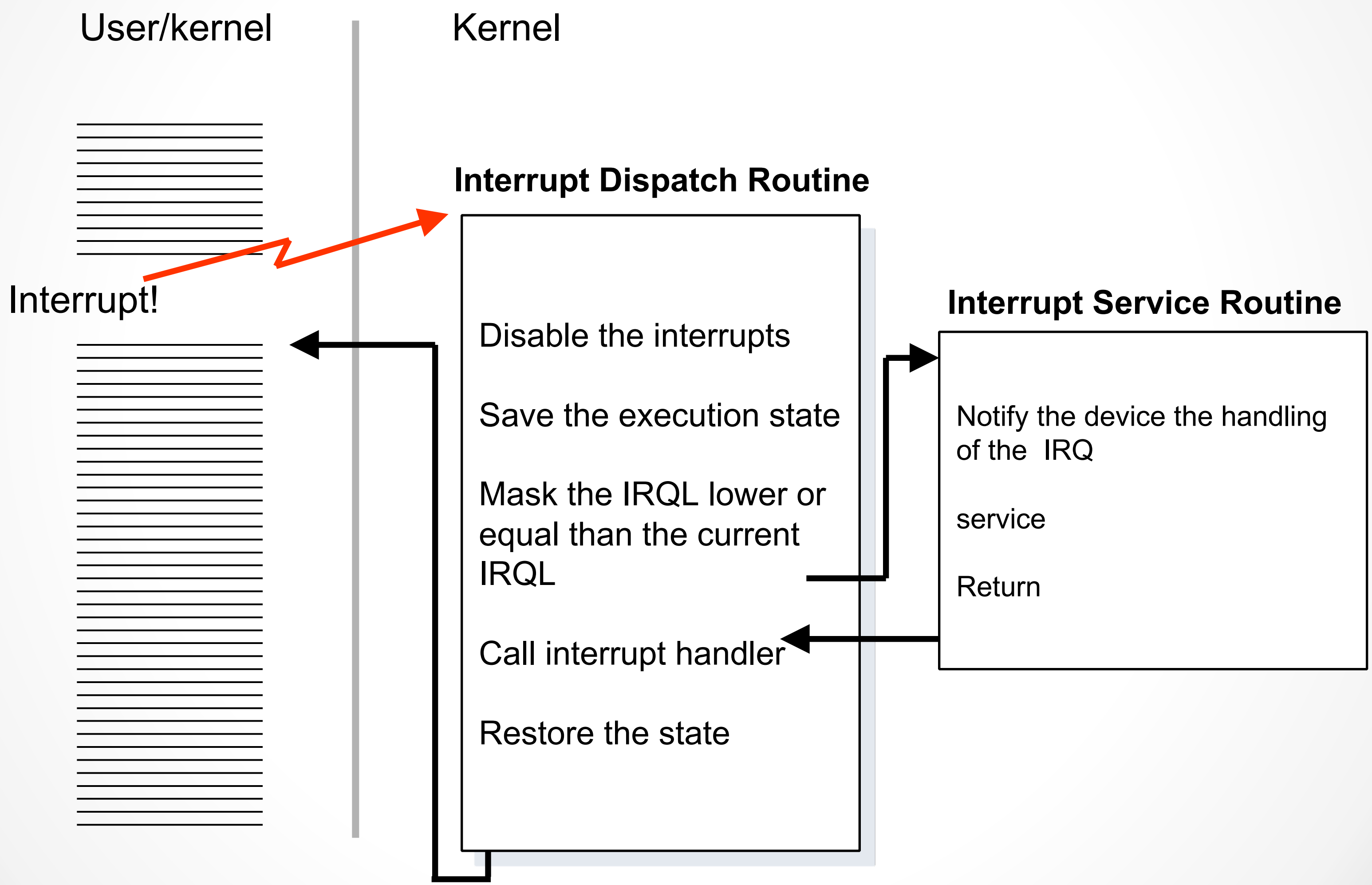



- Each interrupting device have a line for generating IRQ (*Interrupt ReQuest*)
  - Several devices on a line possible => need to query to find out the requester
- All lines are connected to a PIC (*Programmable Interrupt Controller*)
  - Currently APIC (*Advanced Programmable Interrupt Controller*)
- PIC connected to CPU pending interrupt line (INT)
  - PIC ignores interrupt if disabled. If not, send it to CPU by priority
  - Int: number and device



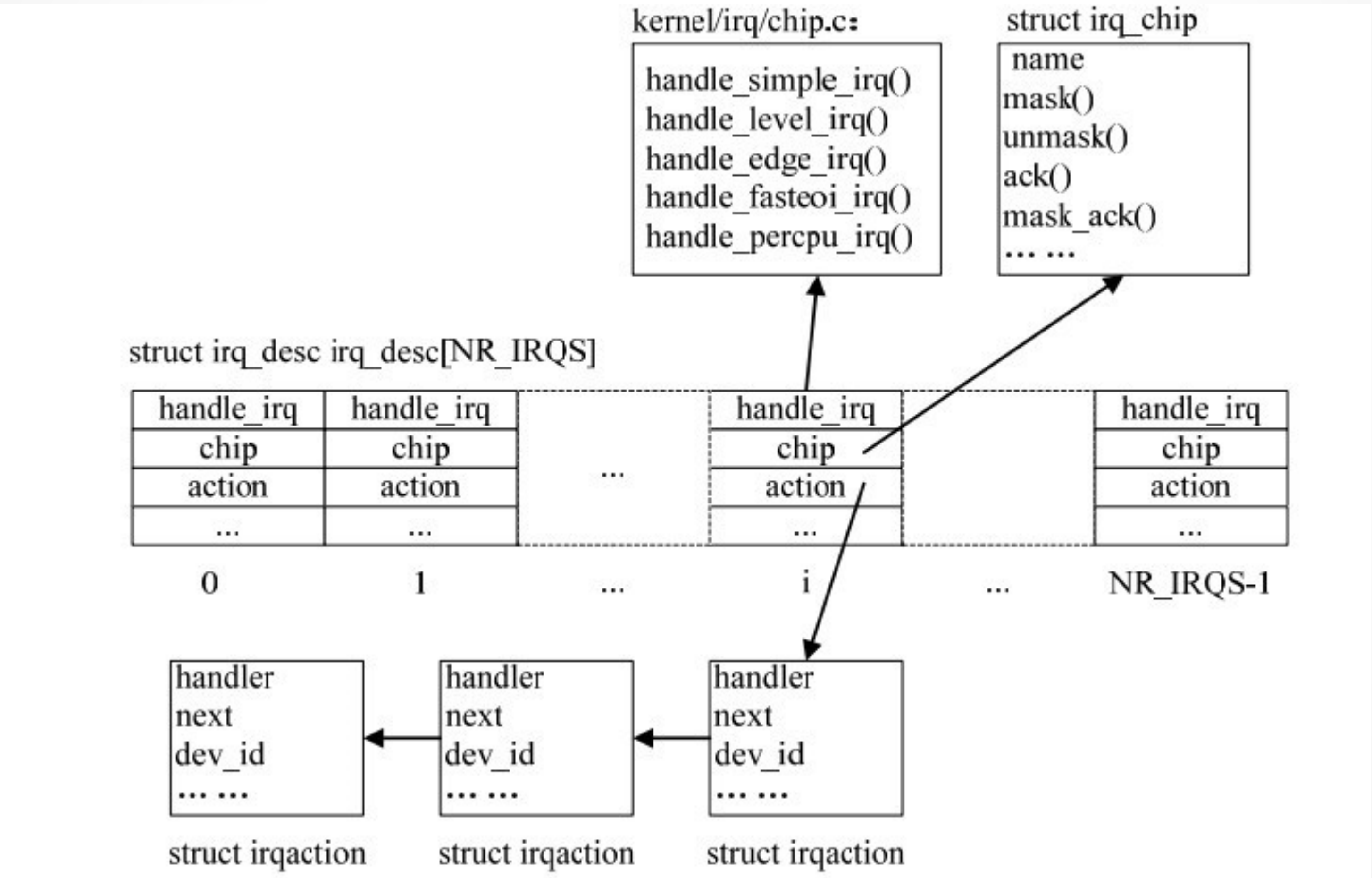
Understanding the Linux kernel (2nd edition)

# Interrupt management routine




Inside Windows 2000 (págie 104)

# Linux hw interrupt management




http://chxxxyg.blog.163.com/blog/static/1502811932010627015098/

# Exceptions

- Synchronous events generated as a result of the execution of an instruction
- Occur either in user or kernel mode :
  - Handled differently
- Source:
  - Programming errors (eg. arithmetic, segmentation violation)
  - Page faults: accessing a page not in memory (no error)
  - Debugging (no error)

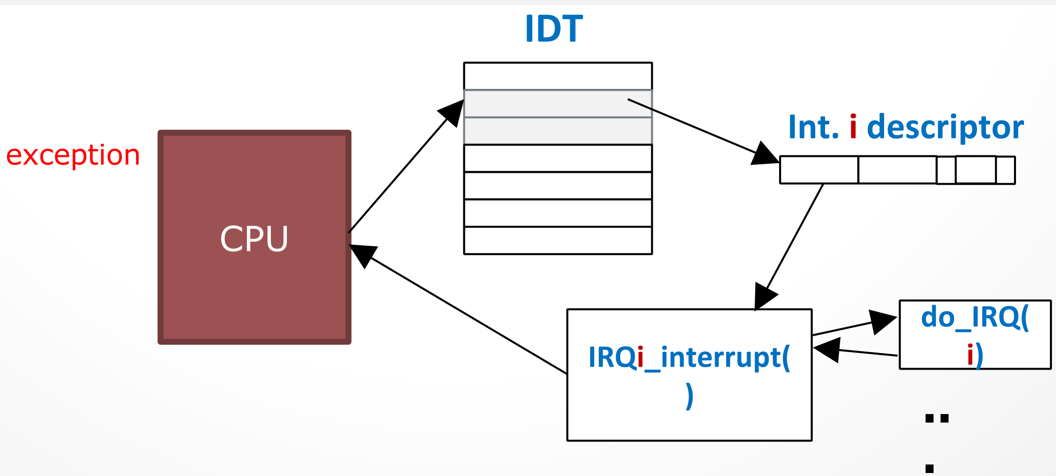
# Quiz

What is the purpose of an exception?

a) Catching any event in the kernel.
b) Executing deferred actions due to kernel activities or hardware interrupts.
c) Allowing the programmer to execute try and catch.
d) Capturing some errors raised by the CPU while processing instructions.

# Exception handling

- The same management mechanisms as hardware interrupts (but no IRQL)
- The result depends on the exception and on the executing mode
  - If error:
    - In kernel mode:
      - Panic: error in the OS code => message + stop the OS
    - In user mode:
      - If debugging, notify the debugger
      - If the program set up an exception handler, call it
      - Otherwise, abort the process
  - If no error: (Eg: page fault)
    - Execute the associated handler (Eg: assign a new page and update page data)
    - Both in user and kernel level



# Exception example

```c
#include "servicios.h"

int main () {
    double result= 3;
    int a = 0;

    result= result / a;

    printf(“result = %d\n“,result);
    return 0;
}
```

Application

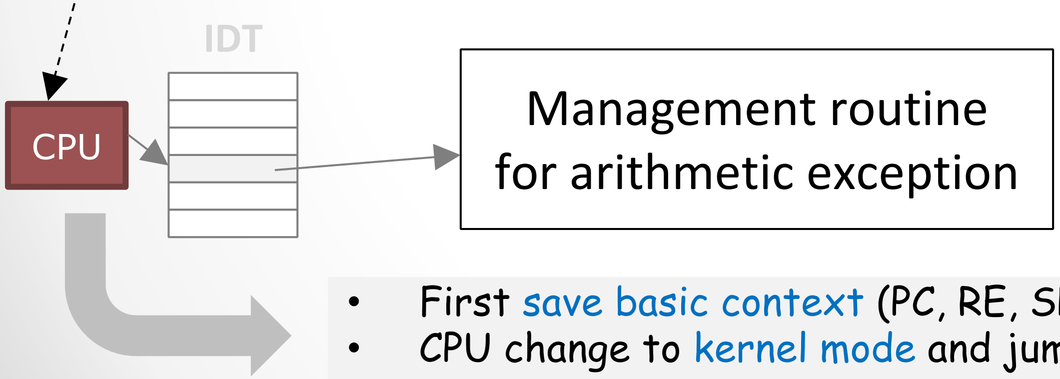
CPU

Aritm.
Exception

User mode

Div by zero

Kernel mode




- First save basic context (PC, RE, SP) to system stack
- CPU change to kernel mode and jumps to exception management routine

# Quiz

If a page fault occurs in kernel mode:

a) The associated handler is executed.
b) The kernel informs the debugger.
c) The kernel aborts the process producing it.
d) It stops the OS.

# System calls

- Synchronous events requesting OS service through a non-privileged instruction
  - Executed only in user mode
- Issued by processes

# Quiz

What is FALSE about a system call?

- It is implemented as a trap.
- Its interface is offered as a system library routine.
- It causes a change in the CPU execution to kernel mode.
- It always causes a context change.

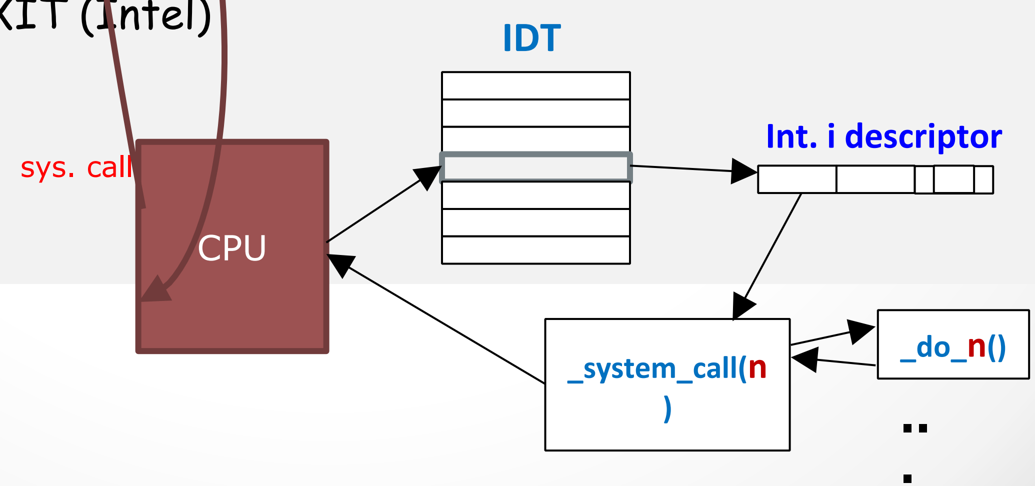
# Quiz

How are system calls notified to the kernel?

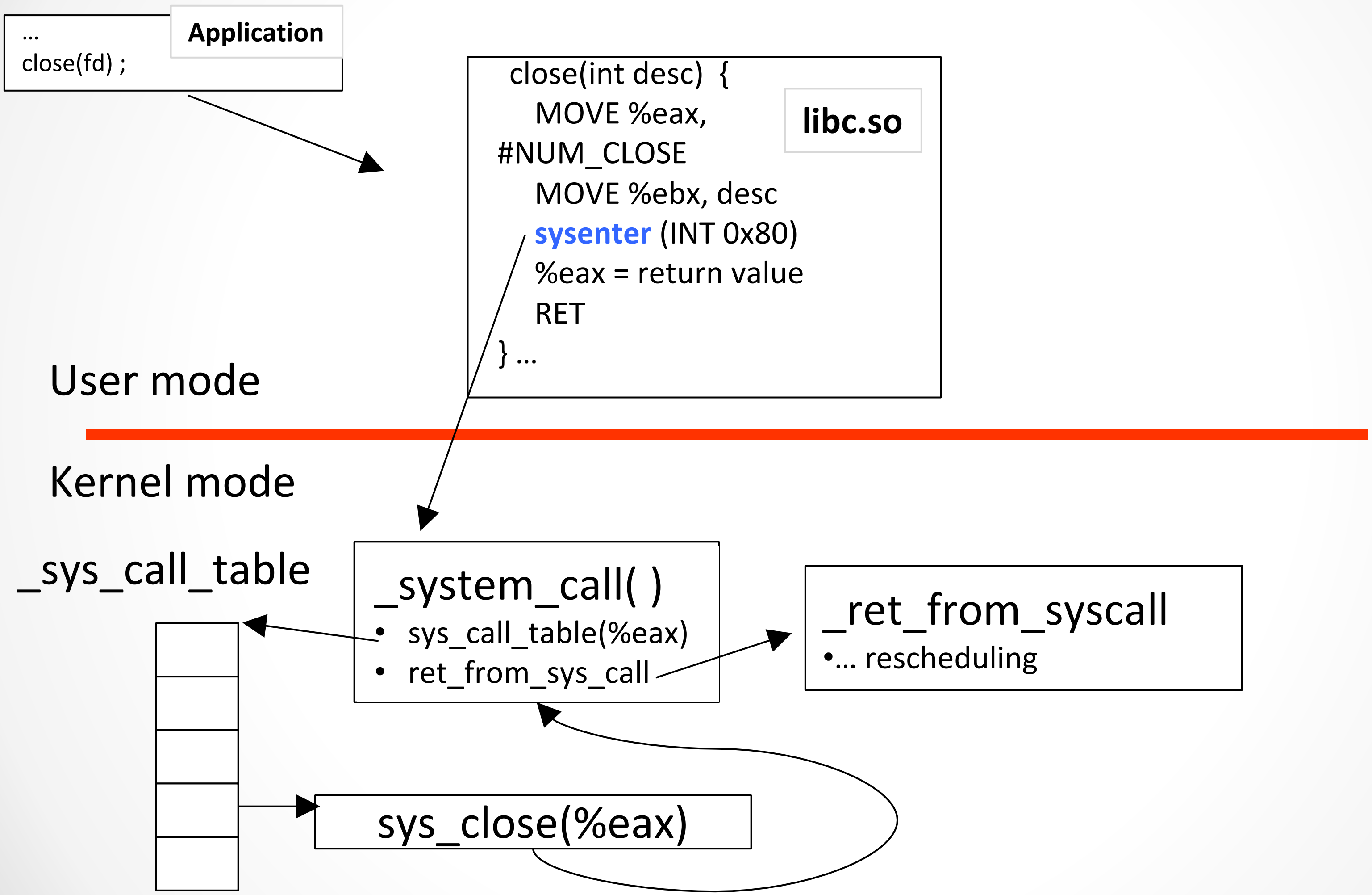
a) By sending a message with the number of the syscall.
b) By activating a software interrupt indicating the number of syscall.
c) By generating an exception in the CPU to activate the kernel.
d) The PC is set to the kernel space address and execution is continued.

# System call handling

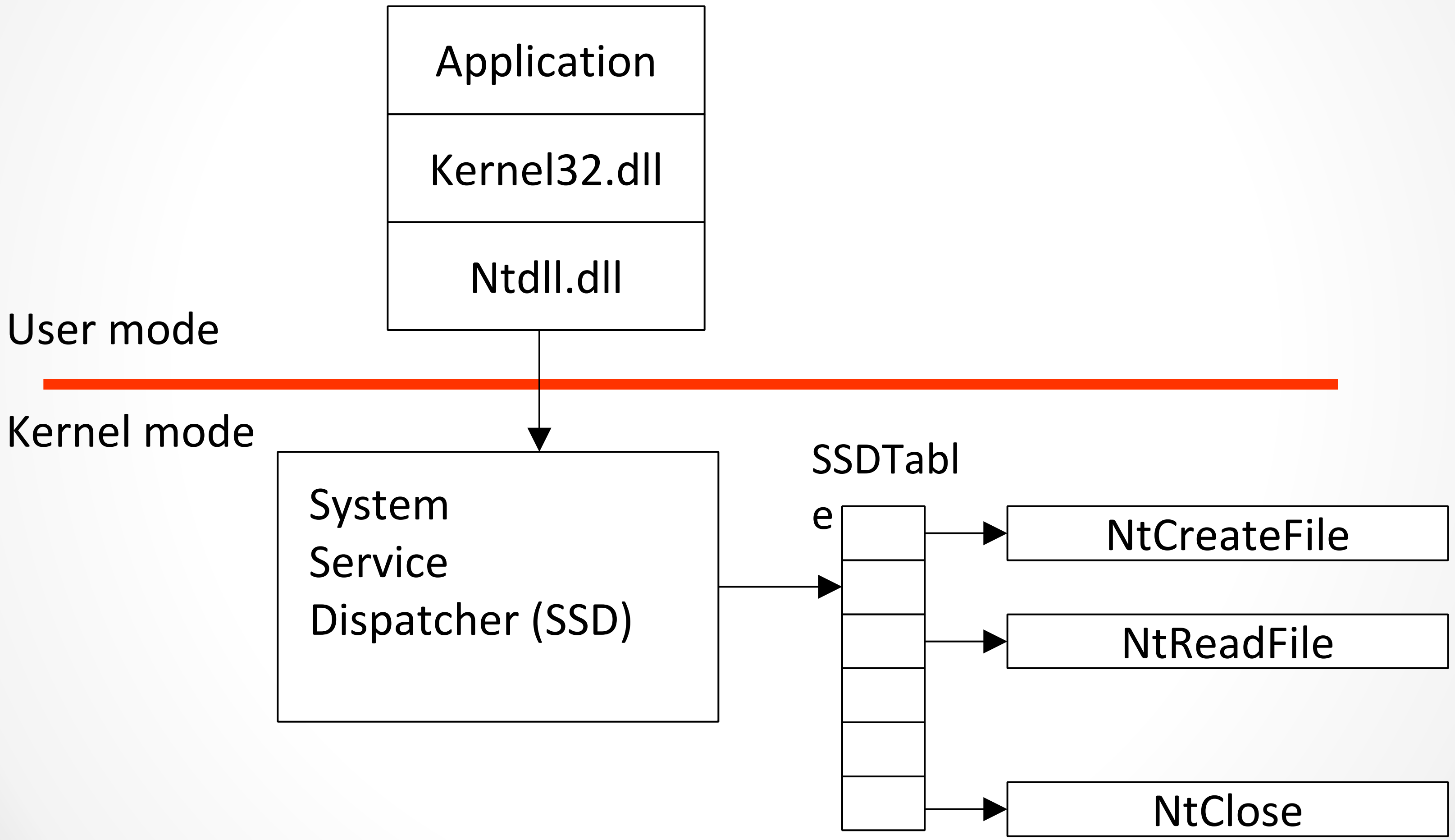
- The syscall code is assigned to a special CPU register
- Activate a software interrupt generated by an assembly instruction
  - Save state (PC, etc.) on the kernel stack
  - Pick up the parameters from registers (Linux) or stack (Windows)
  - Identify the syscall routine in a table
  - Execute the syscall routine
  - The result is passed in special CPU register
  - Restored the state (PC, etc.) and return from RETI
- Newer Alternatives : fast transfer control to the OS and back for a system call without the overhead of an interrupt
  - SYSCALL/SYSRET (AMD)
  - SYSENTER/SYSEXIT (Intel)



# System calls in Linux



# System calls in Windows



# Quiz

Which of the following statements is true?

a) System calls occur at hardware level and are asynchronous events.
b) System calls occur at software level and are asynchronous events.
c) System calls occur at software level and are synchronous events.
d) System calls occur at hardware level and are synchronous events.

# Software interrupts

- Asynchronous events for handling a non-critical part associated to an event
  - Give priority to urgent events
  - Handled later

- Often used to implement system calls because they implement a subroutine call with a CPU ring level change.

- Fired by:
  - In the management of former events, a soft. int. is prepared for the non-critical part of code

# Software interrupts

## deferrable functions in Linux

- Bottom-Halves (BH): removed in kernel version 2.6.x
  - The first implementation of the software int. in Linux
  - Are executed one after another, in series (no mather the CPU number) There are 32 handlers only (previously registered)
- Softirqs: introduced in 2.3
  - Statically defined deferrable functions: 32 handlers only
  - Same kind of softirq can be executed in parallel on different CPUs.
  - The system timer uses softirqs.
- Tasklets: introduced in 2.3
  - On top of softirqs allow for dynamic creation of deferrable functions
  - All tasklets are executed through one softirq
  - One tasklet is executed on one CPU, several tasklets on different CPUs at the same time.
- Work queues: introduced in 2.5
  - Generalization of Tasklet model
  - Tasklets execute atomically until completion, while work queues permit handlers to sleep

http://www.ibm.com/developerworks/linux/library/l-tasklets/index.html

# Software interrupts

## types of managements in Windows

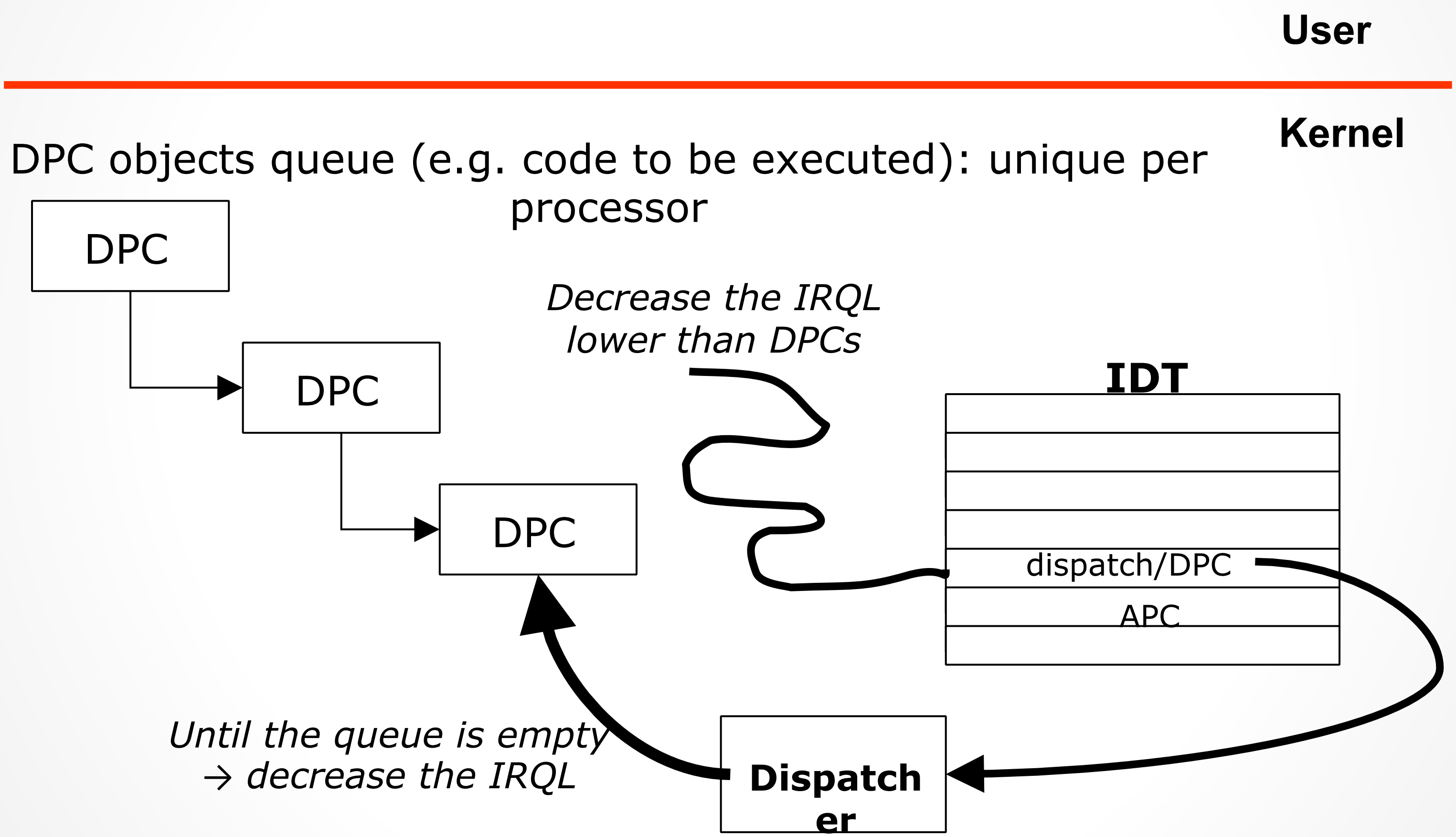
- Deferral Procedure Calls (DPCs):
  - One queue per CPU
  - When IRQL drops, the kernel checks if there are any available DPC and executes them
  - It executes delayed tasks that were previously programmed:
    - To complete I/O operations from drivers.
    - To handle timer expiration.
    - To free waiting threads.
    - To force the scheduling when the time slice ends.

- Asynchronous Procedure Calls (APCs):
  - Function that executes asynchronously in the context of a particular thread (each thread has its own queue)
  - When queued to a thread, the system issues a software interrupt
  - It can be executed from system mode or user mode.

https://en.wikipedia.org/wiki/Deferred_Procedure_Call
https://msdn.microsoft.com/en-us/library/windows/desktop/ms681951(v=vs.85).aspx

# Software interrupts

## types of management in Windows: DPC



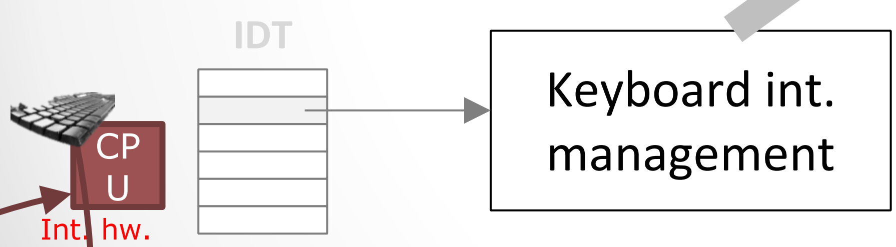
# Example of deferred processing

Just save key code
in kbd buffer

```
void Int_hardware_KBD ( idDevice )
{
    • idDevice -> HardwareID
    • Key = ReadPort(HardwareID)
    • Insert(Key, idDevice, Buffer)
    • insertPendingTask (&listPendTasks,
                         Int_software_Kbd);
    • activate_int_SW();
}
```

User mode

Kernel mode




- Filter key with kbd map
- If special key (CTRL, ESC…)
  - Compose order
- IF CR or buffer full
  - Copy to user space

# Event handler installation

```
int main (int argc, char **argv)
{
        ...

        /* install handlers for exception and interrupt management */
        instal_man_int(EXC_ARITHMETIC,   ArithmeticExceptionRout);
        instal_man_int(EXC_MEMORY,       MemoryExceptionRout);
        instal_man_int(EXC_HW,           HardwareExceptionRout);
        instal_man_int(EXC_IO,           IOExceptionRout);

        ...

        instal_man_int(INT_CLOCK,        ClockinterruptRout);
        instal_man_int(INT_DEVICE_i,     Device_i_interruptRout);
        instal_man_int(SYS_CALL_j,       SysCall_j_Routine);
        instal_man_int(INT_SW_k,         SoftInt_j_Routine);

        ...
```

# Linux event handler setup

/usr/src/linux/arch/x86/kernel/traps.c

```
void __init trap_init(void)
{
        …
        set_intr_gate(X86_TRAP_DE, divide_error);
        set_intr_gate(X86_TRAP_NP, segment_not_present);
        set_intr_gate(X86_TRAP_GP, general_protection);
        set_intr_gate(X86_TRAP_SPURIOUS, spurious_interrupt_bug);
        set_intr_gate(X86_TRAP_MF, coprocessor_error);
        set_intr_gate(X86_TRAP_AC, alignment_check);

#ifdef CONFIG_IA32_EMULATION
        set_system_intr_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
        set_bit(IA32_SYSCALL_VECTOR, used_vectors);
#endif

#ifdef CONFIG_X86_32
        set_system_trap_gate(SYSCALL_VECTOR, &system_call);
        set_bit(SYSCALL_VECTOR, used_vectors);
#endif
        …
```

# Nested events

- Hw.Int./Exception while handling:
  - Hw. Int. / Exception => allow all, none, or only the one with higher priority
  - Syscall / Sw. Int. => interrupt always (disabled in crytical sections)
- Sw. Int/syscall while handling:
  - Hw.Int. / Exception => uninterruptable
  - Syscall / Sw.Int.
    - Non-preemptive Kernel
      - *Can not be interrupted, they are enqueued*
      - *Many UNIX y Linux*
    - Preemptive Kernel.
      - *Interruptible, but have to protect the critical sections (eg.: increase interrupt level)*
      - *Solaris, Windows 2000, etc.*

# Contents

- Operating system booting process
- Operating system execution
- Operating system events
    - Hardware interrupts
    - Exceptions
    - System calls
    - Software interrupts
- **Kernel processes**

# Process types

- User processes
  - No sysadmin (root) privileges
  - Run in privileged mode only:
    - Handle a syscall (fork, exit, etc.)
    - Handle an exception (0/0, *(p=null), etc.)
    - Handle an interrupt
- System processes
  - Sysadmin (root) privileges
  - Run in privileged mode as user processes
- Kernel processes
  - Belong to the kernel (not to a user)
  - Run always in privileged mode
  - Usually high priority  (negative in Linux)

# Quiz

What are system processes?

- All the processes that the Operating System execute.
- Any process executed in kernel mode.
- Any processes initiated by users that run in privileged mode.
- Processes executed by users that only run when there is a syscall, exception or interrupt.

# OS intervention

- Booting the system
- Event management
  - Hardware interrupts
  - Exceptions
  - System calls
  - Software interrupts
- Kernel process
  - OS tasks more appropriate to be done in the context of an independent process
  - It shares the CPU with the rest of processes
    - Typically have higher priority

# Criteria to assign an action to an execution context

- If it is related with an synchronous event (exception or system call)
  - Include the action in the event routine
- If it is related with an asynchronous event:
  - If the action is critical →
    - Include it within the interrupt routine
  - If the actions is not critical
    - If it does not require locks →
      - Include it within the software interrupt
    - If locks are needed (e.g. page faults) →
      - Executed inside a kernel process

# OS and multiprocessors

- **UP**: Uni-Processing.
  - OS and apps run on a single CPU.
  - Simple, but very low performance.
- **ASMP**: *Asymmetric MultiProcessing*.
  - OS running always in the same CPU.
  - Simple, but lower performance.
- **SMP**: Symmetric MultiProcessing.
  - OS running on any CPU.
  - Complex: Synchronization mechanisms needed to protect critical sections.
    E.g.: Raising the int. level does not prevent to execute in another CPU.

# Basic mechanisms in Linux

| Technique | Scope | Prototypes |
|---|---|---|
| **Disable interrupts** | • Single CPU | `unsigned long flags;`<br>`local_irq_save(flags);`<br>`/* ... CS: critical section ... */`<br>`local_irq_restore(flags);` |
| **Spin Locks** | • Any CPU<br>• Busy waiting:<br>  • Can NOT sleep, schedule, etc. in CS | `#include <linux/spinlock.h>`<br>`spinlock_t l1 = SPIN_LOCK_UNLOCKED;`<br>`spin_lock(&l1);`<br>`/* ... CS: critical section ... */`<br>`spin_unlock(&l1);` |
| **Mutex** | • Any CPU<br>• Blocking wait:<br>  • Do NOT use in HW int. | `#include <linux/mutex.h>`<br>`static DEFINE_MUTEX(m1);`<br>`mutex_lock(&m1);`<br>`/* ... CS: critical section ... */`<br>`mutex_unlock(&m1);` |
| **Atomic operations** | • Any CPU | `atomic_t a1 = ATOMIC_INIT(0);`<br>`atomic_inc(&a1);`<br>`printk(“%d\n”, atomic_read(&a1));` |

**Robert Love. Linux Kernel Development. 3rd edition.**

# Complex mechanism examples in Linux

| Technique | Scope | Prototypes |
|---|---|---|
| **RW locks** | • Shared R, exclusive W<br>• Any CPU<br>• Busy waiting:<br>  • Can NOT sleep, schedule, etc. in CS | `rwlock_t x1 = RW_LOCK_UNLOCKED;`<br>`read_lock(&x1);`<br>`/* ... CS:critical section ... */`<br>`read_unlock(&x1);`<br>`write_lock(&x1);`<br>`/* ... CS: critical section ... */`<br>`write_unlock(&x1);` |
| **Spin Locks + irq** | • Any CPU<br>• Busy waiting not interruptible:<br>  • Can NOT sleep, schedule, etc. in CS | `spinlock_t l1 = SPIN_LOCK_UNLOCKED;`<br>`unsigned long flags;`<br>`spin_lock_irqsave(&l1, flags);`<br>`/* ... CS: critical section ... */`<br>`spin_unlock_irqrestore(&l1, flags);` |
| **RW locks + irq** | • Any CPU<br>• Busy waiting not interruptible:<br>  • Can NOT sleep, schedule, etc. in CS | `read_lock_irqsave();`<br>`read_lock_irqrestore();`<br><br>`write_lock_irqsave();`<br>`write_lock_irqrestore();` |

**Robert Love. Linux Kernel Development. 3rd edition.**

ARCOS Group

Operating Systems Design
Grado en Ingeniería Informática
Universidad Carlos III de Madrid