Operating Systems & C

Lecture 7: Processes

Willard Rafnsson
IT University of Copenhagen

ITU CPH

Recap

recall: “operating system manages processes & hardware”

…how? we haven’t *really* seen the OS **do** *anything* yet.

kernel is a collection of **code** and **data structures** (residing in RAM) that are used to manage processes and hardware. but a kernel is not a process/thread. so,

how does the OS do anything? what makes this code execute?

interrupts.

ITU CPH

Topics

- Interrupts
  - X86 Interrupt System (w/ example `write` syscall in xv6)
  - Interrupts (async), Exceptions (sync; Traps, Faults, Aborts)
- Booting
- Processes
  - API (`fork`, `exit`, `wait`, Inter-Process Communication)
  - Kernel State, Context Switching
  - Scheduling

ITU CPH

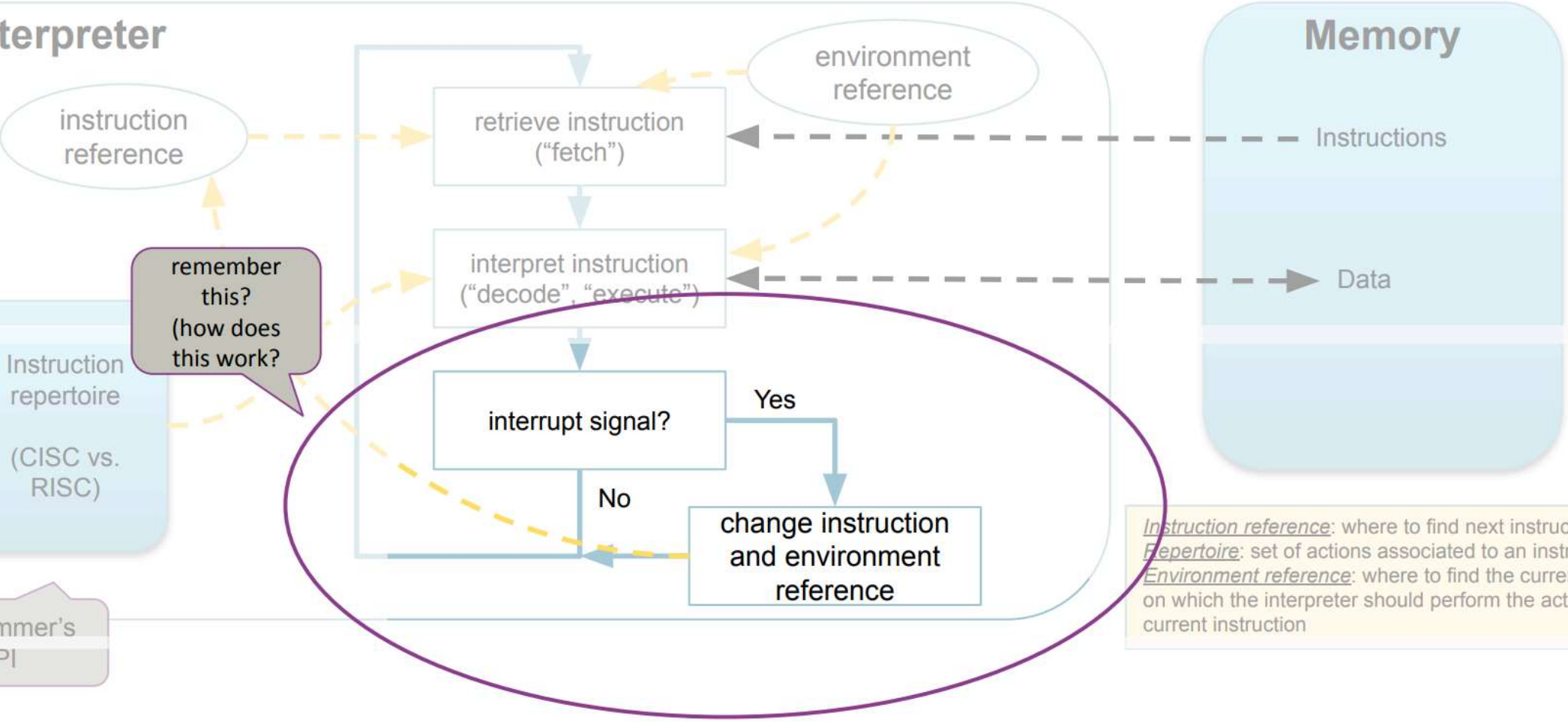
Interrupts

What makes the OS spring into action.

ITU CPH

erpreter abstraction

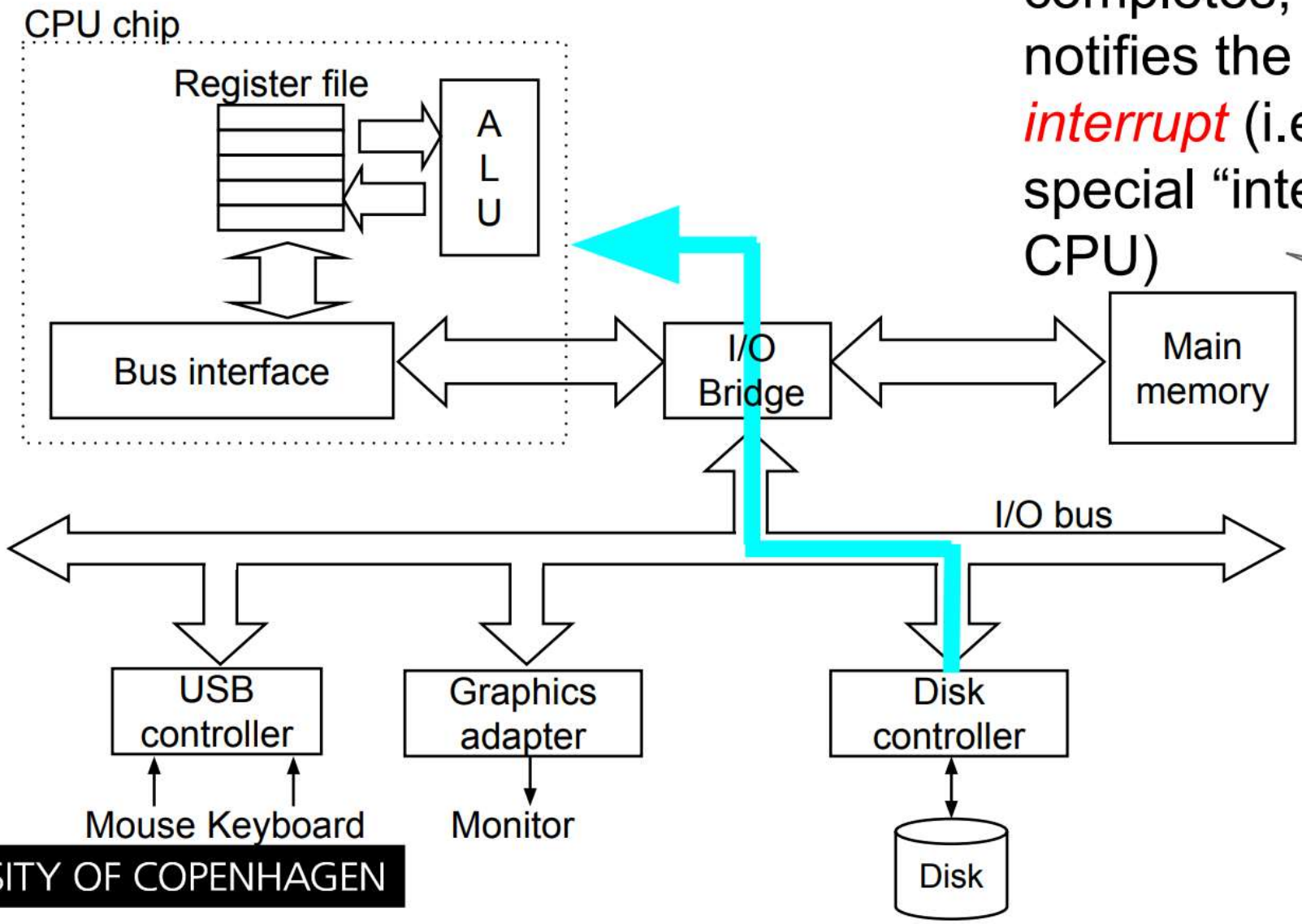
Source: Saltzer and Kaa



ading a Disk Sector (3)

When the DMA transfer completes, the disk contro notifies the CPU with an *interrupt* (i.e., asserts a special “interrupt” pin on t CPU)

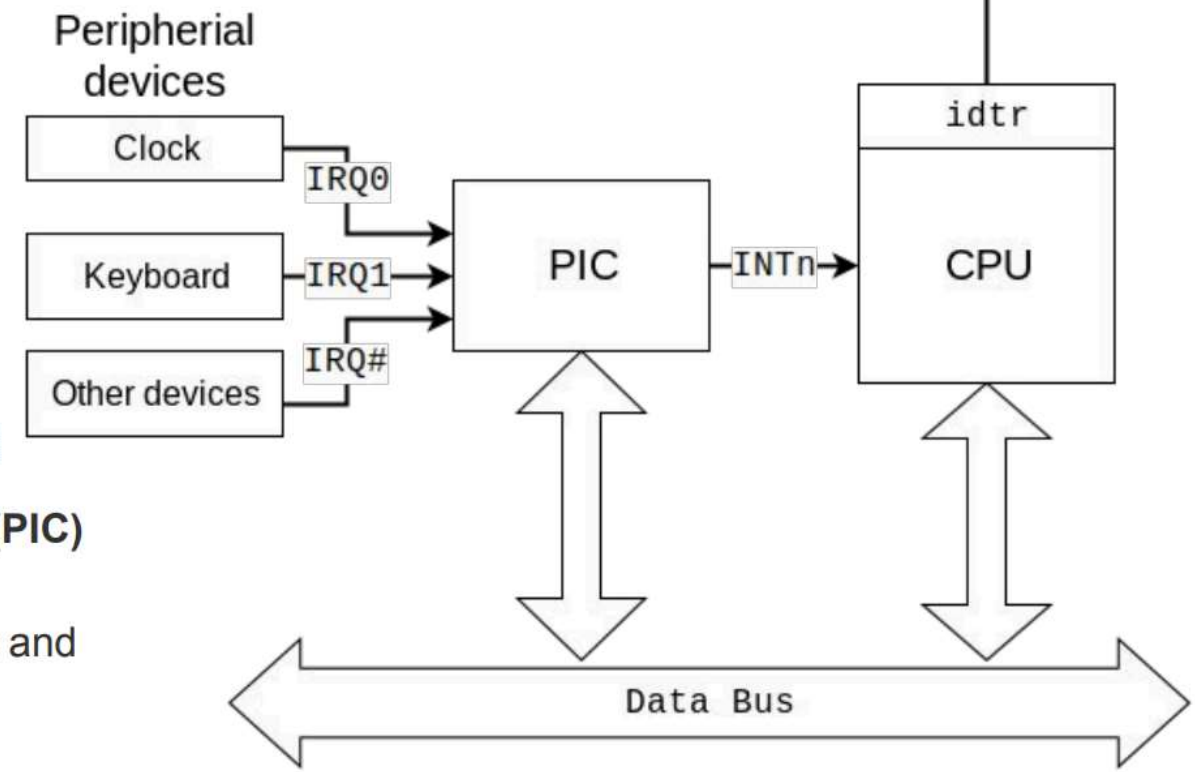
we’ll learn abo interrupts in fu lecture



VERSITY OF COPENHAGEN

ITU CPH

86 interrupt system

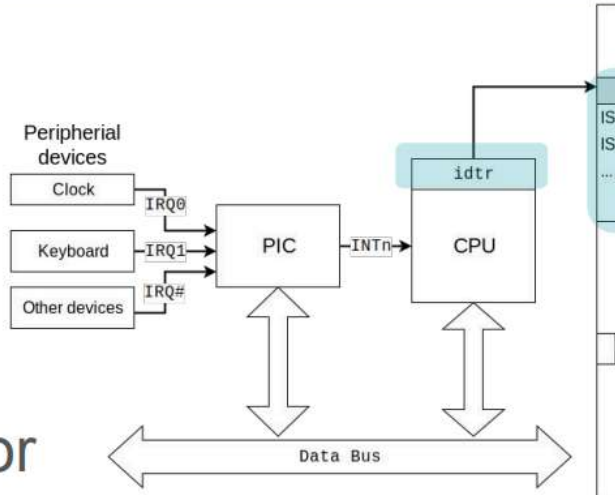


es 3 parts to work conjointly:

ogrammable Interrupt Controller (PIC)
st be configured to receive
errupt requests (IRQs) from devices and
nd them to CPU.
U must be configured to
eive IRQs from PIC and
oke correct interrupt handler, via "gate" described in an
errupt Descriptor Table (IDT).
errupt Service Routines (ISRs) } focus
st be provided by OS kernel to handle interrupts and be ready to be preempted by an interrupt.
lso must configure both PIC and CPU to enable interrupts. (← happens on boot!)

ITU CPH

terrupt descriptor table (IDT)



ble in memory (pointed by `idtr`).

ed by OS (`lidt` instruction.). holds "gate descriptors" for errupt service routines (ISRs), aka. *interrupt handlers*.

ate descriptors":

address to interrupt handler (ISRs)

lags, protection levels

ecution semantics:

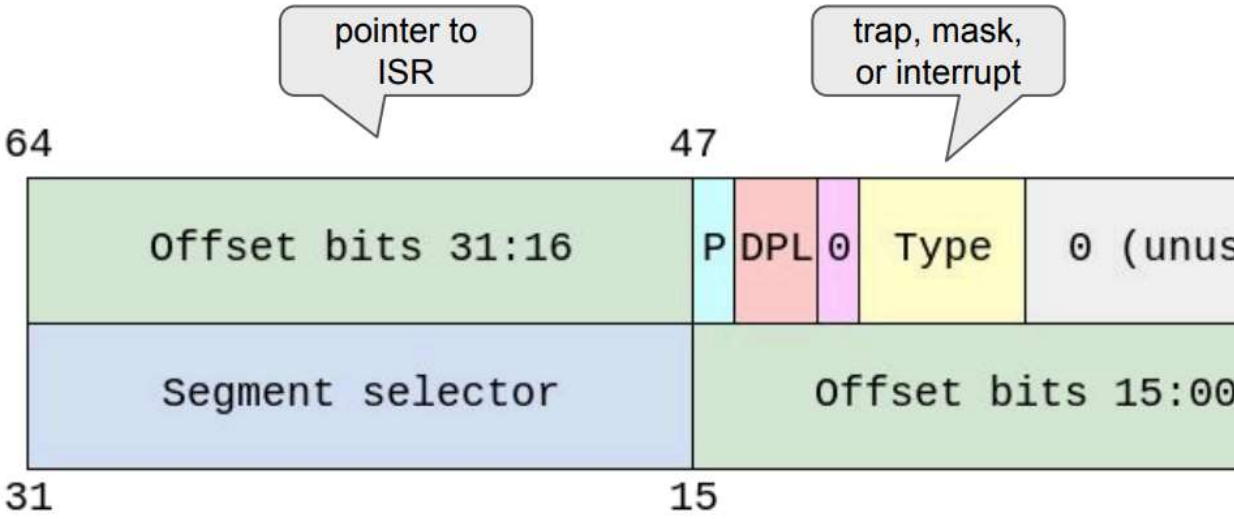
CPU **automatically**

nvokes the handler

egistered for the interrupt.

e.g. in Fig, for IRQ1, it's ISR1)

assuming interrupts have been enabled (`sti`)

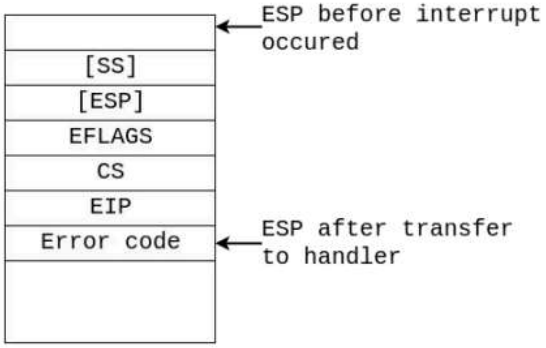


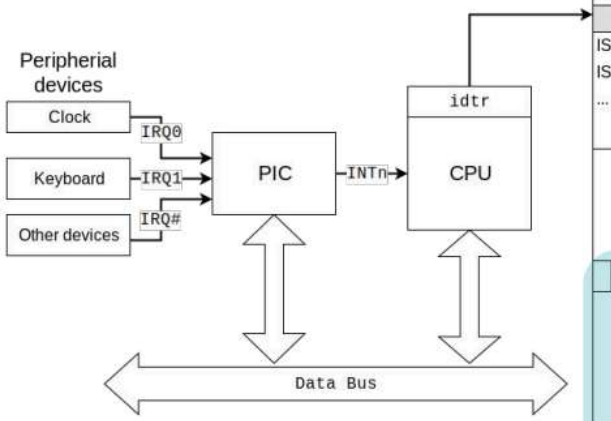
- P - Segment present
- DPL - Descriptor privilege level

ITU CPH

eiving an interrupt

terrupt service routines (ISRs)





-space:

Temporarily **saves** (internally) the current contents of the SS, ESP, EFLAGS, CS and EIP **registers**.
Loads the segment selector and the stack pointer for the **new stack** (that is, the stack for the privilege level being called) from the TSS into the SS and ESP registers and switches to the new stack.
Pushes the **temporarily saved registers** SS, ESP, EFLAGS, CS, and EIP values for the interrupted procedure's stack **onto the new stack**.
Pushes an error code on the new stack (if appropriate).
Loads the segment selector for the new code segment and the **new instruction pointer** (**from** the interrupt gate or trap **gate**) into the CS and EIP registers, respectively.
If the call is through an interrupt gate, clears the IF flag in the EFLAGS register.
Begins execution of the handler procedure at the new privilege level.

kernel-space: (main diff: doesn't switch stacks)

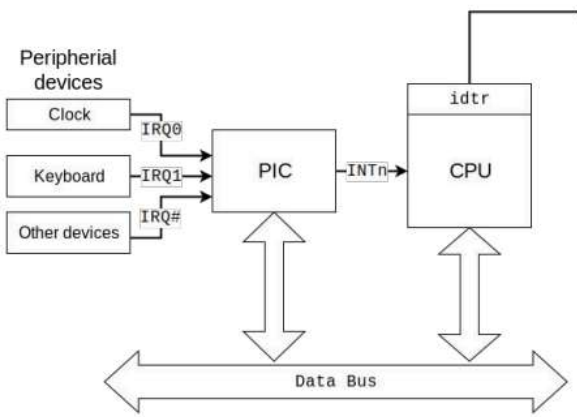
1. **Push** the **current** contents of the EFLAGS, CS, an EIP **registers** (in that order) **on the stack**.
2. Push an error code (if appropriate) on the stack.
3. **Load** the segment selector for the new code segme and the **new instruction pointer** (from the interrup gate or trap gate) into the CS and EIP registers, respectively.
4. Clear the IF flag in the EFLAGS, if the call is throug an interrupt gate.
5. **Begin execution of the handler procedure**

what should the handler do? (next slide)

ITU CPH

ndling an interrupt

terrupt service routines (ISRs)



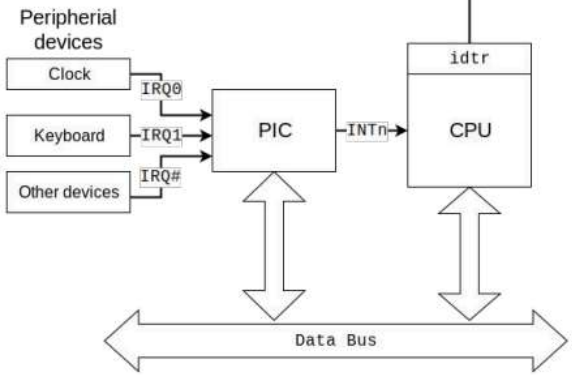
ere is the basic ISR algorithm:

- Save the state of interrupted procedure (`pusha`)
- Save previous data segment
- Reload data segment registers with kernel data descriptors
- Acknowledge interrupt by sending EOI to PIC
- [Do the work]
- Restore data segment
- Restore the state of interrupted procedure (`popa`)
- Enable interrupts (`sti`)
- Exit interrupt handler (with `iret`)

ITU CPH

xample: keyboard press

keyboard's IRQ pre-set to 9 by BIOS (is reconfigurable)



Setup interrupts:

a. Create IDT table
b. Set IDT entry #9 with interrupt gate pointing to keyboard ISR
c. Load IDT address with `lidt`
d. Send interrupt mask 0xfd (11111101) to PIC1 to unmask (enable) IRQ1
e. Enable interrupts with `sti`

Human hits keyboard button

Keyboard controller raises interrupt line IRQ1 in PIC1

PIC checks if this line is not masked (it's not) and send interrupt number 9 to CPU

CPU checks if interrupts disabled by checking IF in EFLAGS (it's not)

(Assume that currently we're executing in kernel mode)

Push EFLAGS, CS, and EIP on the stack

Push an error code from PIC (if appropriate) on the stack

Look into IDT pointed by `idtr` and fetch segment selector from IDT descriptor 9.

Check privilege levels and load segment selector and ISR address into the CS:EIP

Clear IF flag because IDT entries are interrupt gates

Pass control to ISR

the handler

1. Receive interrupt in ISR:
   a. Disable interrupt with cli (just in case)
   b. Save interrupted procedure state with pusha
   c. Push current DS value on the stack
   d. Reload DS, ES, FS, GS from kernel data segme
2. Acknowledge interrupt by sending EOI (0x20) to master (I/O port 0x20)
3. Read keyboard status from keyboard controller (I/O por 0x64)
4. If status is 1 then read keycode from keyboard controlle (I/O port 0x60)
5. Finally, print char via VGA buffer or send it to TTY
6. Return from interrupt:
   a. Pop from stack and restore DS
   b. Restore interrupted procedure state with popa
   c. Enable interrupts with `sti`
   d. `iret`

ITU CPH

write syscall in xv6: user mode

syscall.h

```
...
#define SYS_write   16
...
```

main.c

```
...
write(1,
      "Hello, World!\n",
      14);
...
```

usys.S

(after macro replacement)

```
#include "syscall.h"
// ...
.globl write
write:
    /* 16 = SYS_write */
    movl $16, %eax
    /* 0x40 = T_SYSCALL */
    int $0x40
    ret
```

write syscall in xv6: user mode

syscall.h

```
...
#define SYS_write   16
...
```

main.c

```
...
write(1,
      "Hello, World!\n",
      14);
...
```

usys.S

(after macro replacement)

```
#include "syscall.h"
// ...
.globl write
write:
    /* 16 = SYS_write */
    movl $16, %eax
    /* 0x40 = T_SYSCALL */
    int $0x40
    ret
```

interrupt — trigger an exception similar to a keypress
parameter (0x40 in this case) — type of exception

write syscall in xv6: user mode

syscall.h

```
...
#define SYS_write   16
...
```

main.c

```
...
write(1,
      "Hello, World!\n",
      14);
...
```

usys.S

(after macro replacement)

```
#include "syscall.h"
// ...
.globl write
write:
    /* 16 = SYS_write */
    movl $16, %eax
    /* 0x40 = T_SYSCALL */
    int $0x40
    ret
```

xv6 syscall calling convention:
eax = syscall number
otherwise: same as 32-bit x86 calling convention (arguments on stack)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

`lidt` —
function (in x86.h) wrapping lidt instruction

sets the *interrupt descriptor table*
table of *handler functions* for each interrupt type

50

ITU CPH

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

(from mmu.h):

```
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d)                \
```

50

ITU CPH

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

set the T_SYSCALL (= 0x40) interrupt to be callable from user mode via **int** instruction

(otherwise: triggers fault like privileged instruction)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

set it to use the kernel "code segment"
meaning: run in kernel mode
(yes, code segments specifies more than that — nothing we care about)

50

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

1: do not disable interrupts during syscalls
e.g. keypress handling can interrupt slow syscall
con: makes writing system calls safely more complicated
pro: slow system calls don't stop timers, keypresses, etc. from working

xv6 choice: interrupts *are* disabled during non-syscall exception handling
(e.g. don't worry about keypress being handled while timer being handled

50

ITU CPH

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

`vectors[T_SYSCALL]` — OS function for processor to run
set to pointer to assembly function `vector64`

50

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

vectors[T_SYSCALL] — OS function for processor to run
set to pointer to assembly function vector64

hardware jumps here

vectors.S

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
...
```

trapasm.S

```
alltraps:
  ...
  call trap
  ...
  iret
```

trap.c

```
void
trap(struct trapframe *tf)
{
...
```

50

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

struct trapframe — set by assembly
interrupt type, application registers, ...
example: `tf->eax` = old value of eax

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

myproc() — pseudo-global variable
represents currently running process

much more on this later in semester

ITU CPH

write syscall in xv6: the trap function

trap.c

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  ...
}
```

syscall() — actual implementations uses `myproc()->tf` to determine what operation to do for program

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]  sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

52

ITU CPH

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]   sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

array of functions — one for syscall

'[number] value': syscalls[number] = value

ITU CPH

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]   sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

(if system call number in range)
call sys_...function from table
store result in user's eax register

write syscall in xv6: the syscall function

syscall.c

```
static int (*syscalls[])(void) = {
...
[SYS_write]  sys_write,
...
};

...

void
syscall(void)
{
...
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
...
```

result assigned to eax
(assembly code this returns to
copies `tf->eax` into %eax)

ITU CPH

write syscall in xv6: sys_write

sysfile.c

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

write syscall in xv6: sys_write

sysfile.c

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

utility functions that read arguments from user's stack
returns -1 on error (e.g. stack pointer invalid)
(more on this later)
(note: 32-bit x86 calling convention puts all args on stack)

write syscall in xv6: sys_write

sysfile.c

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return filewrite(f, p, n);
}
```

actual internal function that implements writing to a file (the terminal counts as a file)

write syscall in xv6: interrupt table setup

trap.c (run on boot)

```
...
lidt(idt, sizeof(idt));
...
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
...
```

`trap` returns to `alltraps`

alltraps restores registers from `tf`, then returns to user-mode

hardware jumps here

vectors.S

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
...
```

trapasm.S

```
alltraps:
  ...
  call trap
  ...
  iret
```

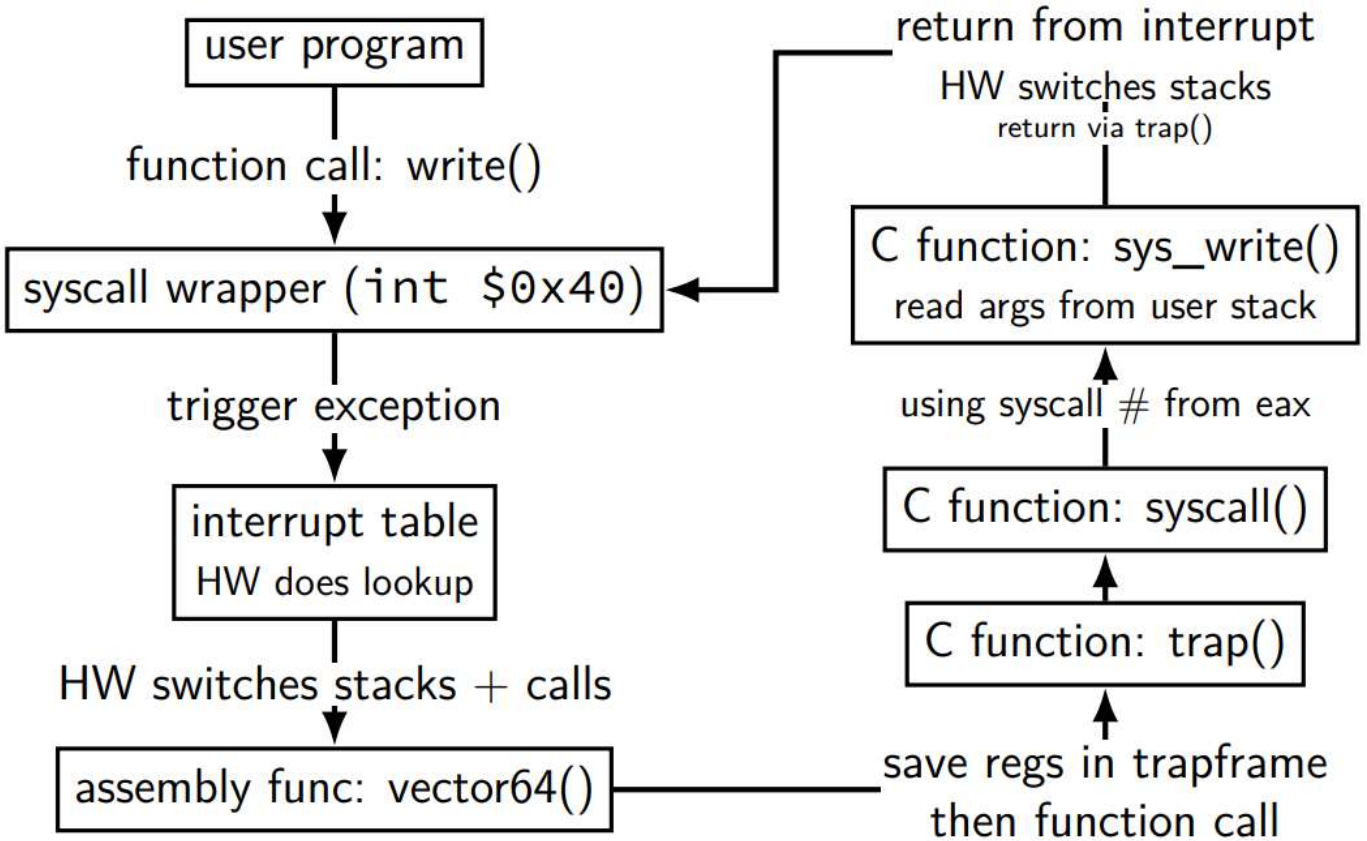
trap.c

```
void
trap(struct trapframe *tf)
{
...
```

54

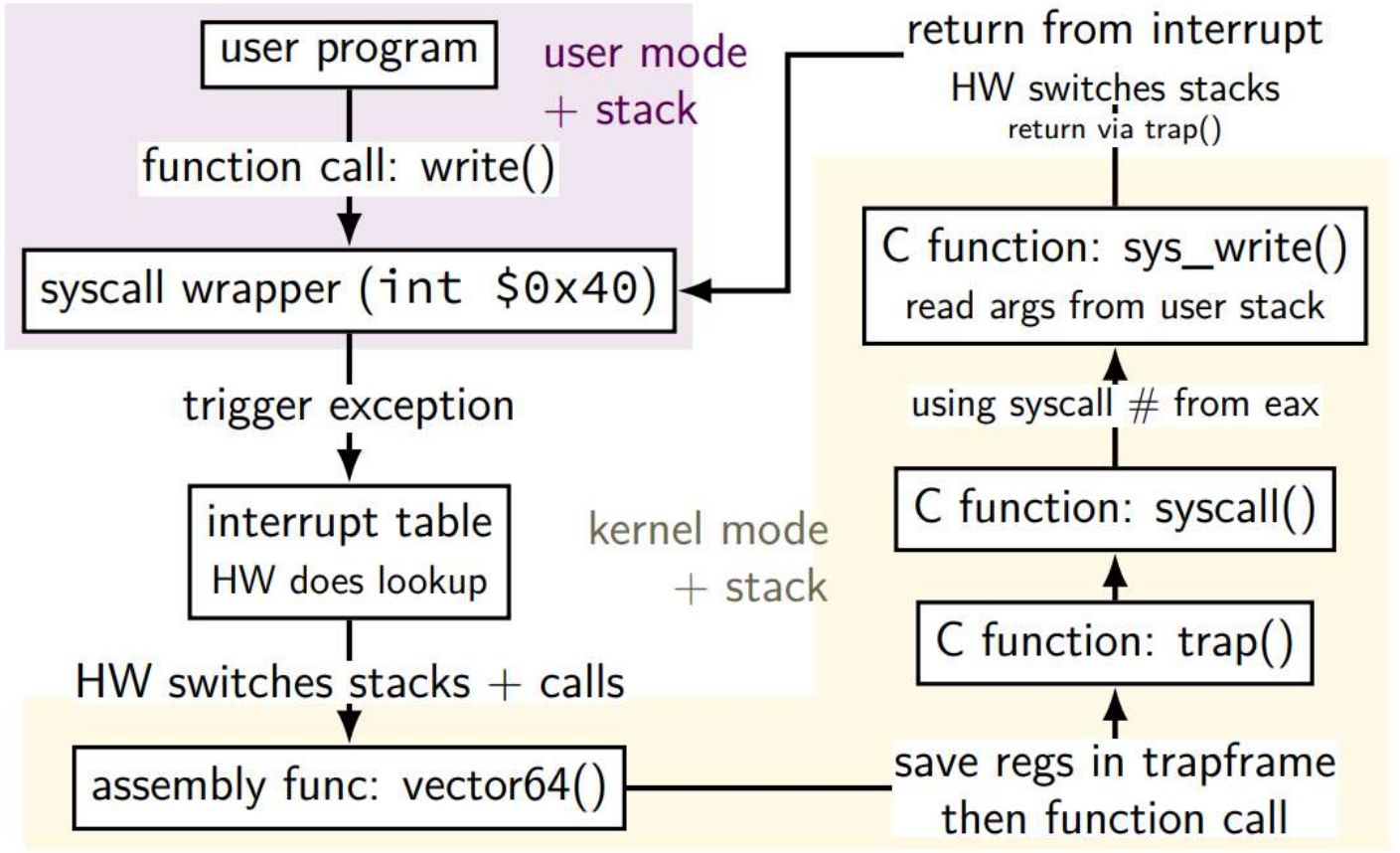
ITU CPH

write syscall in xv6



ITU CPH

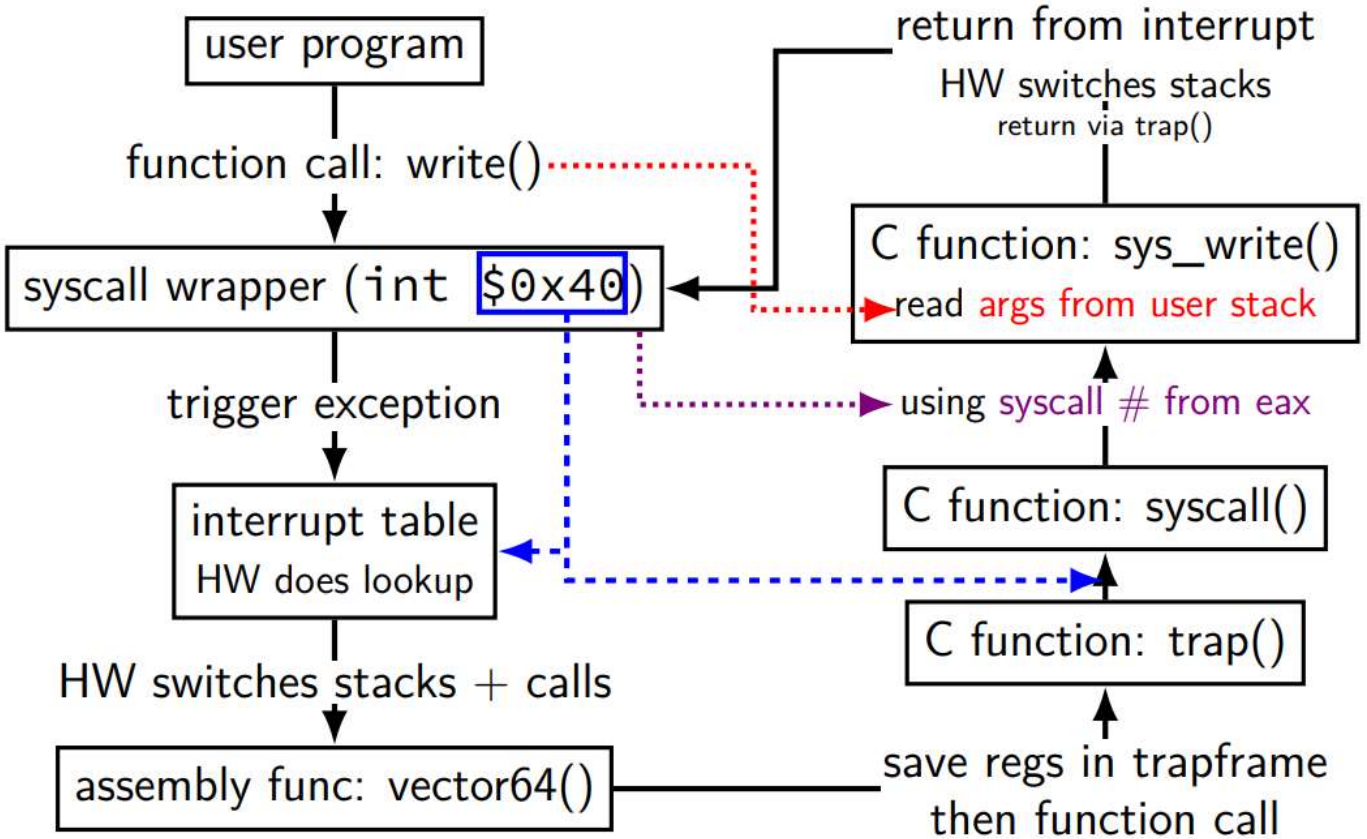
write syscall in xv6



57

ITU CPH

write syscall in xv6



58

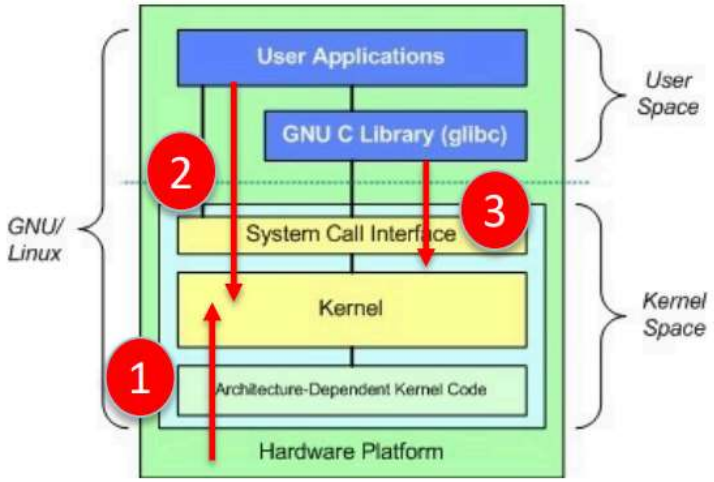
ITU CPH

Exceptional Control Flow

Interruption event:

1. A device needs attention
2. The user program did something illegal
3. The user program asks the OS kernel for a service through a system call

In these cases, **the flow of control is transferred from the user program to the OS kernel.**



Asynchronous Events

Managed in hardware

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - **Handler returns to "next" instruction**
- Examples:
  - I/O interrupts
    - hitting Ctrl-C at the keyboard
    - arrival of a packet from a network
    - arrival of data from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting Ctrl-Alt-Delete on a PC

- **Advanced Programmable Interrupt Controller (APIC)** is a more modern interrupt controller than the earlier 82C59 (see below). It supports multiprocessor/multicore interrupt management by allowing interrupts to be directed to a specific processor. The I/O APIC in the PCH can support up to 24 interrupt vectors and works in conjunction with I/O APICs in other devices to help eliminate the need for share interrupts among multiple devices.

PCH: Platform Controller Hub

https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introductio

Asynchronous Events

Hardware hands it out to software

When an interruption event occurs, hardware saves the minimum processor state required to enable software to resolve the event and continue. The state saved by hardware is held in a set of interruption resources, and together with the interruption vector gives software enough information to either resolve the cause of the interruption, or surface the event to a higher level of the operating system. Software has complete control over the structure of the information communicated, and the conventions between the low-level handlers and the high-level code. Such a scheme allows software rather than hardware to dictate how to best optimize performance for each of the interruptions in its environment. The same basic mechanisms are used in all interruptions to support efficient IA-64 low-level fault handlers for events such as a TLB fault, speculation fault, or a key miss fault.

http://refspecs.linux-foundation.org/IA64-softdevman-vol2.pdf

Synchronous Events

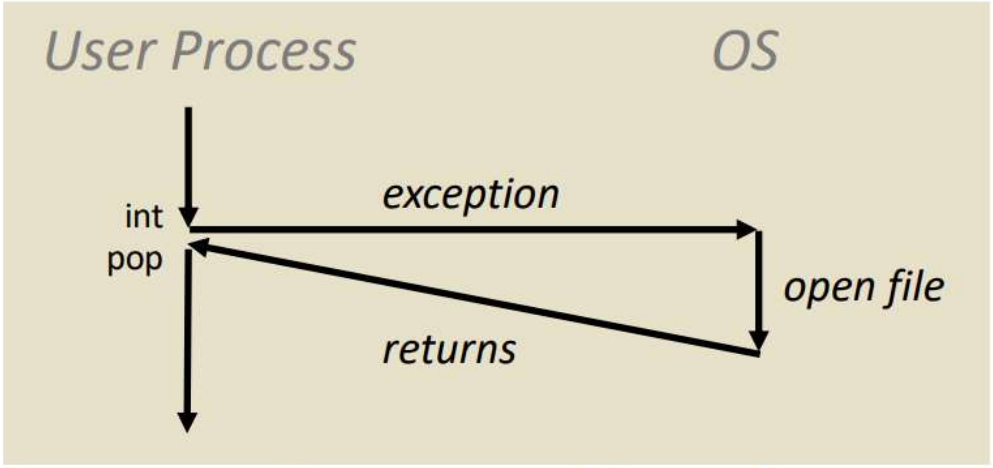
- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - **Returns control to "next" instruction**
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - **Either re-executes faulting ("current") instruction or aborts**
  - ***Aborts***
    - unintentional and unrecoverable
    - Examples: parity error, machine check
    - **Aborts current program**

Trap Example: System call

User calls: `open(filename, options)`

Function `open` executes **system call instruction `int`**

```
0804d070 <__libc_open>:
  . . .
 804d082:     cd 80                    int    $0x80
 804d084:     5b                       pop    %ebx
  . . .
```



OS must find or create file, get it ready for reading or writing

Returns integer file descriptor

syscall_64.tbl

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
6       common  lstat                   sys_newlstat
7       common  poll                    sys_poll
8       common  lseek                   sys_lseek
9       common  mmap                    sys_mmap
10      common  mprotect                sys_mprotect
11      common  munmap                  sys_munmap
12      common  brk                     sys_brk
13      64      rt_sigaction            sys_rt_sigaction
14      common  rt_sigprocmask          sys_rt_sigprocmask
15      64      rt_sigreturn            sys_rt_sigreturn/ptreg
16      64      ioctl                   sys_ioctl
```

syscall_64.tbl

```
520     x32     execve                  compat_sys_execve/ptregs
521     x32     ptrace                  compat_sys_ptrace
522     x32     rt_sigpending           compat_sys_rt_sigpending
523     x32     rt_sigtimedwait         compat_sys_rt_sigtimedwait
524     x32     rt_sigqueueinfo         compat_sys_rt_sigqueueinfo
525     x32     sigaltstack             compat_sys_sigaltstack
526     x32     timer_create            compat_sys_timer_create
527     x32     mq_notify               compat_sys_mq_notify
528     x32     kexec_load              compat_sys_kexec_load
529     x32     waitid                  compat_sys_waitid
530     x32     set_robust_list         compat_sys_set_robust_list
531     x32     get_robust_list         compat_sys_get_robust_list
532     x32     vmsplice                compat_sys_vmsplice
533     x32     move_pages              compat_sys_move_pages
534     x32     preadv                  compat_sys_preadv64
535     x32     pwritev                 compat_sys_pwritev64
536     x32     rt_tgsigqueueinfo       compat_sys_rt_tgsigqueueinfo
537     x32     recvmmsg                compat_sys_recvmmsg
538     x32     sendmmsg                compat_sys_sendmmsg
539     x32     process_vm_readv        compat_sys_process_vm_readv
540     x32     process_vm_writev       compat_sys_process_vm_writev
541     x32     setsockopt              compat_sys_setsockopt
542     x32     getsockopt              compat_sys_getsockopt
543     x32     io_setup                compat_sys_io_setup
544     x32     io_submit               compat_sys_io_submit
545     x32     execveat                compat_sys_execveat/ptregs
546     x32     preadv2                 compat_sys_preadv64v2
547     x32     pwritev2                compat_sys_pwritev64v2
```

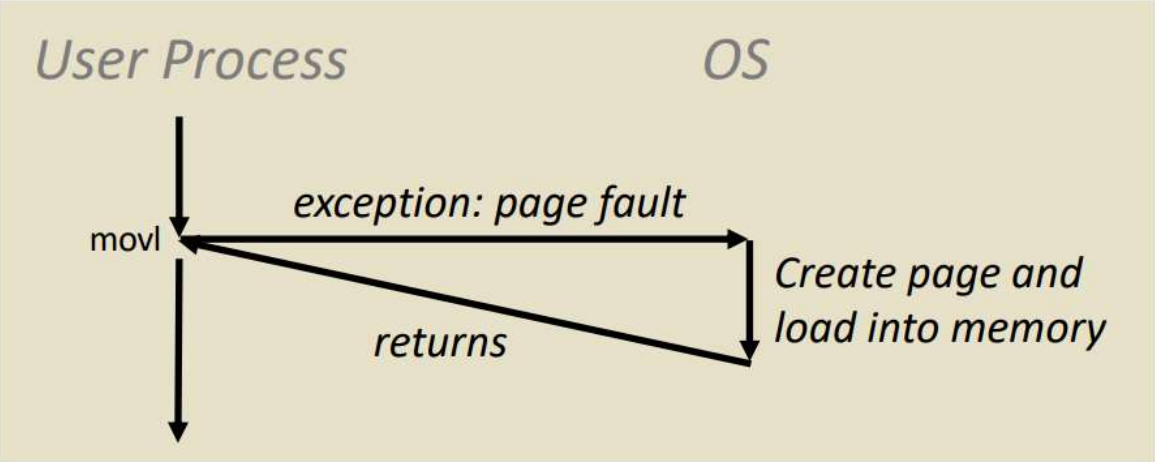
arch/x86/entry/syscalls/sysc

Fault Example: Page Fault

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
80483b7:    c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```

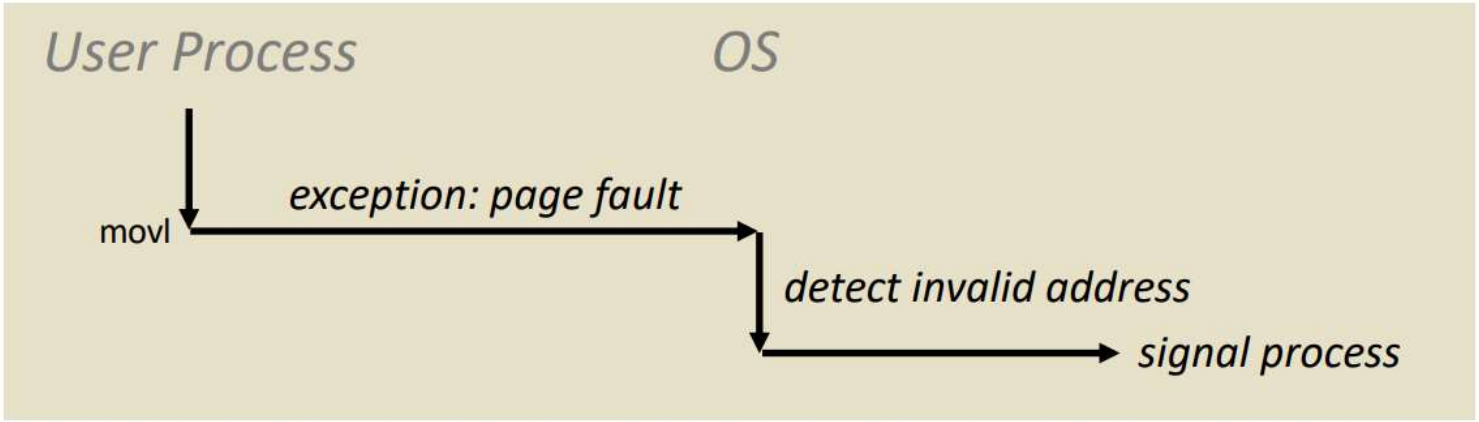


- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

Abort Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:    c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



Page handler detects invalid address
Sends `SIGSEGV` signal to user process
User process exits with “segmentation fault”

Taxonomy

interrupts, exceptions, traps, events, often conflated.

two kinds of vectored events:

- interrupt : asynchronous events
  - consequence of HW, e.g. keypress, timer, disk, …
  - handed off to SW (interrupt handlers) for processing.
- exception : synchronous events
  - consequence of SW executing an instruction.
  - traps, faults, aborts are exceptions.
  - `int`, `syscall` are traps, handled by interrupt handlers.

ITU CPH

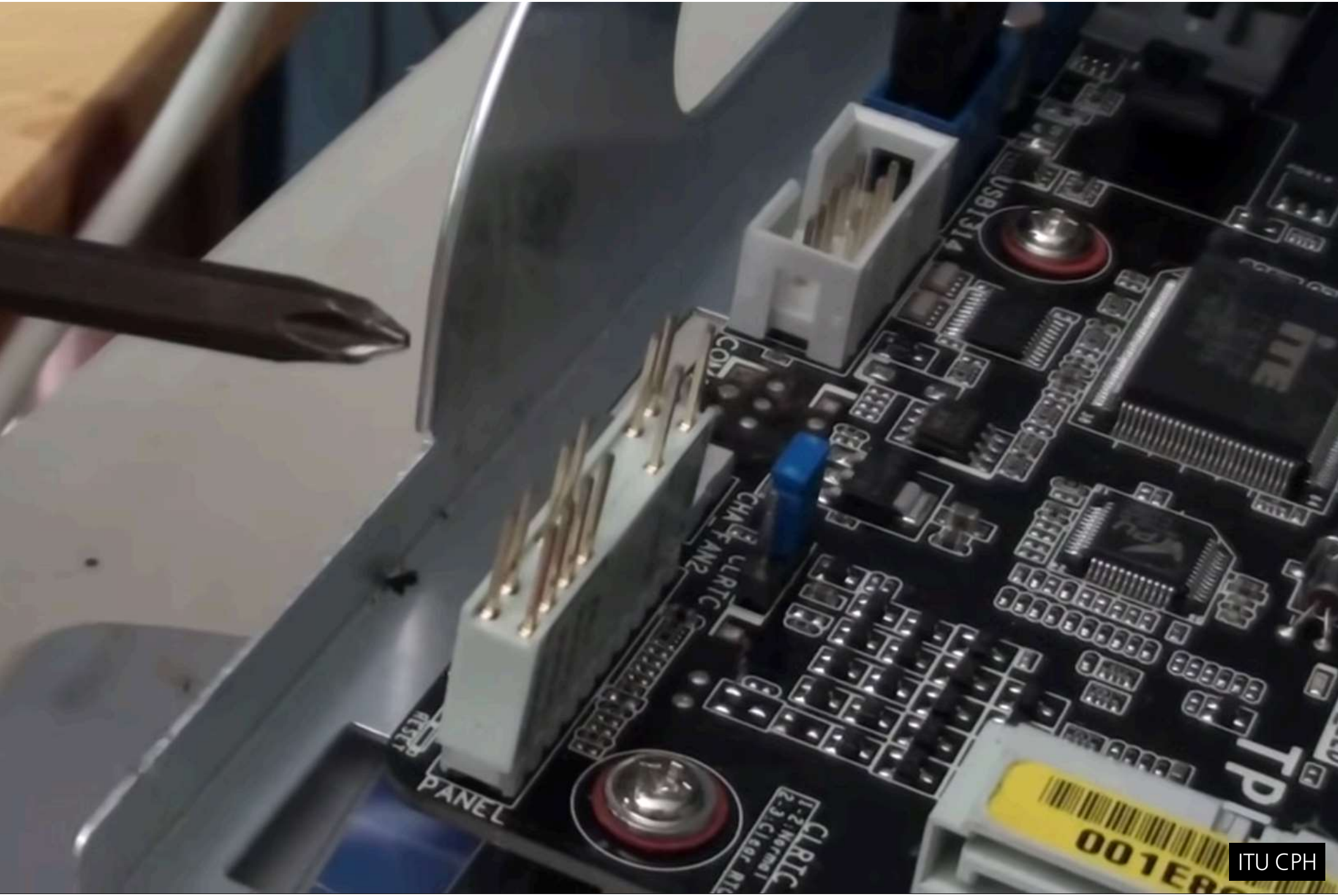
Booting

What puts the OS in place

ITU CPH

Boot Process

ByteByte


PANEL
ITU CPH

Signal Received by PSU

power supply unit (PSU) performs self-test and checks that its output has stabilized.

once it has, sends power good signal to motherboard.

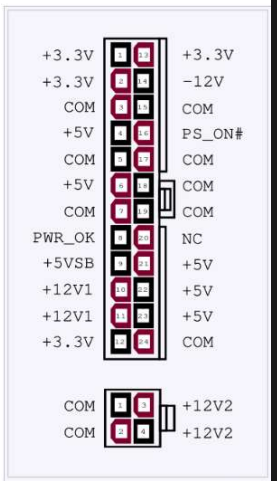
ITU CPH

Power Good Signal

24-pin ATX12V 2.x power supply connector

Color	Signal[A]	Pin[B]	Pin[B][C]	Signal[A]	Color
Orange	+3.3 V	1	13	+3.3 V	Orange
				+3.3 V sense[D]	Brown
Orange	+3.3 V	2	14	−12 V	Blue
Black	Ground	3	15	Ground	Black
Red	+5 V	4	16	Power on[E]	Green
Black	Ground	5	17	Ground	Black
Red	+5 V	6	18	Ground	Black
Black	Ground	7	19	Ground	Black
Grey	Power good[F]	8	20	Reserved[G]	None
Purple	+5 V standby	9	21	+5 V	Red
Yellow	+12 V	10	22	+5 V	Red
Yellow	+12 V	11	23	+5 V	Red
Orange	+3.3 V	12	24	Ground	Black

A. ^ a b Light-blue background denotes control signals.
B. ^ a b Light-green background denotes the pins present only in the 24-pin connector.
C. ^ In the 20-pin connector, pins 13–22 are numbered 11–20 respectively.
D. ^ Supplies +3.3 V power and also has a second low-current wire for remote sensing.[14]
E. ^ A control signal that is pulled up to +5 V by the PSU and must be driven low to turn on the PSU.
F. ^ A control signal that is low when other outputs have not yet reached, or are about to leave, correct voltages.
G. ^ Formerly −5 V (white wire), absent in modern power supplies; it was optional in ATX and ATX12V v1.2 and deleted since v1.3.

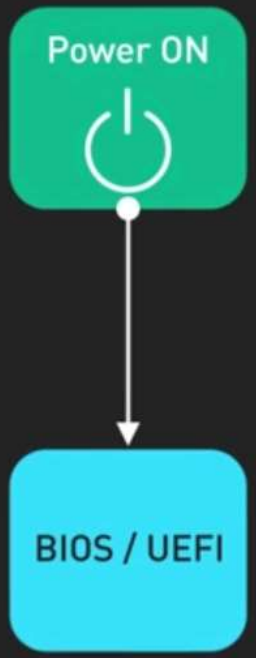


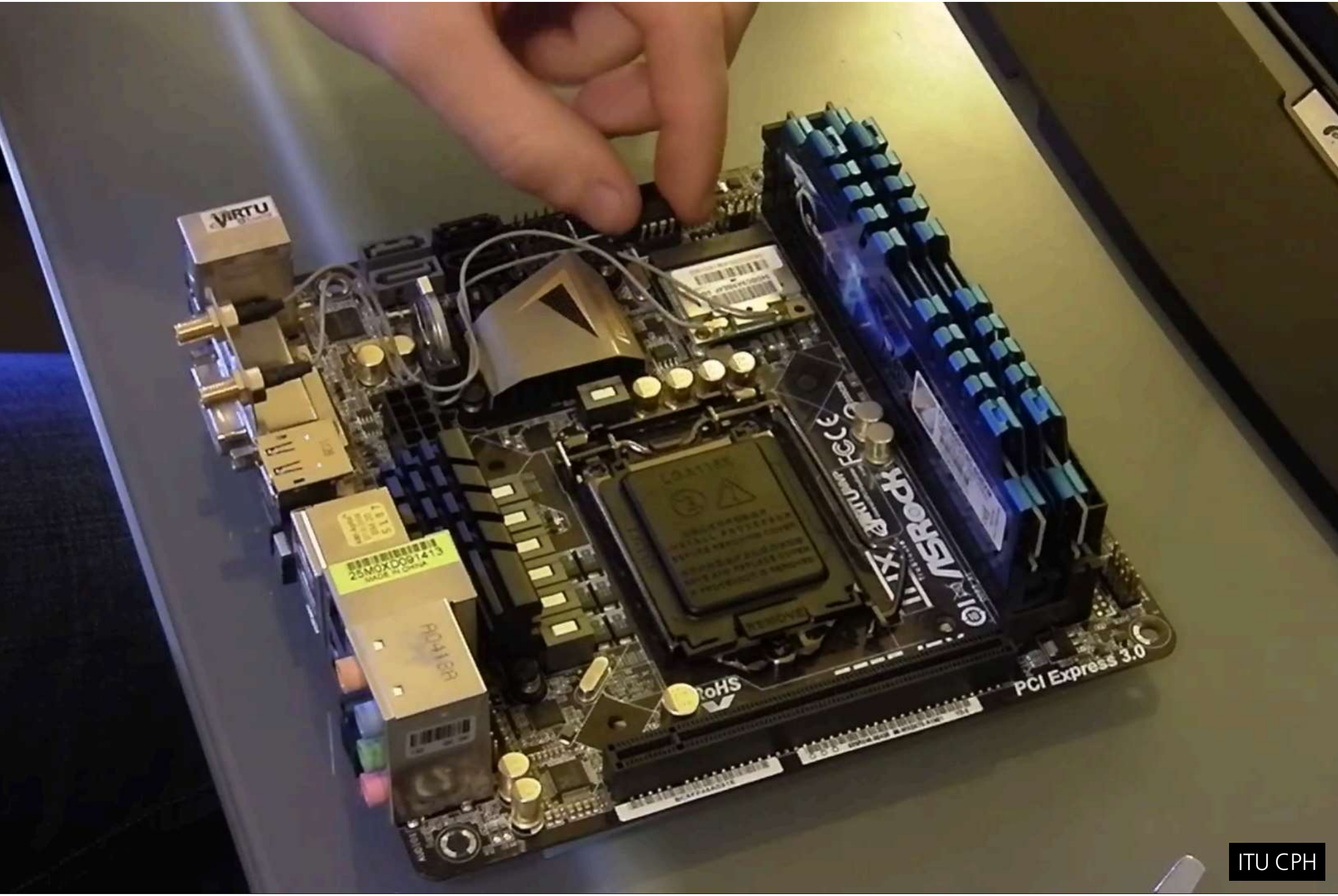
Pinouts of ATX 2.x motherboard power connectors, 24-pin (top) and four-pin "P4" (bottom), as viewed into mating side of the plugs[13]

along one of the wires of that massive PSU-to-motherboard cable.

ITU CPH

Boot Process




VIRTU
PCI Express 3.0
ITU CPH

Execute System Firmware from ROM

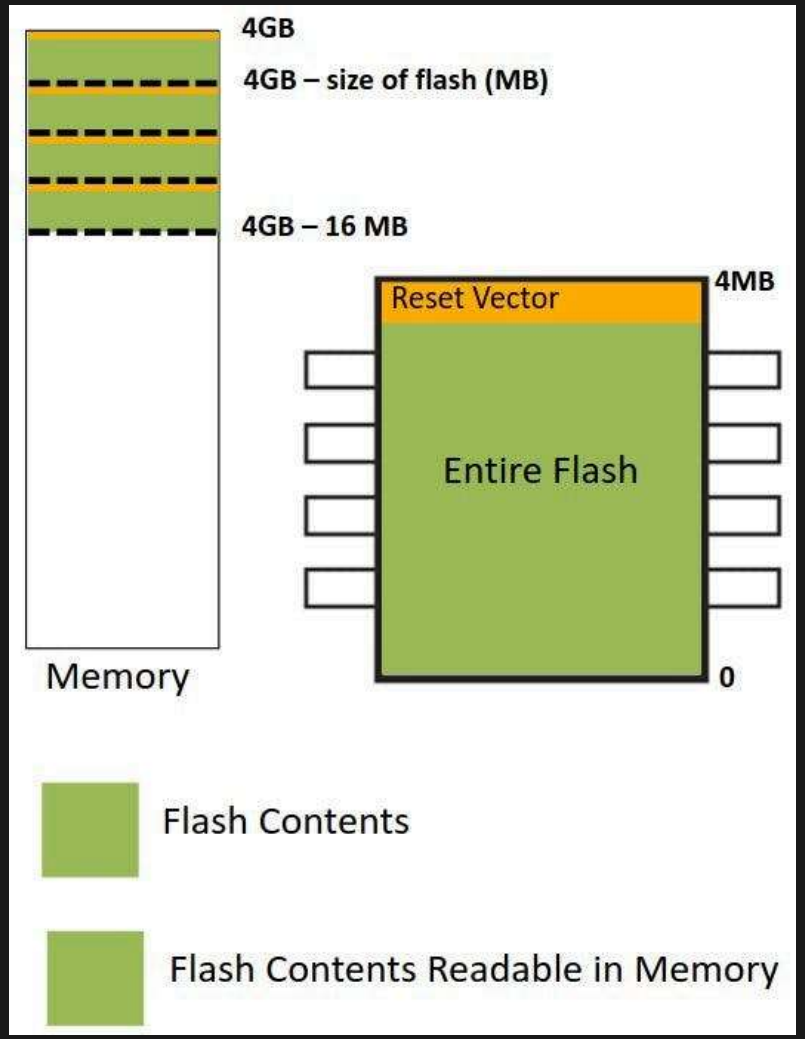
read-only memory (ROM) on motherboard contains (system) firmware (e.g. BIOS, UEFI)

its contents is memory-mapped.

first instruction CPU executes, is one in ROM's last word: the reset vector: a `jmp` into firmware.

(RAM is not configured yet, hence we execute from (slow) ROM)

sets up interrupt handlers (for keyboard, system timer, etc.).



ITU CPH


ITU CPH

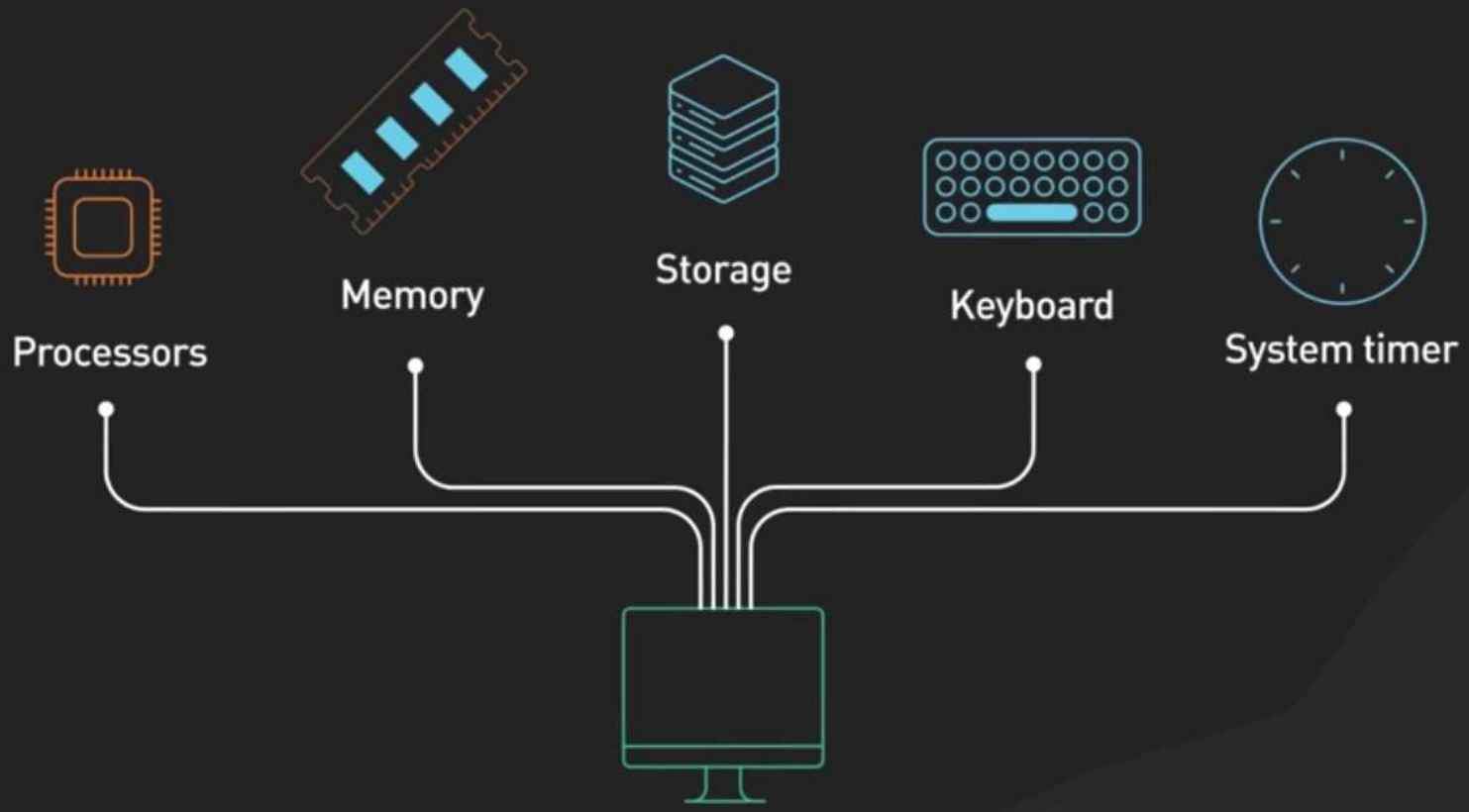
Boot Process

ByteByte

Boot Process

Power-On Self-Test (POST)

ByteByte



Power-On Self-Test (POST)

check all important devices are functioning.

if not, booting halts, and:

ITU CPH

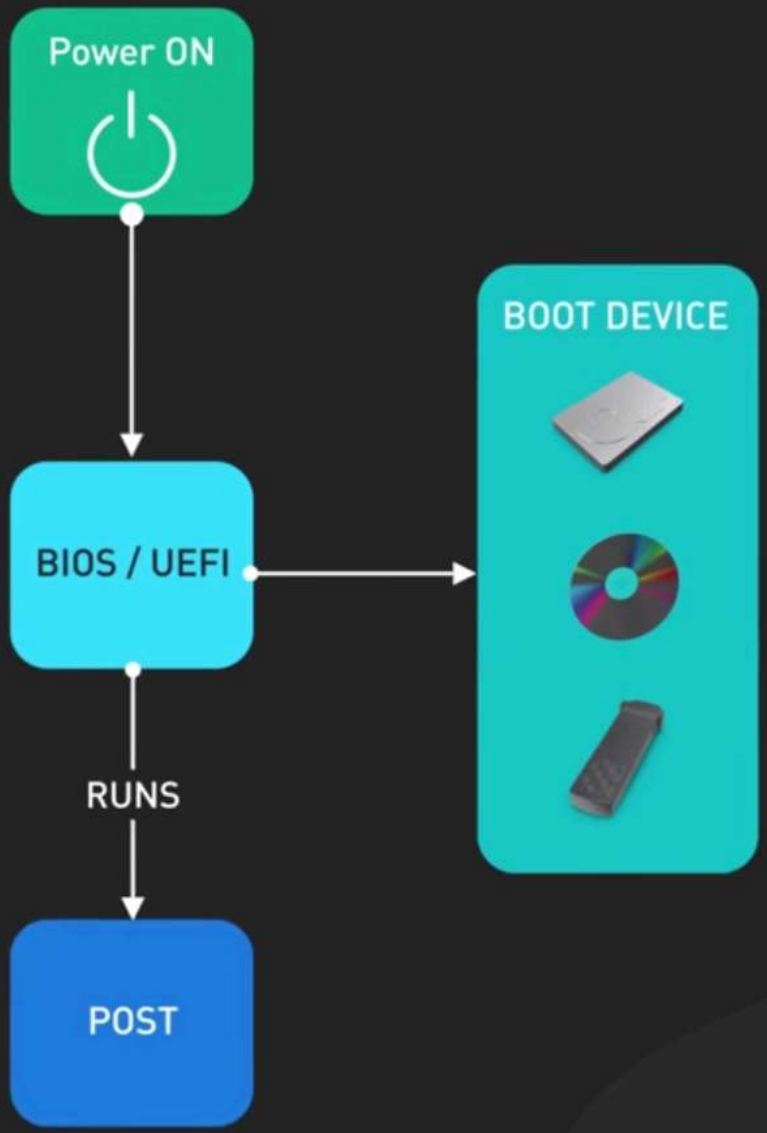
Boot Process

Power-On Self-Test (POST)

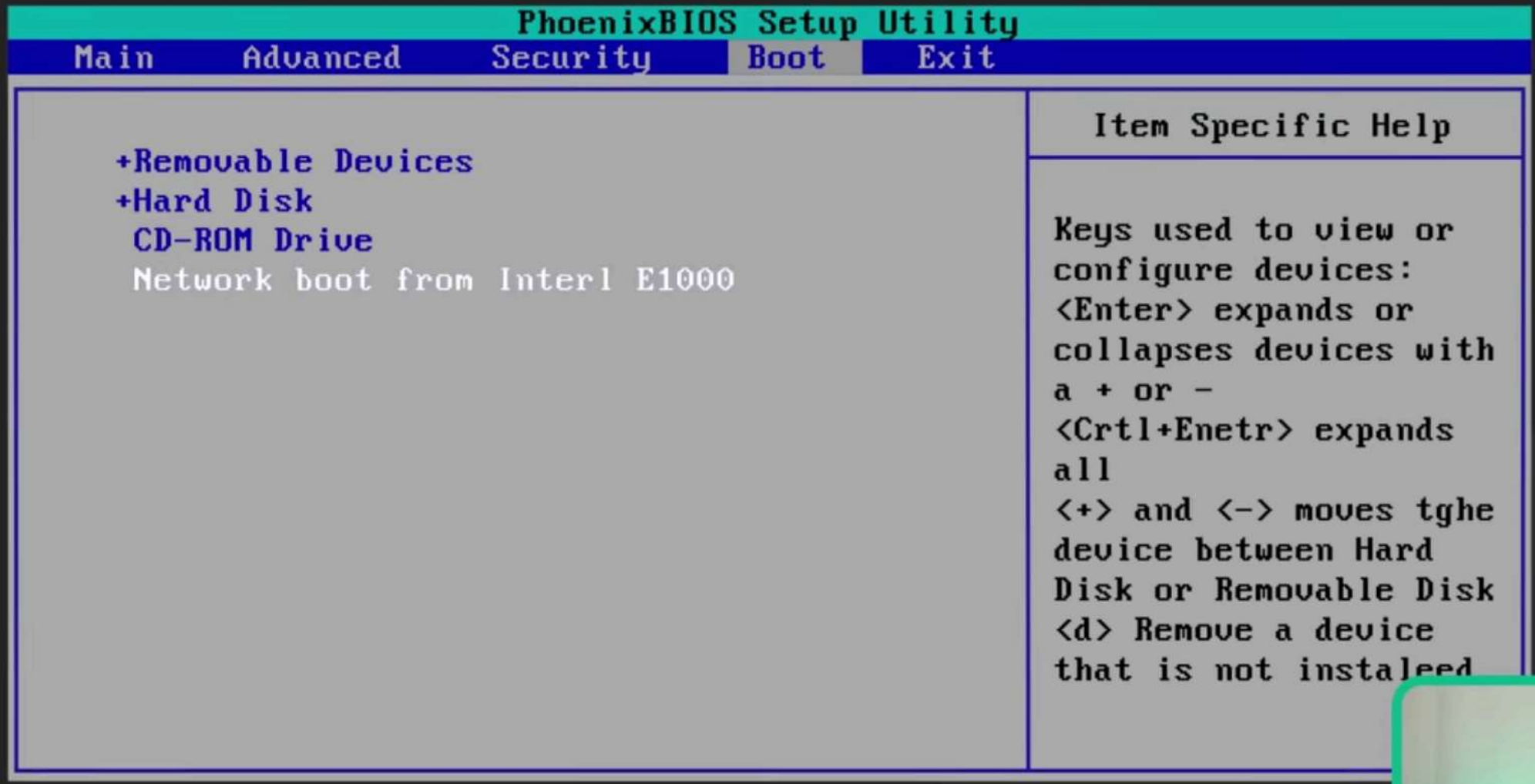
ByteByte

[ERROR MESSAGE]

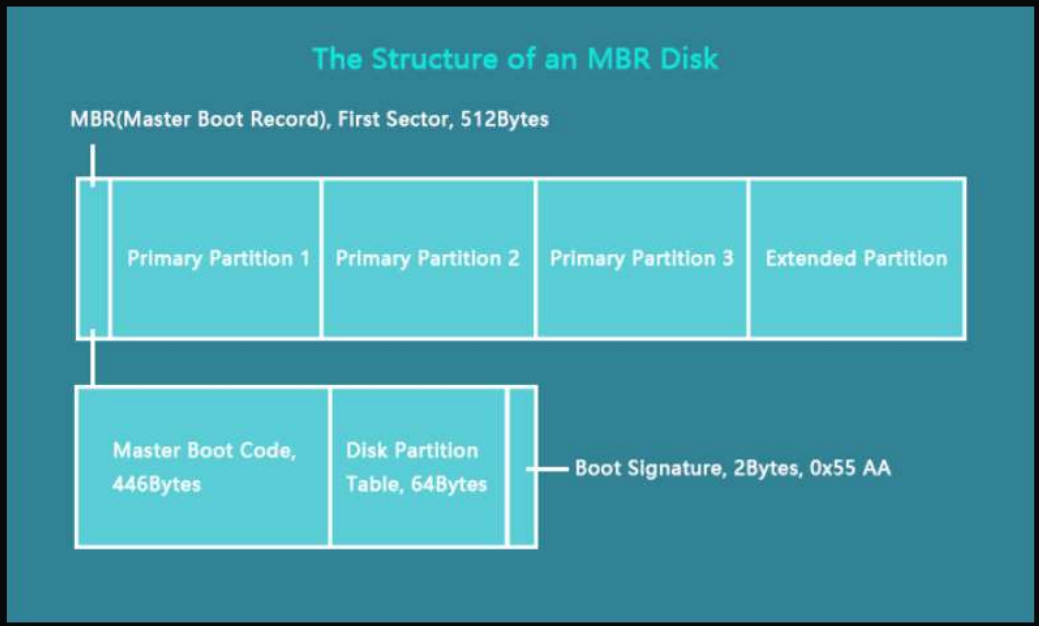
Boot Process



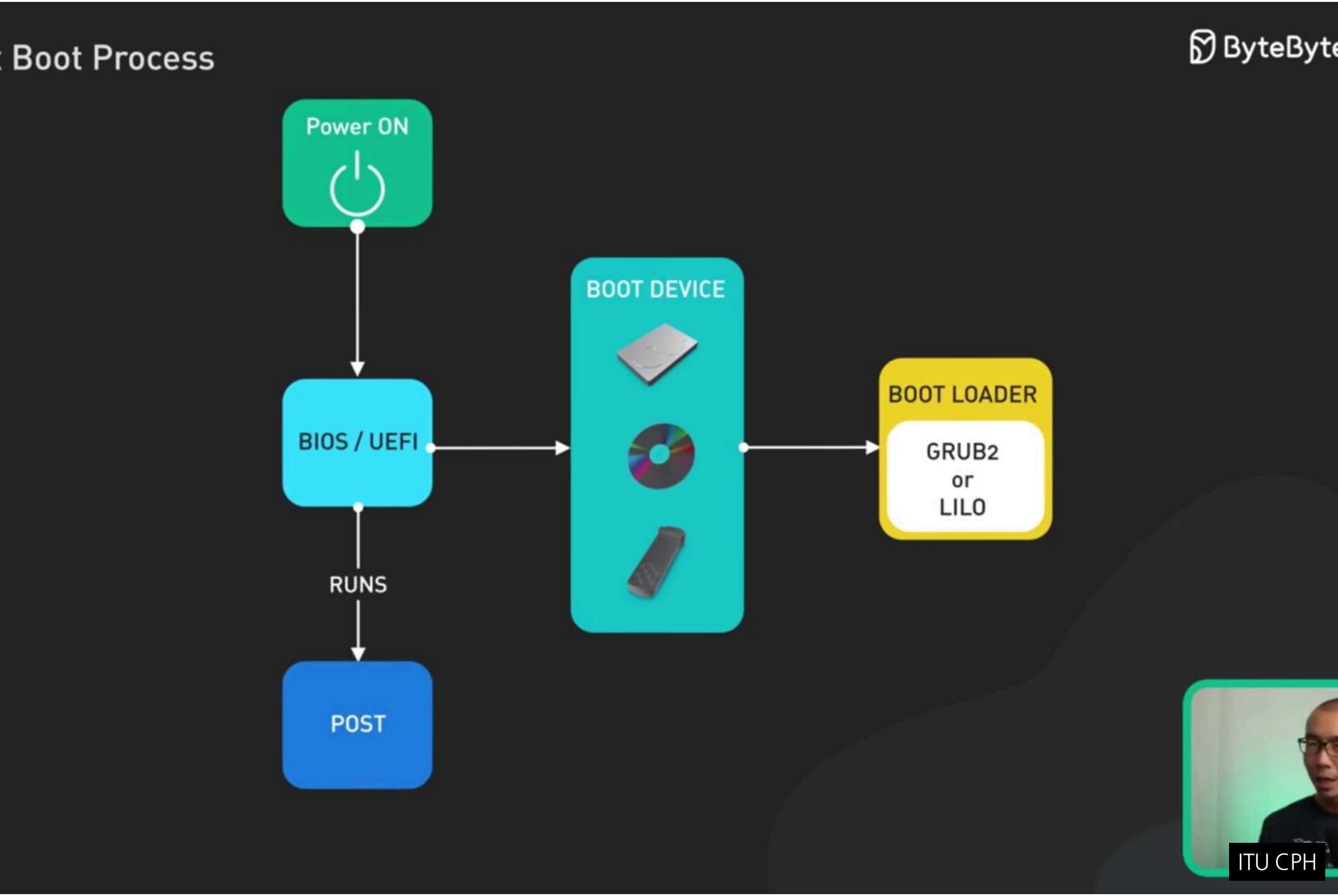
Boot Device Priority



Boot Device Houses A Bootloader



ITU CPH


Boot Process
ByteByte
Power ON
BIOS / UEFI
RUNS
POST
BOOT DEVICE
BOOT LOADER
GRUB2
or
LILO
ITU CPH

Bootloader in MBR “Hello, World!”

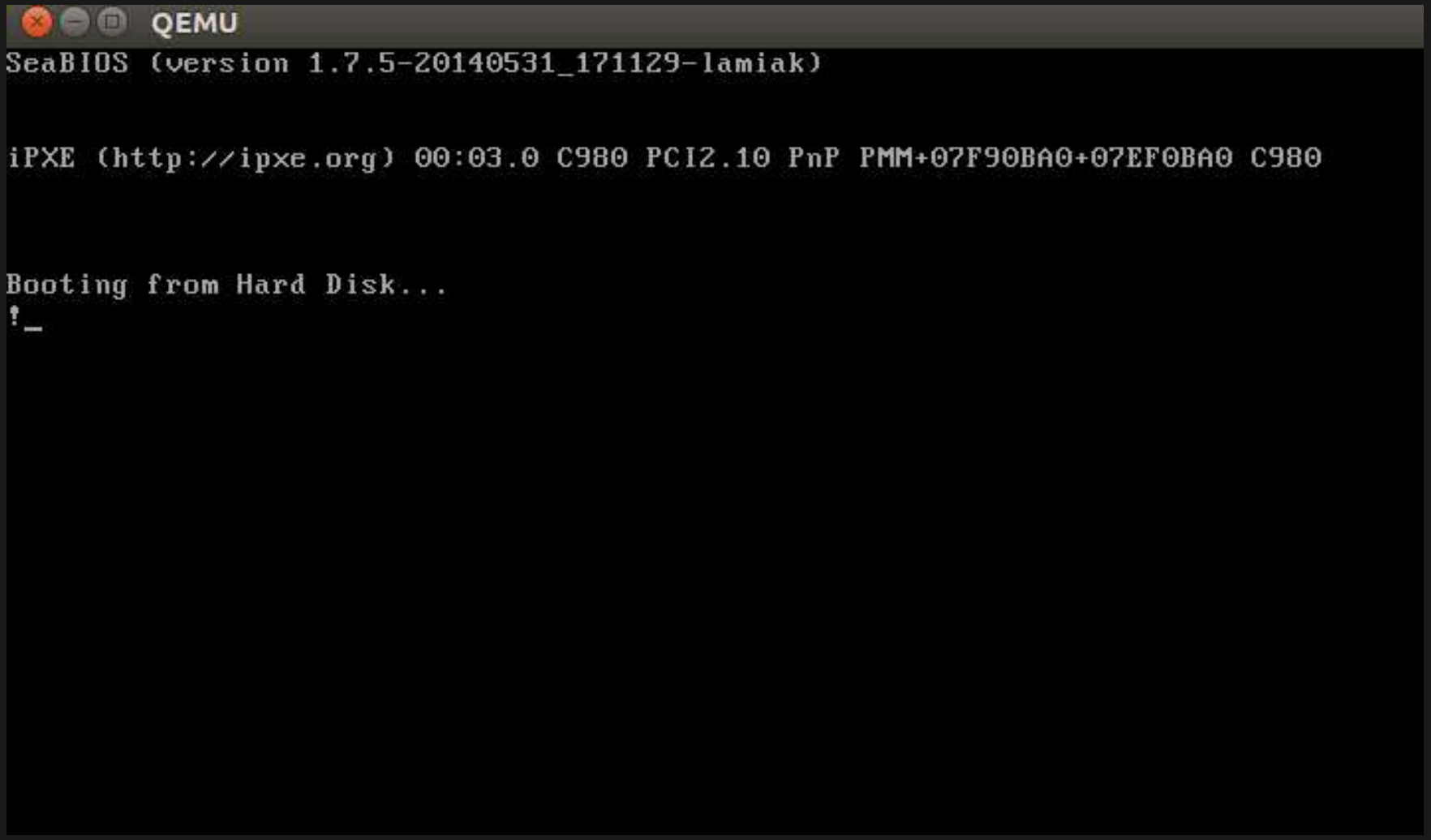
```
1  ;
2  ; Note: this example is written in Intel Assembly syntax
3  ;
4  [BITS 16]
5
6  boot:
7      mov al, '!'
8      mov ah, 0x0e
9      mov bh, 0x00
10     mov bl, 0x07
11
12     int 0x10
13     jmp $
14
15 times 510-($-$$) db 0
16
17 db 0x55
18 db 0xaa
```

(see the two magic bytes at the end; cf. to MBR slide)

(that interrupt is handled by the system firmware)

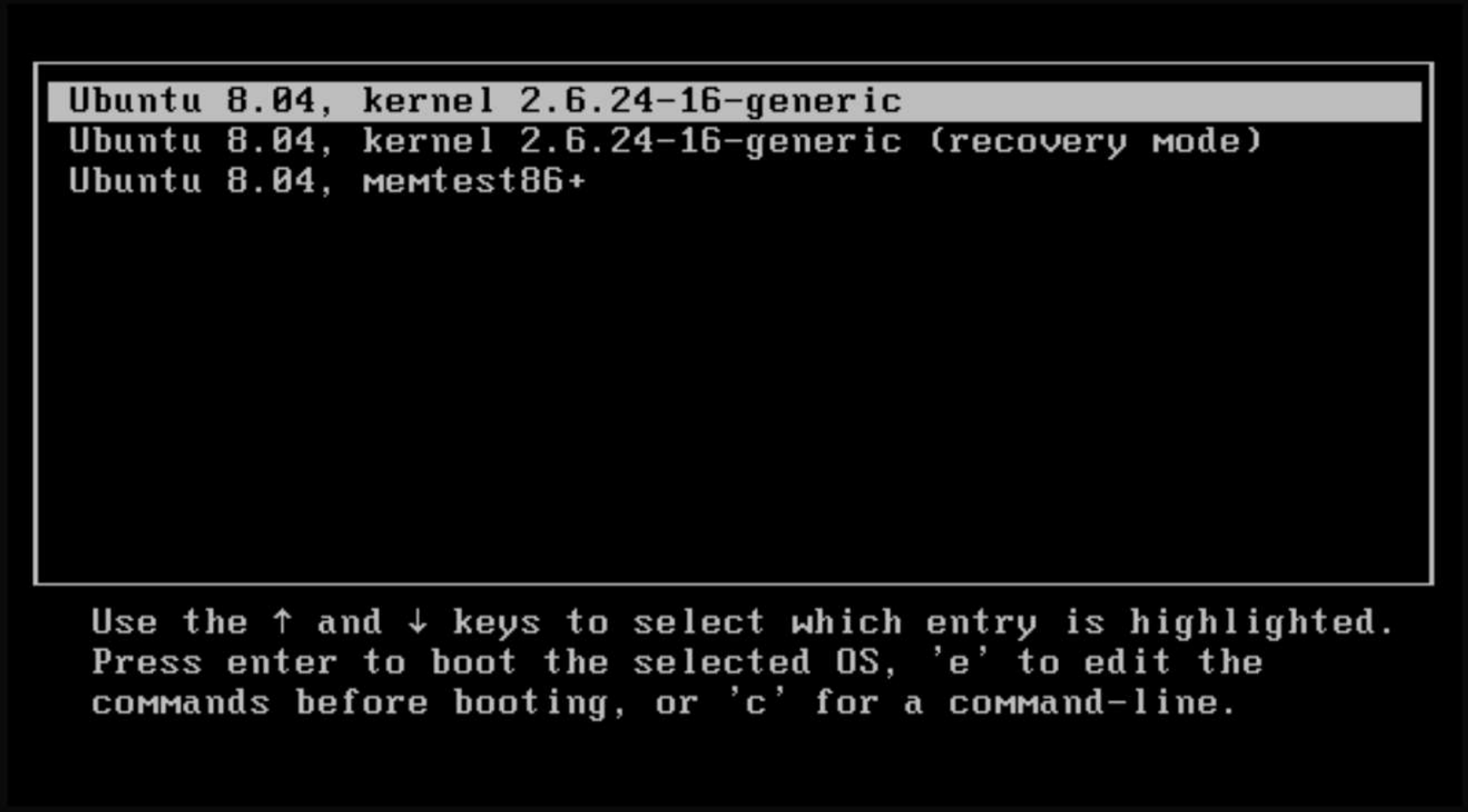
ITU CPH

Bootloader in MBR “Hello, World!”



ITU CPH

Bootloader Typically Looks Like:



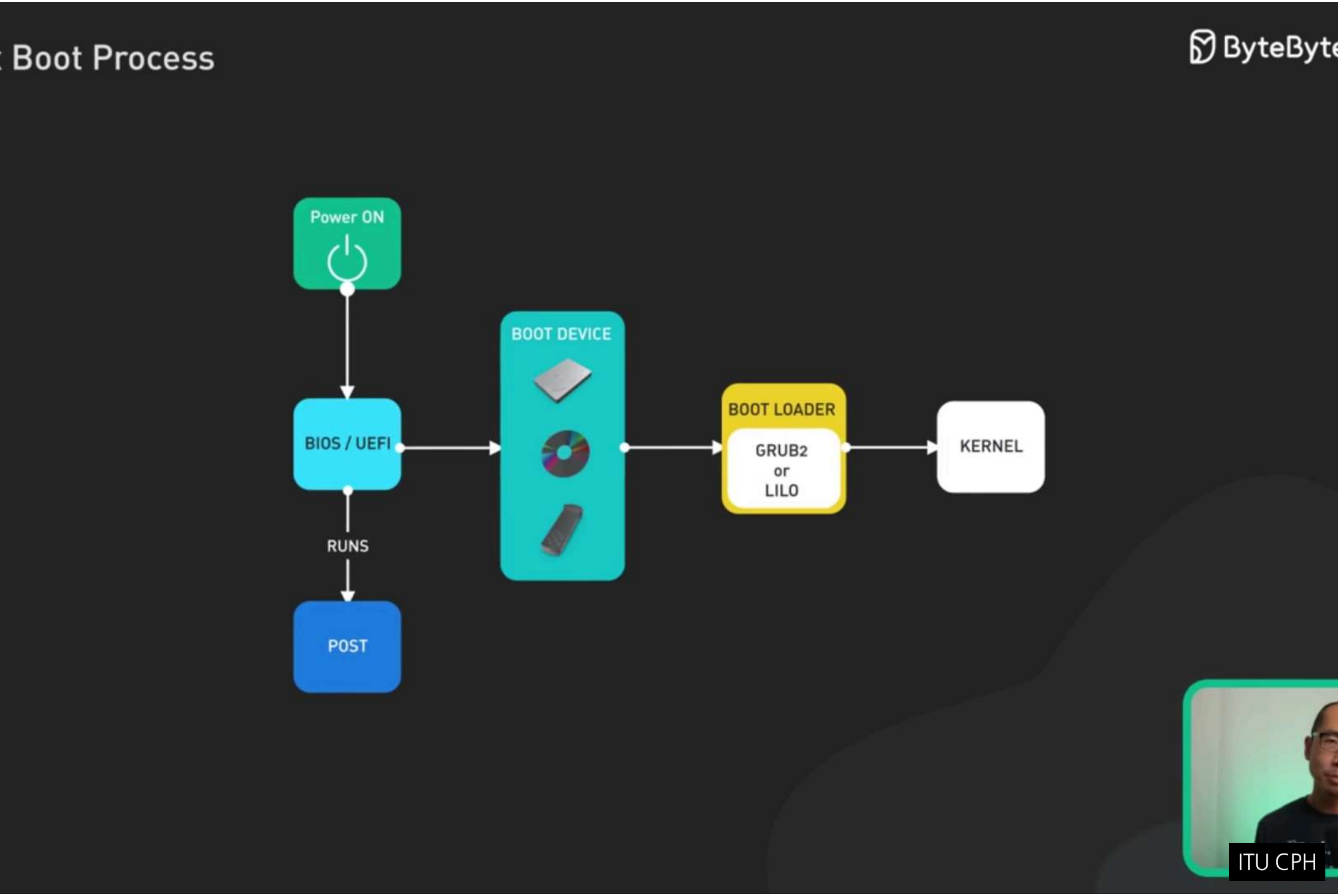
ITU CPH

Boot Process

ByteByte

The key jobs for the boot loader are:

1. Locate the operating system kernel on the disk
2. Load the kernel into the computer's memory
3. Start running the kernel code


Boot Process
ByteByte
Power ON
BIOS / UEFI
RUNS
POST
BOOT DEVICE
BOOT LOADER
GRUB2
or
LILO
KERNEL
ITU CPH

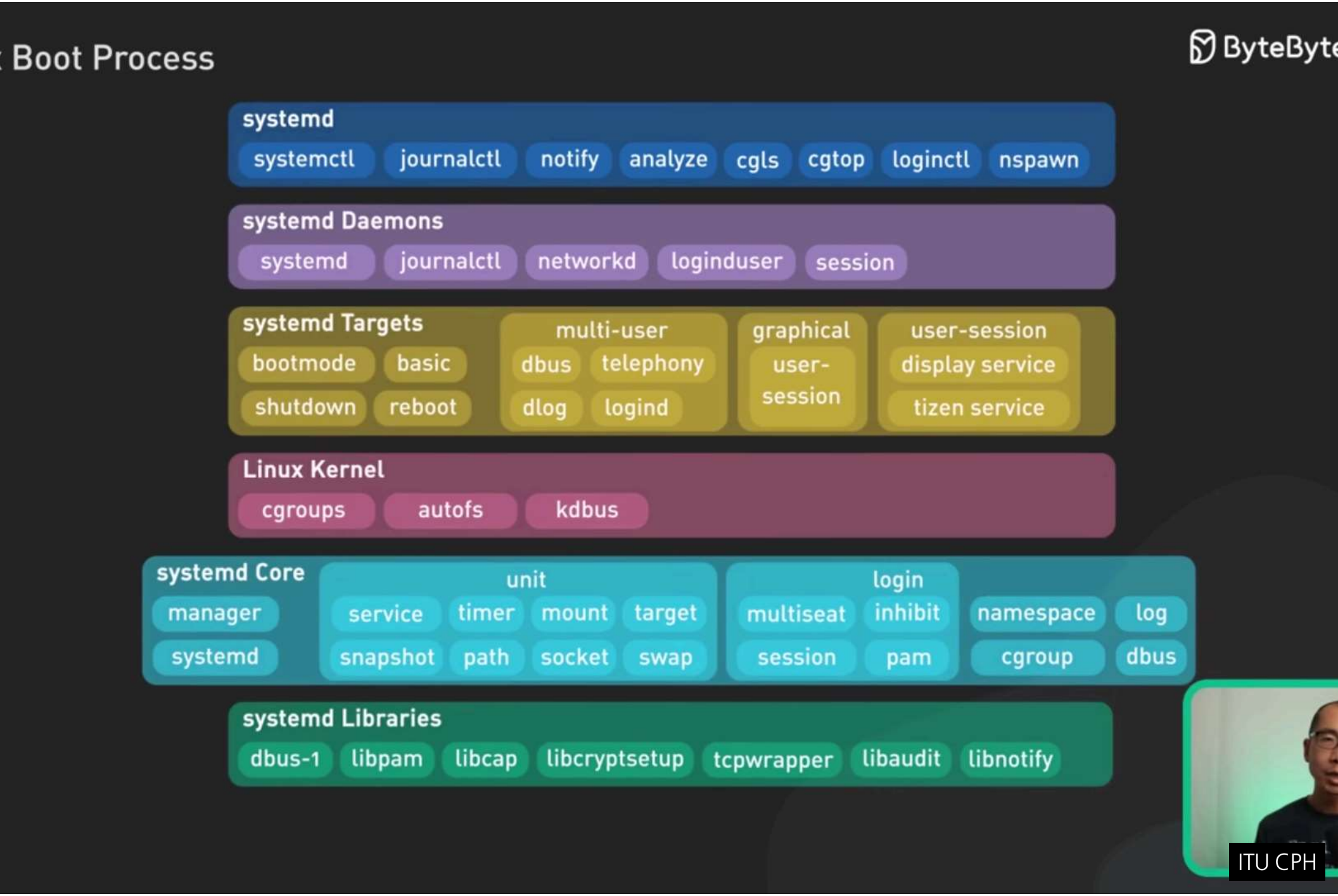
Kernel Starting

```
[  OK  ] Started Apply Kernel Variables.
[  OK  ] Mounted Kernel Debug File System.
[  OK  ] Mounted Huge Pages File System.
[  OK  ] Mounted POSIX Message Queue File System.
[  OK  ] Started Read and set NIS domainname from /etc/sysconfig/network.
[  OK  ] Activated swap /dev/mapper/cl-swap.
[  OK  ] Reached target Swap.
[  OK  ] Started Remount Root and Kernel File Systems.
         Starting Flush Journal to Persistent Storage...
         Starting Load/Save Random Seed...
         Starting Create Static Device Nodes in /dev...
[  OK  ] Started Load/Save Random Seed.
[  OK  ] Started Flush Journal to Persistent Storage.
[  OK  ] Started Setup Virtual Console.
[  OK  ] Started Create Static Device Nodes in /dev.
         Starting udev Kernel Device Manager...
[  OK  ] Started udev Kernel Device Manager.
[  OK  ] Created slice system-lvm2\x2dpvscan.slice.
         Starting LVM event activation on device 8:2...
[  OK  ] Started Monitoring of LVM2 mirrors, snapshots etc. using dmeventd or progress polling.
[  OK  ] Reached target Local File Systems (Pre).
         Starting File System Check on /dev/disk/by-uuid/0868ca58-6212-404b-91d8-b512c612c58a...
[  OK  ] Started LVM event activation on device 8:2.
[  OK  ] Started File System Check on /dev/disk/by-uuid/0868ca58-6212-404b-91d8-b512c612c58a.
         Mounting /boot...
[  OK  ] Mounted /boot.
[  OK  ] Reached target Local File Systems.
         Starting Import network configuration from initramfs...
         Starting Tell Plymouth To Write Out Runtime Data...
         Starting Restore /run/initramfs on shutdown...
[  OK  ] Started Restore /run/initramfs on shutdown.
[  OK  ] Started Tell Plymouth To Write Out Runtime Data.
[  OK  ] Started Import network configuration from initramfs.
         Starting Create Volatile Files and Directories...
[  OK  ] Started Create Volatile Files and Directories.
         Starting Security Auditing Service...
```

ITU CPH

Boot Process

ByteByte


Boot Process
ByteByte
systemd
systemctl
journalctl
notify
analyze
cgls
cgtop
loginctl
nspawn
systemd Daemons
systemd
journalctl
networkd
loginduser
session
systemd Targets
bootmode
basic
shutdown
reboot
multi-user
dbus
telephony
dlog
logind
graphical
user-session
user-session
display service
tizen service
Linux Kernel
cgroups
autofs
kdbus
systemd Core
manager
systemd
unit
service
timer
mount
target
snapshot
path
socket
swap
login
multiseat
inhibit
session
pam
namespace
log
cgroup
dbus
systemd Libraries
dbus-1
libpam
libcap
libcryptsetup
tcpwrapper
libaudit
libnotify
ITU CPH

Processes

Tasks being managed by the OS.

ITU CPH

Processes

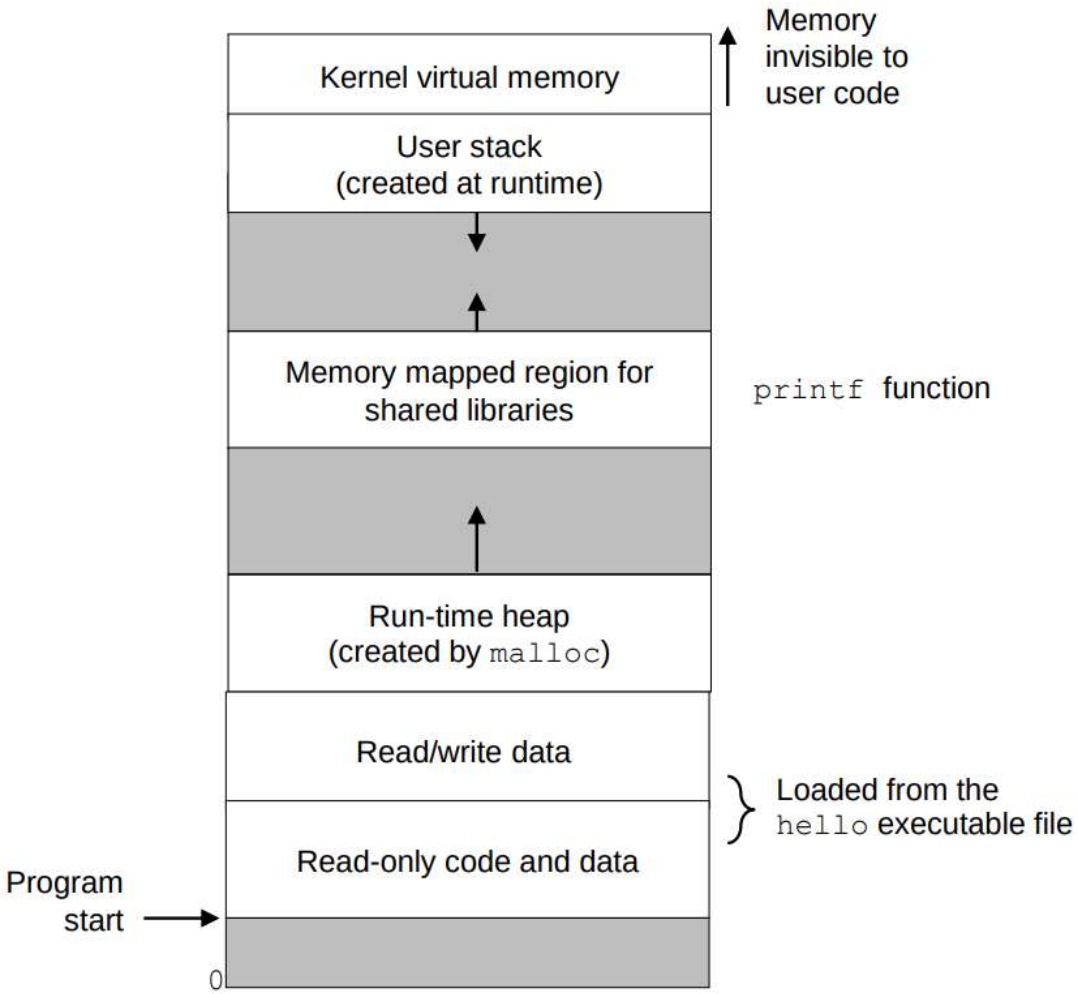
26 Processes

Processes are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.

Processes are organized hierarchically. Each process has a *parent process* which explicitly arranged to create it. The processes created by a given parent are called its *child processes*. A child inherits many of its attributes from the parent process.

https://www.gnu.org/software/libc/manual/html_node/Processes.html

Virtual Memory



Each process has its own address space.

All address spaces are structured in the same way.

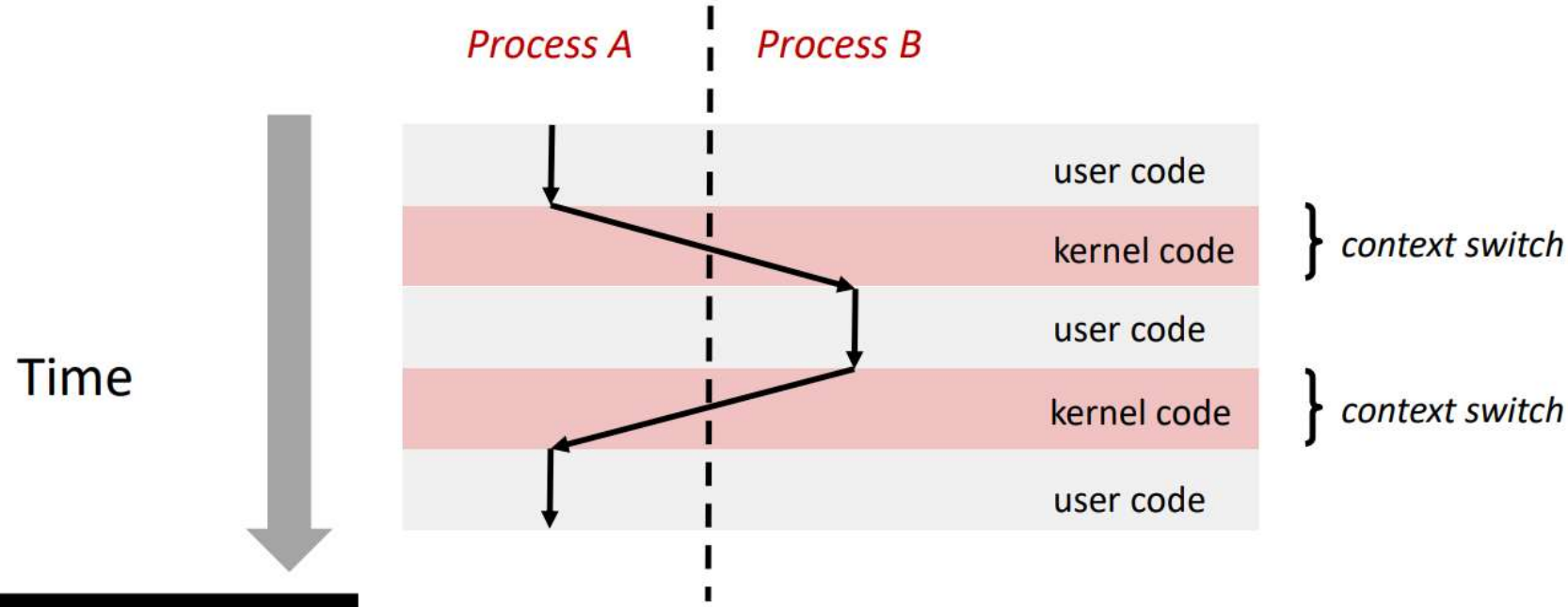
Processes

- Process creation/termination/control defined in standard C library (unistd.h)
- Transferring the thread of control from one process to another is called *context switching*. It is managed by the OS kernel.

Context Switching

Control flow passes from one process to another via a *context switch*

Important: the kernel is not a separate process, but rather runs as part of some user process



UNIVERSITY OF COPENHAGEN

ITU CPH

Context Switching

Processor modes:

- Supervisor vs. user modes: "Supervisor mode may provide access to different peripherals, to memory management hardware or to different memory address spaces. It is also capable of interrupt enabling, disabling, returning and loading of processor status."
- Supervisor mode entered on system call

Context Switching

When a context switch is made the scheduler marks the task as interruptible, saves the process's task_struct and replaces the current tasks pointer with a pointer to the new process's task_struct, marked as running, restoring its memory access and register context.

VM as a Tool for Memory Protection

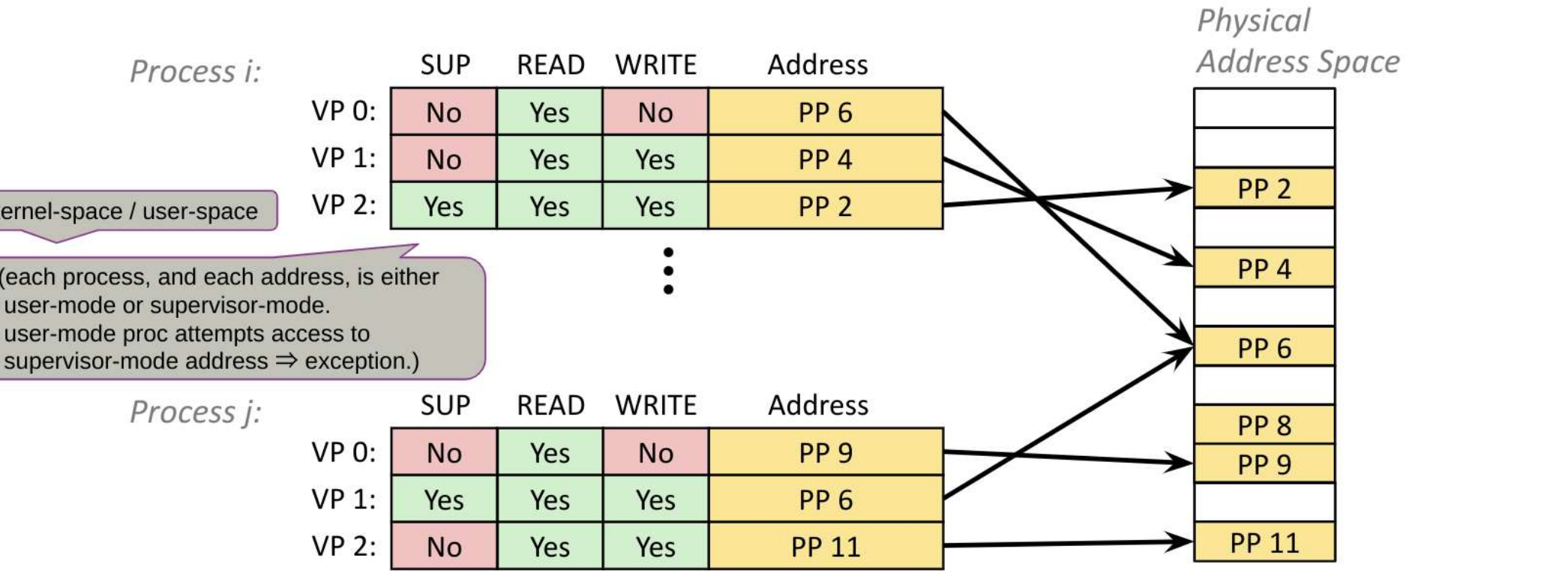
page table entries

Extend PTEs with permission bits

(virt into phys)

Page fault handler checks these bits before remapping

If violated, send process SIGSEGV (segmentation fault)



now MMU can check whether proc has access to region in mem!

UNIVERSITY OF COPENHAGEN

ITU CPH

4

The question is broad, but I can give some general information and links referencing the Instruction Set architecture (ISA) which describes all the instructions.

You can't `MOV` or `POP` a value into *CS* (on 286+[1]) so you are prevented from modifying *CS* that way. For example for `POP` there is a rule:

> The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

And the rule for `MOV` is similar:

> The MOV instruction cannot be used to load the CS register.

You can modify *CS* indirectly through `syscall`, `sysenter`, FAR `jmp` (via call gate), FAR `call` (via call gate), `iret`, `retf` (FAR return) or `int`. You can review the ISA for each instruction and what privilege level checks are applied. You can't arbitrarily change CPL if you don't have the privilege and access rights to do so.

Under most circumstances if you have the privilege to affect a change to CPL it will be changed. If you don't have the required privileges you get an exception (privilege level checks usually involve RPL, CPL, DPL). If using conforming code segments (a different topic) you can request to execute code using a code segment with a higher privileged DPL but the CPL will remain unchanged. It is a case where the CPL and DPL (Descriptor privilege level) of *CS* can be different while code is executing.

Footnotes

[1]You were allowed to modify *CS* via `POP` and `MOV` on 8088/8086 processors.

Share Improve this answer Follow

edited Sep 2, 2019 at 19:00

answered Sep 2, 2019 at 17:27

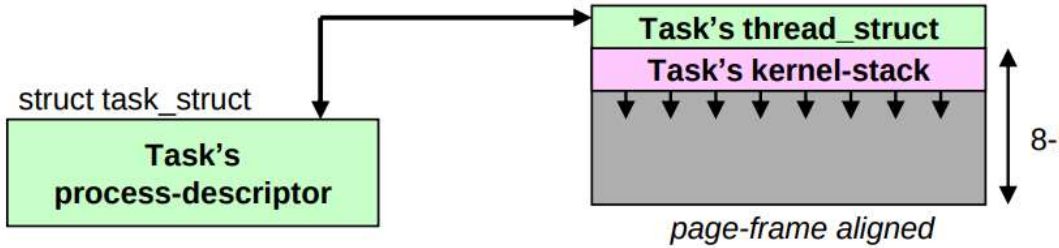
Michael Petch
47.3k ● 9 ● 116 ● 212

Add a comment

Not the answer you're looking for? Browse other questions tagged `x86` `x86-64` `cpu-architecture` `protected-mode` or ask your own question.

ITU CPH

Process State (kernel)

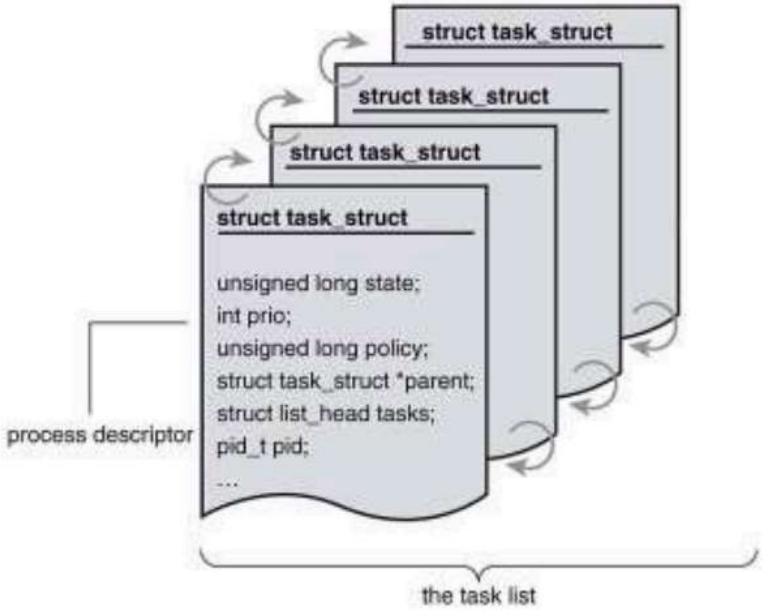


Process context:

- 8KB / process in kernel space to store process descriptor task_struct (/linux/include/linux/sched.h).
  State:

```
#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_ZOMBIE 4
#define TASK_STOPPED 8
```

  Process ID
  + virtual memory info, file system info, open files, signal handlers, …
- The thread of execution thread_struct (/linux/arch/x86/include/asm/processor.h)
  PC, registers, Fault info,



Process Management (libc)

- Spawning process: fork()
- Terminating process: exit()
- Waiting for process: wait()
- Executing a program within a process: execve()

On cos: /usr/include/unistd.h

`fork`: Creating New Processes

```
int fork(void)
```

- creates a new process (child process) that is identical to the calling process (parent process)
- returns 0 to the child process
- returns child's **`pid`** to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

Understanding fork

Process n

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Child Process m

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent`

Which one is first?
No guarantee!

`hello from child`

UNIVERSITY OF COPENHAGEN

ITU CPH

Fork Example #1

Parent and child both run same code

- Distinguish parent from child by return value from **`fork`**

Start with same state, but each has private copy

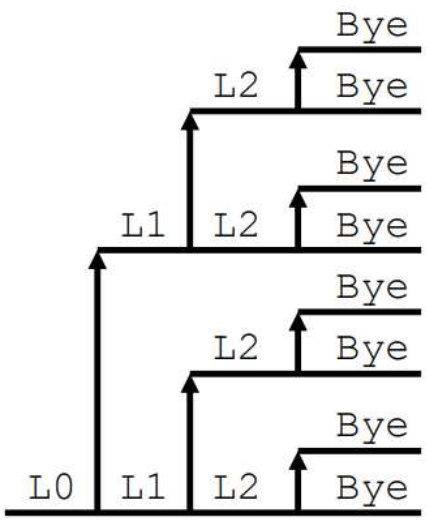
- Including shared output file descriptor
- Relative ordering of their print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
     printf("Child has x = %d\n", ++x);
    } else {
     printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

Both parent and child can continue forking

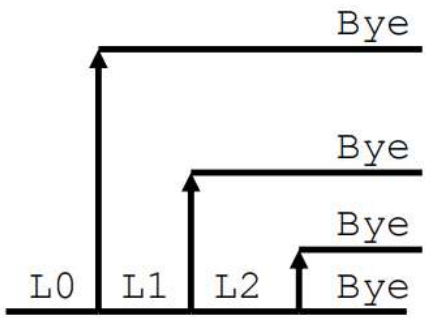
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

Both parent and child can continue forking

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



`exit`: Ending a process

`void exit(int status)`

exits a process

- Normally return with status 0

`atexit()` registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

- Idea
  - When process terminates, still consumes system resources
    - Various tables maintained by OS
  - Called a “zombie”
    - Living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process
- What if parent doesn’t reap?
  - If any parent terminates without reaping a child, then child will be reaped by `init` process
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

```
void fork7()
{
    if (fork() == 0) {
     /* Child */
     printf("Terminating Child, PID = %d\n",
            getpid());
     exit(0);
    } else {
     printf("Running Parent, PID = %d\n",
            getpid());
     while (1)
         ; /* Infinite loop */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

`ps` shows child process as “defunct”

Killing parent allows child to be reaped by `init`

UNIVERSITY OF COPENHAGEN

ITU CPH

Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
     /* Child */
     printf("Running Child, PID = %d\n",
            getpid());
     while (1)
         ; /* Infinite loop */
    } else {
     printf("Terminating Parent, PID = %d\n",
            getpid());
     exit(0);
    }
}
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

Child process still active even though parent has terminated

Must kill explicitly, or else will keep running indefinitely

UNIVERSITY OF COPENHAGEN

ITU CPH

`wait`: Synchronizing with Children

```
int wait(int *child_status)
```

suspends current process until one of its children terminates

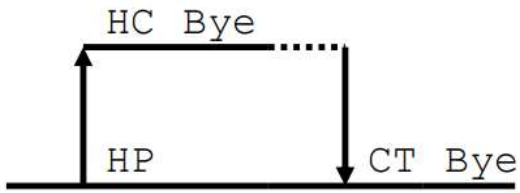
return value is the **`pid`** of the child process that terminated

if **`child_status != NULL`**, then the object it points to will be set to a status indicating why the child process terminated

wait: Synchronizing with Children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



UNIVERSITY OF COPENHAGEN

ITU CPH

wait() Example

If multiple children completed, will take in arbitrary order

Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

waitpid(): Waiting for a Specific Process

waitpid(pid, &status, options)

suspends current process until specific process terminates

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
     if ((pid[i] = fork()) == 0)
         exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
     pid_t wpid = waitpid(pid[i], &child_status, 0);
     if (WIFEXITED(child_status))
         printf("Child %d terminated with exit status %d\n",
             wpid, WEXITSTATUS(child_status));
     else
         printf("Child %d terminated abnormally\n", wpid);
    }
}
```

`execve`: Loading and Running Programs

```
int execve(
      char *filename,
      char *argv[],
      char *envp[]
   )
```

Loads and runs in current process:

- Executable **`filename`**
- With argument list **`argv`**
- And environment variable list **`envp`**

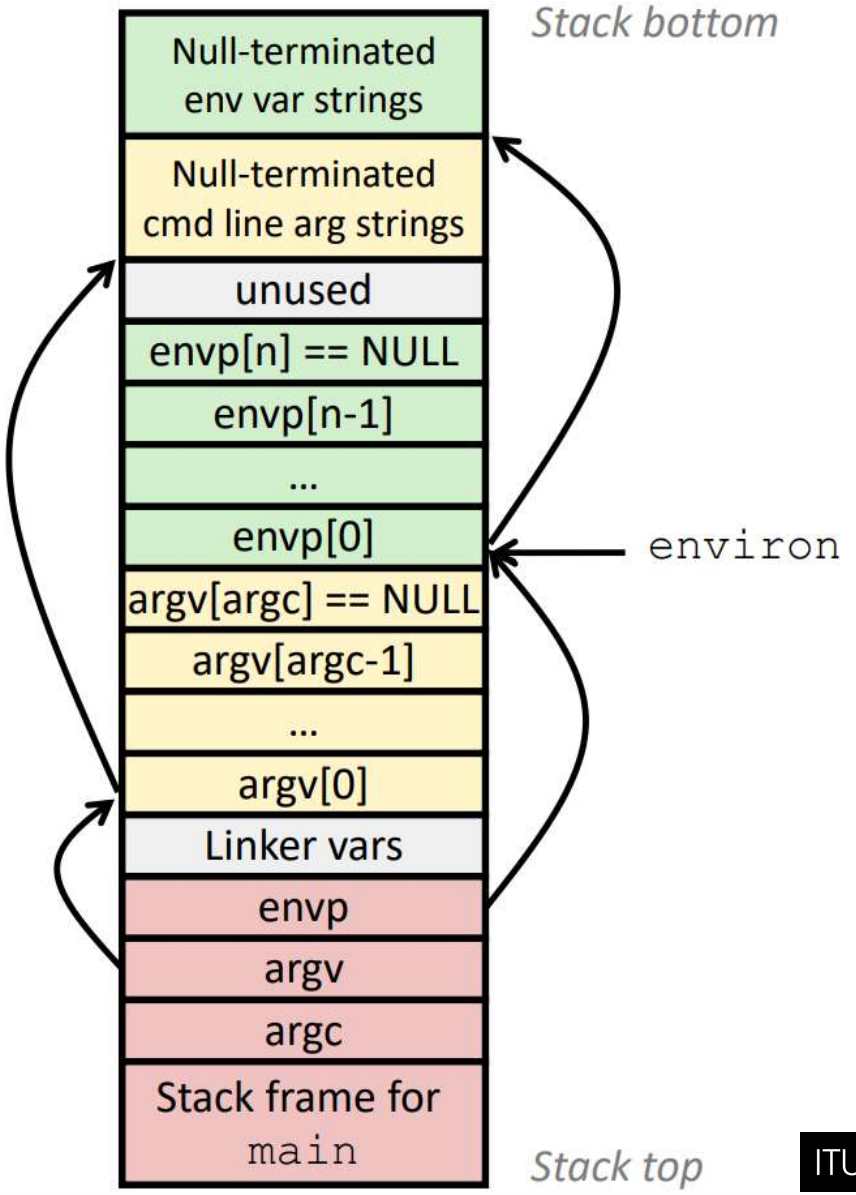
Does not return (unless error)

Overwrites code, data, and stack

- keeps pid, open files and signal context

Environment variables:

- "name=value" strings
- `getenv` and `putenv`

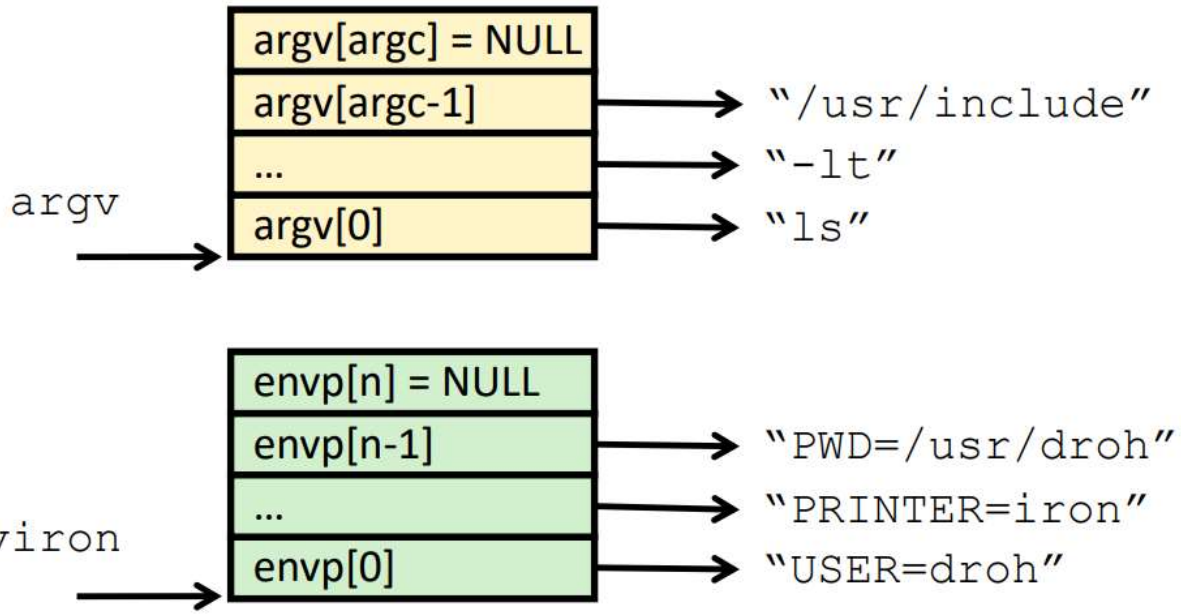


UNIVERSITY OF COPENHAGEN

ITU CPH

execve Example

```
if ((pid = fork()) == 0) { /* Child runs user job */
    if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```



Outline

- Process Management
- Signals

Signals

A *signal* is a small message that notifies a process that an event of some type has occurred in the system

- akin to exceptions and interrupts
- sent from the kernel (sometimes at the request of another process) to a process
- signal type is identified by small integer ID's (1-30)
- only information in a signal is its ID and the fact that it arrived

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Sending a Signal

- Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process

- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
  - Another process has invoked the **`kill`** system call to explicitly request the kernel to send a signal to the destination process

Receiving a Signal

A destination process *receives* a signal. It is forced by the kernel to react in some way to the delivery of the signal

Three possible ways to react:

- ***Ignore*** the signal (do nothing)
- ***Terminate*** the process (with optional core dump)
- ***Catch*** the signal by executing a user-level function called ***signal handler***

Akin to a hardware exception handler being called in response to an asynchronous interrupt

Signal Concepts

Kernel maintains `pending` and `blocked` bit vectors in the context of each process

- **`pending`**: represents the set of pending signals
  - Kernel sets bit k in **`pending`** when a signal of type k is delivered
  - Kernel clears bit k in **`pending`** when a signal of type k is received
- **`blocked`**: represents the set of blocked signals
  - Can be set and cleared by using the **`sigprocmask`** function

Sending Signals with /bin/kill Program

/bin/kill program sends arbitrary signal to a process or process group

Examples

/bin/kill -9 24818

Send SIGKILL to process 24818

/bin/kill -9 -24817

Send SIGKILL to every process in process group 24817

```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> /bin/kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

Example of `ctrl-c` and `ctrl-z`

```
bluefish> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
bluefish> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28107 pts/8    T      0:01 ./forks 17
28108 pts/8    T      0:01 ./forks 17
28109 pts/8    R+     0:00 ps w
bluefish> fg
./forks 17
<types ctrl-c>
bluefish> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28110 pts/8    R+     0:00 ps w
```

STAT (process state) Legend:

First letter:
S: sleeping
T: stopped
R: running

Second letter:
s: session leader
+: foreground proc group

See “man ps” for more details

Receiving Signals

Suppose kernel is returning from an exception handler and is ready to pass control to process *p*

Kernel computes `pnb = pending & ~blocked`

The set of pending nonblocked signals for process *p*

If (`pnb == 0`)

Pass control to next instruction in the logical flow for *p*

Else

Choose least nonzero bit *k* in **`pnb`** and force process *p* to ***receive*** signal *k*

The receipt of the signal triggers some ***action*** by *p*

Repeat for all nonzero *k* in **`pnb`**

Pass control to next instruction in logical flow for *p*

Default Actions

Each signal type has a predefined *default action*, which is one of:

- The process terminates
- The process terminates and dumps core
- The process stops until restarted by a SIGCONT signal
- The process ignores the signal

Installing Signal Handlers

The `signal` function modifies the default action associated with the receipt of signal `signum`:

```
handler_t *signal(int signum, handler_t *handler)
```

Different values for `handler`:

SIG_IGN: ignore signals of type **`signum`**

SIG_DFL: revert to the default action on receipt of signals of type **`signum`**

Otherwise, **`handler`** is the address of a ***signal handler***

- Called when process receives signal of type **`signum`**
- Referred to as ***"installing"*** the handler
- Executing handler is called ***"catching"*** or ***"handling"*** the signal
- When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

Signal Handling Example

```
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork13() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()
            while(1); /* chi
        }
    for (i = 0; i < N; i++)
        printf("Killing proc
        kill(pid[i], SIGINT)
    }
    for (i = 0; i < N; i++)
        pid_t wpid = wait(&c
        if (WIFEXITED(child_
            printf("Child %d
                   wpid, WEX
        else
            printf("Child %d
    }
}
```

```
linux> ./forks 13
Killing process 25417
Killing process 25418
Killing process 25419
Killing process 25420
Killing process 25421
Process 25417 received signal 2
Process 25418 received signal 2
Process 25420 received signal 2
Process 25421 received signal 2
Process 25419 received signal 2
Child 25417 terminated with exit status 0
Child 25418 terminated with exit status 0
Child 25420 terminated with exit status 0
Child 25419 terminated with exit status 0
Child 25421 terminated with exit status 0
linux>
```

UNIVERSITY OF COPENHAGEN

ITU CPH

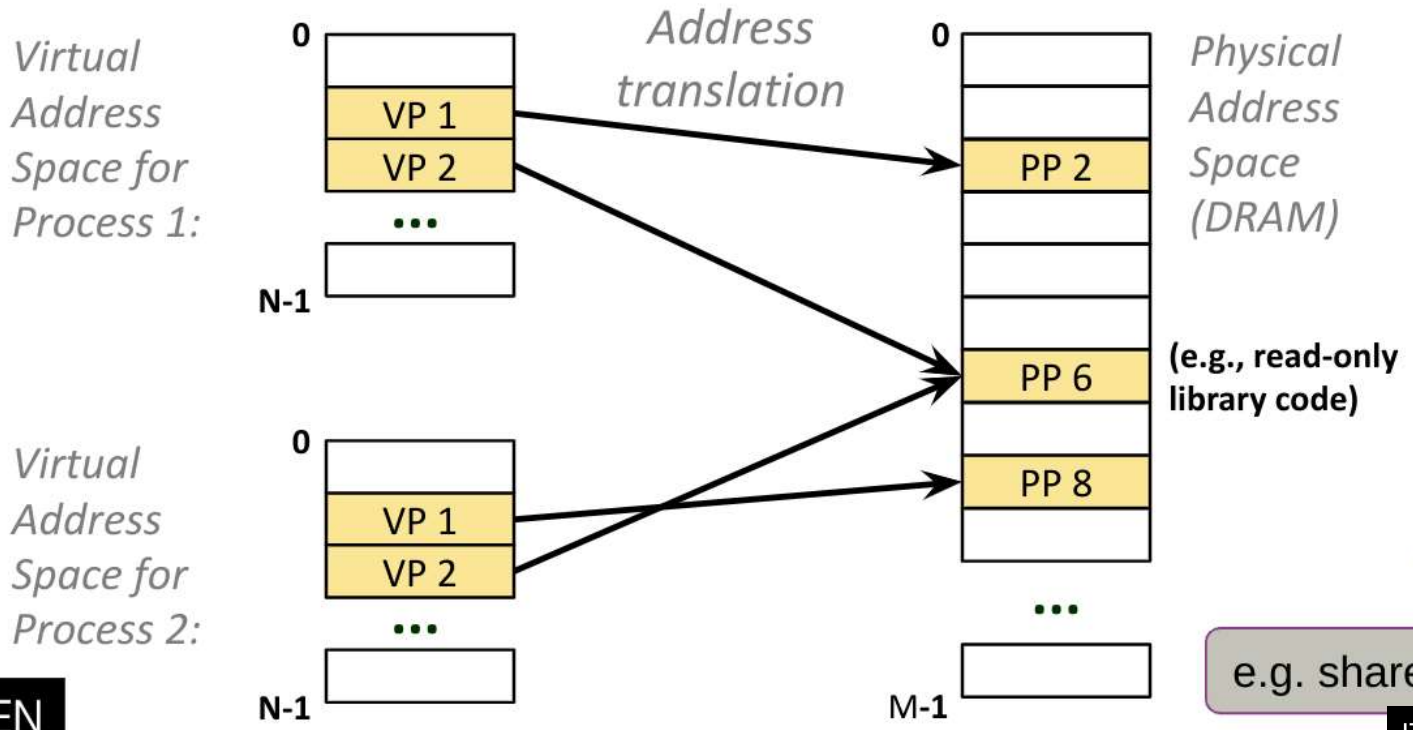
VM as a Tool for Memory Management

Memory allocation

- Each virtual page can be mapped to any physical page
- A virtual page can be stored in different physical pages at different times

Sharing code and data among processes

- Map virtual pages to the same physical page (here: PP 6)



locality:
you want, as much as possible, to have contiguous VM pages mapped to contiguous PM pages. (but not too much; nefficient cache use)

e.g. shared libra

UNIVERSITY OF COPENHAGEN

ITU CPH

Shared Memory in Linux

10s

https://stackoverflow.com/questions/5656530/how-to-use-shared-memory-with-linux-in-c

Shared Memory Segment

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
        fprintf(stderr, "usage: shmdemo [data_to_write]\n");
        exit(1);
    }

    /* make the key: */
    if ((key = ftok("hello.txt", 'R')) == -1) /*Here the file must exist */
{
        perror("ftok");
        exit(1);
    }

    /*  create the segment: */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
        perror("shmget");
        exit(1);
    }
```

Memory Mapped Segment/File

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
  // Our memory buffer will be readable and writable:
  int protection = PROT_READ | PROT_WRITE;

  // The buffer will be shared (meaning other processes can access it), but
  // anonymous (meaning third-party processes cannot obtain an address for it),
  // so only this process and its children will be able to use it:
  int visibility = MAP_SHARED | MAP_ANONYMOUS;

  // The remaining parameters to `mmap()` are not important for this use case,
  // but the manpage for `mmap` explains their purpose.
  return mmap(NULL, size, protection, visibility, -1, 0);
}
```

Linux default: no sharing between processes. if you want sharing (shared memory), then you have to work for it.
there are primitives in the C std lib for this.

UNI

old UNIX way of sharing between processes.

fork; child shares mem w/ parent

like VP2/PP6 before

ITU CPH

Inter-Process Communication, More

We will see more IPC in the next lecture

- File I/O
- Socket I/O

ITU CPH

dback Queue (MLFQ). Hopefully you can now see why it is called t: it has *multiple levels* of queues, and uses *feedback* to determine the ority of a given job. History is its guide: pay attention to how jobs ave over time and treat them accordingly.

The refined set of MLFQ rules, spread throughout the chapter, are reduced here for your viewing pleasure:

- **Rule 1:** If Priority(A) $>$ Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) $=$ Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

ITU CPH

Take-Aways

A *vectored event* is a transfer of control to the OS in response to some *event (asynchronous vs. synchronous (trap, fault, abort)).*

Process has own address space and thread of control. Libs defines primitives for spawning/terminating processes, waiting for processes and executing programs within a process. The kernel takes care of context switching.

Hardware interrupts first handled in hardware then in software (kernel) through interrupt vector. Signals as process-level interrupt handling.