# CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

gayathri@iitdh.ac.in

***Materials in these slides have been borrowed from textbooks and existing operating systems courses***

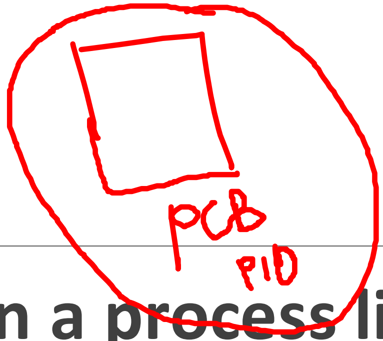
# 13th Jan 2021, Mechanism of Process Execution

How does the OS run a process?

How does it handle a system call?

How does it context switch from one process to the other?

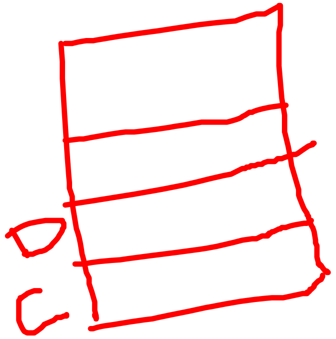
# Process Execution




Z

main()

- OS creates a process entry for the process in a process list
- **Allocates memory and creates memory image**
  - –Code and data (from executable)
  - –Stack and heap




- **Points CPU program counter to current instruction**
  - –Other registers may store operands, return values etc.
- **After setup, OS is out of the way and process executes directly on CPU**

# A simple function call

User


- A function call translates to a jump instruction
- A new stack frame pushed to stack and stack pointer (SP)updated
- Old value of PC (return value) pushed to stack and PC updated
- Stack frame contains return value, function arguments etc.

# How is a system call different?

CPU hardware has multiple privilege levels

–One to run user code: user mode

–One to run OS code like system calls: kernel mode

–Some instructions execute only in kernel mode

•Kernel does not trust user stack

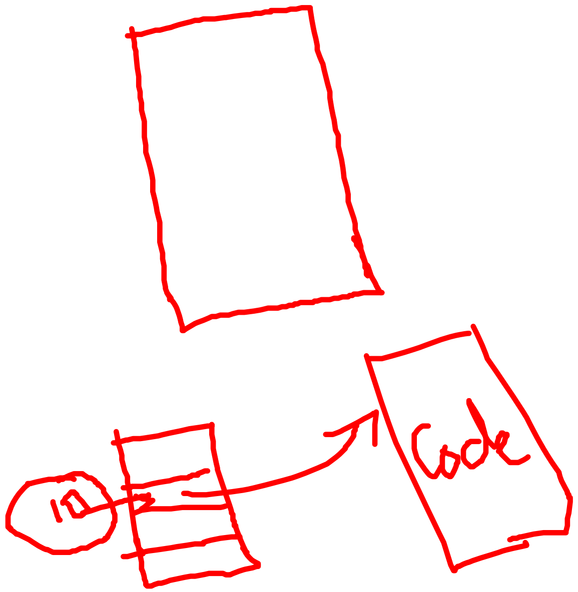
–Uses a separate kernel stack when in kernel mode

•Kernel does not trust user provided addresses to jump to

–Kernel sets up Interrupt Descriptor Table (IDT) at boot time– IDT has addresses of kernel functions to run for system calls and other events

open()

close()



# Mechanism of system call: trap instruction

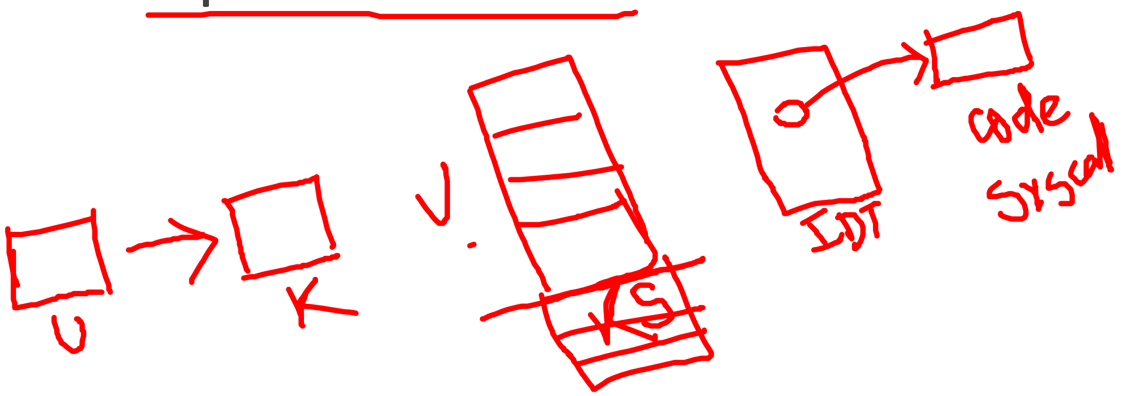
trap

ISA

x86
x86-64

When system call must be made, a special trap instruction is run(usually hidden from user by libc)

- Trap instruction execution
  - Move CPU to higher privilege level
  - Switch to kernel stack
  - Save context (old PC, registers) on kernel stack
  - Look up address in IDT and jump to trap handler function in OS code



# More on trap instruction

Trap instruction is executed on hardware in following cases:

–System call (program needs OS service)

–Program fault (program does something illegal, e.g., access memory it doesn’t have access to)

–Interrupt (external device needs attention of OS, e.g., a network packet has arrived on network card)

- Across all cases, the mechanism is: save context on kernel stack and switch to OS address in IDT
- IDT has many entries: which to use?

–System calls/interrupts store a number in a CPU register before calling trap, to identify which IDT entry to use

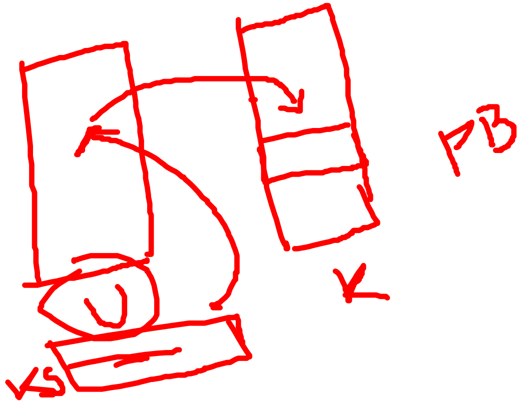
# Return from trap

When OS is done handling syscall or interrupt, it calls a special instruction return-from-trap

–Restore context of CPU registers from kernel stack

–Change CPU privilege from kernel mode to user mode

–Restore PC and jump to user code after trap




- User process unaware that it was suspended,resumes execution as always
- Must you always return to the same user processfrom kernel mode? No
- Before returning to user mode, OS checks if itmust switch to another process

KM

| OS @ boot (kernel mode) | Hardware |
|---|---|
| initialize trap table | |
| | remember address of... syscall handler |

| OS @ run (kernel mode) | Hardware | Program (user mode) |
|---|---|---|
| Create entry for process list<br>Allocate memory for program<br>Load program into memory<br>Setup user stack with argv<br>Fill kernel stack with reg/PC<br>**return-from-trap** | | |
| | restore regs<br>(from kernel stack)<br>move to user mode<br>jump to main main() | |
| | | Run main()<br>...<br>Call system call Sys<br>**trap** into OS<br>Trap |
| | save regs<br>(to kernel stack)<br>move to kernel mode<br>jump to trap handler | |
| Handle trap<br>Do work of syscall<br>**return-from-trap** | | |
| | restore regs<br>(from kernel stack)<br>move to user mode<br>jump to PC after trap | |
| | | ...<br>return from main<br>**trap** (via `exit()`) |
| Free memory of process<br>Remove from process list | | |

# Why switch between processes?

- Sometimes when OS is in kernel mode, it cannot return back to the same process it left

–Process has exited or must be terminated (e.g.,segfault)

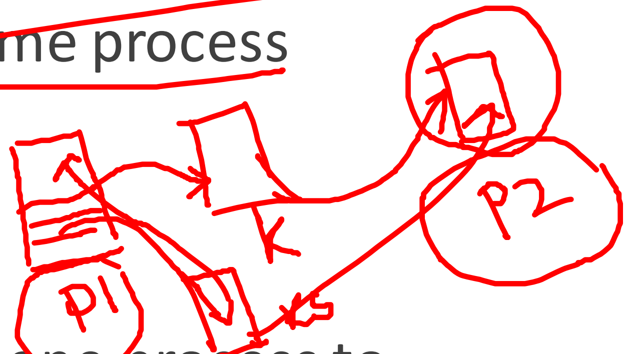
–Process has made a blocking system call

I/o Synchronous

- Sometimes, the OS does not want to return back to the same process

–The process has run for too long

–Must timeshare CPU with other processes




- In such cases, OS performs a context switch toswitch from one process to another

# The OS scheduler

- OS scheduler has two parts
  - –Policy to pick which process to run (next lecture)
  - –Mechanism to switch to that process (this lecture)
- Non preemptive (cooperative) schedulers are polite
  - –Switch only if process blocked or terminated

infinite loop while(1){
}

- Preemptive (non-cooperative) schedulers canswitch even when process is ready to continue
  - –CPU generates periodic timer interrupt X ms
  - –After servicing interrupt, OS checks if the currentprocess has run for too long

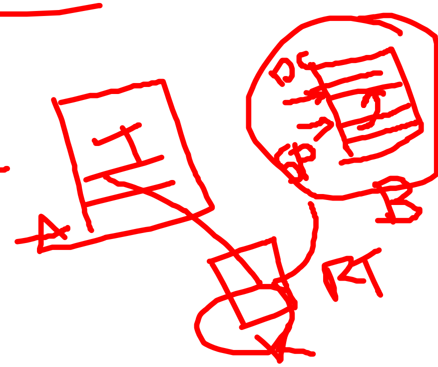
A B

# Context Switch Mechanism

**Example: process A has moved from user to kernel mode, OS decides it must switch from A to B**

Timer

- Save context (PC, registers, kernel stack pointer) of A on kernel stack
- Switch SP to kernel stack of B
- Restore context from B's kernel stack
- Who has saved registers on B's kernel stack? OS did, when it switched out B in the past
- Now, CPU is running B in kernel mode, return-from-trap to switch to user mode of B



# A subtlety on saving context

- Context (PC and other CPU registers) saved on the kernel stack in two different scenarios
- When going from user mode to kernel mode, user context (e.g., which instruction of user codeyou stopped at) is saved on kernel stack by thetrap instruction

–Restored by return-from-trap

- During a context switch, kernel context (e.g.,where you stopped in the OS code) of process A is saved on the kernel stack of A by the contextswitching code

–Restores kernel context of process B

| **OS @ boot<br>(kernel mode)** | **Hardware** |
|---|---|
| **initialize trap table** | |
| | remember addresses of...<br>syscall handler<br>timer handler |
| **start interrupt timer** | |
| | start timer<br>interrupt CPU in X ms |

| **OS @ run<br>(kernel mode)** | **Hardware** | **Program<br>(user mode)** |
|---|---|---|
| | | Process A<br>... |
| | **timer interrupt**<br>save regs(A) → k-stack(A)<br>move to kernel mode<br>jump to trap handler | HW |
| OS<br>Handle the trap<br>Call `switch()` routine<br>save regs(A) → proc_t(A)<br>restore regs(B) ← proc_t(B)<br>switch to k-stack(B)<br>**return-from-trap (into B)** | | |
| | restore regs(B) ← k-stack(B)<br>move to user mode<br>jump to B's PC | |
| | | Process B<br>... |

# Trivia

What is the kernel stack of a process used for?

When a process goes from user mode to kernel mode due to a trap occurring (system call / interrupt / program fault), who saves the context of the process? What exactly happens?

What happens on a trap?

Who saves user's CPU context when moving from user mode to kernel mode?

Are different types of contexts saved on the kernel stack?

```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
  # Save old registers
  movl 4(%esp), %eax  # put old ptr into eax
  popl 0(%eax)        # save the old IP
  movl %esp, 4(%eax)  # and stack
  movl %ebx, 8(%eax)  # and other registers
  movl %ecx, 12(%eax)
  movl %edx, 16(%eax)
  movl %esi, 20(%eax)
  movl %edi, 24(%eax)
  movl %ebp, 28(%eax)

  # Load new registers
  movl 4(%esp), %eax  # put new ptr into eax
  movl 28(%eax), %ebp # restore other registers
  movl 24(%eax), %edi
  movl 20(%eax), %esi
  movl 16(%eax), %edx
  movl 12(%eax), %ecx
  movl 8(%eax), %ebx
  movl 4(%eax), %esp  # stack is switched here
  pushl 0(%eax)       # return addr put in place
  ret                 # finally return into new ctxt
```

We thus have the basic mechanisms for virtualizing the CPU in place.

But a major question is left unanswered: which process should we run at a given time?

It is this question that the scheduler must answer, and thus the next topic of our study.