# Lecture 6

Jan 15, 2024

# Process Execution

- **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

- 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

# Mechanism of system call: trap instruction

- 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 process from kernel mode? No
- Before returning to user mode, OS checks if it must switch to another process

# 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
- 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 to switch from one process to another

# Mechanism to implement Process API

- A Process lifecycle – Who does What ?
- Good to think terms of three layers
  - OS
  - Hardware
  - Process

| Operating System | Hardware | User –Space Process |
| --- | --- | --- |
| | BOOT UP | |
| Setup trap table<br><br>Create 1st Process | Save Syscall address | |
| | | Create Process |
| Assign pid to Proc<br>Allocate Memory<br>Load Program<br>Setup stack with argv<br>Setup Kernel Stack<br>Return from trap | Restore regs from kernel stack<br>Switch to user mode<br>Jump to PC ( main) | Run main() in user space |

| Operating System | Hardware | User –Space Process |
| --- | --- | --- |
| | | Running Program<br>Call syscall (int 0x80) |
| | Save regs to kernel stack, switch to kernel mode, jump to trap handler | |
| Handle trap, run syscall<br>Update kernel stack, user stack<br>Return from trap | | |
| | Restore regs from kernel stack, switch to user mode, jump to PC after trap | |
| | | Continue execution<br>exit() - trap |
| Free Memory, delete pid, manage child pids | | |

# The OS scheduler

- OS scheduler has two parts
  - –Policy to pick which process to run
  - –Mechanism to switch to that process
- Non preemptive (cooperative) schedulers are polite
  - –Switch only if process blocked or terminated
- Preemptive (non-cooperative) schedulers can switch even when process is ready to continue
  - –CPU generates periodic timer interrupt
  - –After servicing interrupt, OS checks if the current process has run for too long

# Context Switch Mechanism

- **Example: process A has moved from user to kernel mode, OS decides it must switch from A to B**
- 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 code you stopped at) is saved on kernel stack by the trap 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 context switching code
  - –Restores kernel context of process B

| Operating System | Hardware | User –Space Process |
| --- | --- | --- |
| Setup trap table | Save Syscall, timer interrupt address | |
| Start Interrupt Timer | Start timer<br>Interrupt CPU in X ms | |
| | Timer Interrupt<br>Save regs (A) to kernel stack (A)<br>Switch to kernel mode<br>Jump to trap handler | Run Process A |
| Handle the trap<br>Decide process to run by scheduling policy<br>Call switch():<br>Save proc-state of A<br>Switch to kernel-stack of B<br>Return from trap | Restore regs from kernel stack (B)<br>Switch to user mode<br>Jump to PC of B | Run Process B |

```
# 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
```

# Homework -2

- 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?

# Scheduling is a vital OS role

- **Many processes with differing roles run together**
- **Deciding which one runs next is a vital role of the OS**

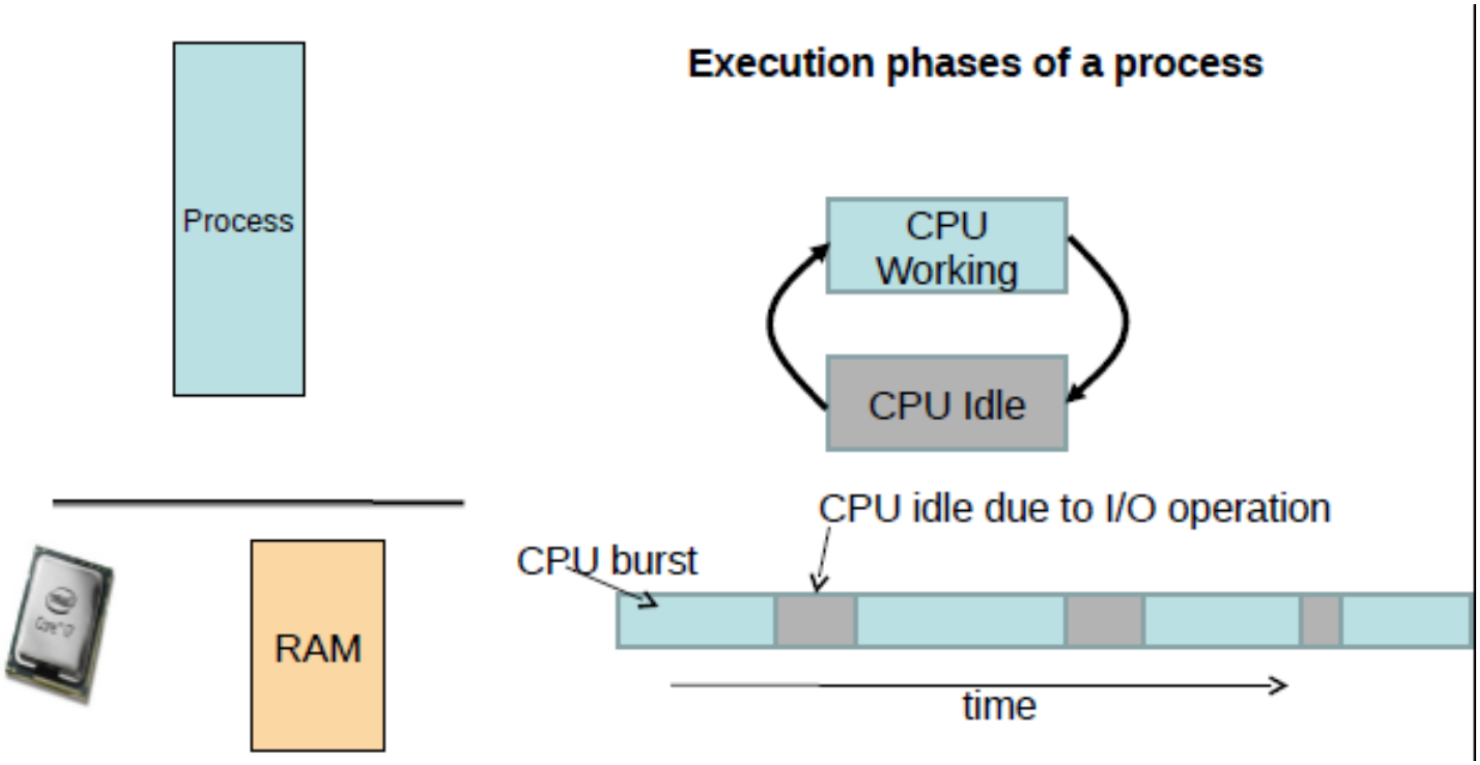
# What is a scheduling policy?

- On context switch, which process to run next, from set of ready processes?
- OS scheduler schedules the CPU requests (bursts) of processes
  - CPU burst = the CPU time used by a process in a continuous stretch
  - If a process comes back after I/O wait, it counts as a fresh CPU burst

## Policy and Mechanism

- **Dispatcher**
  - Main responsibility is context switch
  - Low level hardware-software handling
  - Mechanism
- **Scheduler**
  - Main responsibility is deciding which process to run
  - High level algorithmic view
  - Policy

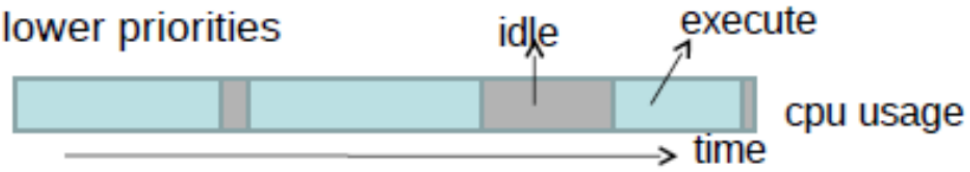
# Execution Phases



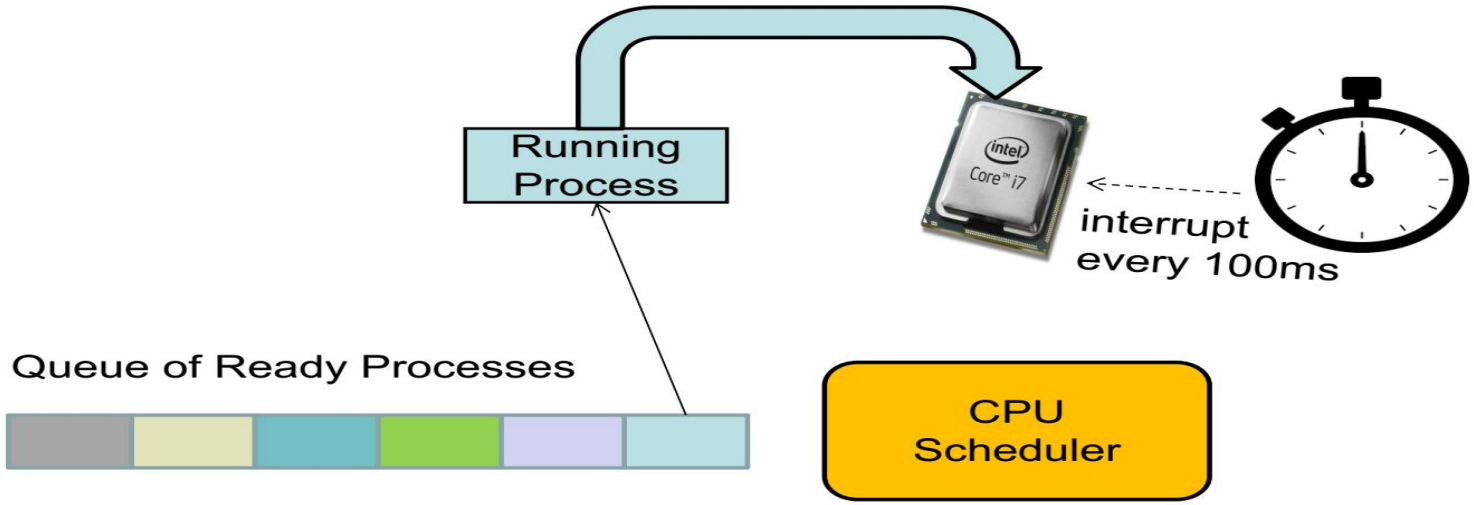
# Types of Processes

- I/O bound
  - Has small bursts of CPU activity and then waits for I/O
  - eg. Word processor
  - Affects user interaction (we want these processes to have highest priority)


- CPU bound
  - Hardly any I/O, mostly CPU activity (eg. gcc, scientific modeling, 3D rendering, etc)
    - Useful to have long CPU bursts
  - Could do with lower priorities



# CPU Scheduler




Scheduler triggered to run when timer interrupt occurs or when running process is blocked on I/O or exits

Scheduler picks another process from the ready queue

Performs a context switch

# Schedulers

- Decides which process should run next.
- Objectives:
  - Minimize waiting time
    - Process should not wait long in the ready queue
  - Maximize CPU utilization
    - CPU should not be idle
  - Maximize throughput
    - Complete as many processes as possible per unit time
  - Minimize response time
    - CPU should respond immediately
  - Fairness
    - Give each process a fair share of CPU

**- Make gaming the scheduler hard**

**- Keep overheads of scheduling small**

# What are we trying to optimize?

- Maximize (utilization = fraction of time CPU is used)
- Minimize average (turnaround time = time from process arrival to completion)
- Minimize average (response time = time from process arrival to first scheduling)
- Fairness: all processes must be treated equally
- Minimize overhead: run process long enough to amortize cost of context switch (~1 microsecond)

# When studying scheduling...

- **Careful abstractions**

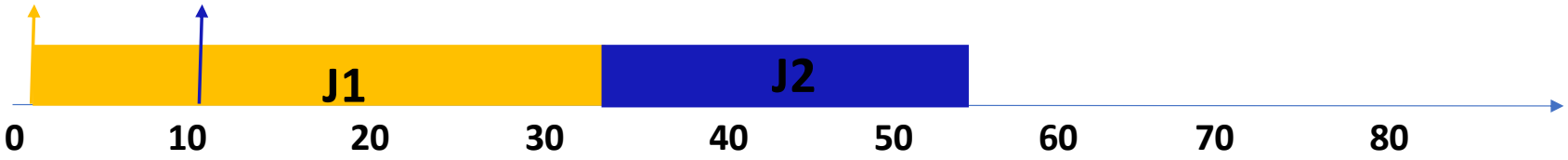
- **Workload** – Set of abstracted job descriptions
- **Scheduler** - The policy/logic that decides what runs
- **Metric** – Measurements/ analysis on scheduling quality

# When studying scheduling...

- **Theoretical analysis**
- **Workload (W), Scheduler (S), Metric (M)**
- **f(W, S) = M**
- **Scheduling problems are of different flavours:**
- **Given any two of these quantities, compute the third**

# Abstraction setup

- A job has an **arrival time (a)** and an **execution time/burst time (c)**




J1 : $a_1 = 0$ ; $c_1 = 30$

J2 : $a_2 = 10$; $c_2 = 20$

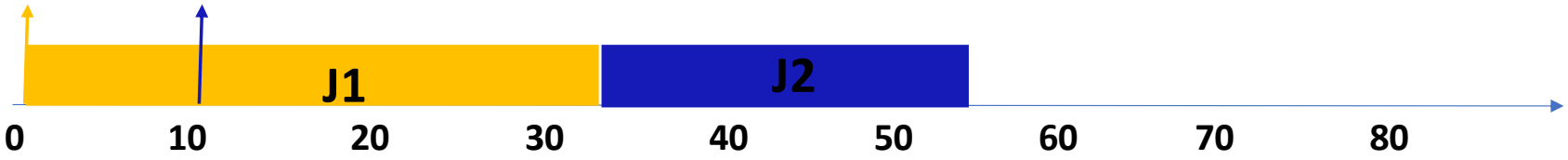
Turnaround time (t) = Completion time – arrival time

$t_1 = 30 - 0 = 30$

$t_2 = 50 - 10 = 40$

# Abstraction setup

- A job has an **arrival time (a)** and an **execution time/burst time (c)**




J1 : $a_1 = 0$ ; $c_1 = 30$

J2 : $a_2 = 10$; $c_2 = 20$

Response time (r) = first execution – arrival time

$t_1 = 0 - 0 = 0$

$t_2 = 30 - 10 = 20$