operating systems

UNIT - 1

INTRODUCTION & PROCESS MANAGEMENT

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

OPERATING SYSTEMS

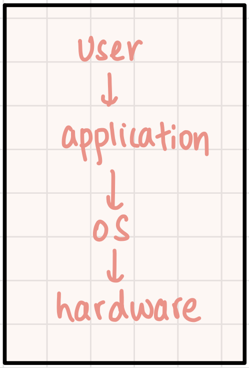
INTRODUCTION

NEED for OS

- Access hardware interfaces
- Manage multiple processes
- Storage management, files
- Protection and Security

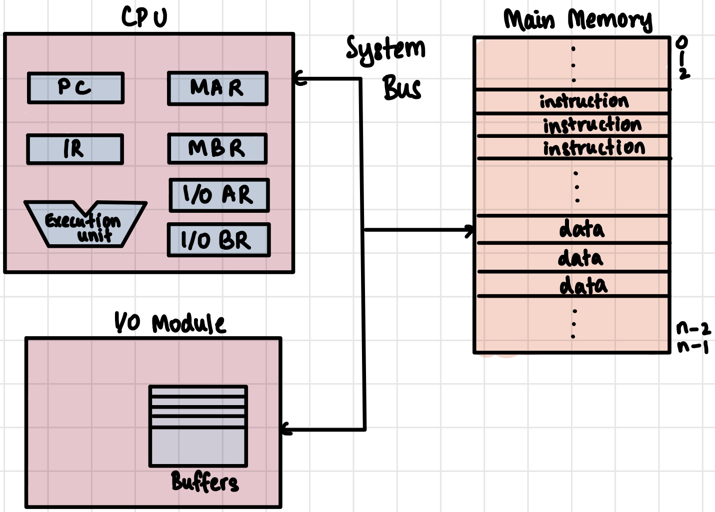
Definition

- Intermediary between user and hardware
- User-friendly
- resource allocator, control program
- Compromise b/w usability and utilisation



Computer Components - Top Level View

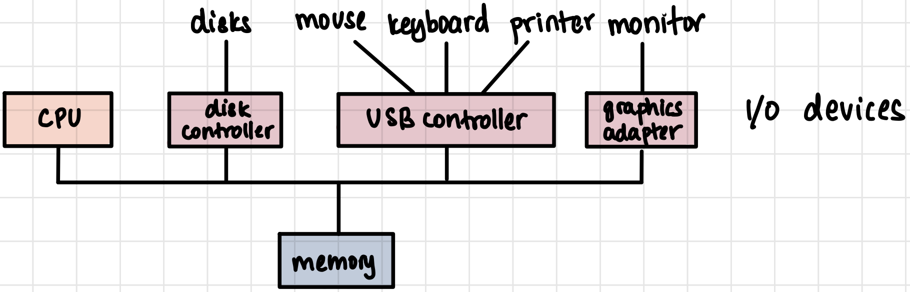
- Processor
- Memory
- I/O modules
- System Bus



PC: prog counter
IR: instruction register
MAR: memory address register
MBR: memory buffer register
I/O AR: input/output address register
I/O BR: input/output buffer register

COMPUTER SYSTEM ORGANISATION

- CPU(s) and device controllers have access to shared memory via a common bus
- CPU(s) and data controllers compete for memory cycles and can run concurrently
- Memory controller is provided to synchronise access to memory



Computer System Operation

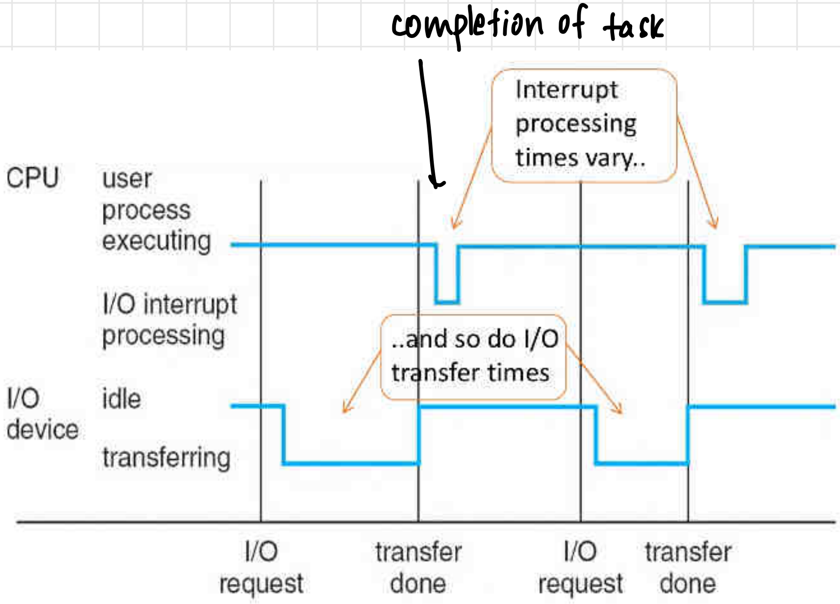
- I/O devices, CPU execute concurrently
- Each device controller in charge of particular device type and has local memory
- Device controller has registers for each action (keyboard input)
- CPU loads data from main memory to local buffer
- Device controller sends interrupt to CPU when task is completed

Bootstrap

- When system is booted, first program to be executed is Bootstrap, which is stored on ROM or EEPROM
- Also referred to as firmware
- Initialises all aspects of system (CPU registers, device controllers, main memory etc)
- Load OS kernel onto memory
- After booting, first program that is created is init; waits for event to occur
  ↘ interrupt

Interrupt

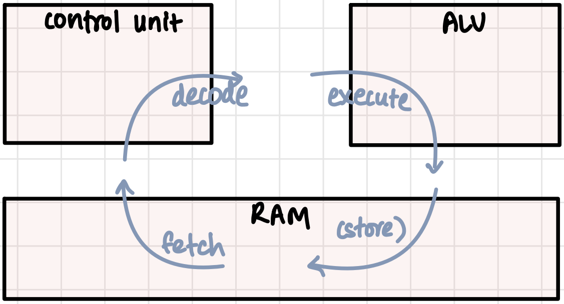
- Transfers control to the interrupt service routine (ISR) through the interrupt vector; return address needs to be saved
- Interrupt vector table contains addresses of all service routines
- OS is an interrupt-driven program
- State of CPU saved by OS by storing registers and PC onto stack
- Type of interrupt
    - polling for device (I/O)
    - vectored interrupt system (timer)
- Action to be taken for each interrupt determined by code segment



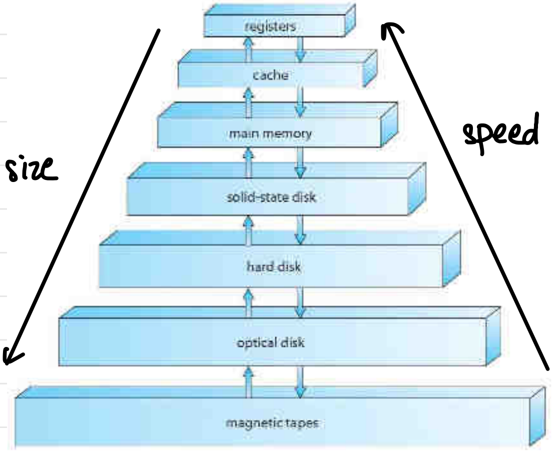
completion of task

storage structure

- hierarchy of memory
- RAM : volatile – main memory, directly accessible by CPU
  - implemented with semiconductor technology
  - DRAM
  - info: charge on capacitor
  - frequent charging required
- ROM, EEPROM → mobile phones: factory installed programs
- Von Neumann model: fetch, decode, execute cycles



- Secondary memory - non volatile
- Hard disk
  - disk surface: tracks, sectors
  - disk controller: interaction
- SSD - solid state disk
  - faster
  - flash memory
- Caching
  - level 1 and level 2
  - faster storage for frequent access
- Device driver
  - interface b/w controller and kernel



Caching

- information copied from slower to faster, temporarily

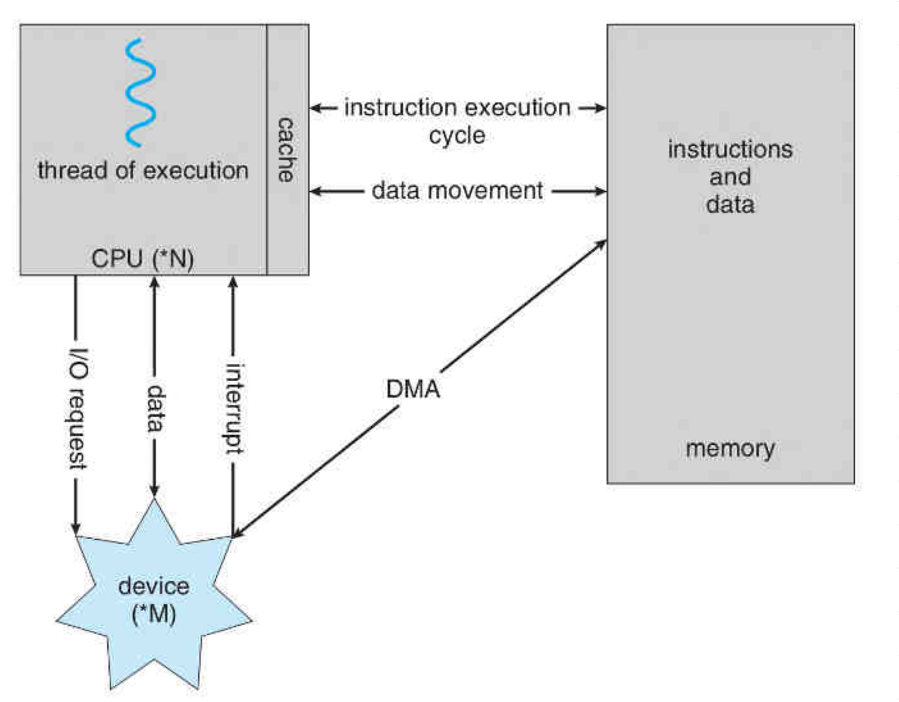
I/O Structure

- After I/O starts control returns to user program only after completion of I/O
  - CPU idles until next interrupt given (wait instruction)
  - No simultaneous I/O processing can occur
- After I/O starts, control returns to user program without waiting for I/O completion
  - System call: request to OS to allow user to wait for I/O completion
  - Device status table: entries for each of the I/O devices (type, address, state)

    input/output ↑
  - OS indexes into I/O device table (check if busy/idle, assign program if idle)

Direct Memory Access Structure

- used for high speed I/O devices (close to memory speeds)
- device controller transfers data from device directly to main memory without CPU intervention
- only one interrupt generated per block (instead of one interrupt per byte)

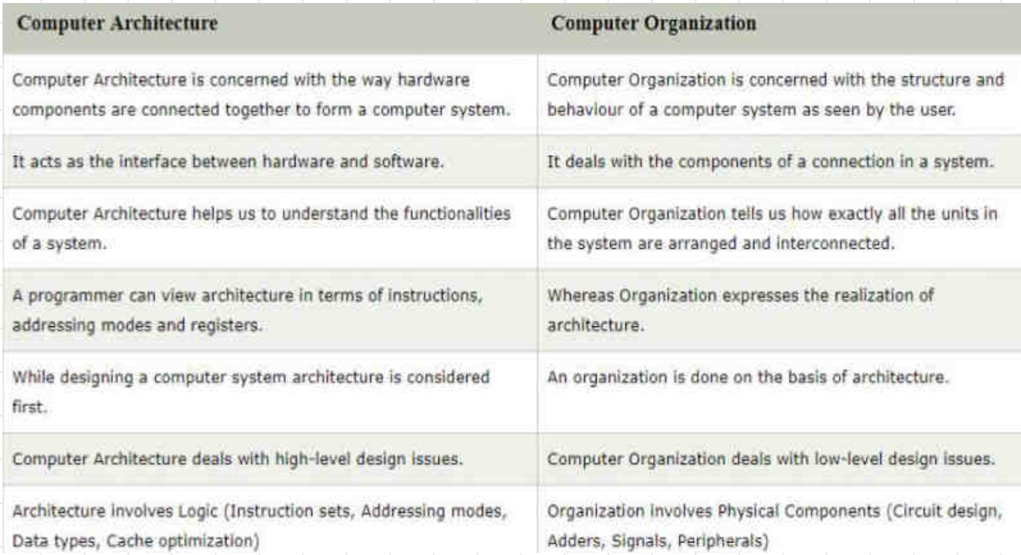


Computer Architecture

way hardware components are connected together to form computer system

Computer Organization

structure and behaviour of computer system as seen by the user



Computer Architecture	Computer Organization
Computer Architecture is concerned with the way hardware components are connected together to form a computer system.	Computer Organization is concerned with the structure and behaviour of a computer system as seen by the user.
It acts as the interface between hardware and software.	It deals with the components of a connection in a system.
Computer Architecture helps us to understand the functionalities of a system.	Computer Organization tells us how exactly all the units in the system are arranged and interconnected.
A programmer can view architecture in terms of instructions, addressing modes and registers.	Whereas Organization expresses the realization of architecture.
While designing a computer system architecture is considered first.	An organization is done on the basis of architecture.
Computer Architecture deals with high-level design issues.	Computer Organization deals with low-level design issues.
Architecture involves Logic (Instruction sets, Addressing modes, Data types, Cache optimization)	Organization involves Physical Components (Circuit design, Adders, Signals, Peripherals)

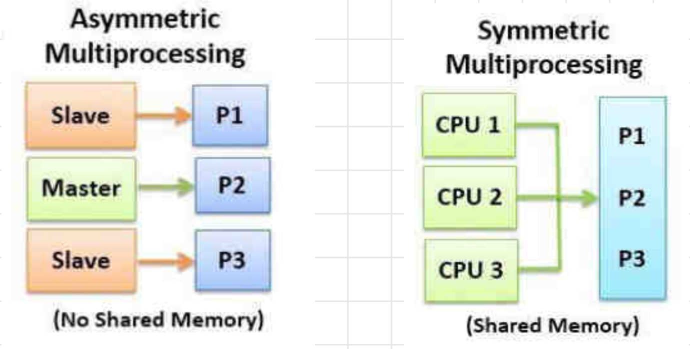
Computer System Architecture

- single general purpose processor
- special purpose processors: disk controller, keyboard (device specific), graphics controller — run limited no. of instructions
- managed by OS
- eg: disk controll microprocessor — receives sequence requests from CPU, implements queue and scheduling algorithm, relieves main CPU

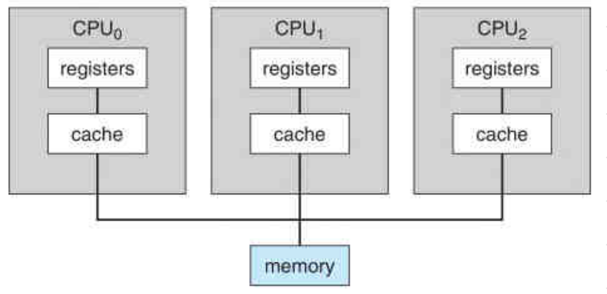
Multiprocessor System

- parallel systems, tightly-coupled systems (multiple processors)
- advantages
  - increased throughput
  - economy of scale cheaper than n single processor systems
  - increased reliability tolerant systems

1. Asymmetric Multiprocessing
   each processor assigned a specific task (boss-subordinate)
2. Symmetric Multiprocessing
   each processor performs all tasks

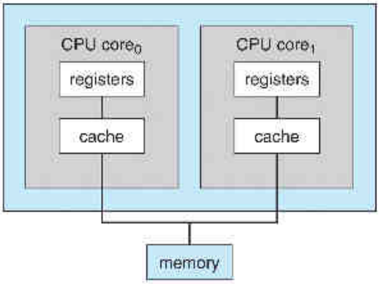


Symmetric Multiprocessor Architecture



DUAL CORE DESIGN

- Multi-chip and multicore
- Systems containing all chips; chassis containing multiple separate systems

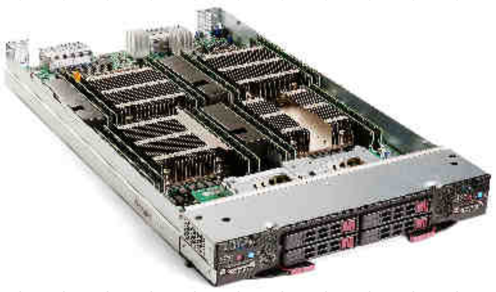


- Command (Linux) `cat /proc/cpuinfo | more`

```
vibhamasti@ubuntu:~$ cat /proc/cpuinfo | more
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-7920HQ CPU @ 3.10GHz
stepping        : 9
cpu MHz         : 3096.000
cache size      : 8192 KB
physical id     : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx
 fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon nopl xtopology tsc_reliable
nonstop_tsc cpuid pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_ti
--More--
```

Blade Servers

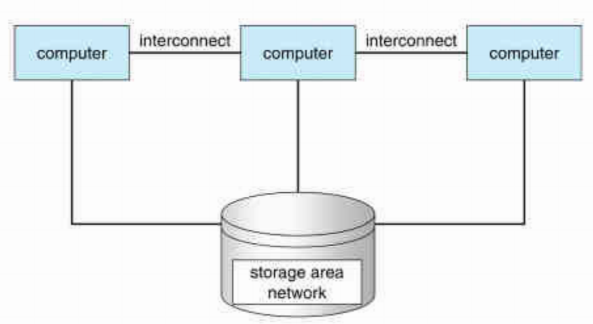
- multiple processor boards, I/O boards, networking boards on same chassis
- board boots independently and runs its own OS
- some blade-server boards are multiprocessor; multiple independent multiprocessor systems



Clustered Systems

- Multiples systems working together (over a network)
- Shares storage via storaage-area network (SAN)
- High-availability service which survives failures
  - Asymmetric clustering: one machine in hot standby mode → if failure, this machine takes over
  - Symmetric clustering: multiple nodes running apps monitoring each other
- Some clusters are for high performance computing (HPC)
  - apps must be written to use parallisation
- Some have distributed lock manager (DLM) to avoid conflicts over shared data

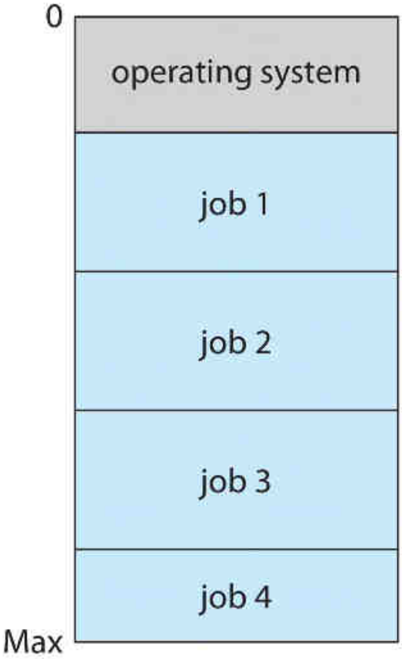
clustered system



SAN allows many systems to attach to a pool of storage

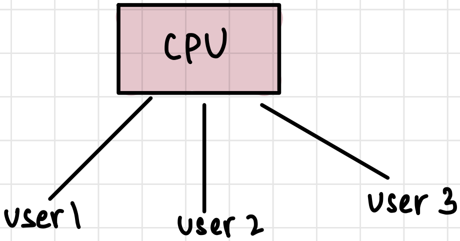
OS Structure - Multiprogramming

- Multiprogramming (batch system) needed for efficiency
- single user cannot keep CPU and I/O devices busy at all times
- Organises jobs (CPU and data) so that CPU always has one job to execute
- Subset of total jobs kept in memory
- One job selected and executed via job scheduling
- When it has to wait (for I/O, etc), OS switches to another job
- Reduce CPU idling



OS Structure - Multitasking

- Timesharing (multitasking)
- CPU switches jobs so frequently that users can interact with each job while it is running (interactive computing)
- Response time <1 second
- Each user has at least one program executing in memory
- CPU scheduling if several jobs to run at same time
- Swapping moves processes in and out of memory
- Virtual memory: execution of processes not completely in memory



Interrupt Driven

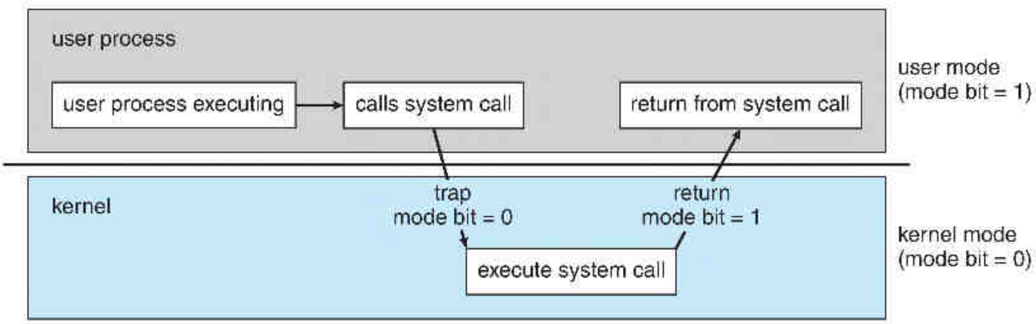
- Hardware interrupt by one of the devices
- Software interrupt (exception or trap)
  - software error (divide by 0)
  - request for OS service
  - infinite loops, processes modifying OS etc

Dual Mode and Multimode Operation

- User mode and kernel mode — dual mode

- Mode bit provided by hardware
  - distinguish
  - privileged: only kernel mode instructions
  - system call changes mode to kernel, return resets it

- multi-mode support by CPUs: VM Manager mode for guest VMs



Timer

- interrupt computer after specified period

- variable timer implemented by fixed rate clock and counter

- every clock tick, counter decrements

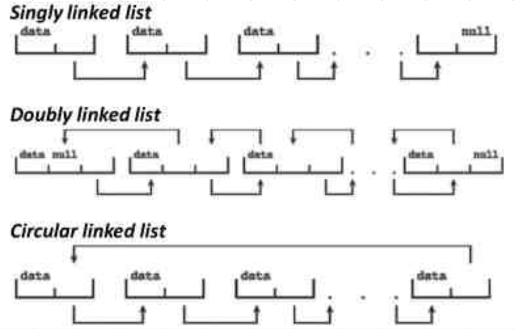
- interrupt occurs when counter reaches 0; prevents prog from running for too long

Kernal Data Structures

(a) Array

- each element can be accessed directly
- main memory
- multiple bytes → no. of bytes
- items with varying size?

(b) Linked List



- SLL
- DLL
- CLL

- advantages:
  - varying size
  - easy insertion / deletion

- disadvantages
  - retrieval : O(n) for size n
  - kernel algorithms
  - stacks and queues

(c) Stack

- LIFO
- OS: stack of function calls
- params, local vars, return address pushed onto stack
- return from function call pops items from stack

(d) Queue

- FIFO
- task scheduling CPU
- printer print jobs

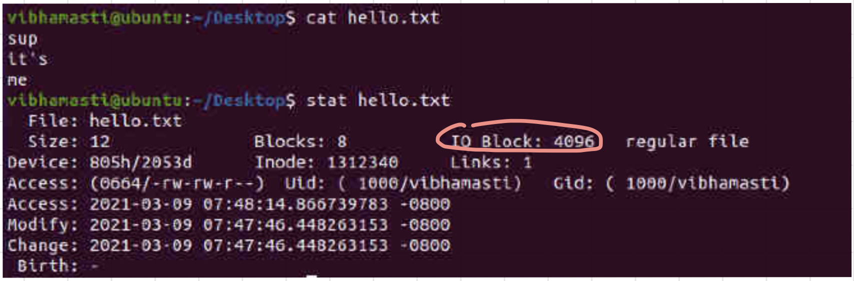
(e) Trees

- BST
    - search : O(n)

- Balanced BST
    - search: O(log n)
    - Linux for CPU scheduling — next task
    - red-black trees

- stat <filename> → I/O block

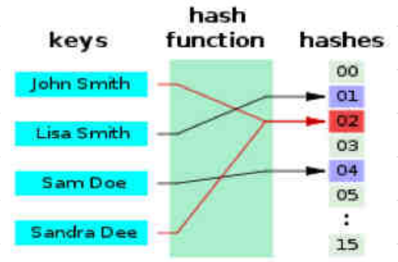
  Unix command

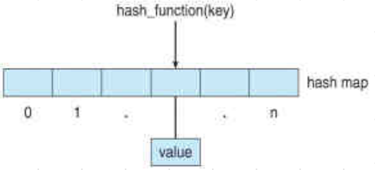


```
vibhamasti@ubuntu:~/Desktop$ cat hello.txt
sup
it's
me
vibhamasti@ubuntu:~/Desktop$ stat hello.txt
  File: hello.txt
  Size: 12              Blocks: 8          IO Block: 4096   regular file
Device: 805h/2053d      Inode: 1312340     Links: 1
Access: (0664/-rw-rw-r--)  Uid: ( 1000/vibhamasti)   Gid: ( 1000/vibhamasti)
Access: 2021-03-09 07:48:14.866739783 -0800
Modify: 2021-03-09 07:47:46.448263153 -0800
Change: 2021-03-09 07:47:46.448263153 -0800
 Birth: -
```

(f) Hash Functions and Maps

- implement hash map
- key: value pairs
- constant search time





(g) Bitmap

- string of n binary digits representing status of n items
- availability of each resource: 0 or 1
    - 0 : available
    - 1 : not available
- eg: bitmap 001011101

  0, 1, 3, 7 available

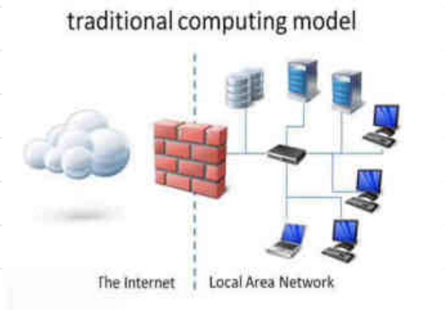
  2, 4, 5, 6 unavailable

Computing Environments

- where task is being performed

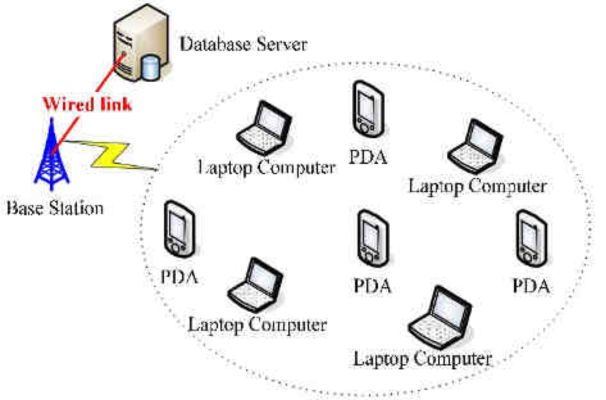
1. Traditional

- stand alone general purpose machine
- blurred — internet
- portals provide web access
- eg: company servers



2. Mobile

- handheld smartphones, tablets
- GPS, gyroscope
- AR
- IEEE 802.11 wireless, cellular data network

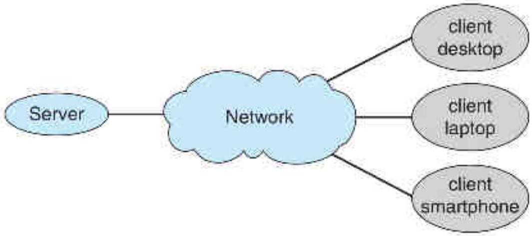


3. Distributed Computing

- collection of separate computers
- TCP/IP
    - LAN
    - WAN
    - MAN Metropolitan
    - PAN Personal - BT
- Network OS

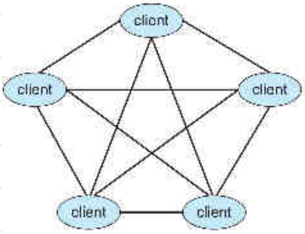
4. Client-Server

- servers respond to client requests
    - compute server system
    - file server system



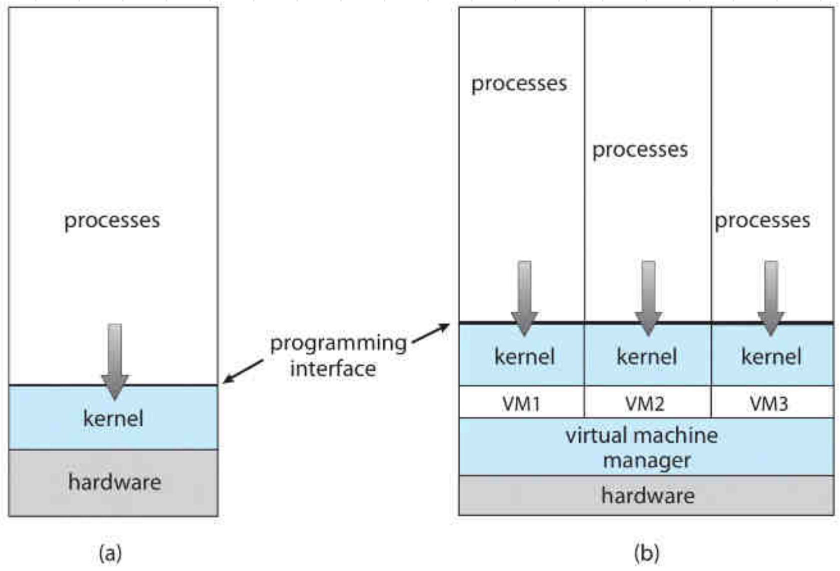
5. Peer-to-peer

- P2P : no client and servers
    - peer can act as client, server or both
    - nodes registered with central lookup table
    - discovery protocal: requests and responses
- Skype (VoIP), Napster, BitTorrent



6. Virtualisation

- host OS run guest OS as application
- emulation – source CPU diff from target CPU (eg: PowerPC to Intel x86 - Rosetta) not compiled to native code; interpretation
- virtualisation: OS natively compiled for CPU running guest OSes also natively compiled
- VMM – virtual machine manager
- JVM — byte code generated is not hardware-specific



(a) No virtual machine (b) Virtual machine

7. Cloud Computing

- computing, storage, apps as service across a network

- logical extension of virtualisation
  - Amazon Elastic Cloud (EC2) has 1000s of servers, millions of VMs

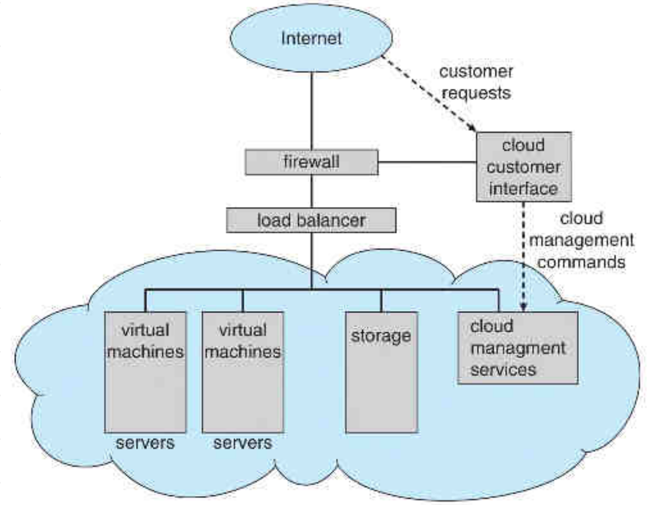
- Public cloud: via internet for anyone willing to pay

- Private cloud: run by a company for its own use

- Hybrid cloud: both public and private components

- Services
  - SaaS: Software as a Service (eg: word processor)
  - PaaS: Platform as a Service (eg: database server - software stack)
  - IaaS: Infrastructure as a Service (eg: storage available for backup)

- Cloud computing environments composed of traditional OSes, VMMs, cloud management tools
  - load balancers spread traffic across apps (servers)



8. Real-Time Embedded Systems

- most prevalent form of computers
  - real-time OS

- special computing environments
  - some OS, some no OS

- real-time OS : well-defined time constraints
  - soft real-time systems (do not hamper results with small delay)
  - hard real-time systems (hamper results with small delay)

services

- OS provides environment for execution of programs and services to progoms and users

- OS services helpful to user:

1) User Interface

CLI , GUI , Batch (eg: shell scripts in Linux)

2) Program Execution

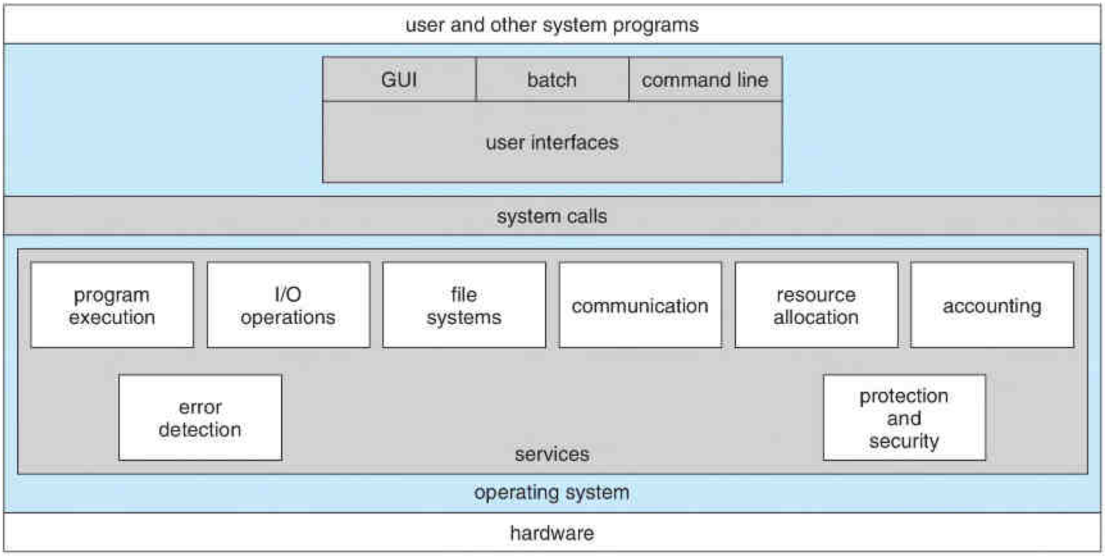
system loads program into memory and execute, terminate (normally or abnormally) → errors, exceptions, abort, interrupt

exit(1) → failure cases

3) I/O Operations

a running program may require I/O (file or device)

View of OS Services



OS Design and Implementation

- policy: what to do
  mechanism: how to do

- separation of policy from mechanism important, allows max flexibility

- creative task

- implementation of OSes:
  - earlier, assembly
  - then system programming langs- Algol, PL/I
  - now C, C++

- mix of languages
  - lowest levels in assembly
  - main body in C
  - system programs in C/C++, scripting languages like PERL, Python, shell scripts

- High level language easier to port to other hardware, but slower

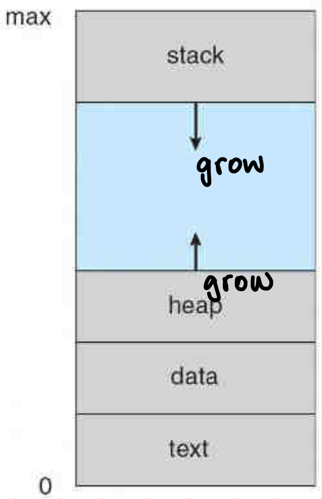
- Emulation: run OS on non-native hardware

Process Concepts

- OS executes various programs
  - batch system - jobs
  - time-shared systems - user programs or tasks

- process: program in execution, sequential

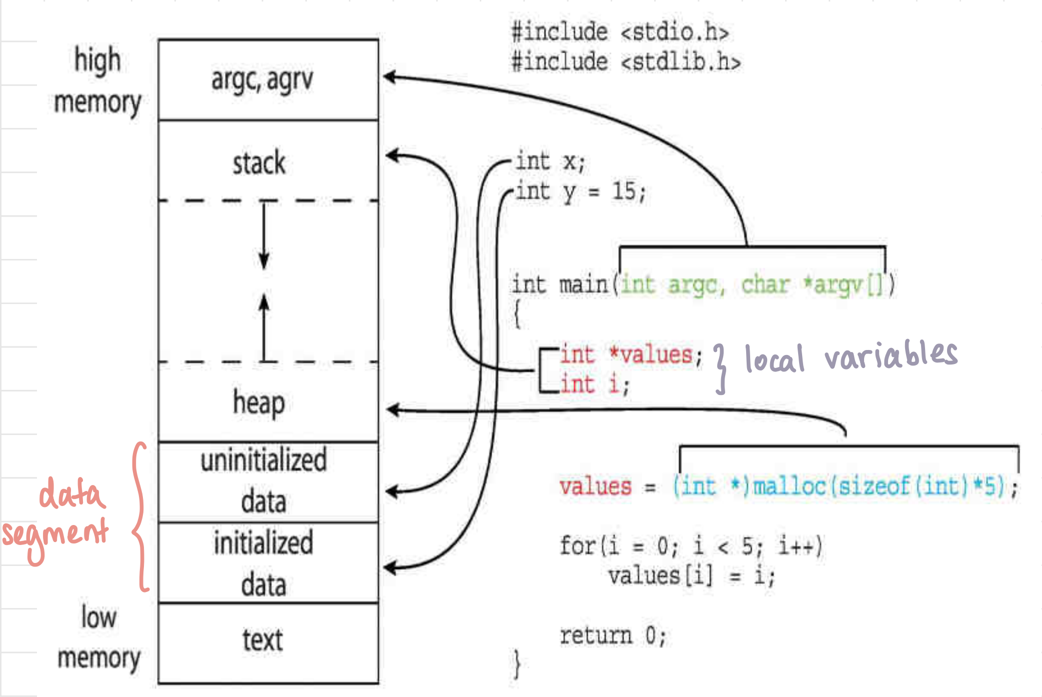
- segments/parts
  - code or text section
  - current activity - PC, processor registers
  - stack - function parameters, local variables, return address
  - data section - global variables
  - heap - dynamically allocated memory



- program: passive entity; file stored on disk
  process: active; when executable file loaded onto memory

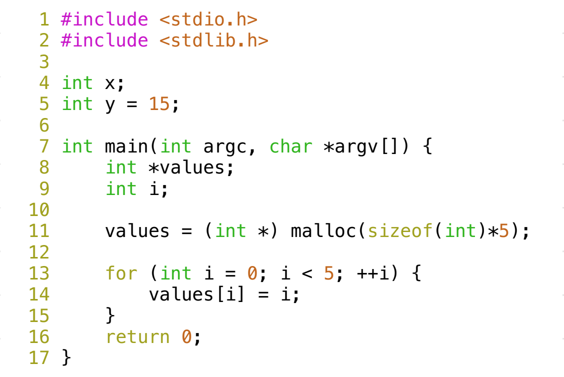
- execution of program via CLI, GUI etc

- one program can be several processes (multiple users executing same program)



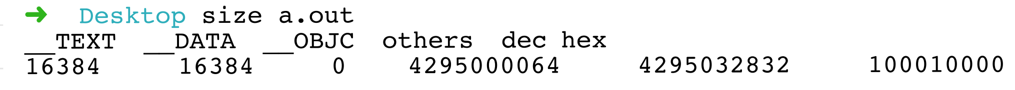
```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Size of a program



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[]) {
    int *values;
    int i;

    values = (int *) malloc(sizeof(int)*5);

    for (int i = 0; i < 5; ++i) {
        values[i] = i;
    }
    return 0;
}
```

in Linux, bss — static



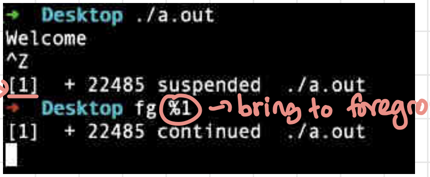
```
➜  Desktop size a.out
__TEXT  __DATA  __OBJC  others  dec hex
16384   16384   0   4295000064   4295032832   100010000
```

Process States

- New: process being created
- Running: instructions being executed
- Waiting: process waiting for event
- Ready: waiting to be assigned to a processor
- Terminated: process finished execution

- ctrl-z: shunted into background ; suspended
  ctrl-c : abort

Suspend into bg

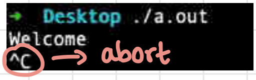


```
→ Desktop ./a.out
Welcome
^Z
[1]  + 22485 suspended  ./a.out
→ Desktop fg %1
[1]  + 22485 continued  ./a.out
```

process no.

→ bring to foreground

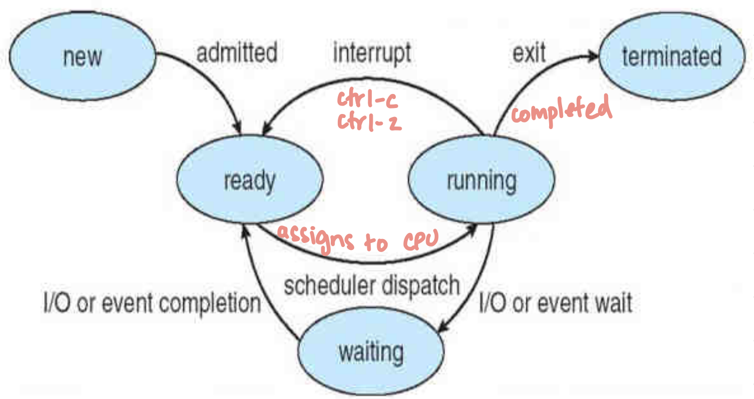
Abort



```
→ Desktop ./a.out
Welcome
^C
```

→ abort

Process State Diagram



Process Control Block (PCB)

Every process has a PCB; info associated with each process (task control block)

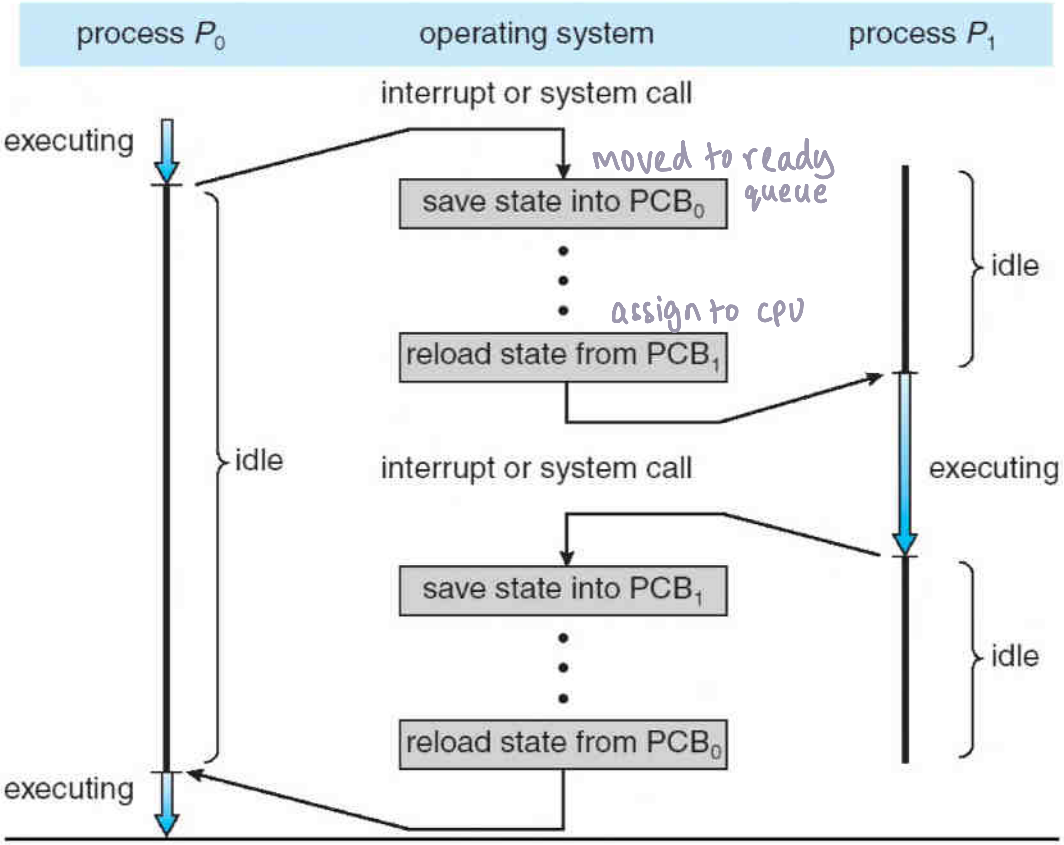
- Process state: running, waiting etc (PID)
- PC: next instruction
- CPU registers: contents of process-centric registers
- Memory management information: memory allocated to the process
- Accounting information: CPU used, clock time elapsed since start, time limits
- I/O status information: I/O devices allocated to process, list of open files

linux: ps -aux works

```
→  Desktop ps aux
USER               PID  %CPU %MEM      VSZ    RSS   TT  STAT STARTED      TIME COMMAND
root                71  84.1  0.4  4588492  68412   ??  Rs   23Dec20  43:43.13 /usr/sbin/systemstats --daemon
vibhamasti        1387   7.5  0.8  5678388 133680   ??  S    23Dec20  12:13.97 /System/Applications/Utilities/Terminal.app/Contents/MacOS/Terminal
_windowserver      140   7.1  0.8 17187596 130364   ??  Ss   23Dec20 767:43.82 /System/Library/PrivateFrameworks/SkyLight.framework/Resources/WindowServ
```

CPU Switch from process to process

- multiprogramming
- context switching



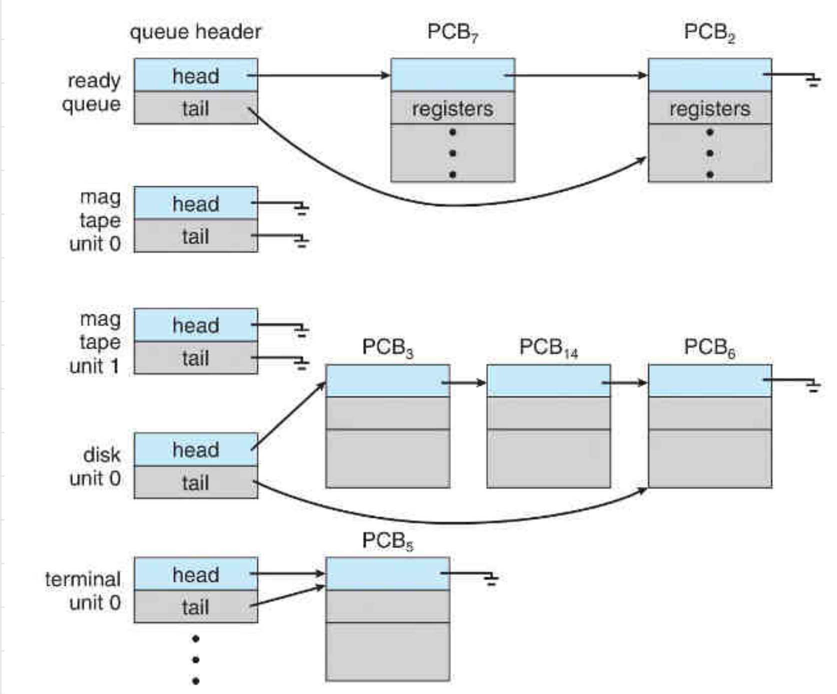
moved to ready queue

assign to cpu

Process Scheduling

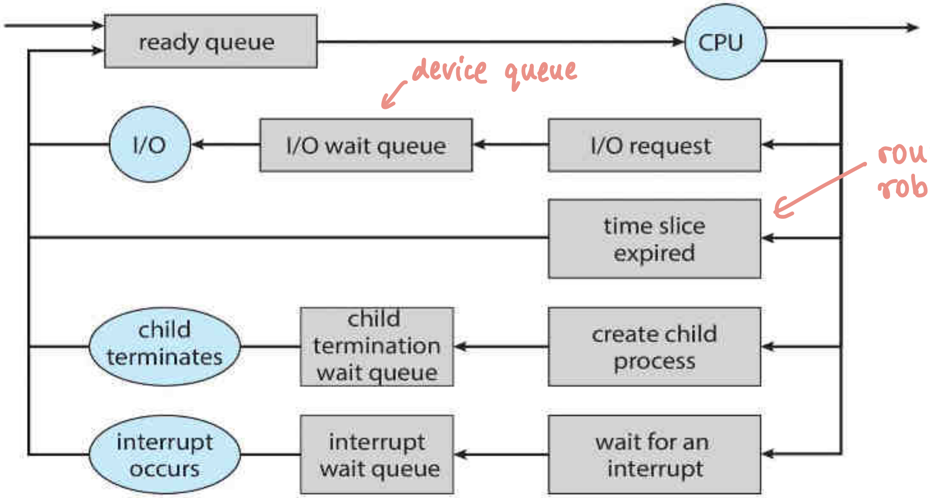
- Process scheduler selects among available processes
- maintains scheduling queues of processes (migrate among queues)
  - Job queue: set of all processes in system
  - Ready queue: set of processes in memory, ready and waiting to execute
  - Device queues: set of processes waiting for I/O devices

Ready Queue & Various I/O Device Queues



linked list of PCBs

Process Scheduling



device queue

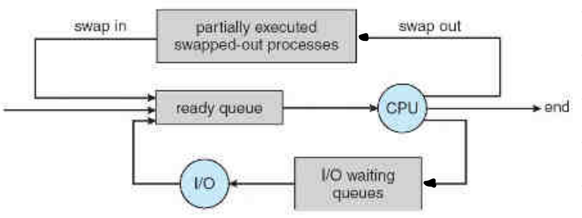
round robin

Schedulers

- Short-term schedulers (CPU schedulers): selects next process to be executed and assigns it to CPU (from ready queue)
  - invoked frequently (ms)
  - sometimes the only scheduler

- Long-term scheduler (job scheduler): selects which process should be brought to ready queue (from job queue) pg 28
  - invoked infrequently (s → min)
  - degree of multiprogramming

- Medium-term scheduler: if degree of multiprogramming needs to decrease
  - remove process from memory swap out
  - store on disk (backing store)
  - bring into memory from disk to continue swap in
  - swapping



- Processes
  I/O bound: more time on I/O, less on CPU
  CPU bound: more time on CPU (long, infrequent CPU bursts)

- Long-term scheduler: good process mix

Context Switching

- CPU switching between processes must save old process state and load saved state while switching back to it
- Context: represented in PCB
- More time spent on context switching, more time wasted ; context switching time is an overhead ; complex OS and PCB means longer context switch
- Time depends on hardware availability

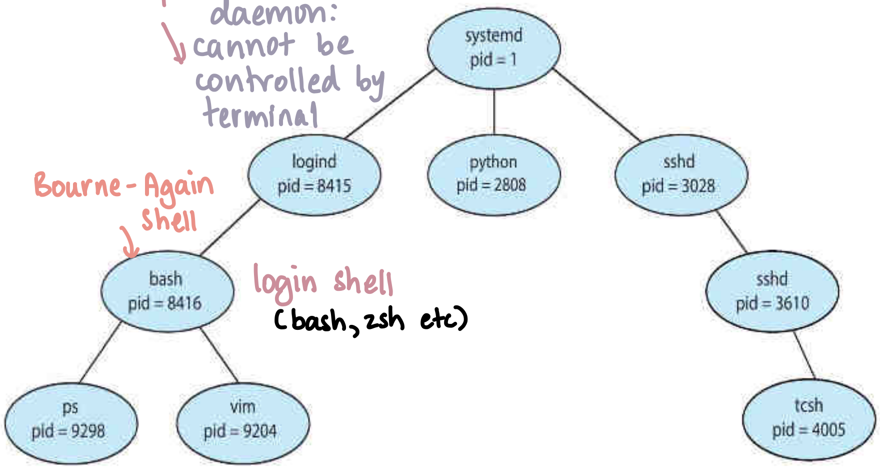
OPERATIONS ON PROCESSES

- creation — create process : windows, fork() : linux
- termination

creation

- parents create children (tree)
- PID : identifier
- resource sharing
  - parents & children share all
  - children share subset of parent's
  - no sharing
- execution
  - simultaneous
  - sequential

TREE OF PROCESSES IN LINUX



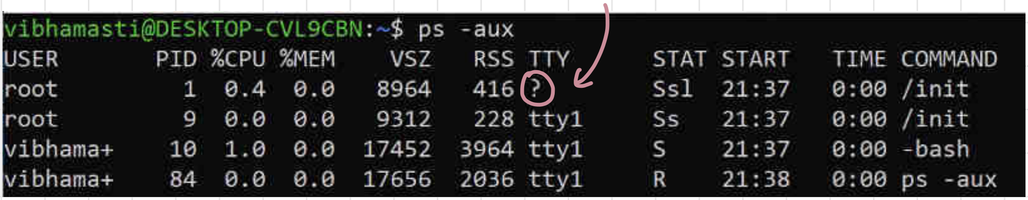
no interrupt

daemon: cannot be controlled by terminal

Bourne-Again shell

login shell
(bash, zsh etc)

daemon



```
vibhamasti@DESKTOP-CVL9CBN:~$ ps -aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.4  0.0   8964   416 ?        Ssl  21:37   0:00 /init
root         9  0.0  0.0   9312   228 tty1     Ss   21:37   0:00 /init
vibhama+    10  1.0  0.0  17452  3964 tty1     S    21:37   0:00 -bash
vibhama+    84  0.0  0.0  17656  2036 tty1     R    21:38   0:00 ps -aux
```

bash

```
vibhamasti@DESKTOP-CVL9CBN:~$ echo $SHELL
/bin/bash
```

zsh

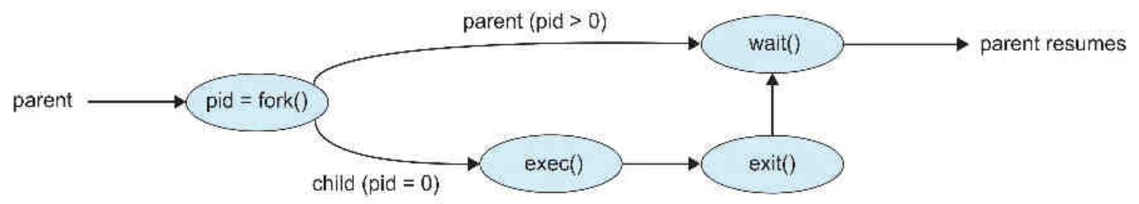
```
➜  ~ echo $SHELL
/bin/zsh
```

fork() : Process Creation

- fork() system call: creates new process

6 variations

- exec() system call: used after fork() to replace process' memory space with new program



termination

- exit() system call: asks OS to delete process
  - wait() returns status data from child to parent
  - resources deallocated by OS

- parent can terminate execution of children using abort()
  - child exceeds allocated resources
  - task no longer required
  - parent is terminating (exiting)

- Some OSes: if parent is terminating, all its children must terminate
  - cascading termination (children → grandchildren)

- wait() : parent waits for children to execute and terminate
  - returns status info and pid

linux: include
<sys/wait.h>

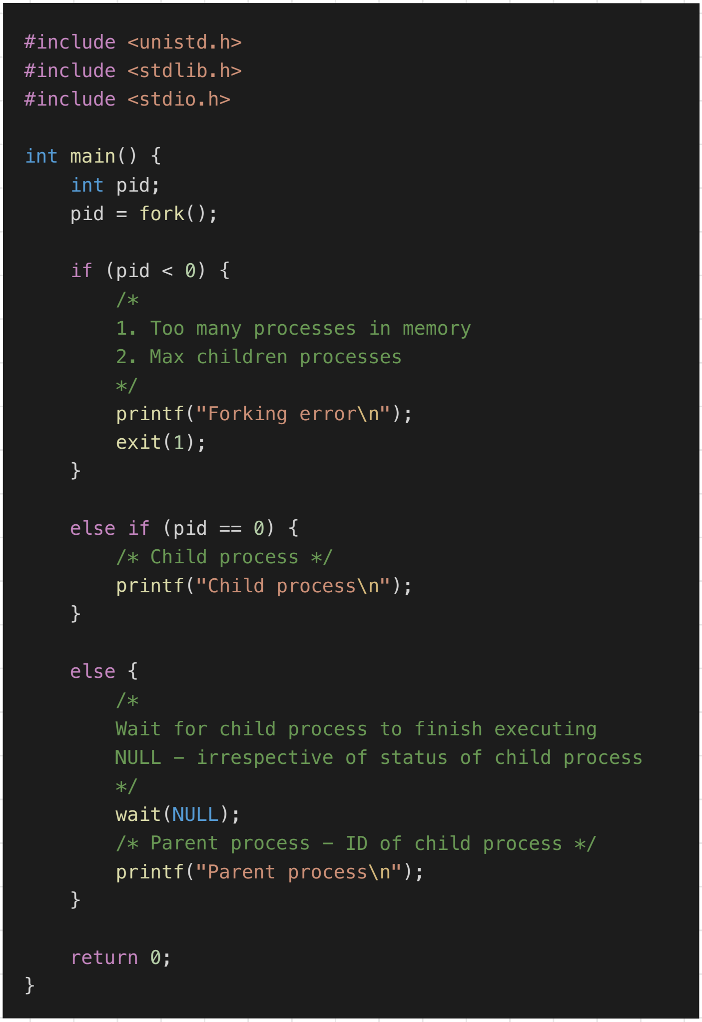
pid = wait(&status);

memory
location

- zombie : no parent waiting (parent sleeping; did not get status)

- orphan: parent terminated without wait() (child still executing)

https://www.geeksforgeeks.org/fork-system-call/

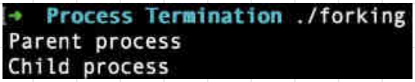


```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
    int pid;
    pid = fork();

    if (pid < 0) {
        /*
        1. Too many processes in memory
        2. Max children processes
        */
        printf("Forking error\n");
        exit(1);
    }

    else if (pid == 0) {
        /* Child process */
        printf("Child process\n");
    }

    else {
        /*
        Wait for child process to finish executing
        NULL - irrespective of status of child process
        */
        wait(NULL);
        /* Parent process - ID of child process */
        printf("Parent process\n");
    }

    return 0;
}
```

Terminal: with wait()

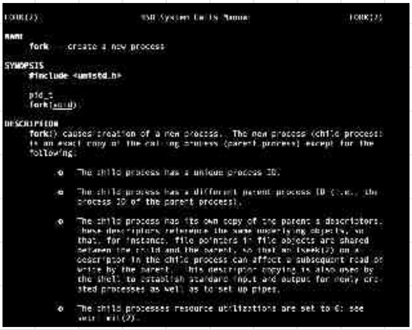
```
→  Process Termination ./forking
Child process
Parent process
```

Without wait()



```
→  Process Termination ./forking
Parent process
Child process
```

For more - man fork
man wait



```
FORK(2)                   BSD System Calls Manual                  FORK(2)

NAME
     fork -- create a new process

SYNOPSIS
     #include <unistd.h>

     pid_t
     fork(void);

DESCRIPTION
     fork() causes creation of a new process.  The new process (child process)
     is an exact copy of the calling process (parent process) except for the
     following:

           o   The child process has a unique process ID.

           o   The child process has a different parent process ID (i.e., the
               process ID of the parent process).

           o   The child process has its own copy of the parent's descriptors.
               These descriptors reference the same underlying objects, so
               that, for instance, file pointers in file objects are shared
               between the child and the parent, so that an lseek(2) on a
               descriptor in the child process can affect a subsequent read or
               write by the parent.  This descriptor copying is also used by
               the shell to establish standard input and output for newly cre-
               ated processes as well as to set up pipes.

           o   The child processes resource utilizations are set to 0; see
               [illegible]
```

exec commands

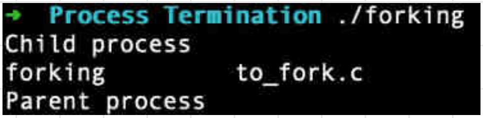
https://www.geeksforgeeks.org/wait-system-call-c/

https://www.geeksforgeeks.org/exec-family-of-functions-in-c/

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
    int pid;
    pid = fork();

    if (pid < 0) {
        /*
        1. Too many processes in memory
        2. Max children processes
        */
        printf("Forking error\n");
        exit(1);
    }

    else if (pid == 0) {
        /* Child process */
        printf("Child process\n");
        execl("/bin/ls", "ls", NULL);
    }

    else {
        /*
        Wait for child process to finish executing
        NULL - irrespective of status of child process
        */
        wait(NULL);
        /* Parent process - ID of child process */
        printf("Parent process\n");
    }

    return 0;
}
```

anything after is not run

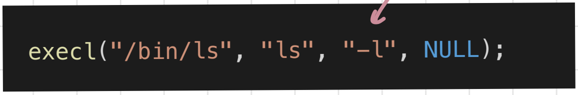
Terminal



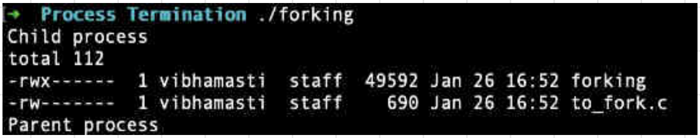
```
→  Process Termination ./forking
Child process
forking         to_fork.c
Parent process
```

Arguments

arguments to command



```c
execl("/bin/ls", "ls", "-l", NULL);
```

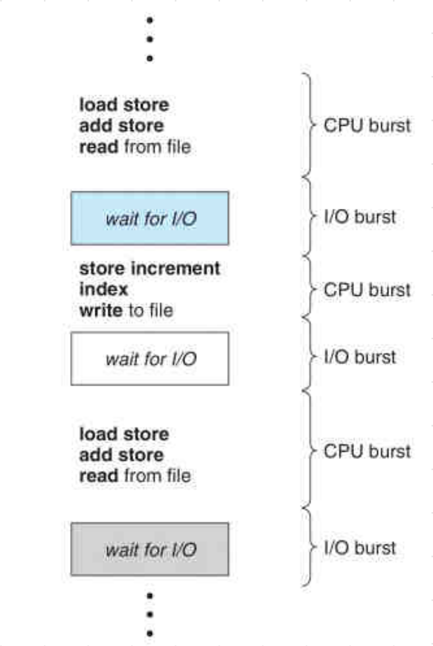


```
→  Process Termination ./forking
Child process
total 112
-rwx------  1 vibhamasti  staff  49592 Jan 26 16:52 forking
-rw-------  1 vibhamasti  staff    690 Jan 26 16:52 to_fork.c
Parent process
```

CPU SCHEDULING

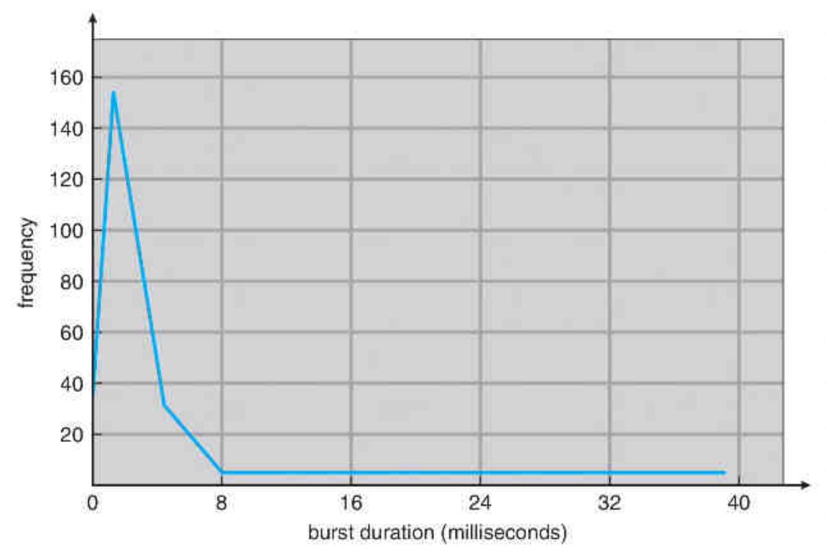
- From RQ to CPU : order of assignment, execution
- Single core system: only one process at a time. Once CPU is free, next process
- Multiprogramming: maximise CPU utilisation and minimise idling
- Several processes in memory at once
- When one process is waiting, OS takes CPU away from it and assigns new process ↖ I/O
- Multi-core : process of keeping CPU busy extended to all cores

Alternate Sequence of I/O and CPU Bursts



maximise CPU utilisation

CPU Burst Time Histogram



- multiple short CPU bursts
- few long CPU bursts

CPU Scheduler

- Short-term scheduler
- Queue: FIFO, queue, tree, unordered linked list
- records: PCB

Preemptive and Non-Preemptive Scheduling

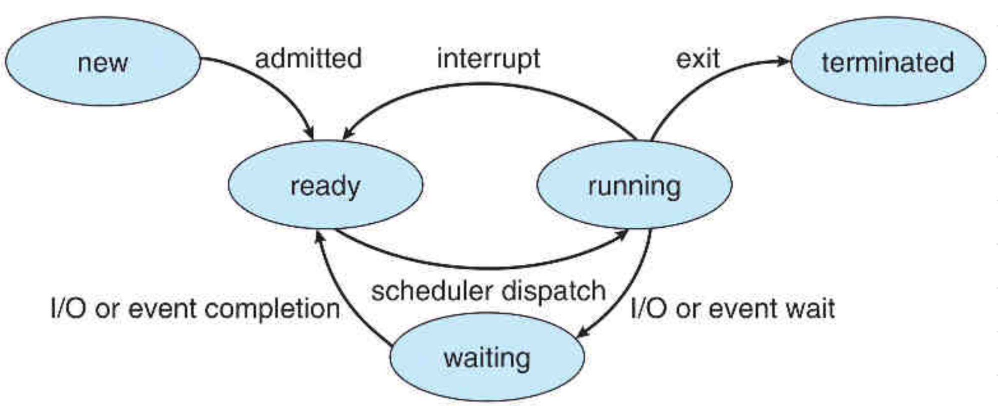
- When a process
  1. switches from running to waiting – I/O, event
  2. switches from running to ready – interrupt
  3. switches from waiting to ready
  4. terminates

- Scheduling under 1 and 4: non-preemptive – allowed to run to completion with CPU, switching to wait state

- Others: preemptive – process asked to release CPU involuntarily
  - access to shared data
  - kernel mode
  - interrupts during crucial OS tasks

  } race condition

- Preemptive scheduling used by most OSes



- **Mutex locks**: prevent race conditions while accessing shared data from kernel data structures

DISPATCHER

- Gives control of CPU to process selected by short-term scheduler
  - context switching
  - switching to user mode
  - jumping to restart

- Dispatch latency: time taken by dispatcher to stop one process and start another

Scheduling Criteria

- CPU Utilisation : ~40% to ~90% (max)

- Throughput : no. of processes executed in unit time (max)

- Turnaround time: time taken to execute particular process ↙ wait + CPU burst
  (min) – performance metric

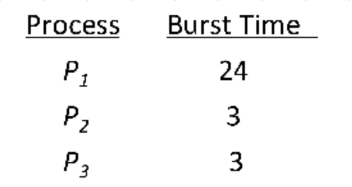
- Waiting time: time for which a process waits in ready queue
  (min)

- Response time: amount of time it takes from when request submitted, until first response (min)

Scheduling ALGORITHMS

1. First Come, First Serve (FCFS)

- executed in order of arrival



Process	Burst Time
P_1	24
P_2	3
P_3	3

convoy effect : shorter tasks after long tasks

- if arrival order : P_1, P_2, P_3

Gantt Chart

- waiting time for $P_1 = 0$, t·a of $P_1 = 24$
  waiting time for $P_2 = 24$, t·a of $P_2 = 27$
  waiting time for $P_3 = 27$, t·a of $P_3 = 30$
- average waiting time $= \frac{0+24+27}{3} = 17$
- average turnaround time $= 27$
- if arrival order P_2, P_3, P_1

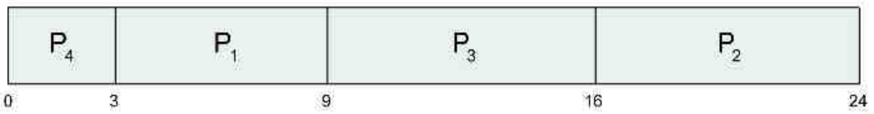
- t_w $P_1 = 6$, $P_2 = 0$, $P_3 = 3$, average $t_w = 3$
- t_{ta} $P_1 = 30$, $P_2 = 3$, $P_3 = 6$, average $t_{ta} = 13$

2 Shortest Job First (SJF) Scheduling

- process with shorter CPU burst time executed first
- optimal - minimum average waiting time
- knowing the length of next process - difficulty

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- Gantt Chart for SJF



- average wait time = $\dfrac{0+3+9+16}{4} = 7$

Predicting Length of Next CPU Burst

- only an estimate (should be similar to previous one)
- using length of previous bursts using exponential averaging

actual length

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

α: 0.5 usually

predicted

$1 \leq \alpha \leq 1$

Q: Calculate exponential averaging with $\tau_1 = 10$, $\alpha = 0.5$ and the algorithm is SJF with previous runs as 8, 7, 4, 16

Initially $\tau_1 = 10$ and $\alpha = 0.5$ and run times 8, 7, 4, 16

Possible order: 4, 7, 8, 16

$$\boxed{\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n}$$

$\tau_2 = 0.5 \times 4 + 0.5 \times 10 = 7$

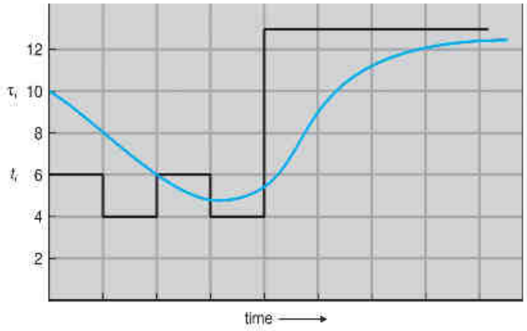
$\tau_3 = 0.5 \times 7 + 0.5 \times 7 = 7$

$\tau_4 = 0.5 \times 8 + 0.5 \times 7 = 7.5$

$\tau_5 = 0.5 \times 16 + 0.5 \times 7.5 = 11.75$

$\tau_1 = 10$
$\tau_2 = 7$
$\tau_3 = 7$
$\tau_4 = 7.5$
$\tau_5 = 11.75$

PREDICTION



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

EFFECT OF α

- $\alpha = 0$: $\tau_{n+1} = \tau_n$

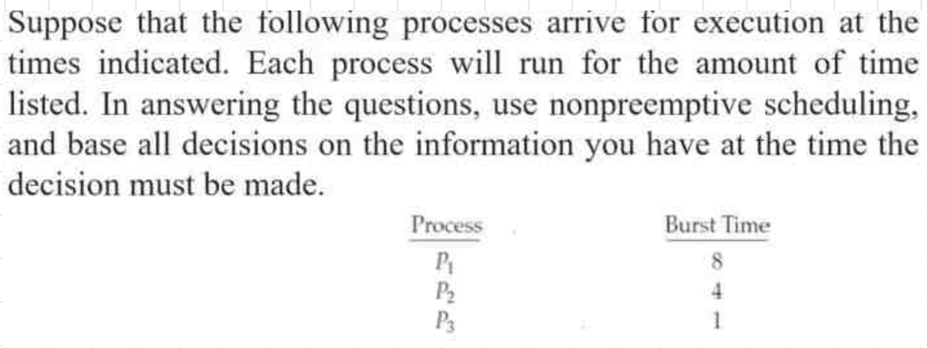
  previous round does not effect

- $\alpha = 1$: $\tau_{n+1} = t_n$

  only previous CPU burst counts

- simplified exponential averaging

Q:



Suppose that the following processes arrive for execution at the times indicated. Each process will run for the amount of time listed. In answering the questions, use nonpreemptive scheduling, and base all decisions on the information you have at the time the decision must be made.

Process	Burst Time
P_1	8
P_2	4
P_3	1

What is average turnaround time with

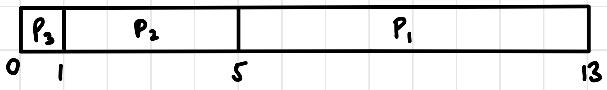
(i) FCFS

(ii) SJF

(i)

average turnaround $= \frac{8+12+13}{3} = 11$

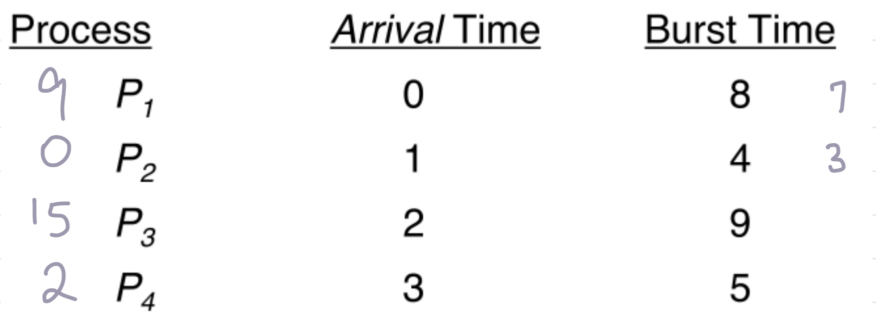
(ii)



average turnaround $= \frac{1+5+13}{3} = 6.33$

3. Shortest Remaining Time First (SRTF) Scheduling

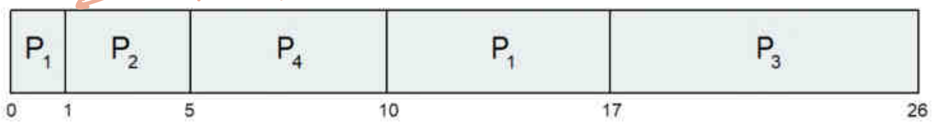
- preemptive version of SJF
- arrival time taken into account (current time - arrival)
- preempt currently executing process if new process has shorter burst time



wait time	Process	*Arrival* Time	Burst Time		
9	P_1	0	8	7	7
0	P_2	1	4	3	2
15	P_3	2	9		9
2	P_4	3	5		5

- Gantt chart

preempted



- average waiting time = $\frac{9+0+15+2}{4} = 6.5$

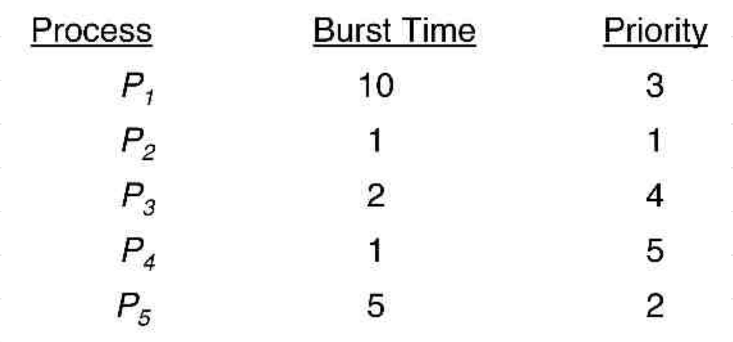
turnaround time = exit time - arrival time

waiting time = turnaround time - burst time

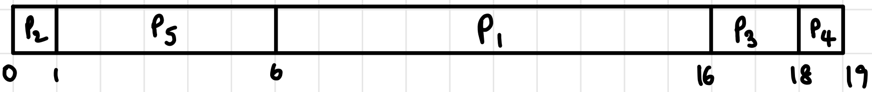
4. Priority Scheduling

- priority defined by integer (smaller → higher priority)
- preemptive (arrival time; shortest remaining time)
  non-preemptive
- SJF is priority scheduling algorithm where priority is inverse of CPU burst time
- Problem ≡ starvation (never executed if priority very low)
- Solution ≡ ageing (as time increases, priority increases)

Q: Non-preemptive priority queue (no arrival). Find avg wait time.



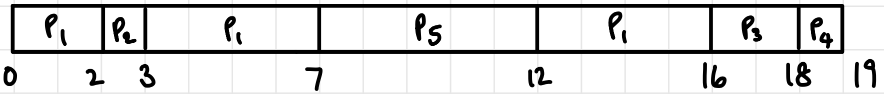
Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



average waiting $= \dfrac{0+1+6+16+18}{5} = 8.2$

Q: Preemptive

Process	Burst Time	Priority	Arrival Time
P_1	10	3	0
P_2	1	1	2
P_3	2	4	4
P_4	1	5	5
P_5	5	2	7



average wait time $= \frac{(16-0-10)+(3-2-1)+(18-4-2)+(19-5-1)+(12-7-5)}{5}$

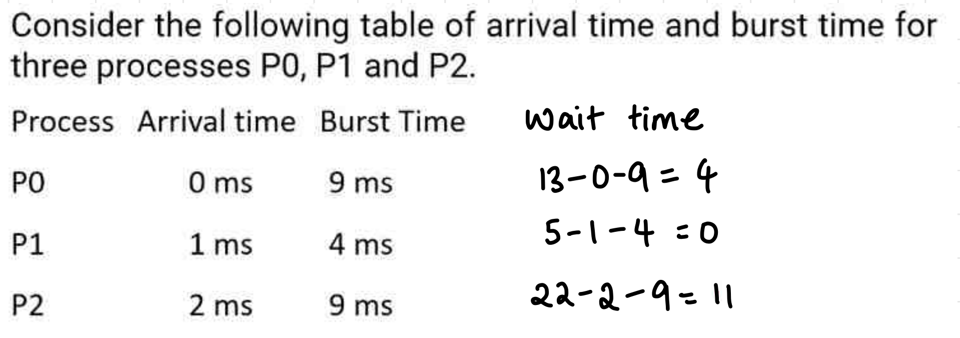
$= \frac{31}{5} = 6.2$

Q: Consider three CPU-intensive processes, which require 10, 20 and 30 time units and arrive at times 0, 2 and 6, respectively. How many context switches are needed if the operating system implements a shortest remaining time first scheduling algorithm? Do not count the context switches at time zero and at the end

Process	Burst Time	Arrival Time
P_1	10	0
P_2	20	2
P_3	30	6

∴ 0 context switches

Q:



Consider the following table of arrival time and burst time for three processes P0, P1 and P2.

Process	Arrival time	Burst Time	wait time
P0	0 ms	9 ms	13-0-9 = 4
P1	1 ms	4 ms	5-1-4 = 0
P2	2 ms	9 ms	22-2-9 = 11

Preemptive SJF = SRTF

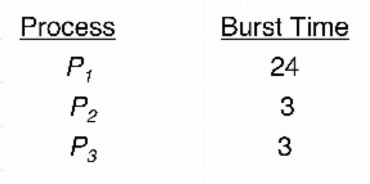
average wait time = $\frac{15}{3}$ = 5

5. Round Robin

- each process gets a small unit of CPU time (time quantum q) usually 10-100 ms
- after this time has elapsed, process is preempted and added to the end of the ready queue
- if there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks, at most q units
- no process waits more than (n-1)q units
- timer interrupts every quantum to schedule next process
- performance
  - q large → FIFO
  - q small → overhead too high (too many context switches)

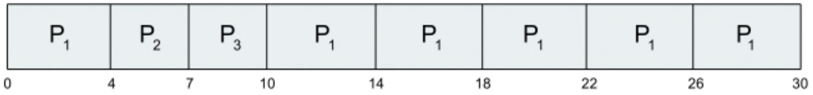
q should be >> context switch time

Q: $q = 4$



Process	Burst Time
P_1	24
P_2	3
P_3	3

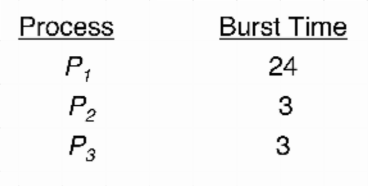
Gantt Chart



- higher average turnaround than SJF, better response

- average waiting time = $\frac{(10-4)+4+7}{3} = \frac{17}{3} = 5.67$

Q: if q = 2 for prev question

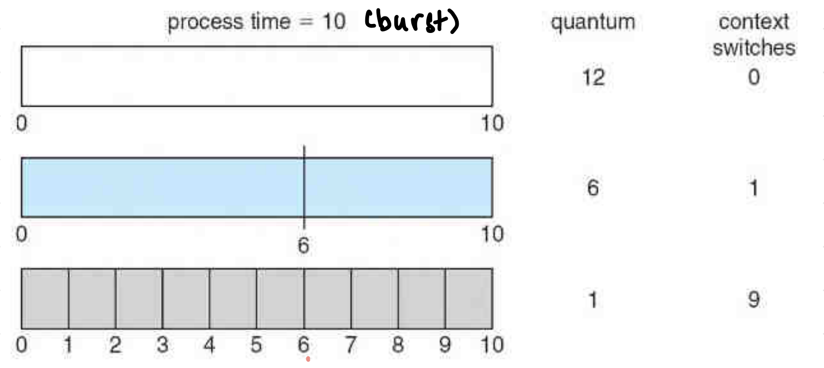


Process	Burst Time
P_1	24
P_2	3
P_3	3

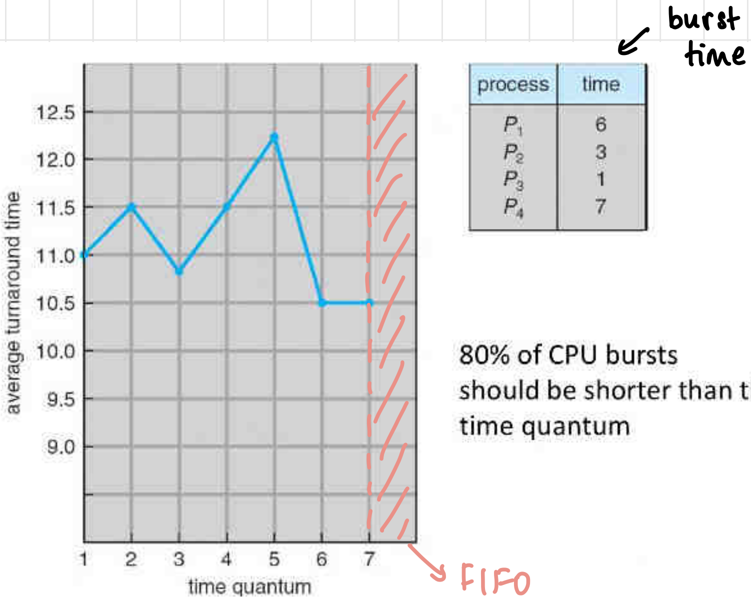
P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1	P_1	P_1	P_1	P_1	P_1

Time Quantum and Context Switching

shorter q, more context switches



Turnaround Time and Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than the time quantum

- Linux, Windows use RR scheduling

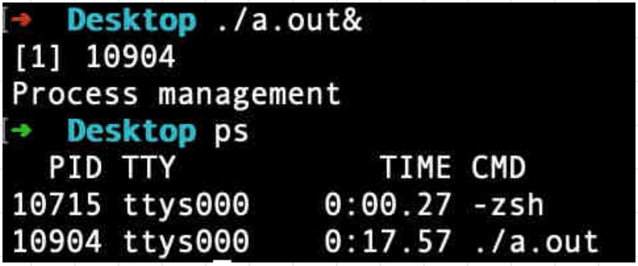
6. Multilevel Scheduling

- ready queue → 2 separate partitions
  - foreground (interactive)
  - background (batch)

- process permanently in a given queue

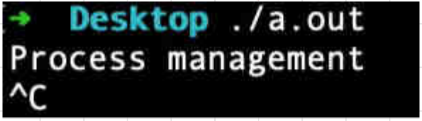
- each queue : scheduling algorithm
  - foreground : RR
  - background : FCFS

run in background



```
[➔  Desktop ./a.out&
[1] 10904
Process management
[➔  Desktop ps
  PID TTY           TIME CMD
10715 ttys000    0:00.27 -zsh
10904 ttys000    0:17.57 ./a.out
```

run in foreground

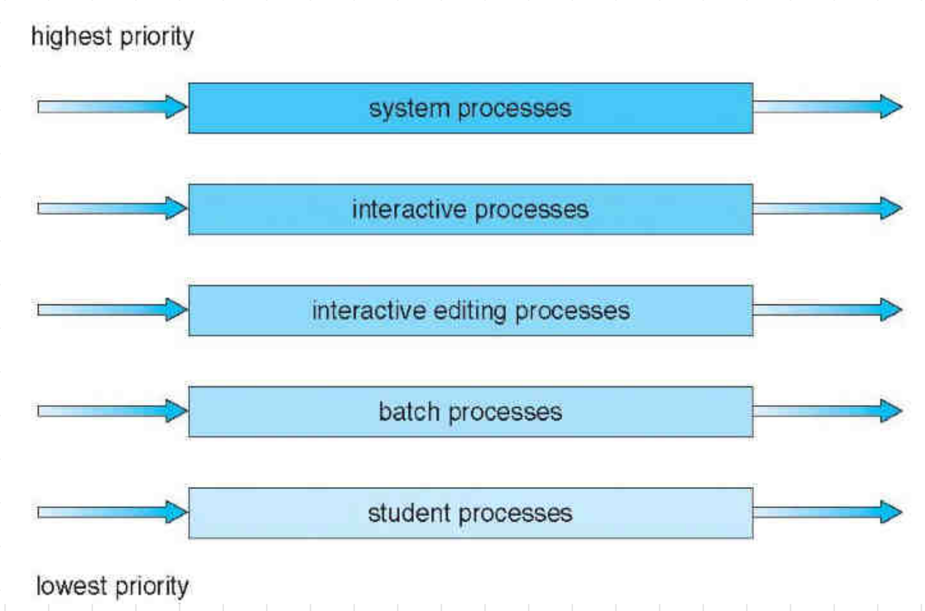


```
➔  Desktop ./a.out
Process management
^C
```

bring back to foreground

```
[➔  Desktop fg %1
[1]  + 10904 running    ./a.out
^C
```

- scheduling done between queues
  - fixed priority scheduling (starvation)
  - time slices (~80% fg)



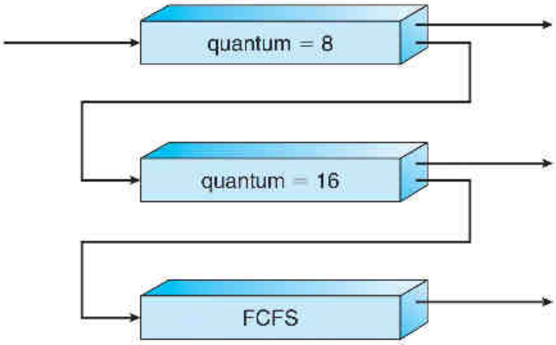
7. Multilevel Feedback Queue Scheduling

- process moves between various queues - ageing

- multilevel feedback queue scheduler:
  - no. of queues
  - each queue's scheduling algorithm
  - method to determine when to upgrade a process
  - method to determine when to demote a process
  - method to determine which queue a process enters when it needs service

- three queues - example
  - Q_0: RR - quantum 8 ms
  - Q_1: RR - quantum 16 ms
  - Q_2: FCFS

- Scheduling
  - new job enters Q_0
  - job receives 8ms
  - if not completed, moved to Q_1
  - jobs receive additional 16 ms
  - if not complete, preempted and moved to Q_2
  - Q_2 is FCFS

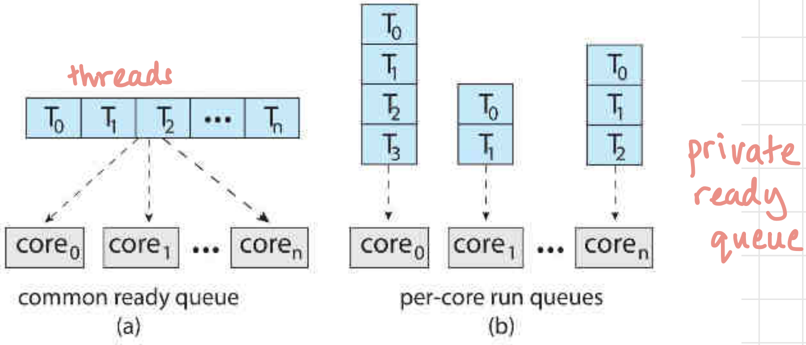


MULTIPLE PROCESSOR SCHEDULING

- Asymmetric scheduling: single master assigns processes to other processor; only master has access to system data structures (communicates with OS)

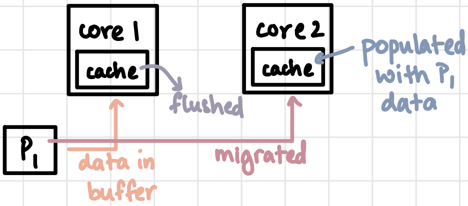
- Symmetric scheduling (SMP): each processor has its own ready queue and scheduling algorithm or all processes have common ready queue

- Modern OSes: Windows, Linux, MacOS support SMP



symmetric multiprocessing (SMP)

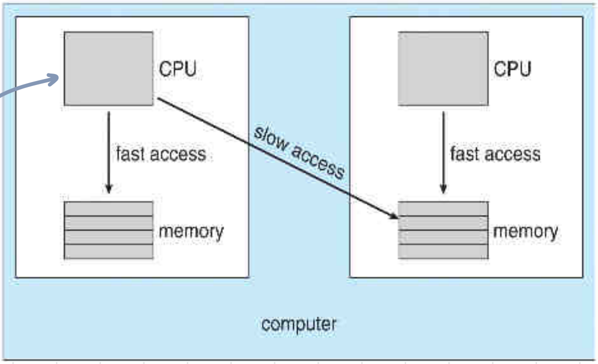
- each processor: own cache ; buffer
  - buffer of cache populated with process data
  - process migrates: cache flushed, new cache filled



- Processor affinity: process has affinity for processor on which it is currently running
    - soft affinity: OS keeps process on same core (tries not to migrate) but not guaranteed
    - hard affinity: doesn't allow process to migrate between processors
    - Linux: soft affinity
    - sched_setaffinity() system call - supports hard affinity

ACCESS to MEMORY

- Main memory architecture can affect processor affinity



on chip memory

- Scheduling and memory placement algorithms work together

LOAD BALANCING

- SMP tasks (each CPU own task)

- Push migration: periodic task checks load on each CPU and pushes overloaded CPU task to other CPU

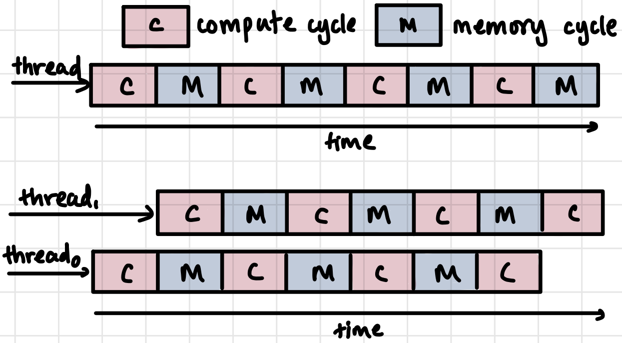
- Pull migration: idle processors pull waiting task from busy processor

- Counteracts benefits of processor affinity

Multicore Processors

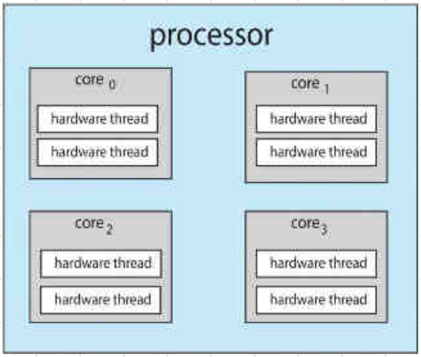
- Multiple cores on single chip
- Faster, less power
- Multiple hardware threads
- Memory stall: request memory for data, takes time; another thread can compute while previous thread is fetching from memory

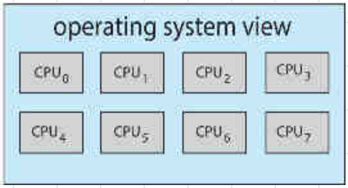
idling CPU



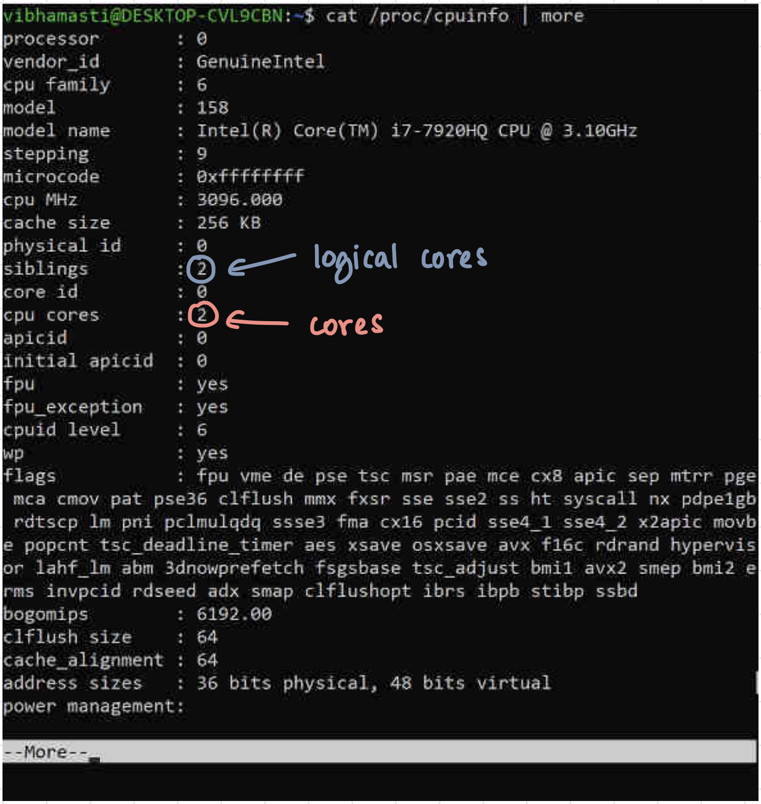
Chip Multithreading

- CMT - each core assigned multiple threads
- Intel: hyperthreading
- Quad-core system with 2 threads per core: logically 8 cores to the OS





`cat /proc/cpuinfo | more` (page 11)



```
vibhamasti@DESKTOP-CVL9CBN:~$ cat /proc/cpuinfo | more
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-7920HQ CPU @ 3.10GHz
stepping        : 9
microcode       : 0xffffffff
cpu MHz         : 3096.000
cache size      : 256 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 6
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
 mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb
 rdtscp lm pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movb
e popcnt tsc_deadline_timer aes xsave osxsave avx f16c rdrand hypervis
or lahf_lm abm 3dnowprefetch fsgsbase tsc_adjust bmi1 avx2 smep bmi2 e
rms invpcid rdseed adx smap clflushopt ibrs ibpb stibp ssbd
bogomips        : 6192.00
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:

--More--
```

siblings: 2 ← logical cores

cpu cores: 2 ← cores

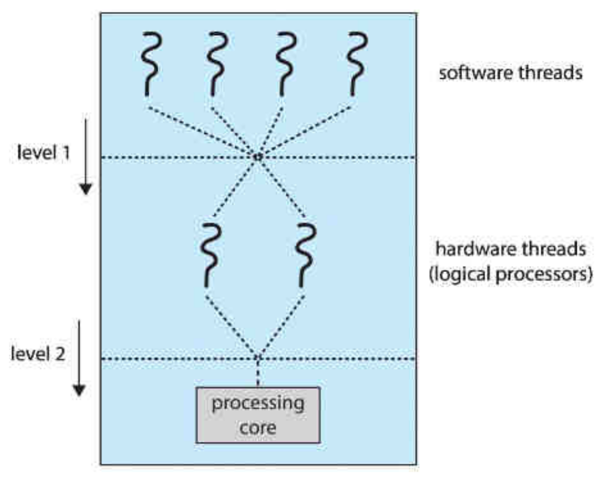
Multithreading

1. Coarse grained
   - thread executed on processor until long-latency event such as memory stall
   - cost of switching is high
   - state is saved

2. Fine grained
   - cost lower
   - finer level of granularity

Multithreaded Multicore Processor

- two levels of scheduling



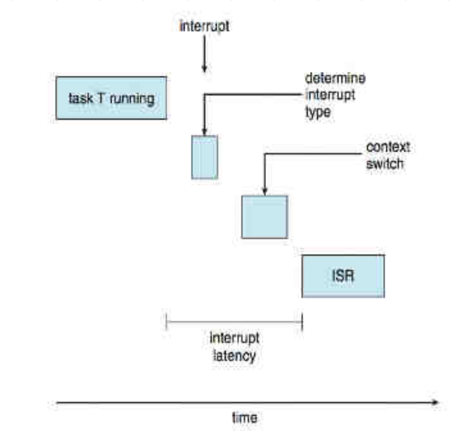
- OS decides which software thread runs on a logical CPU
- How each core decides which hardware thread to run on physical core

Real-Time CPU Scheduling

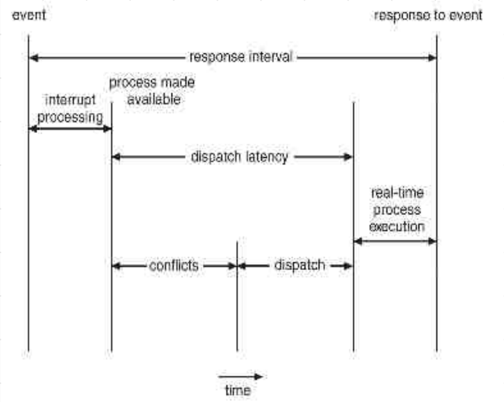
- embedded systems, real-time CPUs

1) Soft Real-Time systems
    - no guarantee as to when task is scheduled
2) Hard Real-Time systems
    - must be serviced by deadline

- interrupt latency interrupt arrival to service
- dispatch latency switch CPUs



- conflict phase of dispatch latency
  - preemption of process running in kernal mode
  - release by low priority process of resources needed by high priority



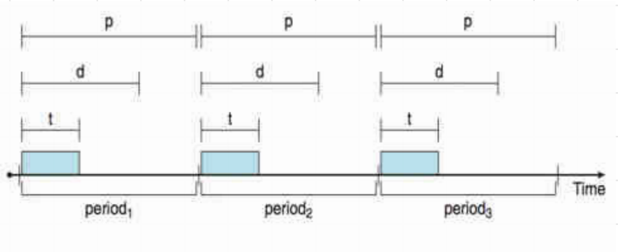
scheduling ALGORITHMS

1. Priority-based Scheduling

- preemptive
- soft real-time, not hard
- CPU required at constant intervals

- t : processing time
  d : deadline
  p : period

$0 \le t \le d \le p$



- rate of periodic task = $1/p$

2. Rate Monotonic Scheduling

- inverse of period : priority

Q: P1 and P2 are 50 and 100, respectively—that is, p1 = 50 and p2 = 100.
The processing times are t1 = 20 for P1 and t2 = 35 for P2. The deadline for each process requires that it complete its CPU burst by the start of its next period.

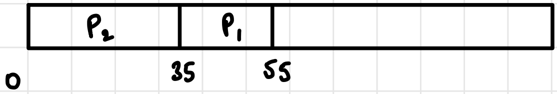
CPU utilisation = $\frac{t_i}{p_i}$

$p_1 = 50$
$t_1 = 20$

$p_2 = 100$
$t_2 = 35$

$P1 = \frac{20}{50} = 0.4$ $P2 = \frac{35}{100} = 0.35$ total = 0.75

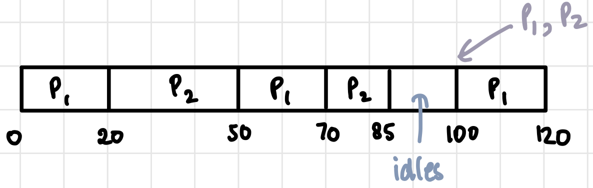
Case 1: P2 priority higher than P1 (priority should be based on period)



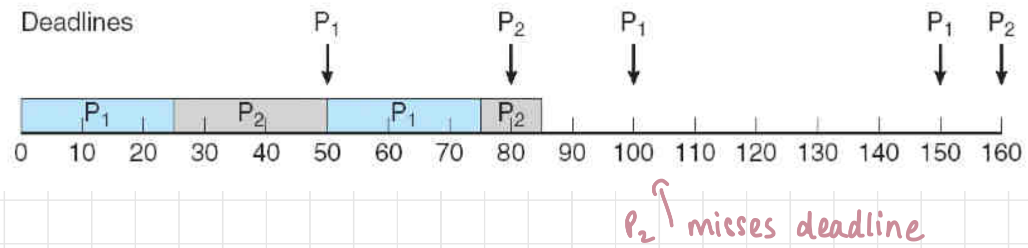
P_1 needs to complete before 50

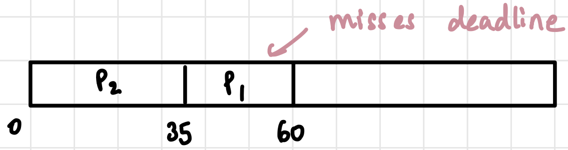
misses deadline

Case 2: P1 priority higher than P2 missed deadlines



Q: Consider processes with $p_1 = 50$, $t_1 = 25$, $p_2 = 80$, $t_2 = 35$





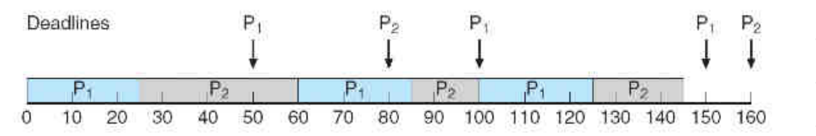
priority order also wrong

- worst case utilisation of CPU for N processes = $N(2^{1/N} - 1)$

3. Earliest Deadline First (EDF) Scheduling

- priorities assigned dynamically according to deadlines
- earlier the deadline, higher the priority
- soft real-time

Q: consider processes with $p_1 = 50$, $t_1 = 25$, $p_2 = 80$, $t_2 = 35$



4. Proportional Share Scheduling

- T shares allocated among all processes
- each app: N/T of processor time (N shares)
- eg: assume total T = 100 shares and 3 processes A, B, C

A ⟶ 50
B ⟶ 15
C ⟶ 20

85 shares

not more than 15 more can be allocated

5. POSIX Real-Time Scheduling

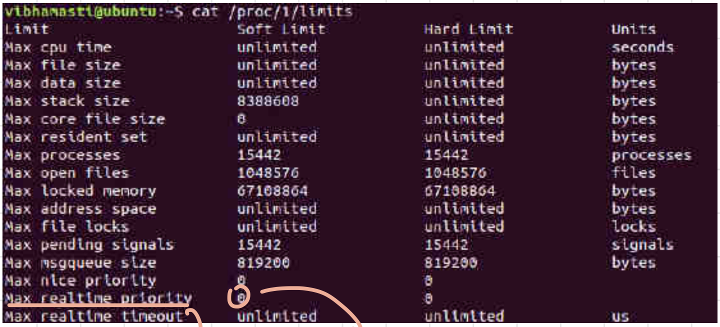
- POSIX.1b standard
- API provides functions for managing threads in real-time
- Scheduling classes
  1. SCHED_FIFO: threads scheduled using FCFS with queue, no time-slicing for equal priority
  2. SCHED_RR: similar to FIFO but time-slicing occurs

get posix scheduling algorithm

```
pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)

pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)
```

set

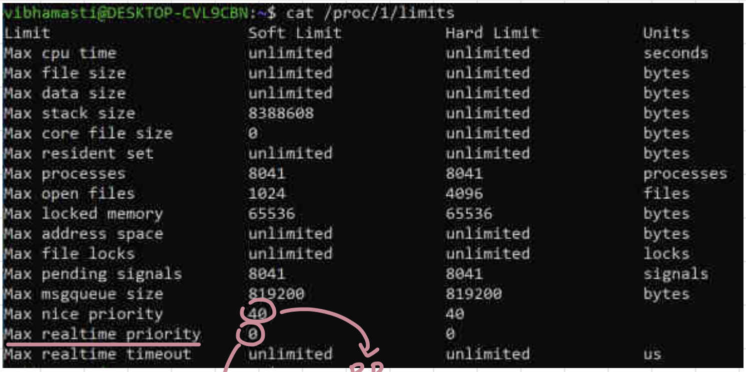
```
$ cat /proc/1/limits
```



```
vibhamasti@ubuntu:~$ cat /proc/1/limits
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        0                    unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             15442                15442                processes
Max open files            1048576              1048576              files
Max locked memory         67108864             67108864             bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       15442                15442                signals
Max msgqueue size         819200               819200               bytes
Max nice priority         0                    0
Max realtime priority     0                    0
Max realtime timeout      unlimited            unlimited            us
```

real time priority scheduling

priority 0



```
vibhamasti@DESKTOP-CVL9CBN:~$ cat /proc/1/limits
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        0                    unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             8041                 8041                 processes
Max open files            1024                 4096                 files
Max locked memory         65536                65536                bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       8041                 8041                 signals
Max msgqueue size         819200               819200               bytes
Max nice priority         40                   40
Max realtime priority     0                    0
Max realtime timeout      unlimited            unlimited            us
```

priority

RR switching

- Linux machines: Completely Fair Scheduler (CFS)
  - check slides
  - see: red-black trees

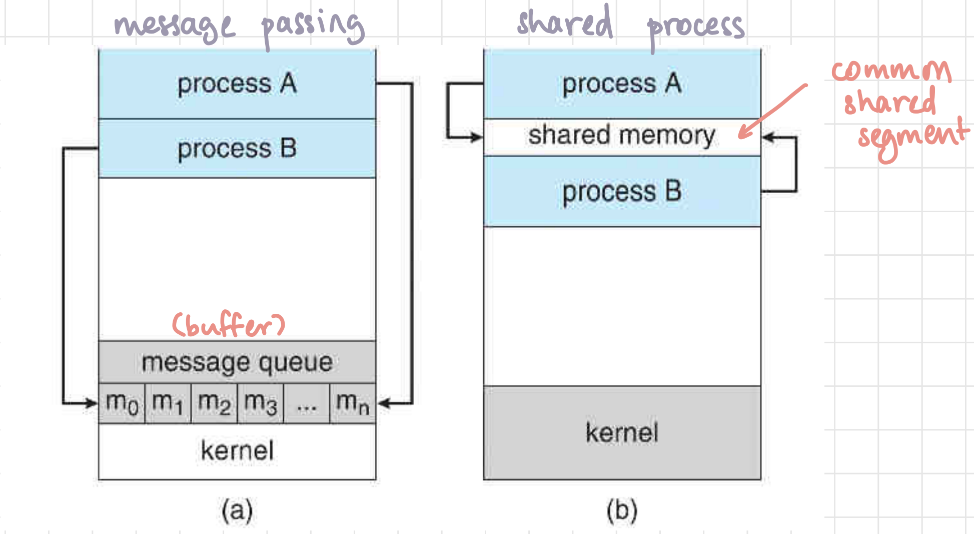
- Windows machine
  - check slides

inter-process communication

- processes: independent or cooperating
  - independent ↳ do not affect each other
  - cooperating ↓ affect each other

- cooperating processes
  - information sharing
  - computation speedup: divide task
  - modularity : cores, dependency
  - convenience

- communication models



Producer - Consumer Model

- producer: process writes to buffer
- consumer: process reads from buffer

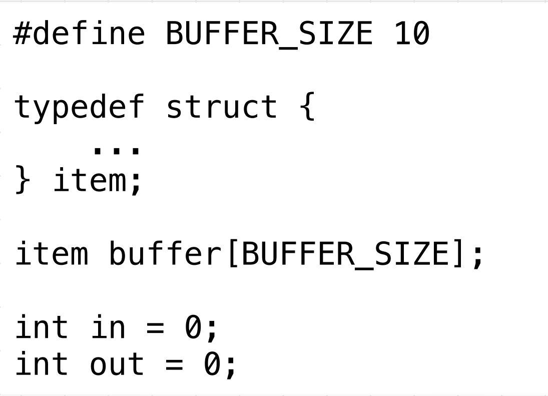
1. Unbounded buffer

- Producer can keep producing data and writing into buffer
- Consumer cannot read from an empty buffer; must wait
- no practical limit on buffer

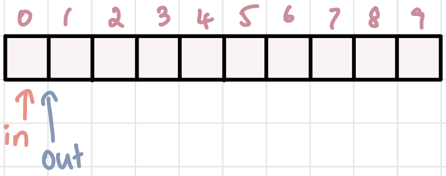
2. Bounded buffer

- Producer waits when buffer full
- Consumer waits when buffer empty

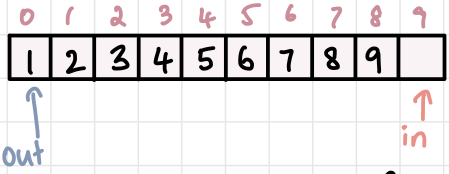
implemented as circular array



```
#define BUFFER_SIZE 10

typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];

int in = 0;
int out = 0;
```



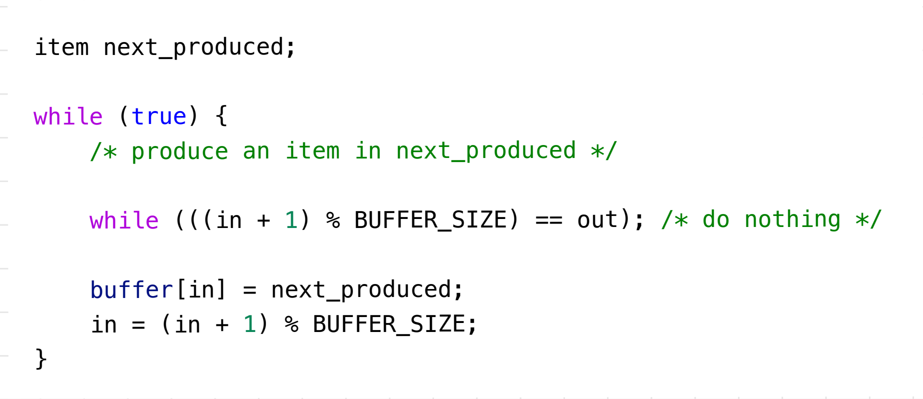
empty: in = out



full queue: one is wasted

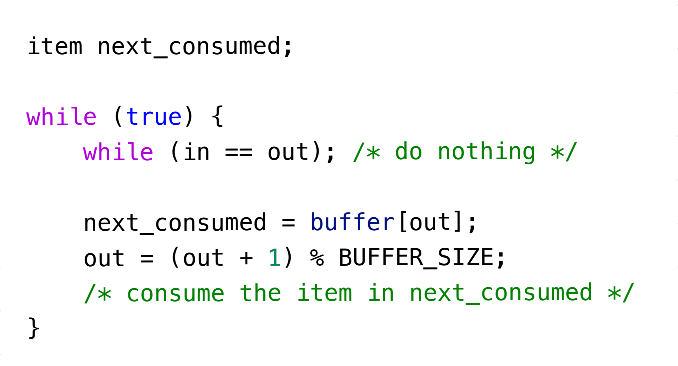
full : (in+1) % BUFFER_SIZE = out

Producer



```
item next_produced;

while (true) {
    /* produce an item in next_produced */

    while (((in + 1) % BUFFER_SIZE) == out); /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

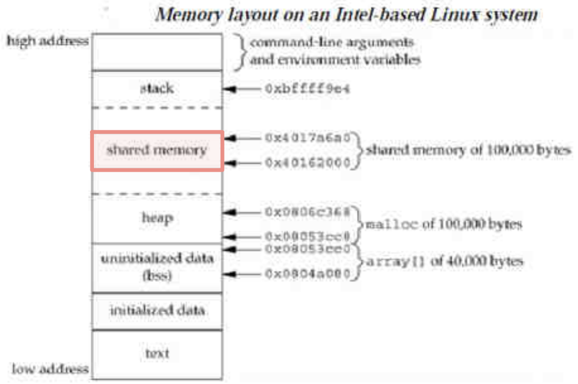
Consumer



```
item next_consumed;

while (true) {
    while (in == out); /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* consume the item in next_consumed */
}
```

shared MEMORY

- two or more processes have access to same memory
- fastest form of IPC
- synchronising access (eg: client & server)
- semaphores: shared memory access


Memory layout on an Intel-based Linux system

message passing

- processes communicate and synchronise
- IPC: send (message), receive (message)
- if P & Q wish to communicate, they must
  1. establish communication link
  2. exchange messages via send / receive
- link between messages – size, direction etc.

- physical link: shared memory, hardware bus, network
- logical link: direct/indirect, sync/async, automatic/explicit buffering

Direct COMMUNICATION

- processes named explicitly
- send(P, message)
  receive(Q, message)
- automatic links, one link per pair
- usually bidirectional

Indirect COMMUNICATION

unique ID

- messages sent/received to/from maiboxes (ports)
- only if processes share a mailbox, they can communicate
- each pair may have several links and each link can connect several process
- link: uni or bidirectional
- create new port/mailbox
- issues: P_1 sends to shared mailbox with R_2, R_3 ; who gets message?

Blocking & Non-Blocking

Blocking

- synchronous
- blocking send: sender blocked until message received
- blocking receive: receiver blocked until message sent

Non-blocking

- asynchronous
- non-blocking send: the sender sends the message, continue
- non-blocking receive: the receiver receives a null or valid message

— if both sender and receiver are blocking, rendezvous between sender and receiver

Buffering

- queues of messages attached to link ; temporary queue for messages
- queues:

message system with no buffering

1. zero capacity: no queue; sender waits for receiver; rendezvous
2. bounded capacity: finite length of n messages, sender waits if full

automatic buffering

3. unbounded capacity: infinite length; sender never waits

Pipes

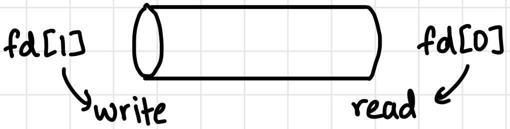
- half-duplex IPC

parent-child · no relationship

- ordinary (unnamed) pipes and named pipes

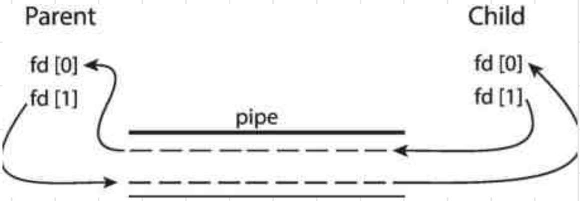
Ordinary Pipes

- producer-consumer style (write-read)



0 – stdin
1 – stdout
2 – stderr

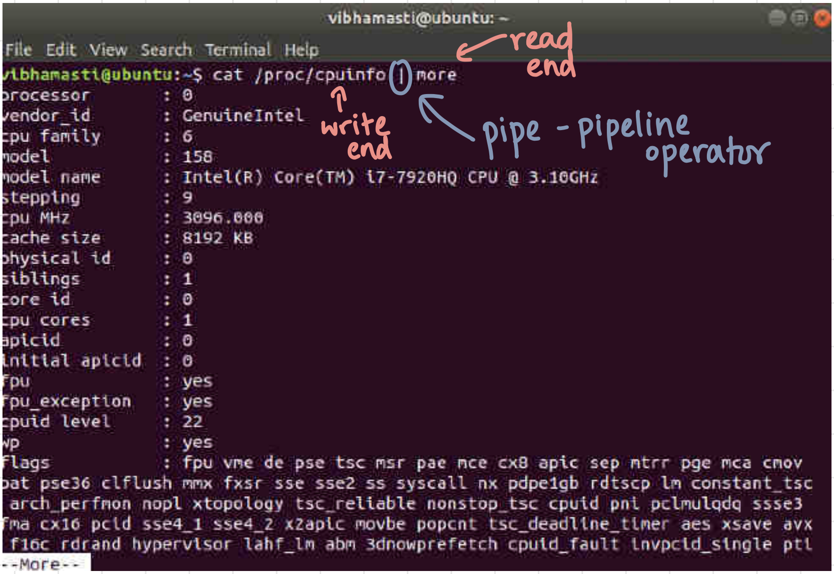
- half-duplex (unidirectional)
- for two-way, two pipes
- Linux: pipe, Windows: anonymous pipes



Named Pipes

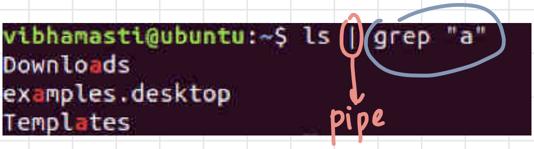
- More powerful than ordinary pipes
- bidirectional, no parent-child relationship

- several processes same pipe
- two pipes for two-way
- FIFO: once retrieved, data removed
- UNIX:
  - mkfifo(), open(), read(), write(), close()
  - byte-oriented
  - half-duplex
  - same machine
- Windows
  - CreateNamedPipe(), ConnectNamedPipe(), ReadFile(), WriteFile(), DisconnectNamedPipe()
  - full duplex
  - same or different machine
  - byte or message oriented



```
ibhamasti@ubuntu:~$ cat /proc/cpuinfo | more
rocessor        : 0
endor_id        : GenuineIntel
pu family       : 6
odel            : 158
odel name       : Intel(R) Core(TM) i7-7920HQ CPU @ 3.10GHz
tepping         : 9
pu MHz          : 3096.000
ache size       : 8192 KB
hysical id      : 0
iblings         : 1
ore id          : 0
pu cores        : 1
picid           : 0
nitial apicid   : 0
pu              : yes
pu_exception    : yes
puid level      : 22
p               : yes
lags            : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
at pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm constant_tsc
 arch_perfmon nopl xtopology tsc_reliable nonstop_tsc cpuid pni pclmulqdq ssse3
ma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx
 f16c rdrand hypervisor lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti
--More--
```

Search for Pattern



```
vibhamasti@ubuntu:~$ ls | grep "a"
Downloads
examples.desktop
Templates
```

search for pattern

pipe

Named pipe - fifo

```
vibhamasti@ubuntu:~$ mkfifo k
vibhamasti@ubuntu:~$ ls
Desktop    Downloads         k      Pictures  Templates
Documents  examples.desktop  Music  Public    Videos
vibhamasti@ubuntu:~$ ls -l k
prw-r--r-- 1 vibhamasti vibhamasti 0 Feb  3 10:46 k
vibhamasti@ubuntu:~$ ls -l examples.desktop
-rw-r--r-- 1 vibhamasti vibhamasti 8980 Feb  2 20:04 examples.desktop
```

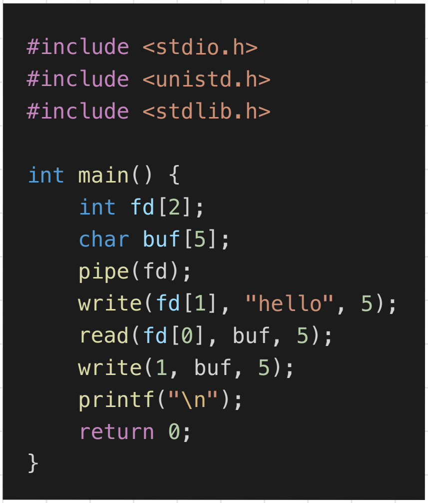
fifo file

pipe

normal

manual entry (2) — system call
manual entry (1) — command

pipe.c

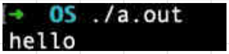


```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    int fd[2];
    char buf[5];
    pipe(fd);
    write(fd[1], "hello", 5);
    read(fd[0], buf, 5);
    write(1, buf, 5);
    printf("\n");
    return 0;
}
```

Ubuntu

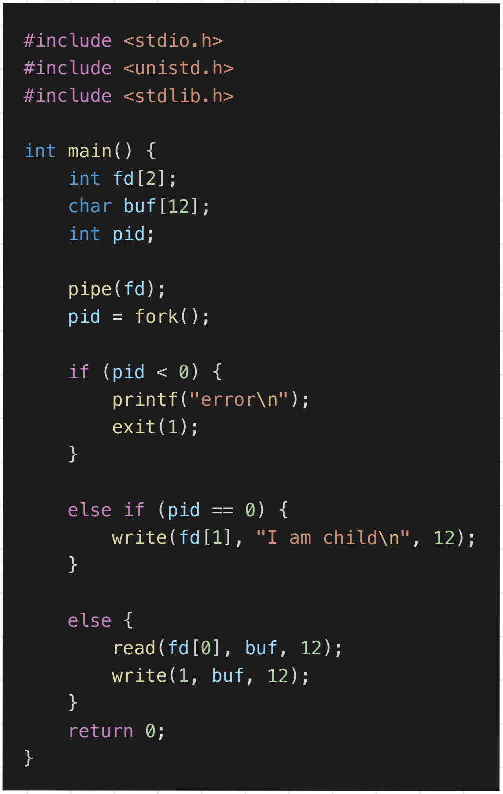
```
vibhamasti@ubuntu:~/dev/college/OS$ ./a.out
hello
```

MacOS



```
➜  OS ./a.out
hello
```

One-way communication

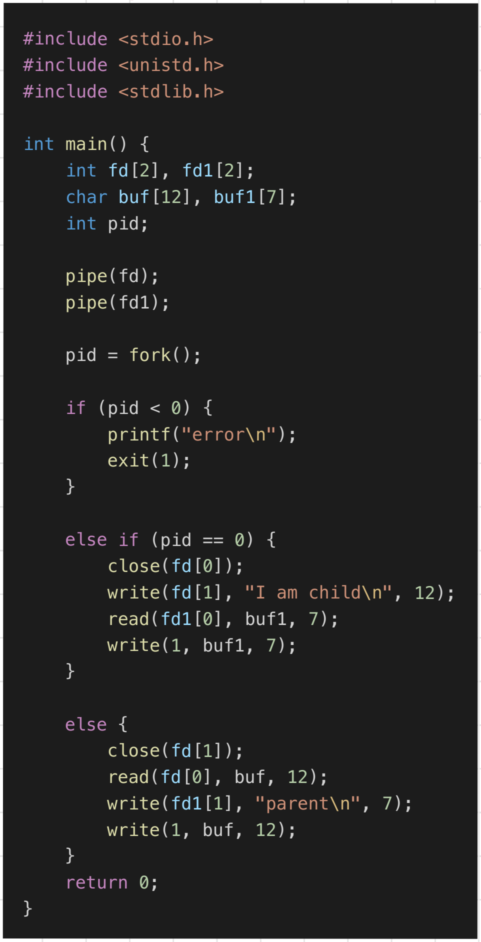


```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    int fd[2];
    char buf[12];
    int pid;

    pipe(fd);
    pid = fork();

    if (pid < 0) {
        printf("error\n");
        exit(1);
    }

    else if (pid == 0) {
        write(fd[1], "I am child\n", 12);
    }

    else {
        read(fd[0], buf, 12);
        write(1, buf, 12);
    }
    return 0;
}
```

Output

```
➜  OS ./pipe2
I am child
```

Two-way Communication



```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    int fd[2], fd1[2];
    char buf[12], buf1[7];
    int pid;

    pipe(fd);
    pipe(fd1);

    pid = fork();

    if (pid < 0) {
        printf("error\n");
        exit(1);
    }

    else if (pid == 0) {
        close(fd[0]);
        write(fd[1], "I am child\n", 12);
        read(fd1[0], buf1, 7);
        write(1, buf1, 7);
    }

    else {
        close(fd[1]);
        read(fd[0], buf, 12);
        write(fd1[1], "parent\n", 7);
        write(1, buf, 12);
    }
    return 0;
}
```

Output

```
➜  OS ./pipe3
I am child
parent
```