beq → branch if equal
rd = rs + rd

lw $t1 [illegible] 32 ($to)
add 32 to the value in to
go to that memory location get value & store

[illegible] → lower byte at lower address
" " " higher address

memory

Byte Addressable Memory

2 # if ($2 == $1)
go to instruction 12

Comp Architecture Theory

memory address
register → 5 bits 00000 to 11111
$t2

lw $t, var1
no.1 $to var1 accessed in
Direct mode
(reg / memory location contain
actual value)

opcode operand
64
constant

Add $to, $to, $7
57 → Immediate addressing mode

offset / Base - displacement):

array: .word 2, 3, 4, 5, 9
la $to, array
lw $ 51, ($to)
lw $ 61, 4 ($to)
first element of array
second element of array

128 ($3)

- 05

addressing mode:

BEQ $3, $4, 12

Direct mode ADD R_1, (R_2) → Register R_2 contains the address of memory from where data is fetched
$R_1 \leftarrow R_1 + M[(R_2)]$

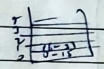
Direct Addressing mode → actual address given in instruction (used to access variables)
eg: lw $to, 400 / lw $to var1

beq → branch if equal
rd = rs + rt

lw $t1 32 ($to)
add 32 to the value in to
go to that memory location get value & store

Little endian → lower byte at lower address
Big endian → " " " higher address

Word Addressable memory



Byte Addressable Memory

beq $2, $3, 12 # if ($2 == $3)
go to instruction 12

Addressing modes:
ways to specify an operand or memory address
if 32 registers then operand → 5 bits 00000 to 11111

(i) Register Addressing mode: [opcode | operand] (operand → Register no.)
op, rs, rt, rd, ...
(operand is present in the register)
lw $t, var1
$to, var1 accessed in
(direct mode (reg/memory location contain actual value))

(ii) Immediate Addressing mode: [opcode | operand] (64)
ADDI $2, $3, #4
operand is directly provided as constant
add $to, $to, $7
57 → immediate addressing mode
* constant must fit in 16 bits of operand in instruction register

(iii) Base (Base - ~~Address~~ offset / Base - displacement):
LW $2, 128 ($3)
array: .word 2, 3, 4, 5, 9
la $to, array
lw $s1, ($to)
lw $s1, 4 ($to)
first element of array
second element of memory

(iv) PC relative addressing mode:
BEQ $3, $4, 12

(v) Register Indirect mode ADD R1, (R2) → Register R2 contains the address of memory from where data is fetched
R1 ← R1 + M[(R2)]

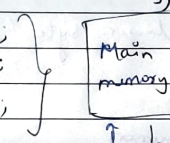
vi) Direct Addressing mode → actual address given in instruction (used to access variables)
eg: lw $to, 400 / lw $to var1

Dataflow model :- instruction is fetched & executed when the operands are ready
→ no instruction pointer
→ many executions at the same time (more parallel)

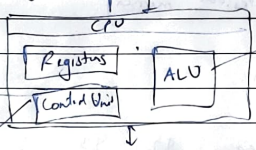
Von Neumann Architecture :- (Stored memory program)
(both data & program stored in memory)

sequential (by default by instruction pointer)

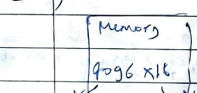
Data → int a = 10; b = 20;
Program c = a + b;

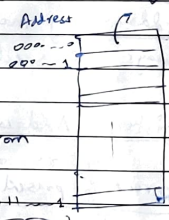


connected via Bus



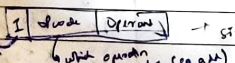
Registers : 8 bits / 16 bits, etc → very fast





Registers :

i) Address Register (AR) (12) (0 to $2^{12}-1$) (stored value from)
ii) Data Register (16) (0 to $2^{16}-1$)
iii) Accumulator (intermediate data) (16)
iv) Program Counter (12) (stores address)
v) Instruction Register (16)

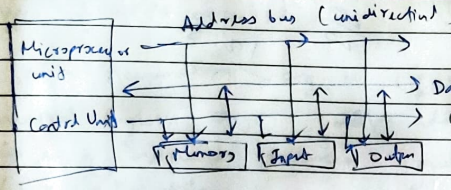


addressing mode → size depends on single accumulator / multiple accumulator
eg: ADD r1, r5, r4

vi) Temporary Register (16)
vii) Input reg (8 bit) } independent
viii) Output reg (8 bit) }

Single accumulator Organisation :
eg: LDA A // Load in accumulator
ADD B // AC ← AC + B
Store X ← AC
(for instruction X = A + B)
(more instructions)

CPU → data section (data path) + control section
datapath → collection of registers, ALU and interconnecting buses



How Basic Computer Works:

load data from memory and store in register

PC → Address Register → Memory Unit → Data Registers

Output device ← Output Register ← Accumulator ← ALU

temp. register

ALU result sent common bus

Types of Instructions

Transfer Instruction	Data Manipulation Instruction	Program Control Instruction
↳ Move (copy)	↳ Arithmetic	↳ if else
↳ load (memory to register)	↳ Logical	↳ loop, for, while
↳ store (register to memory)	↳ shift / conversion	(j, b) - jump to mark
↳ exchange		beq, blt, bgt
↳ input, output		(skip, call, return)
↳ push, pop		

* Performance = $\frac{1}{\text{Execution time}}$

* CPU time = CPU clock cycles × Clock Cycle Time

$= \frac{\text{CPU clock cycles}}{\text{clock rate}}$

$= \frac{\text{Instruction Count} \times \text{Cycles per Instruction}}{\text{clock rate}}$

Amdahl's Law

Overall speedup = $\frac{1}{(1 - \text{frac}_{\text{enh}}) + \frac{\text{frac}_{\text{enh}}}{\text{Speedup}_{\text{enh}}}}$ → % of original execution time affected by enhancement

Types of CPU organisat^n :

i) Single Accumulator org. (1 address)

ii) General Register Org. (multiple address eg 3, 2)
↳ no. of instructions reduces (multiple registers) (fast-)
↳ Instruction size increases, bus size increases
eg: ADD R_1, A, B
ADD R_2, C, D
MUL X, R_1, R_2
X = (A+B) * (C+D)

iii) Register Stack organisat^n (0 address)
Push : SP ← SP+1 ; M[SP] ← DR
Pop : DR ← M[SP] ; SP = SP-1
Push a
Push b
add
Push c
Push d
MUL
(a+b) * (c+d)

iii).ii) Memory stack organisat^n (0 address)
Memory is used as stack (cost effective)
Push : SP ← SP-1 ; M[SP] = DR (data register)
Pop : ~~SP ← SP+1~~ DR ← M[SP] ; SP ← SP+1
same

1 bit | 8 bits | 23 bits
S | E | F

more exponent bits → wider range
more fraction bits → more precision

Value = $(-1)^S \times F \times 2^E$ → sign & magnitude representation

Exponent represent $(-1)^S \times (1 + \text{fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

for single precision format bias = 127

Largest no. that can be represented = 2.0×2^{128}
smallest = 2.0×2^{-128}

HAVELLS

M T W T F S S
Page No.:
Date:
YOUVA

* Double precision format:

1 bit | 11 bits | 52 bits
S | E | F

eg: -0.75_{10} $\quad$ $2^{-1} + 2^{-2}$

binary : -0.11

normalized scientific notation : -1.1×2^{-1} (1 → fraction)

biased exponent = $127 + (-1) = 126$ → biased exponent

1 | 0111110 | 1000...0

$2^{(126-127)} = 2^{-1}$

eg. 1 | 1000 0001 | 0100...00

double precision

$-1.01 \times 2^{2} = -5$

ISA: agreed upon interface b/w software and hardware

→ what the software writer needs to know to write and debug system/programs (eg add instruction)

eg: x86, MIPS (a RISC ISA) CS321 (Microprocessor without Interlocked Pipeline Stages)

- Problem
- Algorithm
- Program
- ISA
- Microarchitecture
- Circuits
- Electrons

Microarchitecture

→ specific implementation of an ISA (eg adder implementation)

C —compiler→ Assembly language —Assembler→ Binary machine language

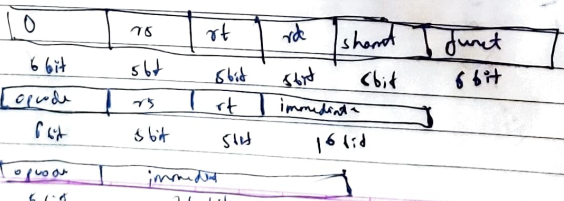
Early pipelined processors

i) Instruction fetch (PC → Instruction memory; PC = PC+1)
ii) Decode / Register read (rs, rt, rd)
iii) Execute (ALU)
iv) Memory (store result in memory)
v) Write back (data memory → register)

Instruction:

consists of: opcode → operat^n to be performed
Operands → on which the operation performs

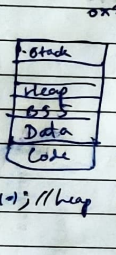
eg: MIPS ISA: (three instruction encoding formats)

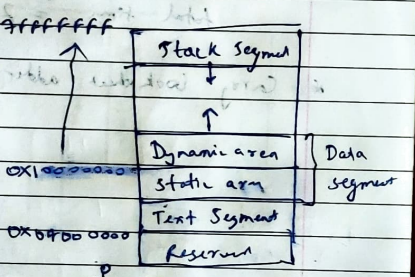


RISC	CISC
→ simple instruction	→ complex instruction
→ fixed length	→ variable length
→ uniform decoder	→ non-uniform decoder
→ few addressing modes	→ many addressing modes
eg: add, xor, multiply	eg: string copy, insert in disk

* MIPS is an example of RISC ISA.

```
int A; // BSS
int B = 10; // data
main() {
  int ALocal; // stack
  int * p; // stack
  p = (int*) malloc(10); // heap
}
```





* R type : add, sub, and, or, slt
  opcode : 000000
  → includes arithmetic & logic with all operands in registers, shift instr, jalr & jr

I type : addi, subi, andi, ori, beq, bne, lw, sw, lui
→ includes load and store, branch instructions,
→ opcode except 000000, 00001X, 0100XX

J type : j and jal
→ opcode : 0000 1X
→ requires a memory address to specify operand

// to call fn jal
// to return jr $ra

BEQ $2, $3, (12)
12 instructions prev to PC (offset)
(relative to next instruction)

* Stack push : addi $sp, $sp, -4
  sw $ra, 0($sp)
  pop : lw $ra, 0($sp)
  addi $sp, $sp, 4

lw $dst imm ($src)
($dst ← M[$src + imm]

M T W T F S S
Page No.:
Date:
YOUVA

* Signed magnitude representation & 1st complement. → 2 zeros
* slt $s1, $s2, $zero if ($s2 < 0) ∴ $s1 = 1
(true for s_1 = 11111111 (unsigned comparison)

* Ripple carry adder:
$S_i = A_i \oplus B_i \oplus C_i$.
$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$
Time taken per full adder = 2
total time = 2n

* Carry lookahead adder.

Post MidSem

M T W T F S S
Page No.:
Date:
YOUVA

$S_0 = 1$
$S_1 = 0$
$S_2 = 1$

MIPS ISA

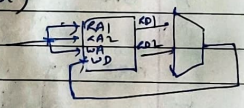
R | opcode (6 bit) | rs (5 bit) | rt (6 bit) | rd (5 bit) | shamt (5 bit) | funct (6 bit)

I | opcode (6 bit) | rs (5 bit) | rt (5 bit) | immediate (16 bit)

J | opcode (6 bit) | immediate (26 bit)

i) Rtype : add t_0 , t_1 , t_2
(rd) (rs) (rt)

→ perform operation on rs and rt and store the result back into register file (at location rd)



ii) I type : a) lw $t0 , 4($t1)
(rt) (rs)

→ compute memory address : SX [I[0,15]] + R[rs]
(sign extend to 32 bit)

R[rt] = M[X]

b) sw $t0, 4($t1)
M[X] = R[rt]

ii) beq $t0, $t1, offset

if (R(rs) = R(rt))
PC = PC + 4 + SL[SX[0,15]] → branch target address

shift left 2 (multiply by 4)

assumed PC is already PC+4

(iii) j offset (Jump instructions)

PC ← PC 31....28 || Off × 4
(pcbits) (28 bits)

PC ← (PC+4)[31..28] || SL2[I[25,0]]
(4bits)

↓
shift left 2 (multiply by 4)

Single Cycle design:

→ fetch, decode and execute all instructions combined in one clock cycle
→ no resource sharing (seperate IM and DM, 2 adders)
→ Cycle time determined by slowest instruction (lw)

Overall clock period:

max { R: t_I | $t_I + t_R + t_A + t_R$
SW: t_I | $t_I + t_R + t_A + t_M$
lw: t_I | $t_I + t_R + t_A + t_M + t_R$
Beq: $t_I + t_R$ | $t_I + t_R$ | $t_I + t_R + t_A$
J: t_I | t_I }

= max { $t_I + t_R + t_A + t_M + t_R$ | $t_I + t_R$ | $t_I + t_R$ }

I = instruction
A = Adder
* = ALU
R = Register file
M = Memory

Multicycle DataPath:

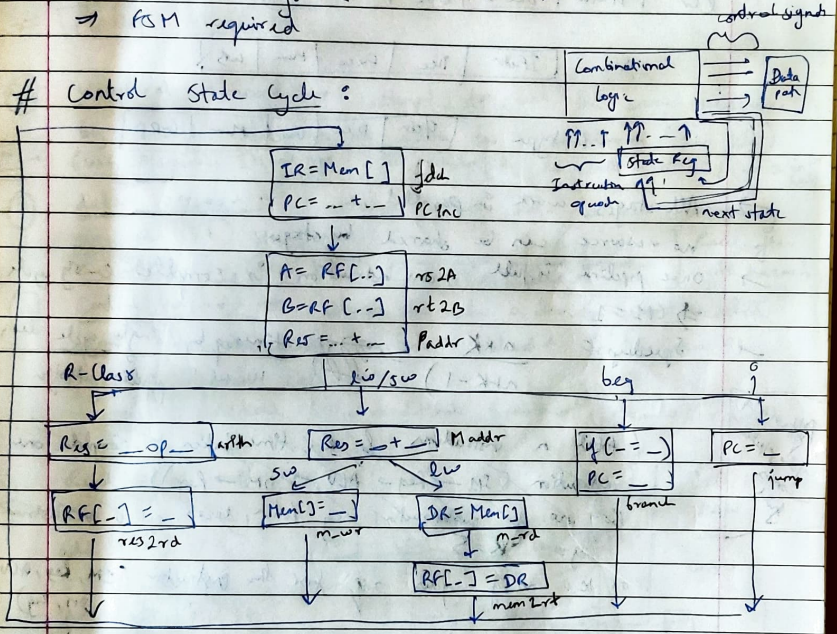
→ diff instructions take diff no. of cycles (load → 5 cycles, jump → 3 cycles)
→ each instruction broken into steps
→ need only one memory → one memory access per cycle
→ need only one ALU / adder → " ALU operation " "
(extra registers & mux required)
→ improved performance & resource sharing

M T W T F S S
Page No.:
Date:
YOUVA

* → Control signals for multicycle datapath not only determined by bits of instruction

→ opcode bits tell what operation to be done but not what instruction cycle to be done next
⇒ FSM required

Control State Cycle :



→ Each box is a state : total 10 states ⇒ 4 bits required
& 6 bits for OP
⇒ 10 bit (4+6) inputs to determine output (control signals)

* CPU Time = $\frac{\text{Instruction count} \times \text{Cycles per instruction}}{\text{Clock rate}}$
↳ no. of cycles per second

Pipelined MIPS Processor

→ start the next instruction (eg sw) before the current (other) one is completed ~~eg lw~~.
(while decode INS_i, fetch INS_{i+1})

lw | IFetch | Decode | Exec | Mem | WB |

sw | IFetch | Dec | Exec | Mem | WB |

R-type | IFetch | Dec | Exec | Mem | WB |

→ All stages work in parallel,
no resource can be shared by stages
→ Once pipeline is full, one instruction is completed every cycle
⇒ CPI = 1
→ Speedup = $\frac{n*K}{(n+K-1)}$ = ~~time~~ time req by single cycle datapath / " " " pipelined design

(total n instructions ~~& K~~ time taken to complete one instruction (IM → Reg → ALU → DM → Reg))
K = no. of subinstruction (reg fetch, decode, etc) in an instruction
eg: K = 5
or K = no. of cycles req for the instruction (IM, Reg, ALU, DM, Reg)

Fetch ← Reg/Dec → ← Exec → ← Mem → ← WR →

IF/ID reg | main control | ID/Ex reg | Ex/Mem reg | Mem/wr reg

control signals

IF → instruction fetch
ID → " decode
EX → execution
Mem → memory
WB → write back to register

→ main control generates control signals during reg / decode
→ Control signals for Exec (ExtOp, ALUSrc) are used 1 cycle later
→ " " " Mem (MemWr, MemRD) " " 2 " "
→ " " " Wr (MemtoReg, MemWr) " " 3 " "

Pipeline Hazards:

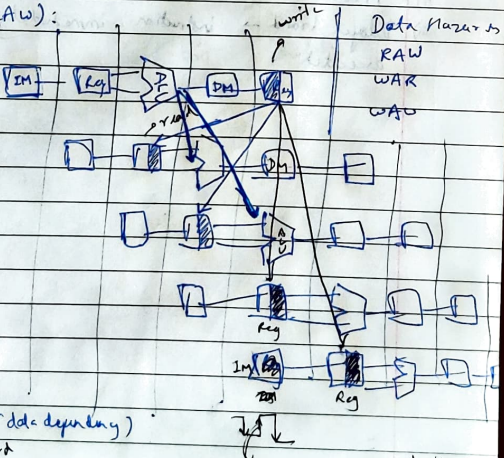
(i) Structural Hazards → attempt to use the same resource by 2 diff instruction at same time (eg memory / Reg file)
sol^n → use separate IM and DM ; for register file, do reads in 2nd half, write in 1st half

(ii) data hazard → attempt to use data before its ready
add $t1, $t0, $t2
# add $t3, ($t1), $t2 → source register for next instruction value yet not calculated
} Read after write hazard

(iii) Control hazard → attempting to make a decision about control flow before the condition has been evaluated & the new PC target address calculated.
(eg conditional & unconditional branches / calls / returns)

pg Read after write (RAW):

add ($t1)
sub $t4, ($t1), $t5
and $6, ($t1), $7
or $8, ($t1), $5
xor $4 ($t1), $5



Data Hazards
RAW
WAR
WAW

→ solution (data forwarding)
→ hazard (data dependency)
→ no hazard (data ready)

clock edge that controls register writing

Solution to RAW hazards: i) stall (halt) (2 cycles)
ii) Reordering
iii) forwarding

→ Similarly load use hazard
(dependencies backward in time causes hazards)

lw $t1, 4($t2)
sub $t4, $t1, $t5

Even after forwarding one stall cycle required (halt)

→ Other data hazards:

add $t1, _ , _ lw $t1, _
sw $t1 sw $t1, _

resolved by data forwarding

Control Hazards:
Which destination to choose → PC = PC + 4 + Immediate
or PC = PC + 4

Solution:
i) stall → stop loading instructions until result (destination address) is available
ii) Predict → assume an outcome & continue fetching (undo if wrong)
iii) Delayed branch → instruction immediately following branch is always executed

05

M T W T F S S
Page No.:
Date:
YOUVA

→ Microprocessor / Processor, ISA, Motherboard, Core, L1, L2, L3, GPU, threads, process, program, interrupt, bus

→ User mode → kernel mode to perform I/O operation. (System calls)

→ DMA → input directly from external device to memory (CPU not involved)

→ Nachos Virtual OS / PINTOS / XV6

→ Thread → lightweight process
Process → a program in execution (program is static) (process is dynamic)
Each thread has its own stack, registers associated with it.

→ Context switch of process more expensive than context switch of thread. So multithreaded preferred, instead of single threaded process.

→ Processes do not share resources well
Threads share resources like code section, data section, OS resources (but has own PC, registers, stack).

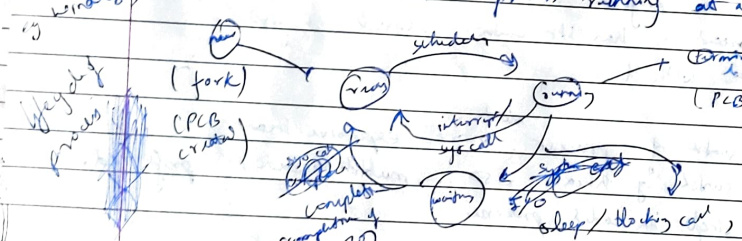
→ 8 → A
16 → AX
32 → EAX
64 → RAX
bit processor

OS design:

i) Monolithic
ii) Layered
iii) microkernel

→ Each process / job is internally represented by the OS by a Process control Block (PCB)
Contains process state (ready / running / waiting), PC, registers, Scheduling info, etc.

→ Multiprogramming → Job1 → Job2 → Job3
Multitasking → jobs switched frequently on process running at a time



→ Job Queue / Ready Queue
Long term scheduler (new → memory)
Short term scheduler (CPU scheduler) (memory → CPU)

Medium term scheduler → swapps process in execution (not complete) which can later be reintroduced

→ Context switching → save state into PCB & load the other

→ Process creation another process (may obtain resources from parent or from OS)
(Parent) / (child)
Root → 1st process (root) & all other child

thread ← task (Linux)

i) running → waiting
ii) running → ready
iii) waiting → ready
iv) terminate

non-preemption / others preemption

non-preemption scheduling:

Process remains in CPU until terminated / switches to waiting state.

Preemption → process can go from running (leave from CPU) to ready queue

Amdahl's law:

$$S = \frac{1}{(1 - \text{fraction enhanced}) + \left(\frac{\text{fraction enhanced}}{\text{factor of improvement}}\right)}$$

(speed up) (max speed up)

$$S = \frac{1}{1 - f_E + \left(\frac{f_E}{f_I}\right)}$$

Max speedup $= \frac{1}{1-p}$ where p = parallel portion

eg: 40% of code can be enhanced to run 2.3 times faster

$f_E = 0.4$

$f_I = T_E / T_E$ (factor of improvement)

$S = \frac{T + T_E}{T + T_E}$ (part which can be improved)

Threads share code section, data section & OS resources (open files, signals) with own threads (no protection b/w threads)

separate PC, registers and stack space

In multicore CPU, multiple threads / processes can run in parallel

In single core CPU, one process running at a time but concurrent ~~(not parallel)~~ (logical parallelism) through time sharing.

User mode (mode bit = 1) —syscall, interrupt, exception, exit→ Kernel mode (mode bit = 0) / supervisor mode / protected mode

(privileged instruction can only be executed in kernel mode)

→ Interrupt → asynchronous, caused by external device
Exception → synchronous, unexpected problem with instruction
Trap → synchronous, intended transition to OS due to instruction

→ Exception Causes:
hardware interrupt, system call, divide by 0, undefine instruction, arithmetic overflow

→ System calls:

fork(), wait pid, execve, exit
(create child process) (wait for child to terminate) (replace a process's core image)

open(), close(), read (fd, buffer, nbytes), write()
lseek(), stat()

printf() in C calls write() system call

→ Booting:
→ address of 1st instruct. is fixed (in ROM)
→ loads the BIOS (Basic I/O system)
→ BIOS perform POST (power on self test) 1st physical sector
→ BIOS loads MBR from boot device (e.g. floppy disk / hard disk)
→ BIOS loads the OS loader (e.g. LILO, GRUB)
→ The second loader loads the kernel image
→ sets the kernel mode
→ Jumps to OS entry point

M T W T F S S
Page No.:
Date:
YOUVA

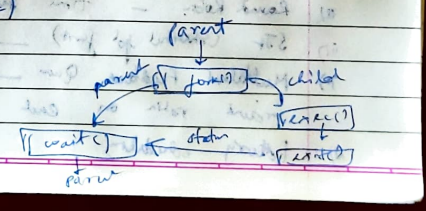
→ **PCB**: data str. to keep track of process's state identified by PID

→ Process consist of i) address space (code + static data + heap data + stack)
ii) CPU state (PC, stack pointer, other registers)
iii) OS resources (open files, network connections, etc)

PCB contains all these + PID of parent PID, execution state (ready, running, etc), UserID, group id, scheduling priority, accounting info, pointers for state queues (ready queue / waiting queue etc)

→ **Context switch**:
Job 1 executing ← interrupt / sys call ← Save state of Job1 into its PCB (registers, SP, PC)
Job 2 executing ← load SP, register, PC from Job 2's PCB

Creating Process:
→ fork() system call. used to create child process
→ child same as parent (registers, variables, etc) except return value (for child return value = 0, for parent = PID of child)
→ Parent can wait for child to terminate (wait() system call) or continue execution
→ Child starts a new program (different) via exec system call
do ps -ef [process tree]



→ When process created OS dynamically allocates PCB for it, then PCB initialized & put into correct queue (ready, waiting, I/O queue, etc)
PCB moved from one queue to another queue as process completes
When process ends, OS deallocates PCB (may wait for a while to receive signals)

Process Scheduling Scheduler → dispatcher → CPU

→ Time taken by dispatcher to switch contexts ~ dispatch latency

→ Throughput → No. of processes completed / time
→ Turnaround time → time to run a process from initialization to termination (including waiting to
→ CPU Utilization → % of time CPU is busy
→ Waiting time → time a process is in ready queue
→ Response time:
time b/w when a process is ready to run and its next I/O request

Interactive → (minimize response time) Batch → FCFS (minimize turnaround time)

→ Scheduling Algo:

i) FCFS — non-preemptive
ii) Round Robin — time slice & preemption
iii) SJF (shortest job first) → throughput ↑
iv) Multilevel Feedback Queue → round robin on each priority queue
v) Lottery scheduling

LINUX

M T W T F S S
Page No.:
Date:
YOUVA

→ Unix based Kernel (Linux + GNU)
→ Multiple distributions → red hat enterprise, fedora, debian
others → Ubuntu, CentOS
→ easy software updates
→ no malware (more secure) (more reliable)
→ free & source code available

Kernel: ~ OS : memory management, task scheduling, file management

→ User communicate with the OS through program called shell
→ Shell is command line interpreter (user commands converted into language which is understood by kernel.
→ Shellscript : list of commands listed in order of execution
→ Bash: ~~most~~ widely used shell (Bourne Again Shell) for scripting and interfacing purposes

Other shells : C Shell, Korn Shell

Some commands (Basic) :

ls, ls -l, pwd, cd, mkdir, rm, rmdir, cp, mv, chmod 777 f1.txt, sudo apt-get install <package>, sort, grep, man <command>, cat, touch, echo
vi f1.txt, clear, history (all previous commands list)
locate -i hello # find file name hello & locate

eg: grep he f1.txt , sort f1.txt
(pattern) (file name) # similar to ctrl+F,
cat -n f1.txt (print file with line no)
cp <file name> <destination address>
history 10 (show last 10 commands used)

Some devop commands:

→ whoami

→ Switch user: su seed # switch to seed user

To switch to root user: su

or sudo bash

→ sudo useradd -u 211 # to add user

sudo passwd -u 211 # to set password

sudo userdel -u 211

sudo groupadd teachers # add group

→ chown root f1.txt (use sudo if permission denied)

change owner to root

→ chgrp <program> f1.txt # change group of a file

→ chmod u+x f1.txt # add execution permission for user

→ id # display user id, groupid, etc

id root # for a specific user

→ tar -xvf tar-file # unzip a file

→ ip, env, ssh, awk, tr, free, top

→ sudo gives root privileges to a user.

When a new user created, no sudo privileges given but can be given sudo privileges later.

(In my OS, seed user has all sudo privileges)