① O_CREAT
② O_RDONLY ③ O_WRONLY

Date:_____

→ int fd = open(): system call returns file descriptor (fd)
→ file descriptor is an int, may be considered as a 'capability', allowing are to access files.

```
prompt > echo hello > foo
prompt > cat foo
hello
prompt >
```
] an example of reading & writing foo.

How does the cat program access the file foo?
We can use strace to trace the system calls made by program.

```
prompt > strace cat foo
...
open ("foo", O_RDONLY | O_LARGEFILE)   = 3   → file descriptor.
read (3, "hello\n", 4096)              = 6   length of string.
write (1, "hello\n", 6)                = 6   → size of buffer.
hello
read (3, "", 4096)                     = 0   // no bytes left in file.
close (3)                              = 0
...
prompt >
```

standard output.

*file descriptor helps identify the file uniquely within same process.

Date:_____

→ each running process already has three files open (i) standard input[0] (ii) standard output [1] and (iii) standard error [2].

→ "cat" stands for concatenate.

- read (file desc., buffer pointer, size of buffer) the value it returns is length of string in above example "hello\n" is of 6 length.
- write (file desc. buffer pointer, size of buffer) has file desc. 1 for standard output and it also returns value 6 the size of string.
- close (3) closes the file with file desc 3.

What if we want to read from random offset within a document?

?! off_t lseek (int fd, off_t offset, int whence);

- fd → file descriptor
- offset → offset in bytes
- whence → determine how seek is performed
  - → from start (SEEK_SET)
  - → from current (SEEK_CUR)
  - → from end (SEEK_END)

fork()

→ here a parent process creates a child process with fork()

→ the parent then waits for child to complete.

→ finally the parent continues from where the child completed its work.

(58-63)

Date:_____

(Refer to slide 57)

int fd=open("file.txt", O_RDONLY) → open filing in read only mode, returns fd if succesfully opened.

assert (fd>=0) → if file doesn't open, program would terminate with assertion failure.

int rc=fork() → to create new process/child process returns 0 & returns PID of child process in parent process.

if (rc==0){ ... } → creation of child process.

else if (rc>0){ ... } → parent process, the wait(NULL) function called making parent process to wait for child function.

Date:_____

## RENAMING FILES

rename (char * old, char * new)

→ rename a file to a different name.

→ it implemented as an atomic call.

prompt > mv foo bar. // change from foo to bar.

flushes all modified file data & metadata from buffers in RAM to physical disk

```
int fd = open ("foo.txt", O_WRONLY | O_CREATE | O_TRUN
write (fd, buffer, size);
fsync (fd);
close (fd);
rename ("foo.txt", "foo.txt.tmp")
```

(the order of flags does not matter.)

→ to get information about files.

stat(), fstat() – show file metadata.

→ The stat information is displayed in a inode structure.

metadata is information about each file.

## REMOVING FILES

rm is linux command to remove file.

→ rm call unlink() to remove file.

Date:_____

## MAKING DIRECTORIES

mkdir( ) - make directory

→ when a directory is created, it is empty

→ empty directory have two entries . (itself) , .. (parent)

```
prompt> ls -a
./   ../
```

## DELETING DIRECTORIES

rmdir( ) - delete directory

→ requires that directory be empty

→ if you call rmdir( ) to a non-empty, it will fail

## HARD LINKS

link(old pathname , new one) - link new file to old.

→ create another way to refer to same file.

→ the command-line link program : ln

→ the way link works:

→ create another name in directory

→ refer it to same inode number of original file.

→ the file is not copied anyway

→ Now we ~~have~~ now just have two human names (file and file2) that both refer to same file.

* a hard link is an additional name for an existing file on same filesystem.
  → direct reference to file's inode.
* symbolic link is a special type of file that serves as a reference or pointer to another file/directory.
  → has their own inodes and store path to the target file.

Date:_____

→ The result of link()
  → two files have same inode number, but two human name (file, file2)
  → there is no difference b/w file and file 2.

→ Thus, to remove a file, we call unlink()
  → when unlink() is called reference count decrements
  → if reference count reaches zero, the filesystem free the inode and related data & blocks → truly "delete" the file.

## SYMBOLIC LINKS

→ Symbolic Links is more useful than Hard Link.
  → Hard link cannot create to directory
  → Hard link cannot create to file to other partition.

What is the difference b/w Symbolic Link and Hard Link?
→ Symbolic links are a third type file system knows about.
→ The size of symbolic links (file2) is 4 bytes.
→ A symbolic link holds the pathname of linked-to file as data of link file.
→ if we link to larger pathname, our link file would be bigger.

* hard links remains valid if original file is deleted; file data not removed until all links are deleted. Symbolic links become 'dangling' if target file is moved/deleted as they store path to file.

Date:

→ Dangling reference.
→ when remove original file, symbolic link points noting.

## MAKING AND MOUNTING FILE SYSTEM

mkfs tool : make file system.
→ write an empty file system starting with a root directory, on to disk partition.
→ Input:
→ a device (such as disk partition e.g /dev/sda1)
→ a file system type (e.g ext3)

mount()
→ take an existing directory as a target mount point
→ essentially paste a new file system into directory tree at point

## CHAPTER 40 : File System Implementation

→ There are two different aspects to implement filesystem.
① Data Structures.
* What types of on-disk structures are utilized by the file system to organize its data & metadata?
② Access Methods
* How does it map the calls made by process as open(), read() and write() etc.

Date: _____

* Which structures are read during the execution of particular system call?

→ Divide the disk into blocks.
  → block size is 4KB.
  → the blocks are addressed from 0 to N-1.


→ filesystem has to track which data block comprise a file, the size of file, it's owner etc.

How we store these inodes in file system? → stores information about file/directory, contains metadata about files/directories

→ reserve some space for inode table.
  → this holds an array of on-disk inodes.
  → Ex) inode tables: 3~7, inode size: 256 bytes.
  → 4-KB block can hold 16 inodes.
  → the filesystem contains 80 inodes (maximum num of files)

→ The Bitmap is to track whether inodes or data blocks are free. or allocated.

Date:_____

→each ~~bitmap~~ bit indicates free (0) or in-use (1)
→ data bitmap: for data region
→ inode bitmap: for inode table. (tells which inodes are free and which are allocated)

→ Superblock contains this information for particular file system
→ Ex) The number of inodes, begin location of inode table etc.

→ Each inode is refered to by inode number.
→ filesystem calculate where inode is on disk.
→ Ex) inode number: 32.
① Calculate offset into the inode region (32 x sizeof(inode) (256 bytes) = 8192.
② Add start address of inode table (12KB) + inode region (8KB) = 20KB.

blk : (inumber * sizeof(inode)) / blocksize
sector: ((blk * blocksize) + inodeStratAddr ) / sectorsize.

inode have all of the information about a file.
→file type (regular file, directory, etc)
→ size, their number of blocks allocated to it
→ protection information (who owns the file, who can access, etc).
→ time information.

Date:_____

→ To support bigger files, we use multi-level index.

→ Indirect pointer points to a block that contains more pointers

→ inode have fixed number of direct pointers (12) and single indirect pointer:

→ if a file grows large enough, an indirect block is allocated, inode's slot for an indirect pointer is set to point to it

→ (12+1024) x 4K or 4144KB.

→ Double indirect pointer. points to a block that contains indirect blocks

→ Allow file to grow with an additional 1024 x 1024 or 1 million 4KB blocks.

→ Triple indirect pointer. points to block that contains double indirect blocks

→ Multi-level index approach to a pointing to file blocks.

→ Ex) twelve direct pointers, a single and a double indirect block.

→ over 4GB in size $(12 + 1024 + 1024^2) \times 4KB$.

→ Many file system use multi-level index

→ Linux EXT2, EXT3, NetApp's WAFL, Unix file system.

→ Linux EXT4 use extents instead of simple pointers.

Date:_____

## DIRECTORY ORGANIZATION

→ directory contais a list of (entry name, inode number) pairs

→ each directory has two extra filles ". dot" for current directory and ".. dot dot" for parent directory.

→ for example, dir has three files (foo, bar, foobar)

| inum | reclen | strlen | name |
|---|---|---|---|
| 5 | 4 | 2 | . |
| 2 | 4 | 3 | .. |
| 12 | 4 | 4 | foo |
| 13 | 4 | 4 | bar |
| 24 | 8 | 7 | foobar |

→ filesystem track which inode & data block are free or not

→ in ader to manage free space, we have two simple bitmaps.

→ When the file is newly created, it allocated inode by searching the inode bitmap and update on-disk bitmap.

→ Pre-allocation policy is camonaily used for allocate cautinguos blocks.

Date:_____

## READING FILE FROM DISK

open ("/foo/bar", O_RDONLY)

→ traverse the pathname & thus locate desired inode.

→ begin at the root of fllesystem (/)

→ in most Unix file systems, the root inode number is 2. ** important.

→ filesystem reads in block that contains inode number 2.

→ look inside of it to find pointers to data blocks, ~~it with~~ (contents of root).

→ by reading in one or more directory data blocks it will find "foo" directory.

→ traverse recursively the path name until the desired inode ("bar")

→ check finale permissions, allocate a file descriptor for this process & returns file descriptor to user.

→ Issue read() to read from the file.

→ read in the first block of file, consulting the inode to find location of such block.

→ update the inode with a new last accessed time.

→ update in-memory open file table for file descriptor, the file offset.

Date:_____

→ When the file is closed
  → file descriptor should be deallocated, but for new, that is all file system really needs to do.
  No disk I/O take place.

→ Issue write() to update file with new contents
→ File may allocate a block (unless the block is being written)
→ need to update data block, data bitmap.
  → generates five I/Os.
    (1) to read data bitmap
    (2) to write bitmap
    (3) two more to read & write inode.
    (4) one to write actual block itself.

## CACHING & BUFFERING

→ Reading & writing files are EXPENSIVE, incurring many I/Os.
  → for e.g long pathname
    → literally perform hundreds of reads just to open file.
→ In order to reduce I/O traffic, file systems aggressively use system memory (DRAM) to cache.
  → modern system use dynamic partitioning approach, unified page cache.
→ Read I/O can be avoided by large cache.

Date:_____

→ Write traffics has to go to disk for persistent, Thus, cache does not reduce write I/Os.
→ filesystem use write buffering for write performance benefits
  → delaying writes
  → by buffering number of writes in memory, filesystem can then schedule the subsequent I/Os.
  → by avoiding writes.

→ Some application force flush data to disk by calling fsync() or direct I/O.

## CHAPTER 4: The Abstraction: The Process.

→ CPU virtualizing
  → The OS can promote the illusion that many virtual CPUs exist
  → Time Sharing: Running one process, then stopping it and running another.
    → The potential cost is performance.

### PROCESS

A process is a running program.

→ comprising of a process:
  → Memory
    → instructions
    → data section
  → Registers
    → program counter.
    → stack pointer.

Date:_____

## Process API

→ These APIs are available on any modern OS.

→ Create

→ Create a new process to run a program.

→ Destroy

→ Halt a runaway process

→ Wait

→ Wait for process to stop running

→ Miscellaneous Control.

→ Some kind of method to suspend a process and then resume it.

→ Status

→ Get some status info about process

## PROCESS CREATION

① Load a program code into memory into the address space of process.

→ programs initially reside on disk in executable format

→ OS perform loading process lazily.

→ loading pieces of code or data only as they are needed during program execution.

★ The code/text segment is a region of ~~processors~~ process's virtual address space reserved for holding executable code.

Date: _____

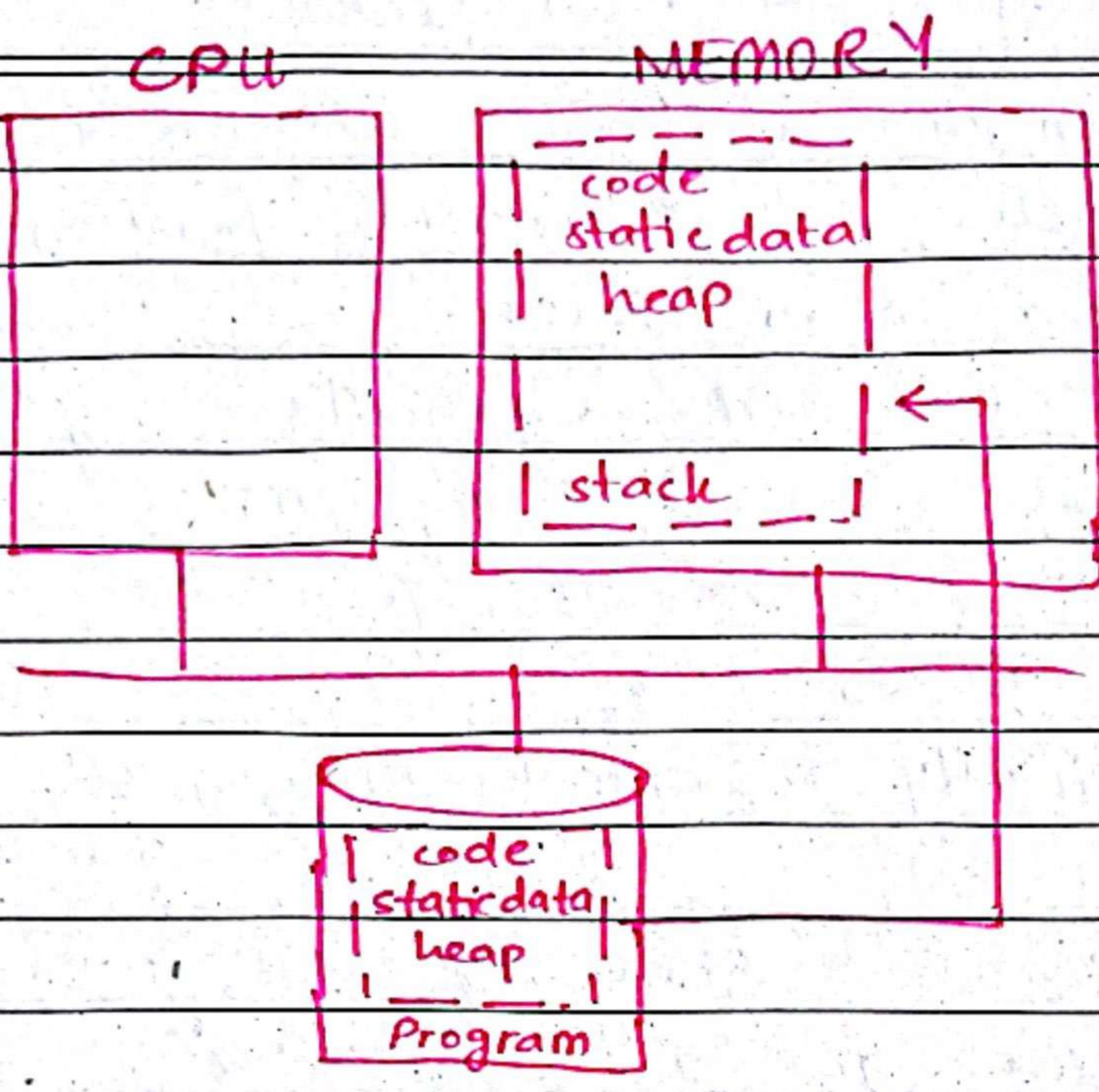
(2) The program's run-time stack is allocated.
→ Use the stack for local variables, function parameters and return address.
→ Initialize the stack with arguments → argc and argv array of main() function.

(3) The program's heap is created.
→ Used for explicitly requested dynamically allocated data.
→ Program request such space by calling malloc() and free it by calling free().

(4) The OS do some other initialization tasks.
→ input/output (I/O) setup.
→ each process by default has three open file descriptors
→ standard input, output and error.

(5) Start the program running at entry point, namely main()
→ OS trasfers control of CPU to newly-created process.

Date:_____




Loading:
Takes on-disk program & reads it into address space of process.

→ A process can be one of three states

→ Running

→ A process is running on a processor.

→ Ready

→ A process is ready to run but for some reason the OS has chosen not to run it at this given moment.

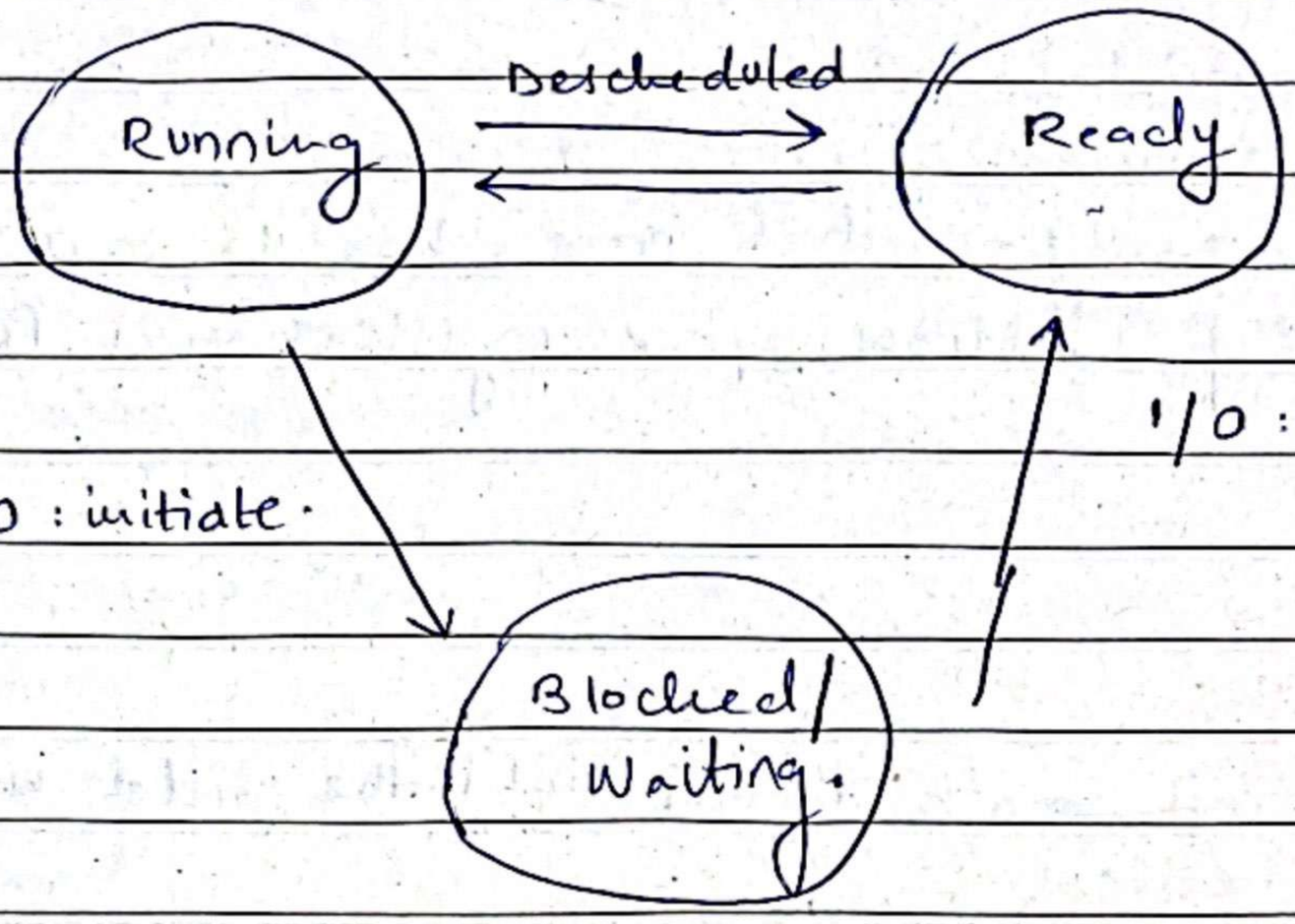
→ Blocked

→ A process has performed some kind of operation.

→ When a process initiates an I/O request to disk, it becomes blocked and thus some other process can use the processor.

↗ a process in running state will go to ready if interrupt is raised.

Date: _____




→ The OS has some key data structures that track various relevant pieces of information.
  → Process List
    → Ready processes.
    → Blocked process
    → Current running process
  → Register context.

→ PCB (Process Control Block)
  → A C-structure that contains information about each process.

* understanding of codes (refer to slide).

CHAPTER 5: Intelude : Process API → Date: _____

fork () system call

→ create new process

→ the newly-created process has it's own copy of address space, registers and PC.

wait() system call

→ this system call won't return until the child has run and exited.

~~system~~ exec() system call

→ run a program that is difficult from calling program.

CHAPTER 6: Mechanism : Limited Direct Execution.

How to efficiently virtualize the CPU with control?

→ The OS needs to share the physical CPU by time sharing

→ Issue

- Performance: How can we implement virtualization without adding excessive overhead to the system?
- Control: How can we run processes efficiently while retaing control over CPU?

Date:_____

## DIRECT EXECUTION

→ Just run the program directly on the CPU.

| OS | Program |
|---|---|
| ① Create entry for process list | |
| ② Allocate memory for program. | |
| ③ Load program into memory | |
| ④ Set up stack with argc/argv | |
| ⑤ Clear registers | |
| ⑥ Execute call main() | |
| | ⑦ Run main() |
| | ⑧ Execute return from main() |
| ⑨ Free memory of process | |
| ⑩ Remove from process list | |

Without limits on running programs, the OS wouldn't be in control of anything and thus would be "just library"

An interrupt within OS occurs 3 times:

① Program misbehaves falls into segmentation fault.
② During input/output.
③ system calls.

→ BIOS loads OS and also loads Interrupt Distrupt Table (IDT)
→ IDT knows where system call is.

→ Interrupt Distrupt Table (IDT)

## PROBLEM 1 Restricted Operation.

→ What if a process wishes to perform some kind of restricted operation such as ...
→ issuing an I/O request to disk
→ gaining access to more system resources such as CPU or a memory.

## SOLUTION

Using protected control tranfer.
→ **User Mode:** Applications do not have full access to hardware resources.
→ **Kernel Mode:** The OS has ~~full~~ access to the full resources of machine.

## SYSTEM CALL.

→ Allow the kernel to carefully expose certain key pieces of functionality to user program, such as ...
↦ accessing the filesystem
↦ creating and destroying processes.
↦ communicating with other processes
↦ allocating more memory.

User Mode
Kernel Mode (highest priviledge)

→ In Direct Execution, instruction directly goes to CPU and there's nothing which can stop it.

Q) how is function call different from system call?

→ you know where Date.function call exists but don't know where system call exists.

TRAP → OS is right now just a massive library of functions.

→ jump into the kernel.

→ raise the priviledge level to kernel mode

RETURN-FROM-TRAP

→ return into calling user program.

→ reduce the priviledge level back to user mode.

PROBLEM 2 Switching Between Processes.

→ How can the OS regain control of the CPU so that it can switch b/w processes?

→ a cooperative approach: Wait for system calls

→ a non-cooperative approach: The OS takes control.

COOPERATIVE APPROACH (when control is given to OS)

→ Processes periodically give up the CPU by making system calls such as yield

→ The OS decides to run some other task.

→ Application also transfer control to the OS when they do something illegal.

→ Divide by zero

→ try access memory that it shouldn't be able to access.

A process gets stuck in an infinite loop

→ Reboot the machine.

Date:

## NON-COOPERATIVE APPROACH

→ A timer interrupt
  → during boot sequence, the OS start the timer.
  → The timer raise an interrupt every so many ~~middl~~ milliseconds.
→ When the interrupt is raised:
  → The currently running process is halted
  → Save enough of the state of the program.
  → A pre-configured interrupt handler in OS runs.

A timer interrupt gives OS the ability to run again on a CPU.

## SCHEDULER (Saving & Restoring Context)

→ Scheduler makes decision
  → whether to continue running the current process or switch to different one.
  → if the decision is made to switch, the OS executes context switch

→ save pro register of old process and load new-

Date:_____

## CONTEXT SWITCH

→ A low-level piece of assembly code.

→ Save a few register values for the current process onto it's kernel stack

→ general purpose registers

→ PC

→ Kernel Stack Pointer

→ Restore a few for the soon-to-be-executing process from it's kernel stack

→ switch to the kernel stack for soon-to-be-executing process.

# CHAPTER 7: Scheduling.

## Purpose of Scheduling.

Manages execution of processes or threads on a CPU to optimize certain performance metrics.

## Goals

① Minimize CPU utilization ② Minimise response time ③ Minimize turnaround time.

## Workload assumptions

→ each job runs for same amount of time.

→ all jobs arrive at same time.

→ all jobs only use CPU (ie they perform no I/O)

→ the run-time of each job is known.

* time from when the job arrives to first time it is scheduled.

$T_{response} = T_{firstrun} - T_{arrival}$

* Decreasing Response Time increase turnaround time.

Date: _____

- Turnaround Time: Total time taken from submission to completion of process.
- Response Time: Time from submission to first response produced.
- Throughput: Number of processes completed per time unit

$T_{turnaround} = T_{completion} - T_{arrival}$.

* the time at which the job completes minus the time at which the job arrived in the system.

→ Another metric is fairness

→ performance and fairness are often at odds in scheduling..

## SCHEDULING ALGORITHMS

① First Come, First Served (FCFS)

→ very simple & easy to implement.

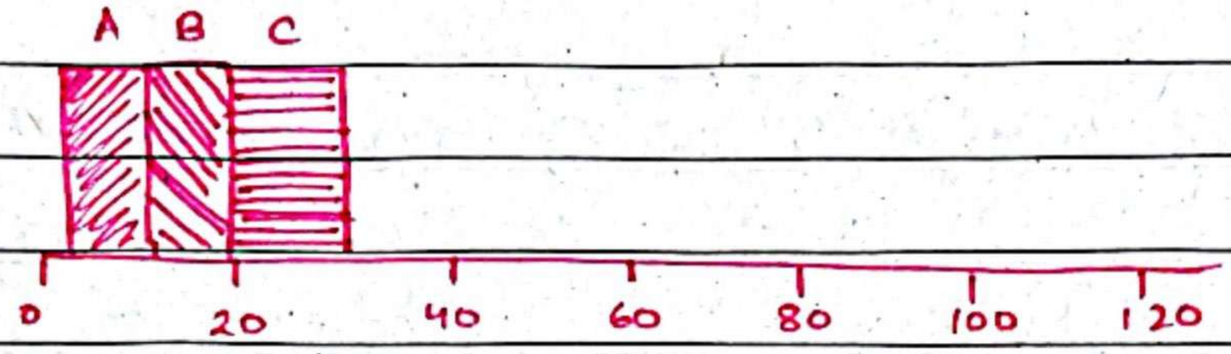
→ processes are scheduled in the order they arrive.

**Example**

→ A arrived just before B which arrived just before C.

→ each job runs for 10 seconds.

Date:_____




Average turnaround time = $\frac{10+20+30}{3}$ = 20 sec.

→ let's relax assumption 1: Each job no larger runs for the same amount of time.

Example

→ A just arrived before B which arrived just before C.

→ A runs for 100 seconds, B and C runs for 10 each.

Average turnaround time = $\frac{100+110+120}{3}$ = 110 sec.

② Shortest Job First (SJF).

→ executes the shortest upcoming job next.

→ run the shortest job first, then next shortest and so on.

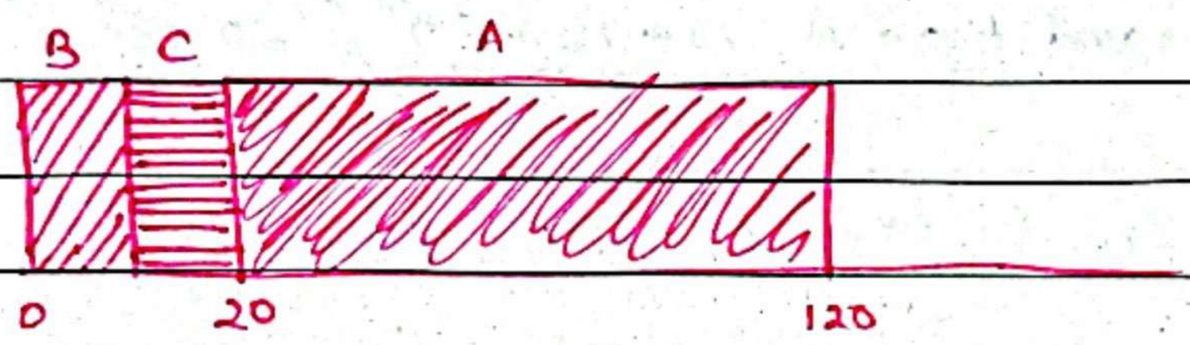
→ NON PREEMPTIVE SCHEDULER.

Date:

**Example**

→ A arrived just before B which arrived just before C

→ A runs for 100 seconds, B and C run for 10 each.



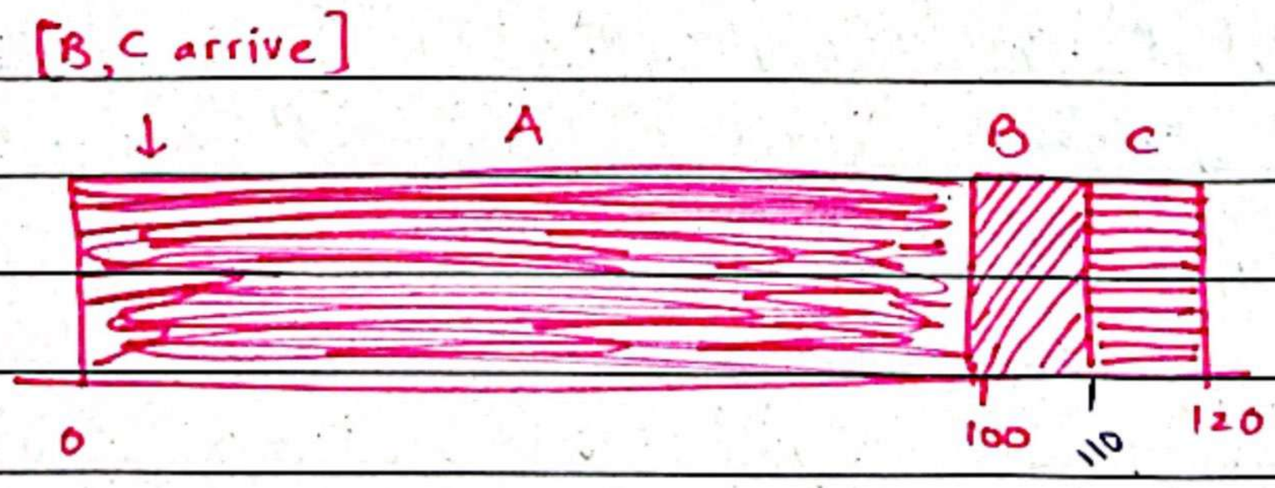

$$\text{Turnaround time} = \frac{10 + 20 + 120}{3} = 50 \text{ sec.}$$

→ Let's relax assumption 2: Jobs can arrive at any time

→ **Example:**

→ A arrives at t=0 and needs to run for 100 seconds.

→ B and C arrive at t=10 and each need to run for 10 seconds.



★ STCF optimizes turnaround time, RR optimizes response time.

Date:_____

$$\text{Average turnaround time} = \frac{100 + (110-10) + (120-10)}{3} = 103.33 \text{ sec}$$

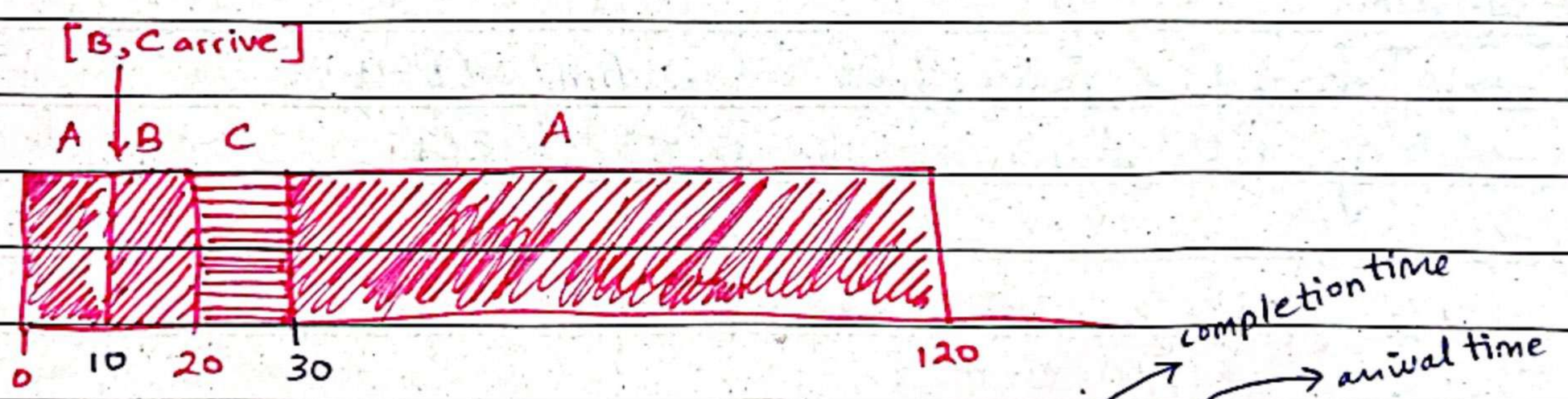
(3) Shortest Time-to-Completion First (STCF)

→ Preemptive version of SJF, where current running process can be interrupted.

Example

→ A arrives at t=0 and needs to run for 100 seconds.

→ B and C arrive at t=10 and each need to run for 10 seconds.




$$\text{Average turnaround time} = \frac{(120-0) + (20-10) + (30-10)}{3} = 50 \text{ sec.}$$

(4) Round Robin (RR) Scheduling

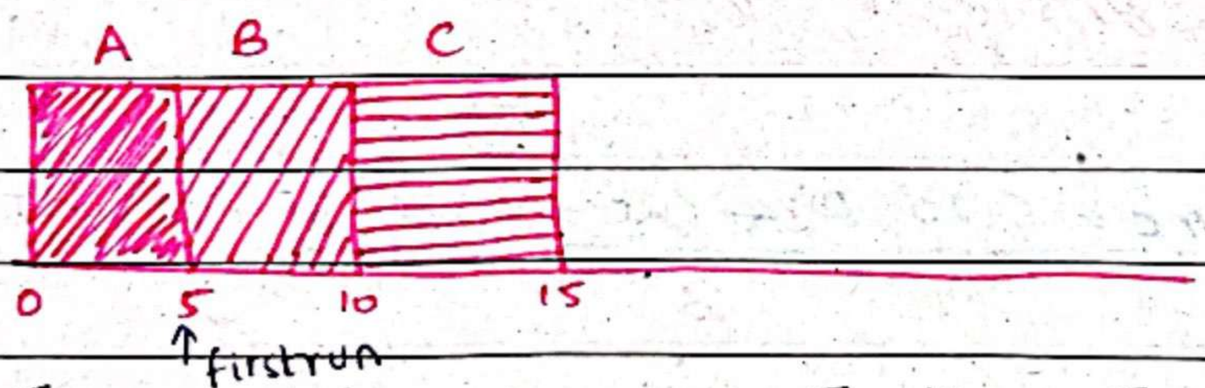
→ each process receives a fixed time slot, rotating through queue.

Date: _____

→ Time Slicing
  → run job for a time slice and then switch to next job in the run queue until the jobs are finished.
    → time slice is sometimes called scheduling quantum
  → it repeatedly does so until the jobs are finished
  → The length of time slice must be multiple of timer-interrupt period.

RR is fair, but performs poorly on metrics such as turnaround time.

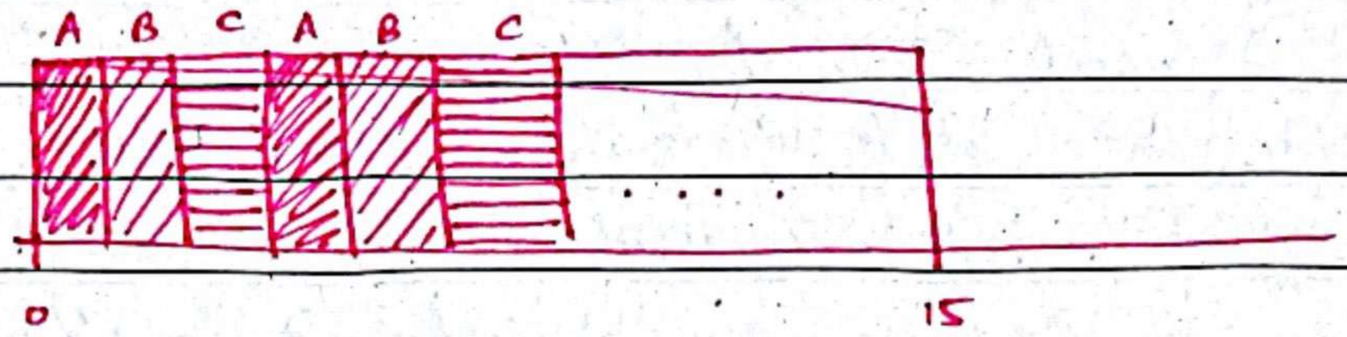
Example
→ A, B and C arrive at same time. (t=0)
→ they each wish to run for 5 seconds.




$T_{average\ response} = \frac{0+5+10}{3} = 5\ sec.$

$= T_{firstrun} - T_{arrival}$

Date: _____




$$T_{average\ response} = \frac{0+1+2}{3} = 1\ sec.$$

(RR with time-slice of 1 sec - Good for Response Time)

The length of time slice is critical.

→ The shorter time slice
  → better response time
  → the cost of context switching will dominate overall performance

→ The larger time slice.
  → amortize the cost of switching
  → worse response time.

Deciding on length of time slice presents a trade-off to a system designer.

Date:_____

→ let's relax assumption 3: All programs perform I/O

Example

→ A and B need 50 ms of CPU time each.

→ A runs for 10ms and then issues an I/O request

→ I/O ~~each~~ each take 10ms.

→ B simply uses CPU for 50ms & performs no I/O.

→ The scheduler runs A first, then B after.

## CHAPTER 8: Scheduling: Multilevel Feedback Queue.

### Multi-Level Feedback Queue (MLFQ)

→ A complex scheduler designed to optimize both turnaround time & response time without prior knowledge of job lengths.

→ In simple words, a scheduler that learns from the past to predict future

→ Objective:

① Optimize turnaround time → run shorter jobs first

② Minimize response time ~~to~~ without a prior knowledge of job length.

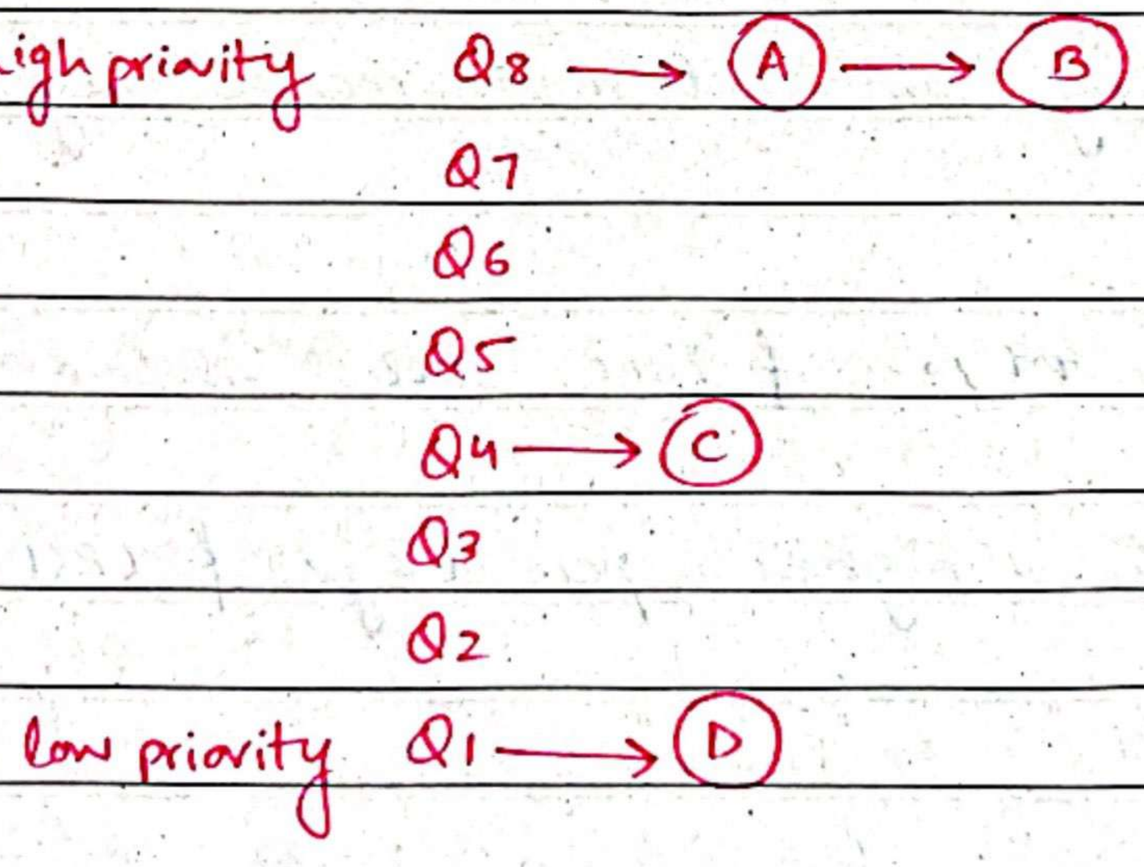
Date:_____

## MLFQ : Basic Rules

→ MLFQ has number of ~~distict~~ distinct queues.
  → each queues is assigned to different priority level.
→ A job that is ready to run is on a single queue.
  → a job on higher queueue is chosen to run
  → use round-robbing scheduling among jobs in the same queue:

Rule 1: If Priority (A) > Priority (B), A runs (B doesn't)
Rule 2: If Priority (A) = Priority (B), A & B run in RR.

→ MLFQ varies the priority of job based on it's observed behaviour.



Date:_____

## How To CHANGE PRIORITY ?

→ MLFQ priority adjustment algorithm:

Rule 3: When a job enters the system, it is placed at highest priority.

Rule 4a: If a job uses up an entire time slice while running, it's priority is reduced.

Rule 4b: If a job gives up CPU before time slice is up, it stays at same priority level.

In this manner, MLFQ approximates SJF.

## PROBLEMS WITH MLFQ:

① Starvation

→ if there are "too many" interactive jobs in the system

→ long-running jobs will never receive any CPU time.

② Game Scheduler.

→ After running 99% of time slice, issue an I/O operation

→ The job gain a higher percentage of CPU time.

Date:_____

→ A program may change its behaviour time.
→ CPU bound process → I/O bound process

Rule 5: After some time period S, move all the jobs in the system to topmost queue.

How to prevent gaming of our scheduler?

Solution

→ Rule 4: Once a job uses up it's time allotment at given level, it's priority is reduced.

Lower Priority, Longer Quanta.

→ The higher-priority queues → short time slices.
→ the low-priority queues → longer time slices.

SUMMARY

Rule ① If Priority (A) > Priority (B), A runs (B doesn't)
Rule ② If Priority (A) = Priority (B), A & B run in RR.
Rule ③ When a job enters the system, it is placed at the highest priority.
Rule ④ Once a job uses up it's time allotment at given level (regardless of how many times it has given up CPU) it's priority is reduced.
Rule ⑤ After some time period S, move all jobs in system to topmost queue.

→ Address space isolate process and provide illusion of dedicated memory.

→ The address space abstraction provides logical view of memory to a process

Code → Data → Heap → Stack
Data → global variables
Stack → local variables

Date: _____

## CHAPTER 13: The Abstraction – Address Space.

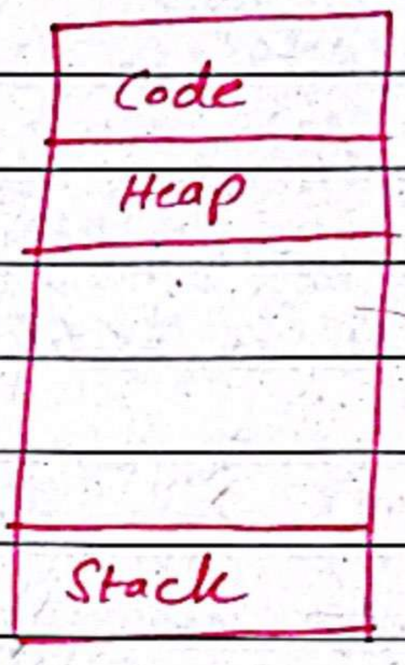
→ An address space serves as an abstraction that provides each process with illusion of it's own private memory

→ **MAIN COMPONENTS of an Address Space:**

① **Code Section**: Contains the executable instructions of program

② **Stack**: Used for local variables function parameters etc. args, uninitialized variables

③ **Heap**: Provides memory for dynamic allocation (eg using malloc() in C)




→ **HISTORICAL CONTEXT**

→ early computing systems used a single address space shared by all processes, leading to issues of memory protection and interference.

→ Multiprograming and time-sharing emerged as methods to allow multiple processes to run concurrently with isolated memory spaces.

**IMPORTANCE OF ADDRESS SPACE**

→ Allows multiple programs to coexist in memory without infering with each other.

→ Enables OS to enforce boundaries for memory access, prevention a process from accidentally modifying another process's memory..

* OS is loaded at OKB in address space.

Date:

## GOALS OF ADDRESS SPACES.

① Transparency: invisible to running program.

Each process operates as if it has it's own contiguos block of memory, creating simplified programing model.

② Protection:

Process are isolated from are another and memory access is restricted to prevent unauthorized interactions.

③ Efficiency:

Efficient memory management is essential to suppart multitasking, minimize overhead, and maximise resource utilization.

## CHAPTER 14: Memory API

malloc()

* not a system call.
* found in #include <stdlib.h>

void* malloc(size_t size).

→ Allocate a memory region on the heap.

→ Argument

① size_t size: size of memory block (in bytes)

② size_t is an unigned integer type.

→ Return

① Success: a void type pointer to the memory block allocated by malloc.

② Fail: a null pointer.