

Operating Systems (CS3000)

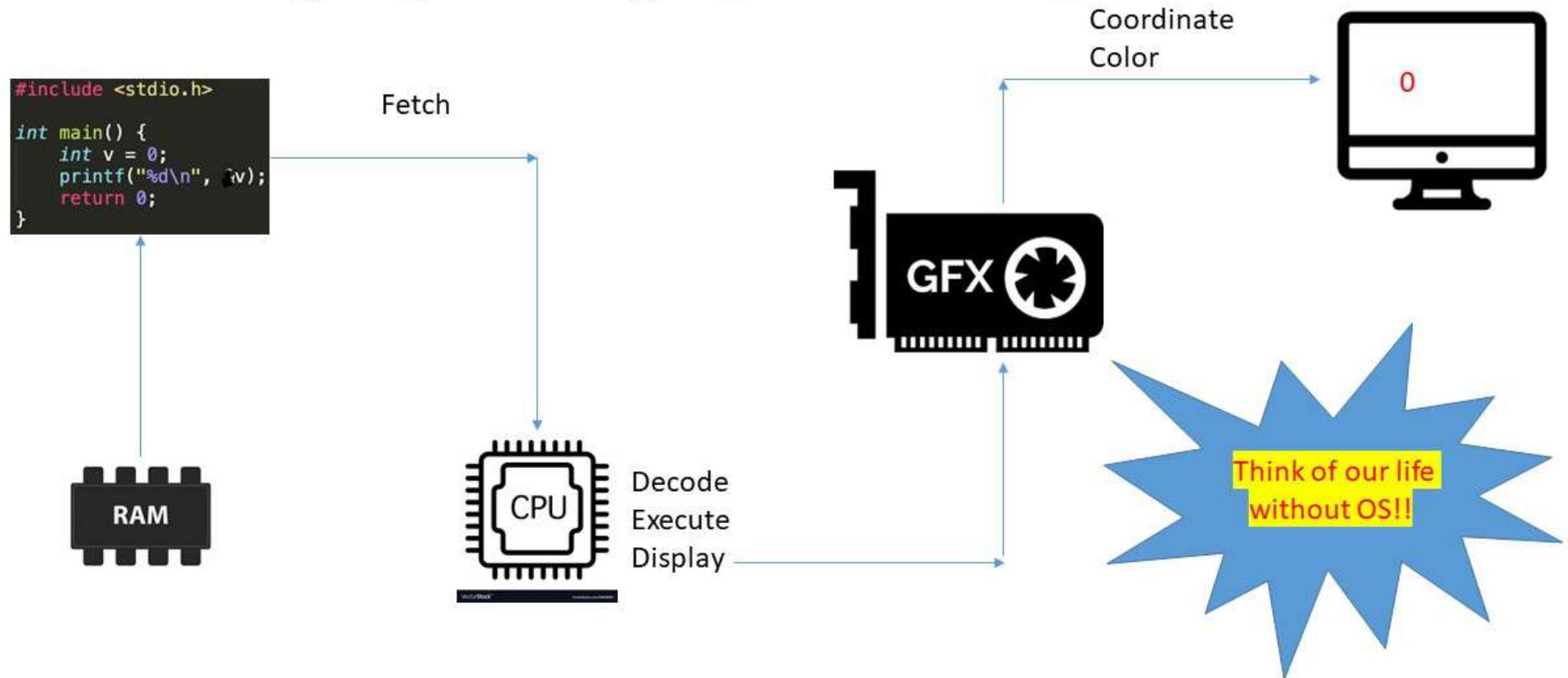
Lecture – 1
(Course Overview)



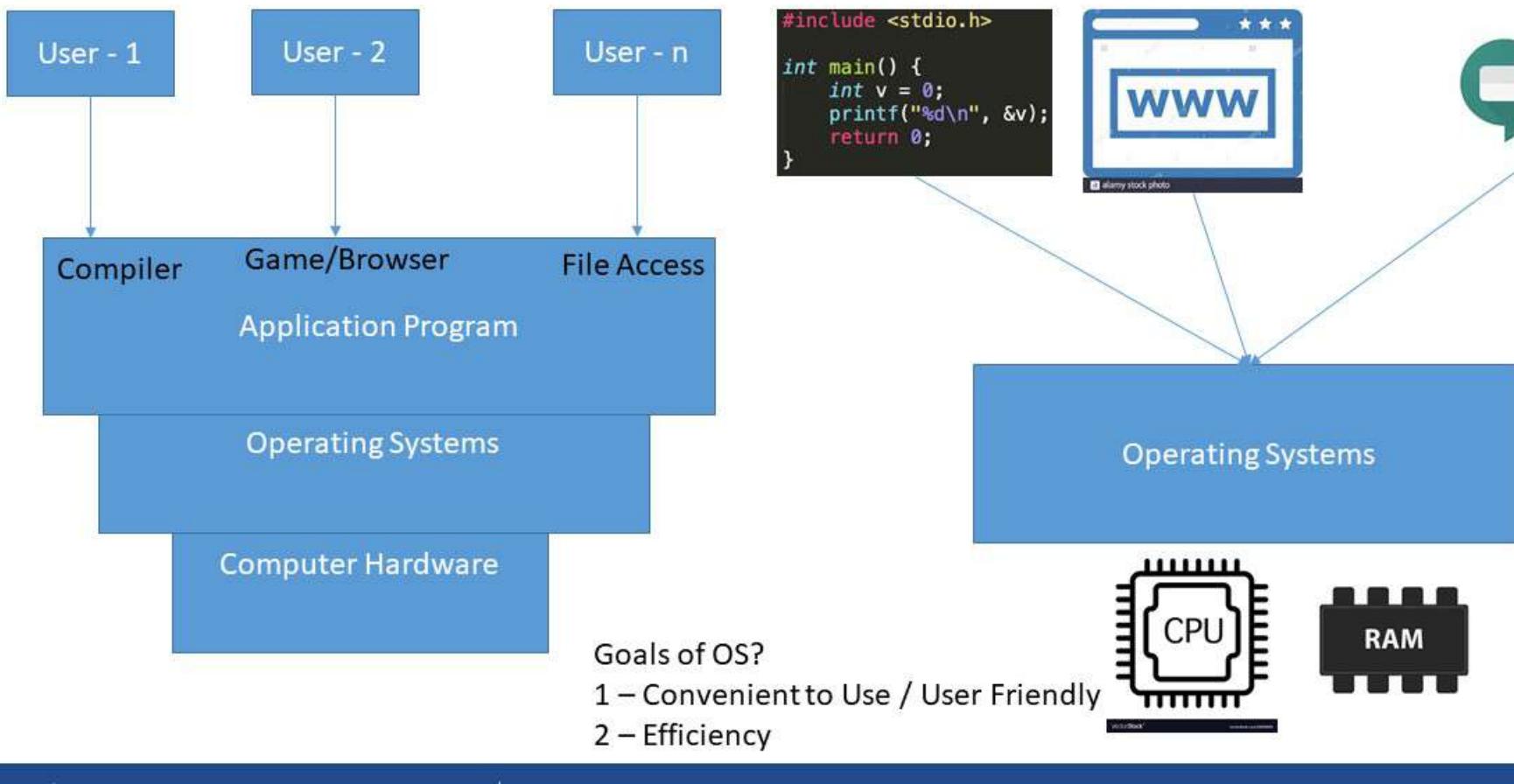
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Why Operating Systems Required?



Why this Course?

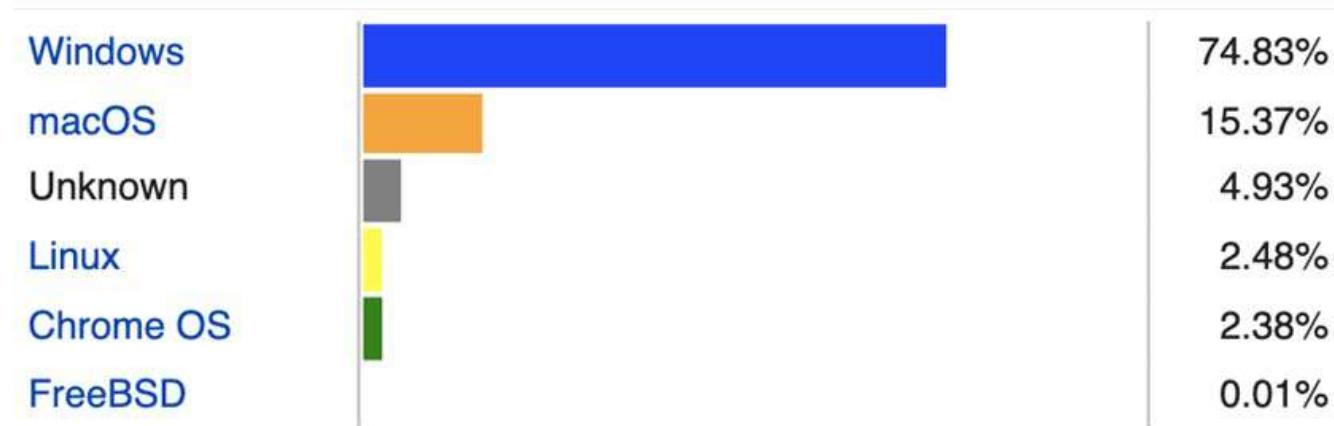


Why this Course?

- Most Essential Part of a Computer System
- A program that acts as an intermediary
 - between a user of a computer and the computer hardware
- A program that is a resource manager
 - Memory, CPU, I/O
- Acts like Government
 - No useful function by itself
 - Sets up environment for other applications to achieve their tasks
- Time/Deadline Based
- Event Driven
- Challenges in the OS design
- Tradeoffs in OS design



Operating System Market Share



Source:StatCounter



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Types of OS (Types of Applications)

- Desktops
- Servers
- Embedded OS
- Mobile OS
- RTOS
- Secure Environment
- Mac OS, Windows, Ubuntu
- Windows Server, Redhat
- Contiki OS
- Android, iOS
- RTLinux
- SeLinux



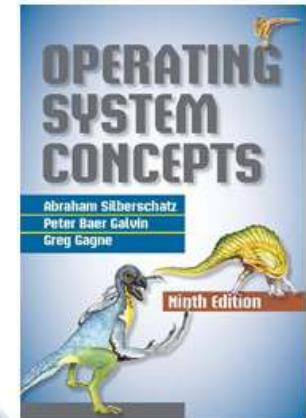
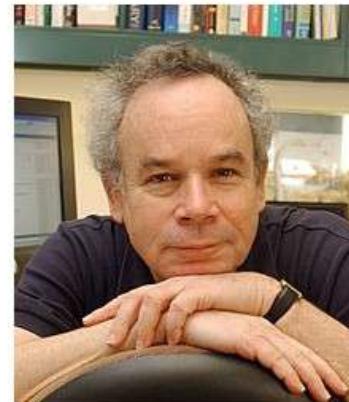
Course Contents

- Functionalities & Services of an Operating System
- Process Concept - System Calls & Management
- Process Synchronization
- Process Scheduling
- Deadlock
- Memory Management
- I/O Management

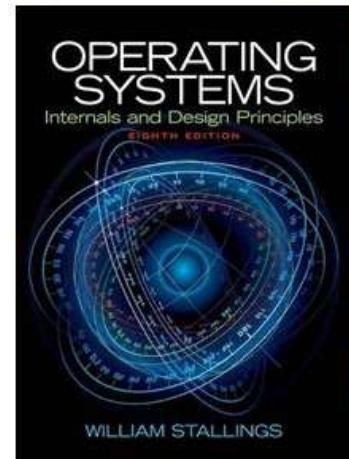


Books & Reference Materials

1. Operating System Concepts,
**8th Edition by Silberschatz,
Galvin, Gagne**



2. Operating Systems: Internals
and Design Principles, **8th
Edition by William Stallings**



3. Online Resources



Course Administration

- **Prior knowledge:** C, Data Structures, Computer Organization
- Lecture slides will be available on moodle after lecture.
- Some reading material will be provided before/after the lecture.
- **Discussion Time:** Anytime except when we have class or laboratory (prior email is preferable).
- **Lab:** Once a week (Good Learning Experience).
- Easiest way to get a good grade in CS3000 is to pay attention in the class.
- 85% attendance is mandatory.
- Total = 42 Lectures, 14 Tutorials

CS3000			
L	T	P	Credits
3	1	0	4

Time Table @ H13

Monday – 12:00 PM – 12:50 PM

Tuesday – 09:00 AM – 09:50 AM

Thursday – 10:00 AM – 10:50 AM

Thursday – 05:00 PM – 05:50 PM

Lab (@L509) – Friday – 02:00 PM – 06:00 PM



Course Evaluation Components (Tentative)

- Quiz – 1 : 20 (28 – 30th August 2023)
- Quiz – 2 : 20 (04 – 06th October 2023)
- ETE : 40 (17 – 29th November 2023)
- Assignment : 20 (1st Week of November 2023)

- ***CRs – CC Meeting**



Thank You

Any Questions?



Operating Systems (CS3000)

Lecture – 2
(Types of OS)



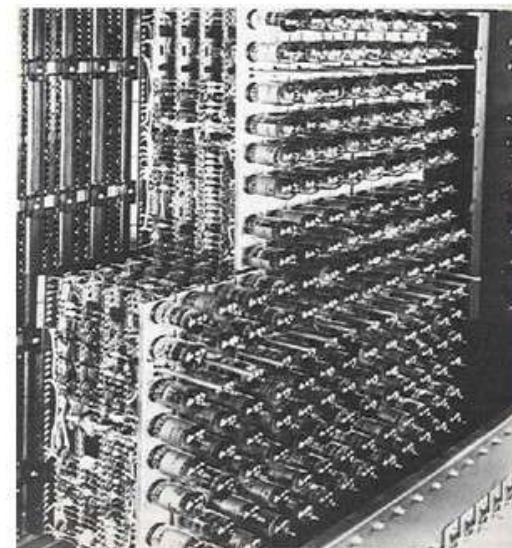
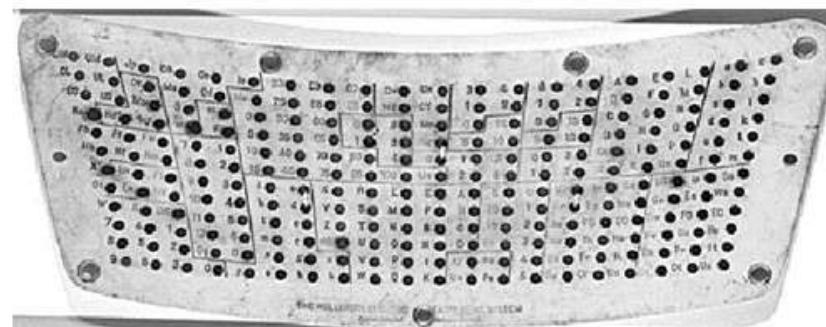
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Types of OS (Technique Used)

- Old Days

- Vacuum Tubes with Punch Cards/Paper Tapes
- Directly Entering Machine Language
- No OS
- Slot is Allocated to Each User.
 - Setup the Environment
 - Perform the Task
 - Take the result using print out.



Source: https://youtu.be/YByu_1S2VeU



Types of OS

- Batch OS

- Transistors
- Assembly Language
- Starts another job only after the present job is **completed entirely**
- both CPU and IO Parts to be over
- Poor CPU Utilization
- Low Throughput (Efficiency Aspect) **No of Jobs or Tasks completed per unit time**

Every Job or Task has
CPU Time
IO Time



- Stored Program Architecture
- Multiple Programs or Jobs are allowed to be in Main Memory



Types of OS

- Multi Programming OS

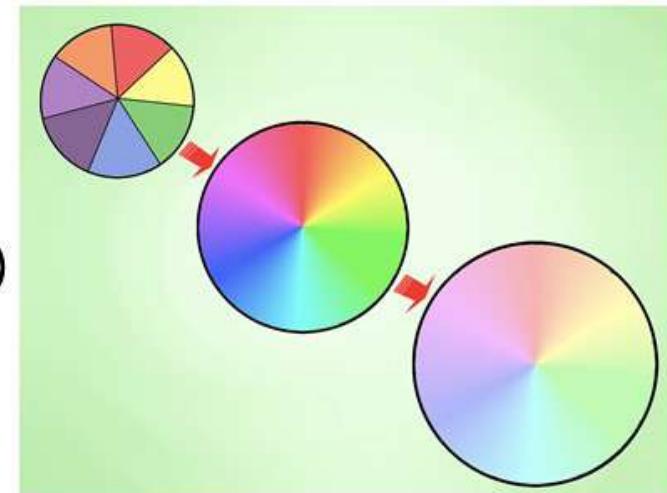
- Overlapped execution of CPU and IO Operations Tasks
- When CPU is idle, switch to other Job
- Better CPU Utilization – when J1 is busy on IO; J2's CPU part is allowed
 - Betters Throughput



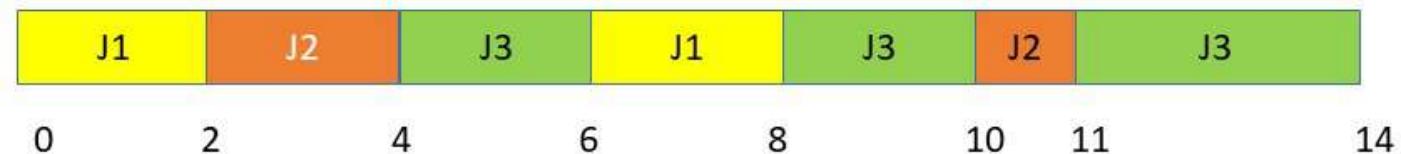
Types of OS

- Multi Tasking OS

- Based on the concept of Time Sharing (= 2sec)
- Time Sliced Execution of Tasks
- Illusion of Simultaneous Execution of Tasks



J1 – 4 Sec
J2 – 3 Sec
J3 – 7 Sec



Types of OS

- Multi Processing OS

- More than 1 Processor
- Modern day Multicore Systems
- True Simultaneous or Parallel Processing
- High Throughput
- High Reliability – Fault Tolerant Systems
- Economical – from a user and application management view

Processor - 1



Processor - 2



Thank You

Any Questions?



Operating Systems (CS3000)

Lecture – 3
(Process Introduction)

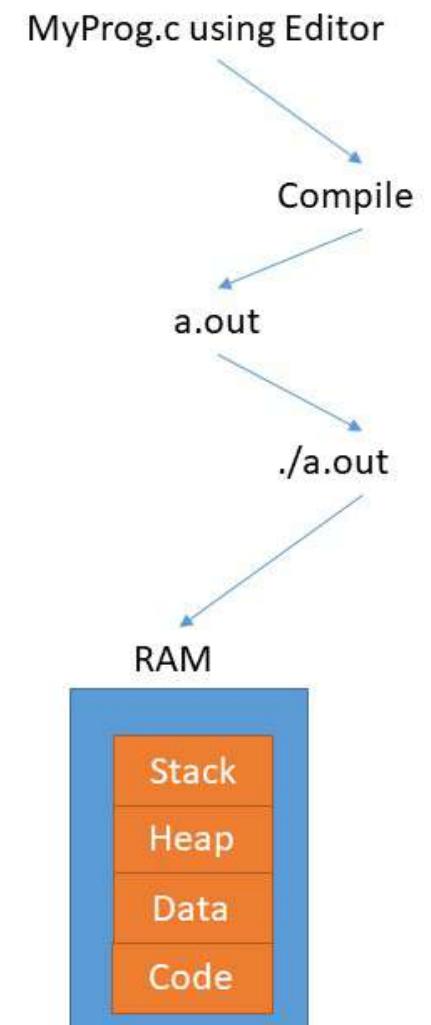


INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Process

- Program in execution is known as **process**.
- Each Process has a PCB.
 - Process Control Block.
 - Stores info. Related to process.



Process

- PCB is a data structure that stores
 - Process id (unique for each process)
 - Process state
 - Size of process memory
 - List of opened files
 - List of opened devices
 - Program counter
 - Memory management info (Page Table, Memory Limit, ...)
 - ...
- Why PCB?
 - Allows process to resume execution after a while
 - Keep track of resources used
 - Track the **process state**



Thank You

Any Questions?



Operating Systems (CS3000)

Lecture – 4
(Process State Diagram)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Process

- From the time process is created till it finishes, it passes through several states.

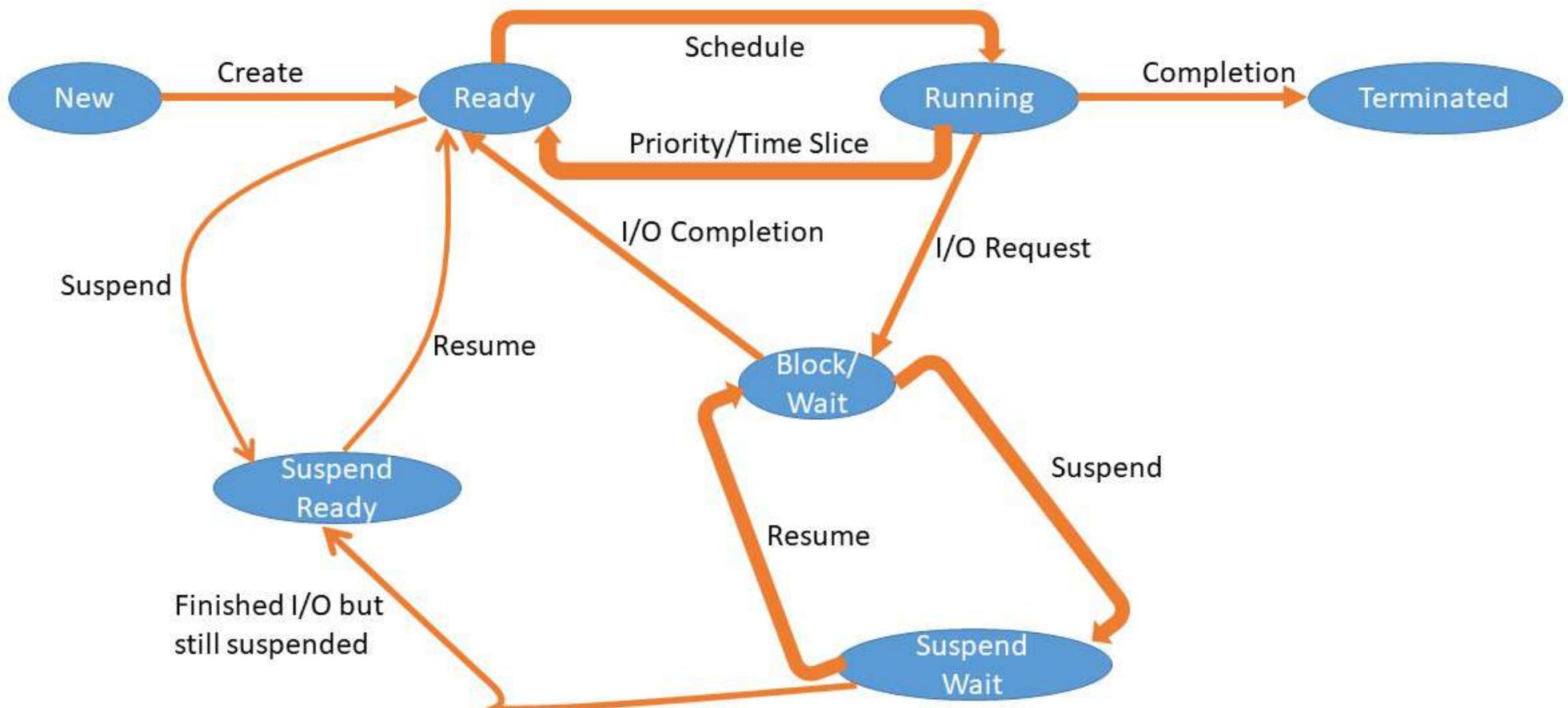
- New
- Ready
- Running
- Wait (Block)
- Terminated (Completed)
- Suspend Ready
- Suspend Wait (Block)

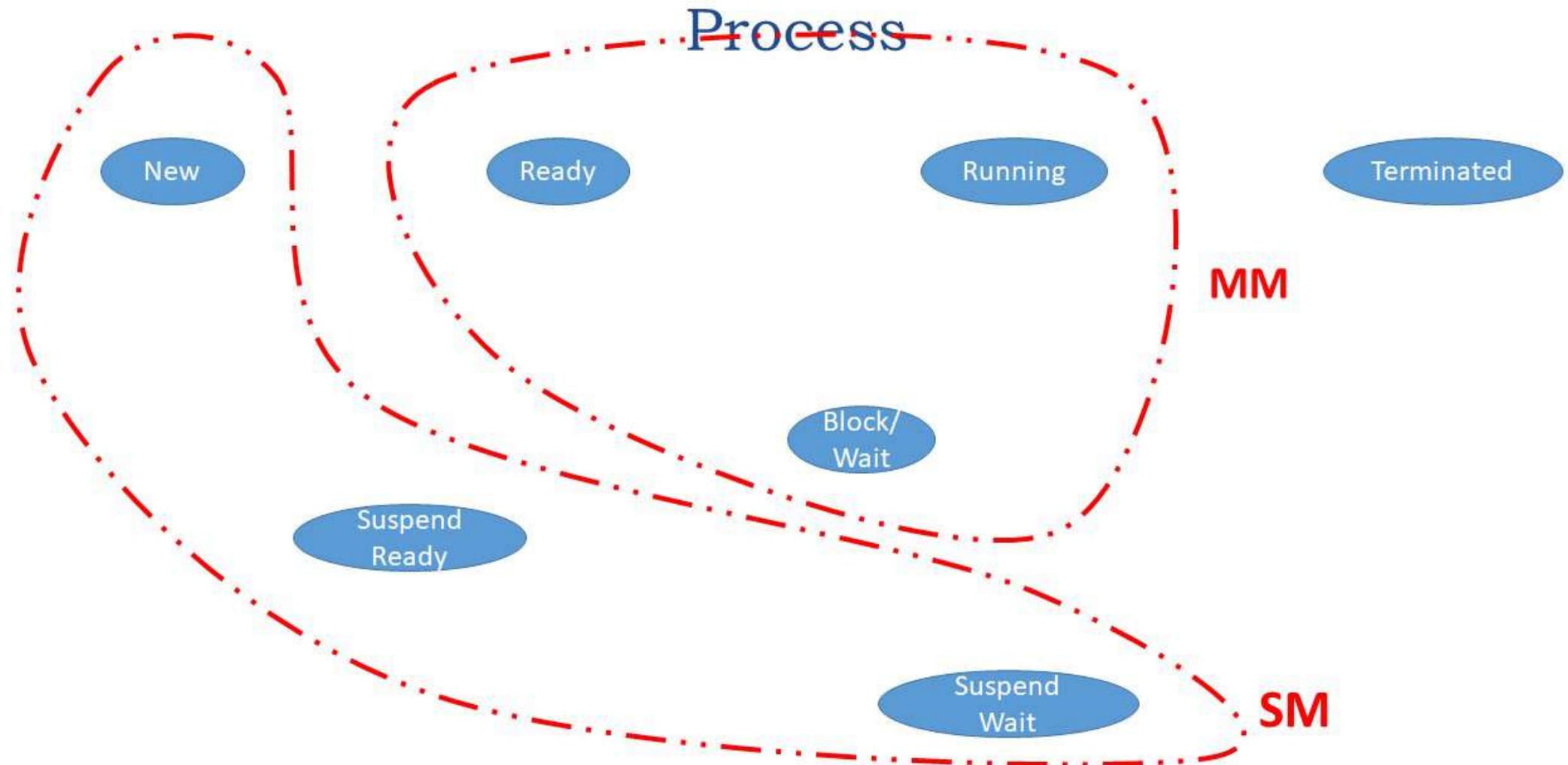
Degree of multiprogramming –

The number of processes that can reside in the ready state at maximum



Process





Thank You

Any Questions?



Operating Systems (CS3000)

Lecture – 6
(CPU Scheduling-1)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Context Switching

- The process of saving the context of one process and loading the context of another process.
 - loading and unloading the process from the **running state** to the **ready state**.
- When does context switching happen?
 - Preemptive CPU scheduling used/TQ Expires
 - When a high-priority process comes to a ready state
 - with higher priority than the running process



Multi-Programming

- We have many processes ready to run.
- **Pre-emption** – Process is forcefully removed from CPU. Pre-emption is also called as **time sharing or multitasking**.
- **Non pre-emption** – Processes are not removed until they complete the execution.
- **Degree of multiprogramming** – The number of processes that can reside in the ready state at maximum



CPU Scheduling

- Decide the order of execution of processes when multiple are competing for system resources
- **Where** – In the Ready State
- **Who(se) Responsibility** – Short Term Scheduler
- **When** - Wherever there is queuing for the CPU!
 - Ready to Running
- **Function** – allocate CPU to process
- **Goal** –
 - Increase CPU Utilization
 - Increase Throughput
 - Minimize Average Wait Time
 - Minimize Average Turn Around Time



CPU Scheduling

- **Non preemptive**

- FCFS
- SJF/SPN
- HRRN

- **Preemptive**

- SRT
- RR

- **Priority**

- Preemptive
- Non preemptive



Different Time based Parameters w.r.t. Process

- **Arrival Time (AT)** - Time the Process comes to the Ready State
- **Burst Time (BT)** – Execution time of the process – also referred as Service Time
- **Completion Time (CT)** – Time at which process completes its execution.
- **Turn Around Time (TAT)** - Time required for an application (process) to give an output to the end user
- $TAT = CT - AT$
- **Waiting Time (WT)** - Time Difference between turn around time and burst time.
- $WT = TAT - BT$
- **Response Time** – Time for the System to Respond to Process or User (First Response time on System Clock)
 - Time Since the Request is Submitted (AT) and the First Response Time

FCFS CPU Scheduling



SJF/SPN CPU Scheduling



Thank You

Any Questions?



Operating Systems (CS3000)

Lecture – 6
(CPU Scheduling-2)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

SRT CPU Scheduling

Mechanisms to Predict BT

Priority CPU Scheduling (Non-Premptive)



Thank You
Any Questions?



Operating Systems (CS3000)

Lecture – 9
(Advanced CPU Scheduling)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Multilevel Queue Scheduling

- **Ready queue is partitioned into several queues.**
 - Foreground (interactive) (Word processor, game application)
 - Background(batch) (System update, Application for generating phone bills at the end of month)
- **Each queue has its own scheduling algorithm**
 - Foreground - **RR**
 - Background - **FCFS**



Scheduling must be done between Queues

- **Strategy 1:** Using Fixed Priority Scheduling

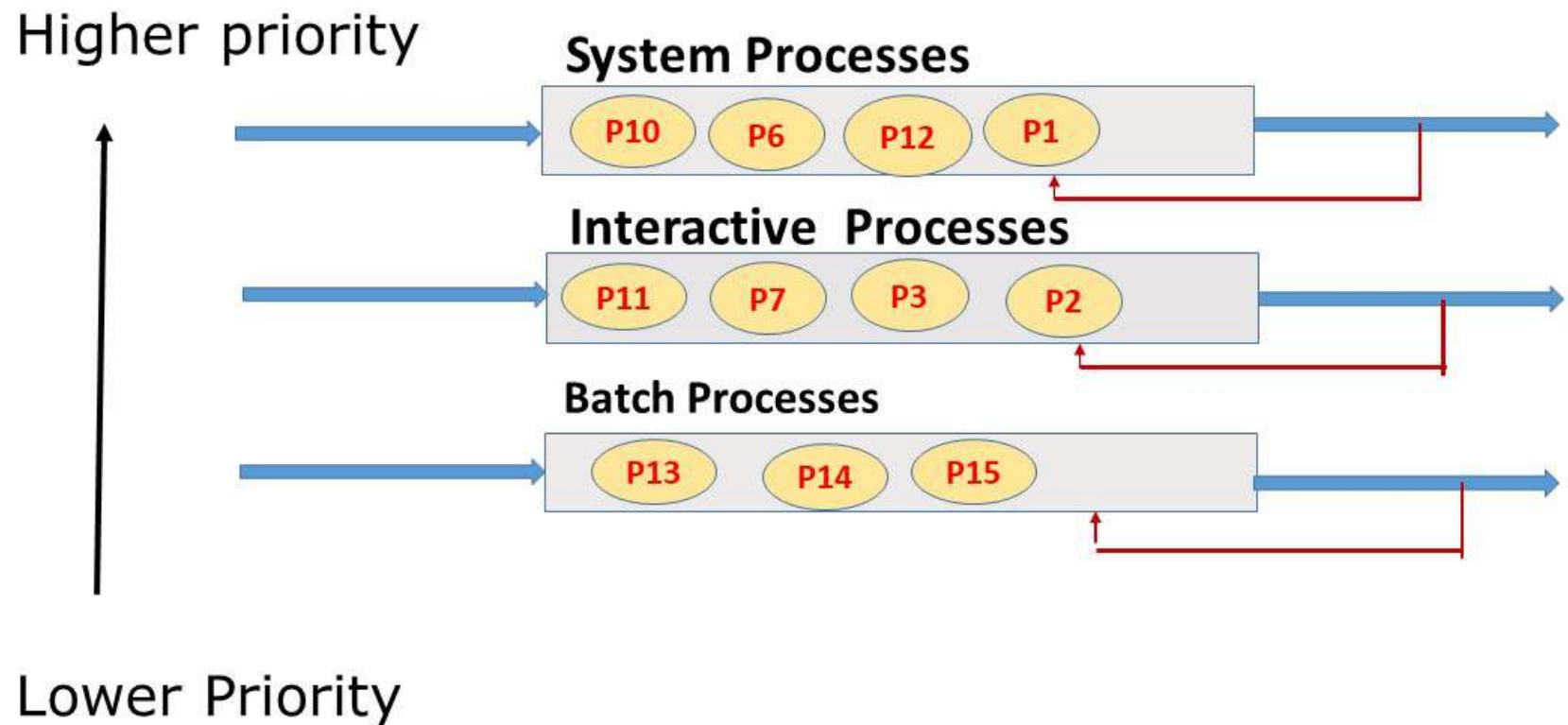
- Each Queue has priority
- Serve all from foreground then from background
- Issue:- Possibility of Starvation

- **Strategy 2:** Using Time Slice

- Each queue gets a certain amount of CPU time which it can schedule amongst its processes
- 80% to foreground in RR
- 20% to background in FCFS



Multilevel Queue Scheduling



Multilevel Feedback Queue

- A process can move between the various queues
- Multilevel feedback queue defined by the following parameters
 - Numbers of queues
 - Scheduling algorithm for each queue
 - Methods used to determine when to upgrade/demote a process



Multilevel Feedback Queue

- Example: Consider 3 queues
 - Q_0 - time quantum 8 milliseconds
 - Q_1 –time quantum 16 milliseconds
 - Q_2 – FCFS
- Scheduling:
 - A new job enters queue Q_0 which is served *for 8 ms.*
 - When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1
 - At Q_1 job is again served RR and receives 16 additional milliseconds. If it still does not complete, it is pre-empted and moved to queue Q_2 .
 - At Q_2 it executes as FCFS.



Multilevel Feedback Queue



Comparison b/w different scheduling approach

Algorithm	Strategy	Average waiting Time	Preemption	Starvation	Comments
FCFS	Arrival time	Large	No	No	Slow performance Convoy effect
SJF	Burst Time	Smaller than FCFS	No	Yes	Min Avg Waiting time
SRTF	Burst Time	Smaller than FCFS	Yes	Yes	Min Avg waiting time
RR	Fixed time quantum (TQ)	Large as compared to SJF and Priority	Yes	No	Each process has given a fairly fixed time
Priority Preemptive	Priority	Smaller than FCFS	Yes	Yes	Starvation



Comparison b/w different scheduling approach

Algorithm	Strategy	Average waiting Time	Preemption	Starvation	Comments
Priority Non-Preemptive	Priority	Smaller than FCFS	No	Yes	Starvation
HRRN	Response Ratio	Smaller than FCFS	No	No	Helps for process with longer waiting time
MLQ	Process that resides in the higher priority queue	Smaller than FCFS	Yes/No	Yes	Starvation
MLFQ	Longer BT process moves to Lower Priority Queue	Smaller than FCFS	Yes/No	No	No Starvation



Other Advanced Scheduling Techniques

- Multiprocessor Scheduling
 - Symmetric
 - Asymmetric
- Real-Time Scheduling
 - Earlier Deadline First
 - Rate Monotonic
- Distributed System Scheduling
- Linux Scheduling
 - Completely Fair Scheduler



Scheduling Criteria

- CPU utilization
 - Maximize
- Throughput
 - Maximize
- TAT
 - Minimize
- Response Time
 - Minimize
- Waiting Time
 - Minimize
- Fairness

Thank You

Any Questions?



Operating Systems (CS3000)

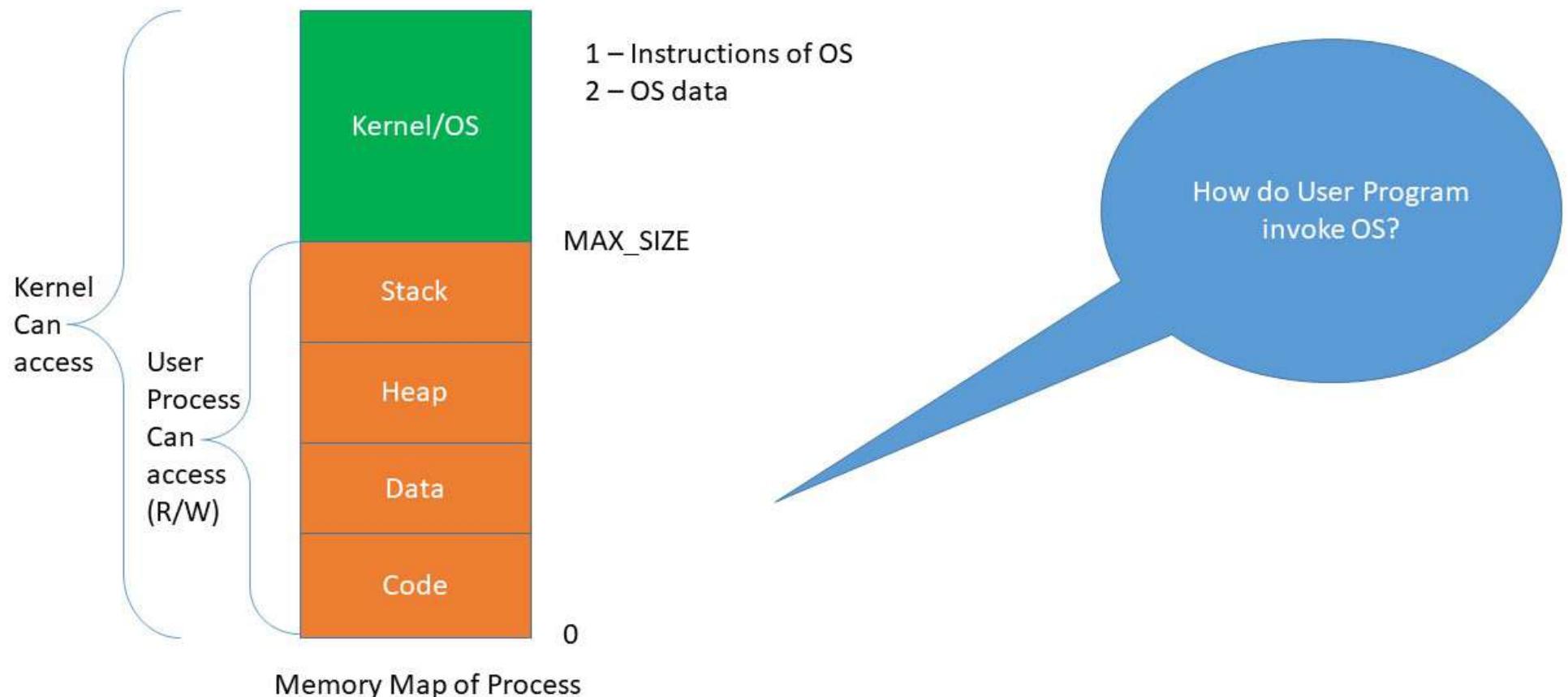
Lecture – 10
(System Calls)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Process Memory Map



Communicating with the OS (System Calls)

- System Calls are a set of special functions which the OS support.
- User Process can invoke any of the system calls
- Why?
 - to get information
 - to access hardware/ resources within the Kernel.

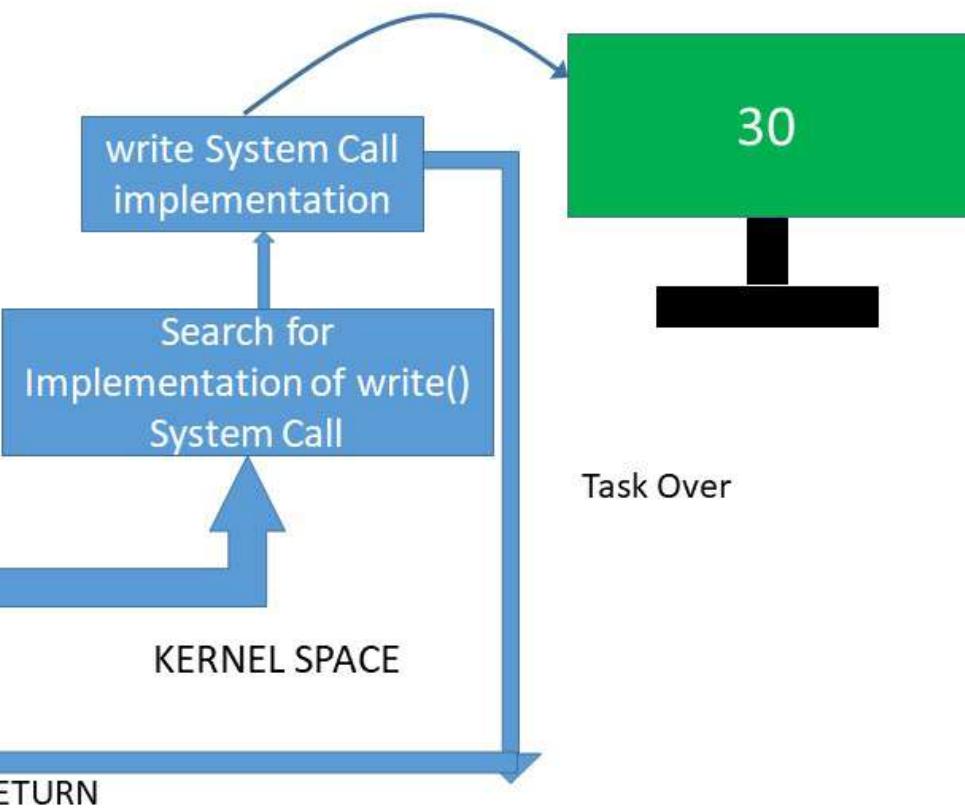
printf() System Call

```
int x = 30;  
printf("%d", x);
```

call Library
present in libc

write(STDOUT)

USER SPACE



Task Over

What Happens During System Calls?

- process (user mode) → process (kernel mode)
 - allow the kernel or the operating system to actually execute the task
- System Call over: process (kernel mode) → process (user mode)

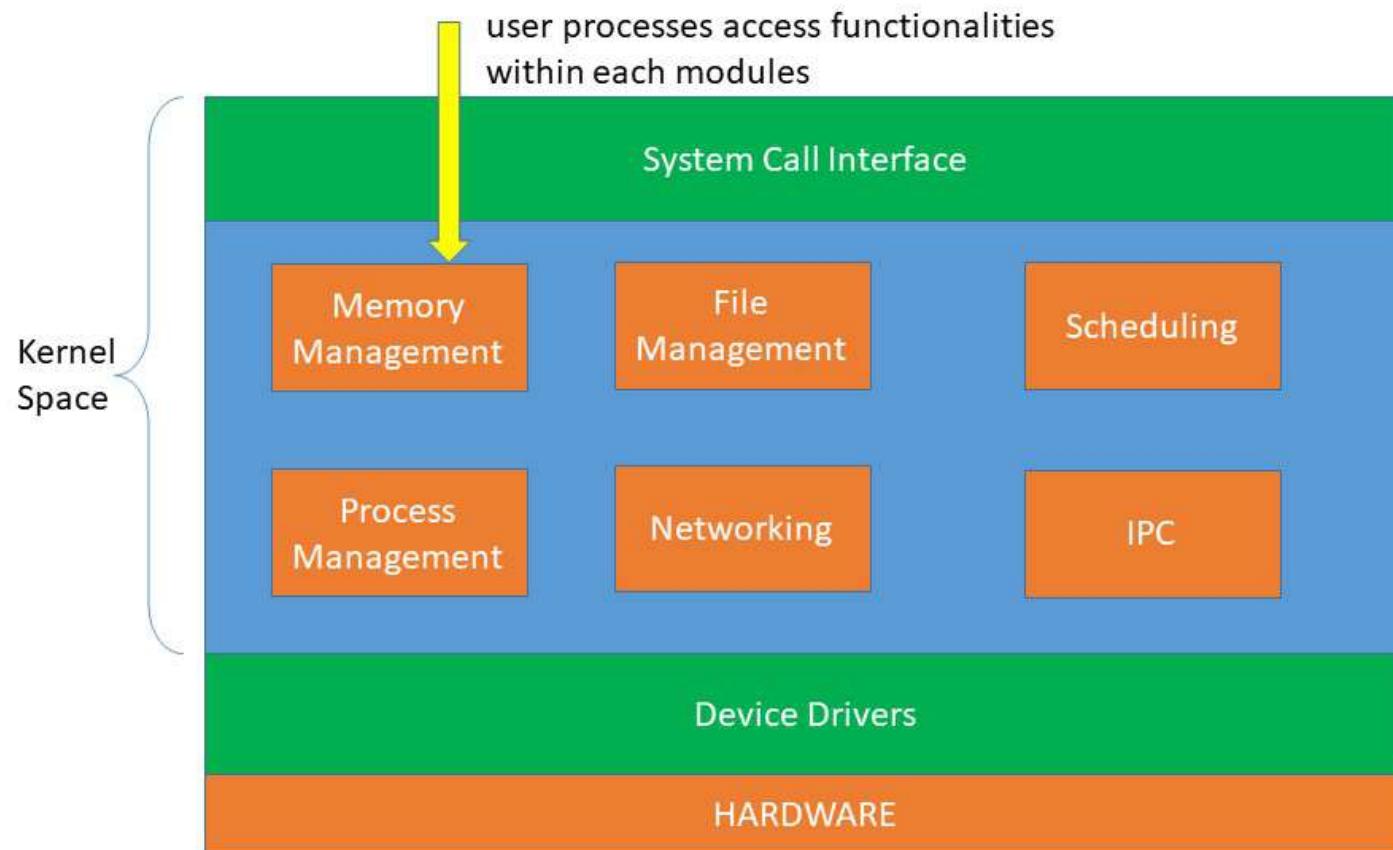
Function Call v/s System Call

- CALL instruction
- User Space
- CALL jumps to a relocatable address
- TRAP instruction (s/w interrupt)
- User Space → Kernel Space
- TRAP jumps to a fixed address

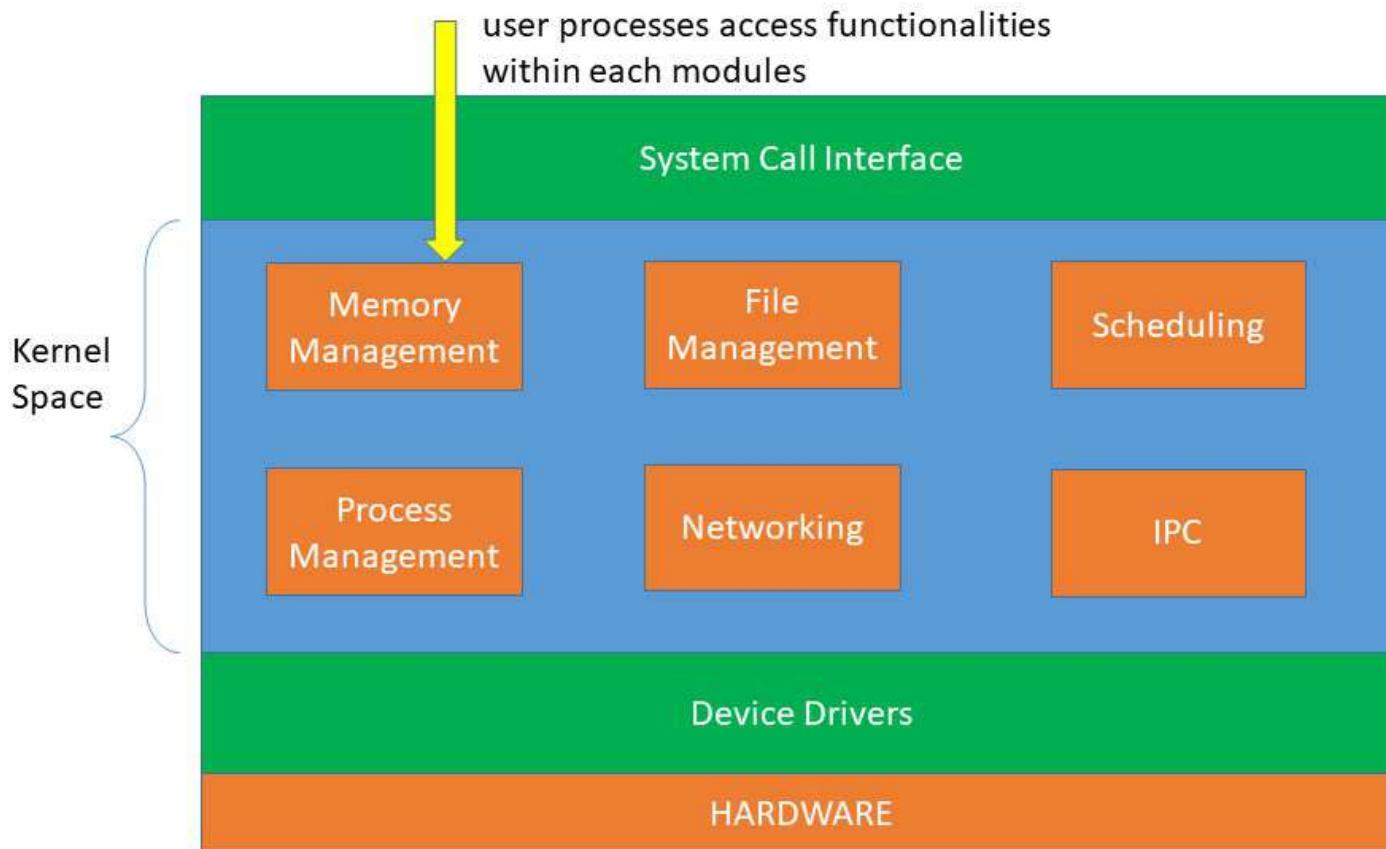
System Calls for Files

- Files: Data remains even if power is off
- Operations on Files:
 - open()
 - close()
 - read()
 - write()
- Files are stored in HDD. System call required to access Hardware.

OS Structure

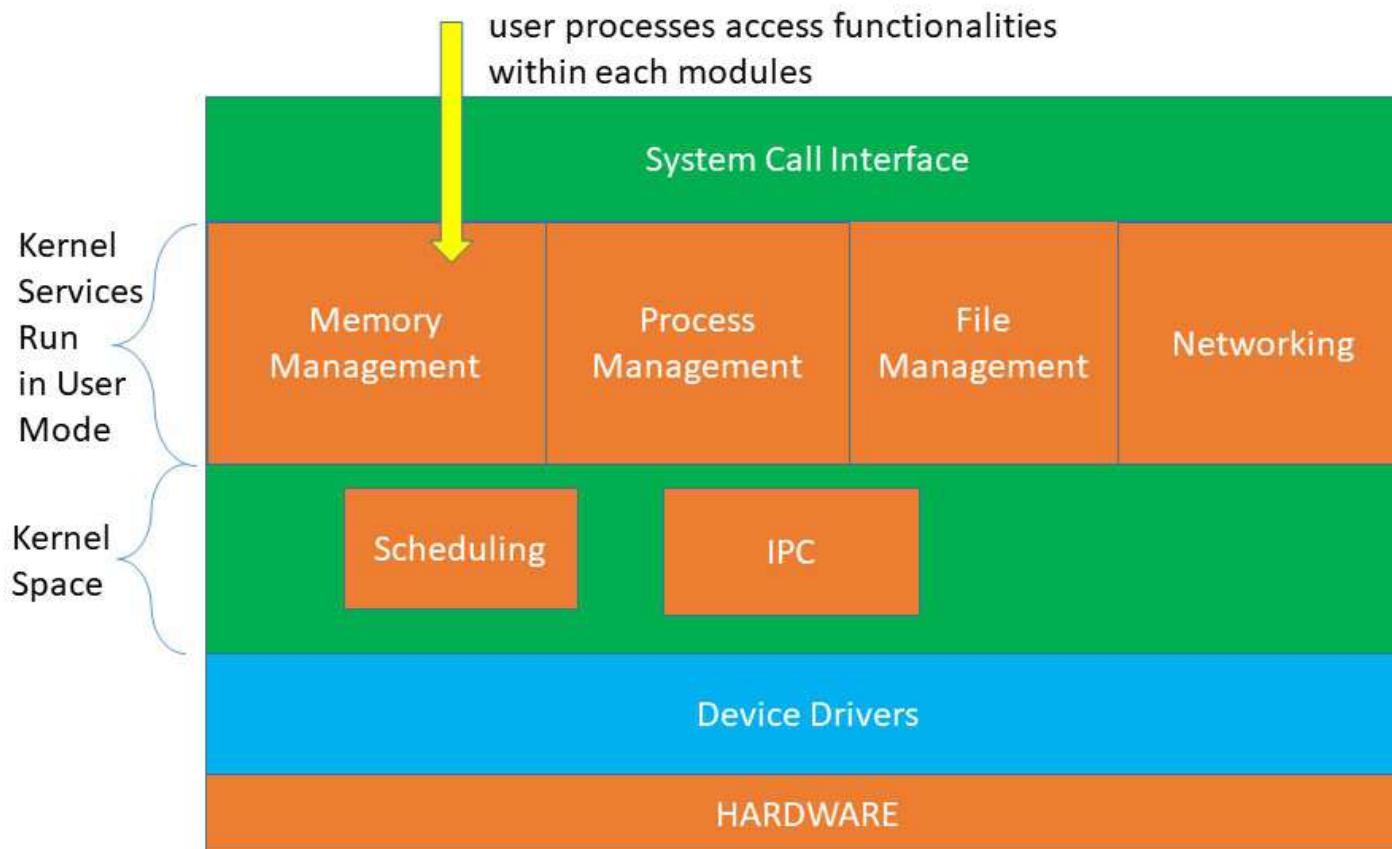


Monolithic OS Structure



1. All Components of OS/All functionality of OS are present in Kernel Space
2. Kernel is a single process where all functionalities share the same address space.
3. Direct Function Calls between modules of Kernels
4. Large Size Kernel, Difficult to maintain and manage

Microlithic OS Structure



1. Few Components of OS/few functionality of OS are present in Kernel Space
2. Some Kernel services can be modified by users based on needs.
3. Direct Function Calls not possible. IPC
4. Small Size Kernel, Easy to maintain and manage

Attendance policy

- **85% minimum** attendance required in a course
- **10% waiver allowed** for accommodating leave due to major illness with/without hospitalization, family calamity and on-duty.
- Medical leaves/on-duty should be applied on the portal & relevant forms should be submitted to office not **later than 10 days from the end day of the leave period**.
- In case of not fulfilling the attendance criteria, student will be **awarded W grade** in the subject concerned and will not be eligible to appear for end semester examination.



Examination Protocols – Code of Conduct

- To report **10 mins prior** to the examination. Possession of the ID card is a must. Advised to bring a bottle of water
- **No Mobile Phones, Smart watches inside the examination hall**
- You will **NOT be permitted to use restrooms** during the examination
- **NOT permitted to carry any handwritten/printed material**
- **Communicating with neighbors** for any reasons is strictly not allowed
- **Exchange of pens / pencils / drawing instruments / calculators, tables,** are not allowed
- Cooperating with Invigilators and Squad in case of frisking.
- Invigilator may ask you to empty the pockets and you are instructed to comply with such instructions
- For any assistance, you may contact the faculty/staff in-charge



Disciplinary actions for malpractice during exams

Malpractice/Act of indiscipline	Disciplinary action
Communicating with neighbor(s) in the examination hall for any reason(s)	The erring student(s) shall be awarded 'U' grade in the subject concerned and 16 hours of Community Service
Possessing incriminating materials (or) individual referral of material irrespective of its usage / discussion with other students	20 hours of Community Service and the erring student(s) shall be awarded 'U' grade in the subject concerned and one grade less in all the other subjects in the concerned semester.
Possessing the answer book of another candidate (or) Passing on answer book to another student (or) Exchange of question papers, with some answers jotted down on them.	30 Hours of Community Service and the erring student(s) shall be awarded 'U' grade in all subjects in the concerned semester
Involved in malpractice in the examination for the second time, in a premeditated manner	The concerned student(s) shall be awarded 'U' grade in all subjects in the concerned semester and will not be eligible for Supplementary Exam or Contact Course in the respective subject. The student will also be debarred from attending classes and taking examinations in the subsequent semester.
Impersonation in the examination	The concerned student(s) shall be awarded 'U' grade in all subjects in the concerned semester and will not be eligible for Supplementary Exam or Contact Course in the respective subject. The student will also be debarred from attending classes and taking examinations in the next two subsequent semesters.



Thank You

Any Questions?

Operating Systems (CS3000)

Lecture – 11
(fork() System Call)

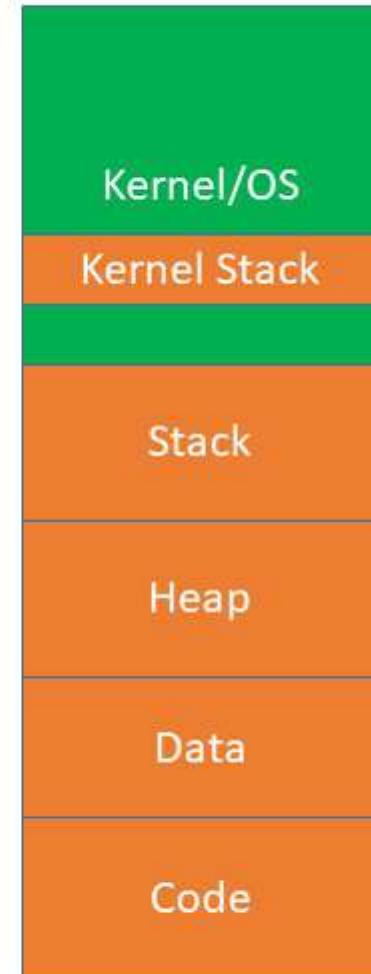


INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

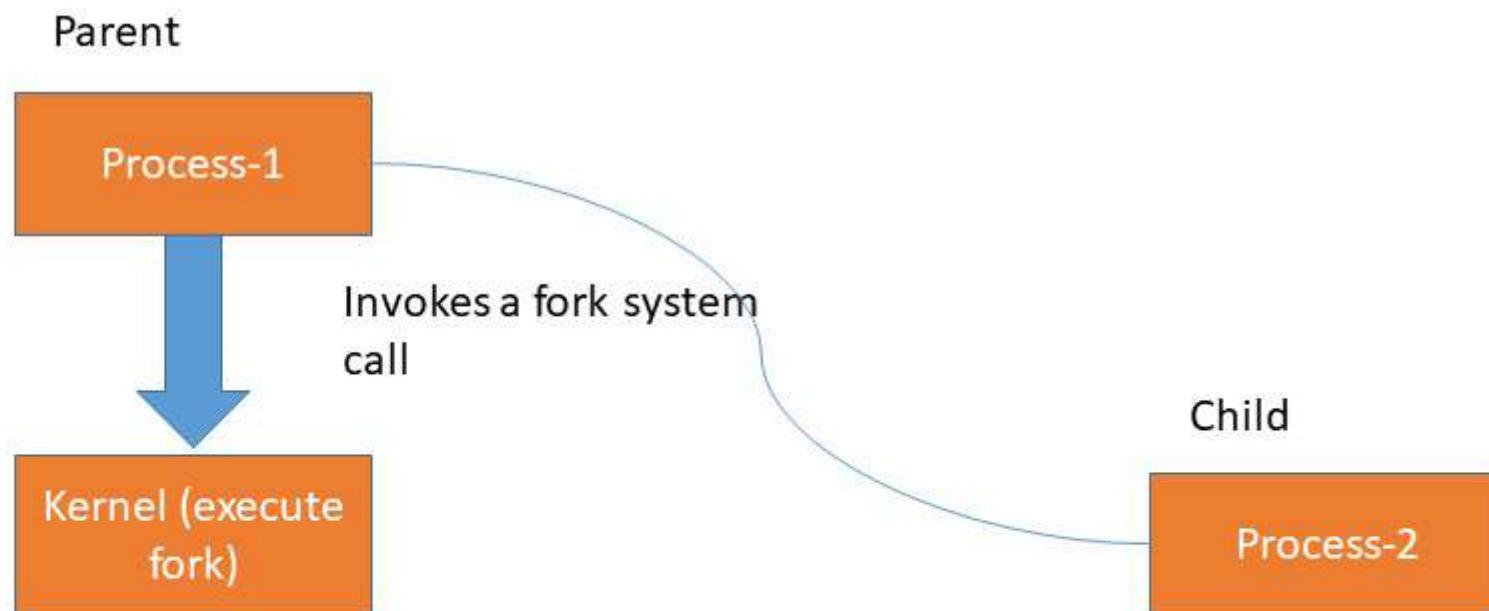
What Metadata of a Process Kernel Stores?

- PCB
- Kernel Stack for User Process
 - During System Calls
- Page Table for that User Process

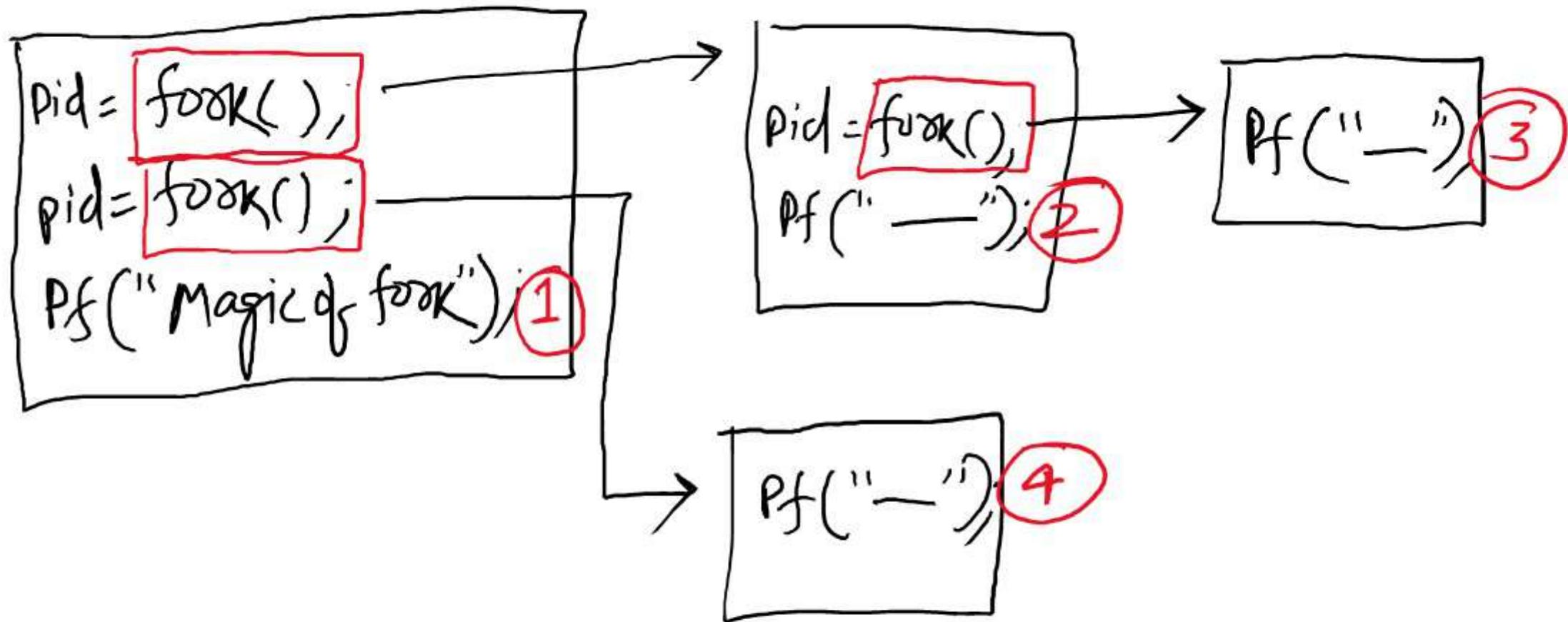


Creating a Process by Cloning

- `fork()`
 - Child Process is duplicate of parent process
 - PID → Parent process is Child's PID
 - PID → Child process is 0



fork() Ex-1



fork()

```
{  
    pid_t pid;  
    pid= fork();  
  
    if (pid<0)  
        printf("error in fork \n");  
    else if (pid==0)  
    {  
        fork();  
        printf("child print \n");  
    }  
    else if (pid>0)  
        printf("Parent Print \n");  
    printf("Main Print \n");  
    return 0;  
}
```

```
    pid_t pid;  
  
    if (pid<0)  
        printf("error in fork \n");  
    else if (pid==0)  
    {  
        fork();  
        printf("child print \n");  
    }  
    else if (pid>0)  
        printf("Parent Print \n");  
    printf("Main Print \n");  
    return 0;  
}
```

Ex-2

```
{  
    pid_t pid;  
  
    if (pid<0)  
        printf("error in fork \n");  
    else if (pid==0)  
    {  
        printf("child print \n");  
    }  
    else if (pid>0)  
        printf("Parent Print \n");  
    printf("Main Print \n");  
    return 0;  
}
```

Thank You
Any Questions?



Operating Systems (CS3000)

Lecture – 12
(exec(), wait(), exit() System Call)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

The classic fork() Bomb!

- Overall 8 processes in Main Memory as indicated at the leaf level nodes count
- In general n fork class (non conditional) will result in 2^n processes

```
int main()
{
    fork();
    fork();
    fork();
    printf("Magic of fork()\n");
    return 0;
}
```

exec() system call

- So Far fork example we did, child process carried the same image as the parent process.
- Practicality requires child to have new definition.
- Is it possible?
 - Yes

exec() system call

- **fork()** system call is used to create a **SEPARATE, DUPLICATE** process from which process (**parent**) it is called.
 - The new **child** process will have different PID.
- When **exec()** system call is invoked from (p1), the program specified in the parameters of exec() will **replace the entire process**.
 - exec() takes one parameter, which is another program(p2)
 - p1 will be replaced by p2.
 - Replace one process with another process (PID don't change)
 - As we are not creating new process
 - We are replacing an existing one
 - **Same PID with different content**

wait() system call

- called in parent process
- Parent goes to block state
 - Until one of it's children terminates
 - -1: if no child is executing
- When the child process exits i.e exit(0), it would cause the parent process to wake up
- the wait function returns the child process's pid
- The parent waits for the child process using this system call.

Why wait() system call ?



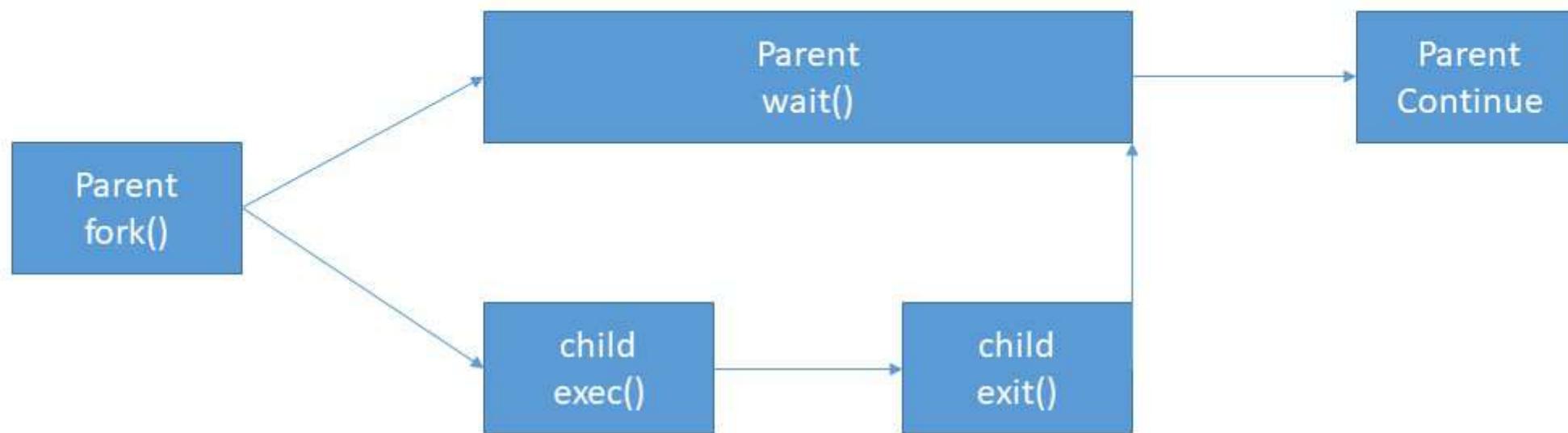
Zombie & Reaper Process



exit() system call

- called in **child process**
- Results in process termination
- Return status is passed to the parent (Voluntary Termination)
- Forcefully termination
 - kill() system call

Process system calls Possible Flow



Orphan Process

- When a parent process terminates before its child
 - The first process will adopt the orphan child.
-
- First process (init) in the system never exits.

Thank You
Any Questions?



Operating Systems (CS3000)

Lecture – 14
(Inter Process Communication)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

IPC

- Communication between the Processes
- Why IPC?
 - Sharing of data with Modular Programming
- 3 Ways
 - Shared Memory
 - Message Passing
 - Signals

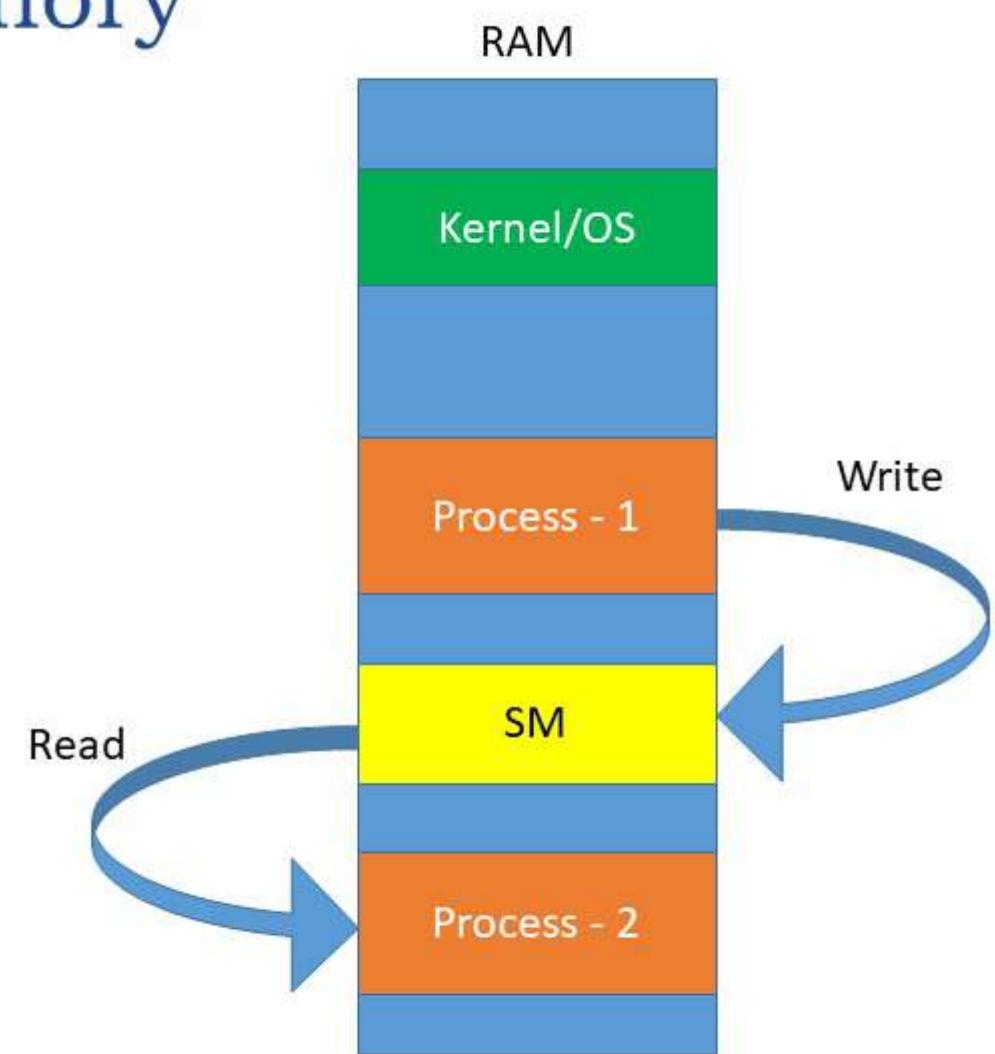
Shared Memory

- **Writer**

- Create SM
- Attach SM to it's address Space
- Write Data into SM

- **Reader**

- Attach SM to it's address Space
- Read Data from SM written by Writer



Functions used in SM

- `shmget()` → to Create the SM
- `shmat()` → to attach the SM with the address space of the process
- `shmdt()` → to detach the SM
- `shmctl()` → to Destroy the SM

shmget()

- int shmget(key_t key, size_t size, int shmflg);
- key-> Unique value that identifies the SM.
- size-> Size of the SM in bytes
- shmflg -> Permissions on the SM
- Returns valid identifier of SM
 - Used in shmat()
- In case of Unsuccessful → Returns -1
- #include<sys/ipc.h>
- #include<sys/shm.h>

shmat()

- void* shmat(int shmid, const void * shmaddr, int shmflg);
- shmid -> value returned by shmget().
- shmaddr -> where to attach the SM in the address space of the calling function
 - Address not known so write NULL. If shmaddr is a NULL pointer, the segment is attached at the first available address as selected by the system.
 - OS will assign it at a suitable location.
- shmflg -> if shmaddr is NULL, shmflg is 0.
- In case of Unsuccessful → Returns -1
- #include<sys/types.h>
- #include<sys/shm.h>

few more ...

- `ftok(const char *path, int id)`
 - Generates an IPC key
 - Path name of an existing file
 - returns the same key value for all paths that name the same file, when called with the same id value
 - If a different id value is given, or a different file is given, a different key is returned
- `shmdt(void * shmaddr)`
 - program detach itself.
- `shmctl(shmid,IPC_RMID,NULL)`
 - It is used to destroy the SM

SM Program-1

server.c

```
17 # define SHMSIZE 27 /* Size of Shared Memory*/
18 int main()
19 {
20     char c; int shmid; key_t key; char *shm, *s;
21     key = 5678; /* Some Key to Uniquely Identify the Shared Memory*/
22     /* Create the Segment*/
23     if((shmid = shmget(key, SHMSIZE, IPC_CREAT|0666))<0)
24     {
25         printf("Error in shmget\n");
26         exit(1); //Abnormal Termination
27     }
28     /*Attach the Segment to our Data Space*/
29     if((shm = shmat(shmid, NULL, 0)) == (char *) -1)//returns void char
30     {
31         printf("Error in shmat\n");
32         exit(1);
33     }
34     /* Now Put Something into the Shared Memory*/
35     s = shm;
36     for(c = 'a'; c <= 'z'; c++)
37     {
38         *s++ = c;
39     }
40     *s = 0; /* End with a NULL termination*/
41     /* Wait Until the other process changes the first character A to
42      '#' in the shared Memory*/
43     while(*shm != '#')
44     {
45         sleep(1); //delay for a 1 sec
46     }
47 }
```

```
shmdt(shm);
shmctl(shmid,IPC_RMID,NULL);
```

```
sanjeet@sanjeet-pc:~$ gcc server.c -o Server
sanjeet@sanjeet-pc:~$ ./Server
sanjeet@sanjeet-pc:~$ █
```

SM Program-1

```
17 # define SHMSIZE 27 /* Size of Shared Memory*/
18 int main()
19 {
20     int shmid; key_t key; char *shm, *s;
21     /* We need to get the segment named "5678" created by the server*/
22     key = 5678;
23     /* Locate the Segment*/
24     if((shmid = shmget(key, SHMSIZE, 0666))<0)
25     {
26         printf("Error in shmget\n");
27         exit(1);
28     }
29     /*Attach the Segment to our Data Space*/
30     if((shm = shmat(shmid, NULL, 0)) == (char *) -1)
31     {
32         printf("Error in shmat\n");
33         exit(1);
34     }
35     /* Read what the server Put in the Memory*/
36     for(s = shm; *s != \0; s++)
37     {
38         putchar(*s);
39     }
40     putchar('\n'); /* End with a NULL termination*/
41
42     /* Finally change the first character of the segment to '#',
        indicating we have read the segment*/
43     *shm = '#';
44     exit(0);
45 }
```

```
shmdt(shm);
```

```
sanjeet@sanjeet-pc:~$ gcc client.c -o ./Client
sanjeet@sanjeet-pc:~$ ./Client
abcdefghijklmnopqrstuvwxyz
sanjeet@sanjeet-pc:~$
```

Thank You
Any Questions?



Operating Systems (CS3000)

Lecture – 15
(Inter Process Communication - 2)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

sender1.c

```
17 int main()
18 {
19     int shmid;
20     void *SM;
21     char str[50];
22
23     shmid = shmget((key_t)1823,1024,0666|IPC_CREAT);
24     //Create SM with identifier 1823 of 1024 bytes, R/W permission for O, G, O.
25     printf("ID of SM in Sender: %d\n", shmid);
26
27     SM = shmat(shmid,NULL,0);
28     printf("SM is attached to %p in Sender\n", SM);
29
30     printf("Write Some Data into SM\n: ");
31     read(0, str, 50); //0--> read from KB, to str, max 50;
32     strcpy(SM, str); // copy data to SM
33     printf("The following Data written in SM: %s\n", (char *)SM);
34
35     return 0;
36 }
```

SM Program-2

receiver1.c

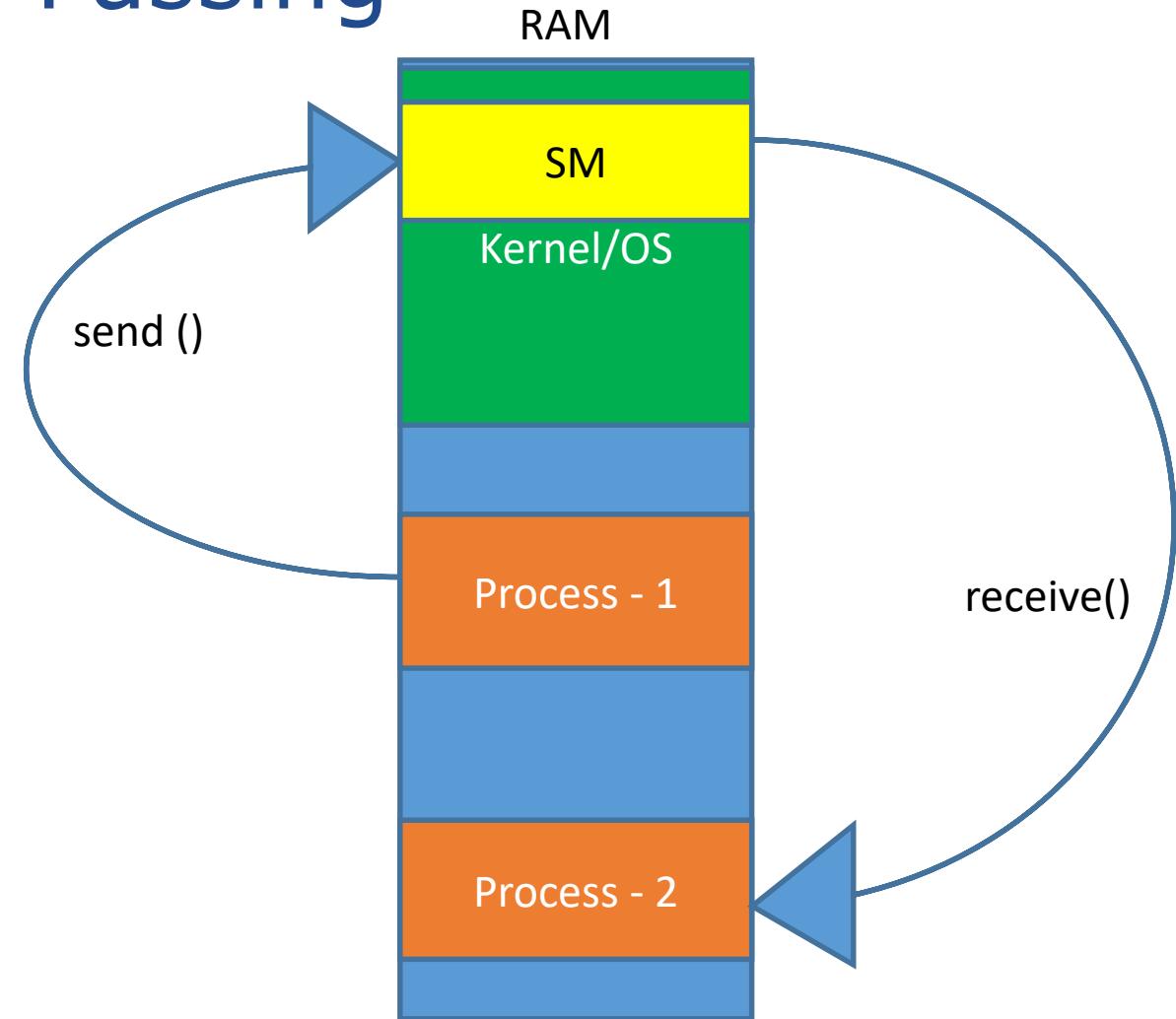
```
18 {
19     int shmid;
20     void *SM;
21
22     shmid = shmget((key_t)1823,1024,0666);
23     //we need the shmid, // needed as input for shmat; No IPC_CREATE
24
25     printf("ID of SM in Receiver: %d\n", shmid);
26
27     SM = shmat(shmid,NULL,0);
28     printf("SM is attached to %p in Receiver\n", SM);
29
30     printf("The following Data Read from SM: %s\n", (char *)SM);
31
32     return 0;
33 }
```

Problem with SM

- Synchronization needed between the processes

Message Passing

- SM is created in Kernel
- System calls are used
 - `send()`: Write to SM
 - `receive()`: Read from SM



IPC using Message Queues

- linked list of messages
- stored within the kernel
- identified by a message queue identifier.
- Functions
 - msgget() → To create a message queue/Open Existing
 - msgsnd() → To write message to message queue by Sender/New messages are added to the end of the queue
 - msgrcv() → To retrieve message from message queue by Receiver
 - msgctl() → The control function

IPC using Message Queues

- Sender
 - Create the MQ
 - Add data to the MQ
- Receiver
 - Retrieve the data from MQ
 - Delete the MQ

IPC using Message Queues

- `int msgget(key_t key, int msgflg);`
 - creates a new message queue
 - returns the message queue identifier associated with the key parameter for an existing message queue.
 - -1 : not successful
- `int msgsnd(int msqid, void *msgp, size_t msgsz, int msgflg);`
 - send a message to the message queue specified by the **msqid** parameter. It is returned by the `msgget()` function and used to identify the message queue to send the message to.
 - The ***msgp** parameter points to a user-defined buffer that must contain the following:
 - A field of type long int that specifies the **type of the message**.
 - A data part that contains the **data bytes of the message**.

IPC using Message Queues

- int msgsnd(int msqid, void *msgp, size_t msgsz, int msgflg);
 - The following structure is an example of what the user-defined buffer might look like for a message that has 5 bytes of data.

```
struct mymsg
{ long int mtype; /* message type */
char mtext[5]; /* message text */ }
```

- The value of mtype must be greater than zero. When messages are received with msgrcv(), the message type can be used to select the messages.
- The message data can be any length up to the system limit.

IPC using Message Queues

- int msgsnd(int msqid, void *msgp, size_t msgsz, int msgflg);
 - **msgsz**: Length of the data part of the message to be sent.
 - **msgflg**: If the message queue is full, the msgflg parameter specifies the action to be taken. The actions are as follows:
 - 0: Suspended
 - IPC_NOWAIT: do not wait for space to become available on the message queue and return immediately.

IPC using Message Queues

- `int msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);`
- The `msgrcv()` function reads a message from the message queue specified by the **msqid** parameter and places it in the user-defined buffer pointed to by the ***msgp** parameter.
- The ***msgp** parameter points to a user-defined buffer that must contain the following:
 - A field of type long int that specifies the **type of the message**.
 - A data part that contains the **data bytes of the message**.

IPC using Message Queues

- int msgrecv(int msqid, void *msgp, size_t msgsiz, long int msgtyp, int msgflg);
struct **mymsg**
{ long int **mtype**; /* message type */
char **mtext**[5]; /* message text */ }
- The value of **mtype** is the **type of the received message**, as specified by the sender of the message.
- The **msgsz** parameter specifies the size in bytes of the data part of the message.
 - The received message is truncated to **msgsz** bytes if it is larger than **msgsz** and the **MSG_NOERROR** flag is set in the **msgflg** parameter. The truncated part of the message is lost.

IPC using Message Queues

- `int msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);`
- The **msgtyp** parameter specifies the type of message to receive from the message queue as follows:
 - If `msgtyp = 0`, read the first message in the queue.
 - the messages will be retrieved in the same order in which they were written into the message queue
 - If `msgtyp > 0`, the first message of type “`msgtyp`” is only received.
 - If `msgtyp < 0`, the first message of the lowest type that is less than or equal to the absolute value of `msgtyp` is received.

IPC using Message Queues

- `int msgrcv(int msqid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);`
 - If a message of the desired type is not available on the message queue, the `msgflg` parameter specifies the action to be taken. The actions are as follows:
 - If the `IPC_NOWAIT` flag is set in the `msgflg` parameter, `msgrcv()` returns immediately with a return value of -1.
 - If 0 : suspend the process

IPC using Message Queues

- int msgctl(int msqid, int cmd, struct msqid_ds *buf);
- The msgctl() function allows the caller to control the message queue specified by the **msqid** parameter.
 - **msqid** Message queue identifier, a positive integer. It is returned by the msgget() function and used to identify the message queue on which to perform the control operation.
 - **cmd** Command, the control operation to perform on the message queue.
 - **buf** Pointer to the **message queue data structure** to be used to get or set message queue information.
 - msqid_ds

IPC using Message Queues

- A message queue is controlled by setting the **cmd** parameter to one of the following values:
 - **IPC_RMID** (0x00000000): Remove the message queue identifier **msqid** from the system and destroy any messages on the message queue.
 - **IPC_SET** (0x00000001): Set the user ID of the owner, the group ID of the owner, the permissions, and the maximum number of bytes for the message queue to the values in the **msg_perm.uid**, **msg_perm.gid**, **msg_perm.mode**, and **msg_qbytes** members of the **msqid_ds** data structure pointed to by ***buf**.
 - **IPC_STAT** (0x00000002): Store the current value of each member of the **msqid_ds** data structure into the structure pointed to by ***buf**.

```

8 #include<stdlib.h>
9 #include<stdio.h>
10 #include<string.h>
11 #include<unistd.h>
12 #include<sys/types.h>
13 #include<sys/ipc.h>
14 #include<sys/msg.h>
15 #define MAX_TEXT 512 //maximum length of the message that can be sent
16     allowed
17 struct my_msg{
18     long int msg_type;//Fixed long int
19     char some_text[MAX_TEXT];//store data in this and send to MQ
20 };
21 int main()
22 {
23     int running=1;//while loop T/F 1=Send 0=Stop Sending
24     int msgid;
25     struct my_msg some_data;
26     char buffer[50]; //array to store user input
27     msgid=msgget((key_t)12345,0666|IPC_CREAT);//MQ with Queue 14534
28     if (msgid == -1) // -1 means the message queue is not created
29     {
30         printf("Error in creating queue\n");
31         exit(0);
32     }
33     //on success return the MQ Identifier.
34     while(running)
35     {
36         printf("Enter some text:\n");
37         fgets(buffer,50,stdin);//what user inputs keep in
38             buffer, howmuch data , from where to read KB
39         some_data.msg_type=1;//used in Receive
40         strcpy(some_data.some_text,buffer);//copy from buffer
41             to structure
42         if(msgsnd(msgid,(void *)&some_data, MAX_TEXT,0)==-1)
43         {
44             printf("Msg not sent\n");
45         }
46         if(strncmp(buffer,"end",3)==0)
47         {
48             running=0;
49         }
50     }

```

MQ_Send1.c

```

8 #include<stdlib.h>
9 #include<stdio.h>
10 #include<string.h>
11 #include<unistd.h>
12 #include<sys/types.h>
13 #include<sys/ipc.h>
14 #include<sys/msg.h>
15 struct my_msg{
16     long int msg_type;
17     char some_text[BUFSIZ];//stdio.h -->1024
18 };//Both elements of the stucture are sent and here also received.
19 int main()
20 {
21     int running=1;
22     int msgid;
23     struct my_msg some_data;
24     long int msg_to_rec=0;//the messages will be retrieved in the
25                             same order in which they were written into the message queue.
26     msgid=msgget((key_t)12345,0644);
27     while(running)
28     {
29         msgrcv(msgid,(void *)&some_data,BUFSIZ,msg_to_rec,0);
30         printf("Data received: %s\n",some_data.some_text);
31         if(strncmp(some_data.some_text,"end",3)==0)
32         {
33             running=0;
34         }
35     }
36     msgctl(msgid,IPC_RMID,0);//delete the MQ

```

MQ_Receive1.c

Thank You

Any Questions?

Operating Systems (CS3000)

Lecture – 15
(Inter Process Communication - 3)

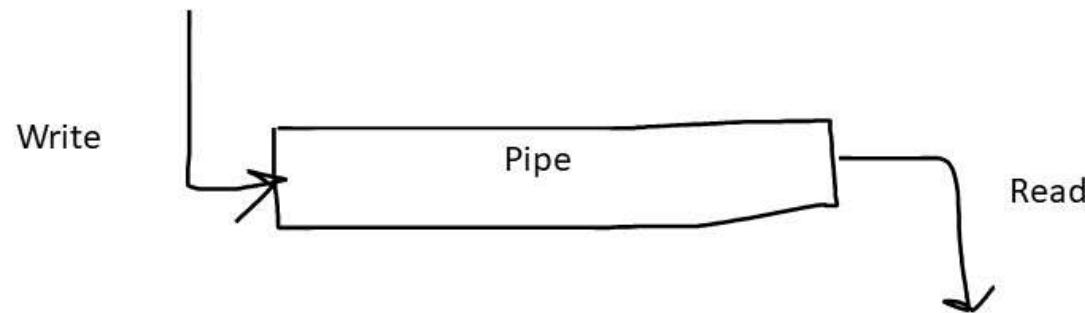


INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Message Passing using Ordinary Pipes

- Pipe is a communication between parent and child process
- Communication is achieved by one process writing into the pipe and other reading from the pipe
- To achieve the pipe system call, create two descriptors, one to write into the file and another to read from the file.
- Parent \leftrightarrow Child



Message Passing using Ordinary Pipes

- int pipe(int pipedes[2]);
- This system call would create a pipe for one-way communication
- creates two descriptors,
 - one is connected to read from the pipe pipedes[0]
 - one is connected to write into the pipe. pipedes[1]
- Whatever is written into pipedes[1] can be read from pipedes[0].
- This call would return zero on success and -1 in case of failure.

Message Passing using Ordinary Pipes

- `ssize_t read(int fd, void *buf, size_t count)`
- to read from the specified file with arguments of file descriptor `fd`, proper buffer with allocated memory and the size of buffer.
- This call would return the number of bytes read (or zero in case of encountering the end of the file) on success and -1 in case of failure.

Message Passing using Ordinary Pipes

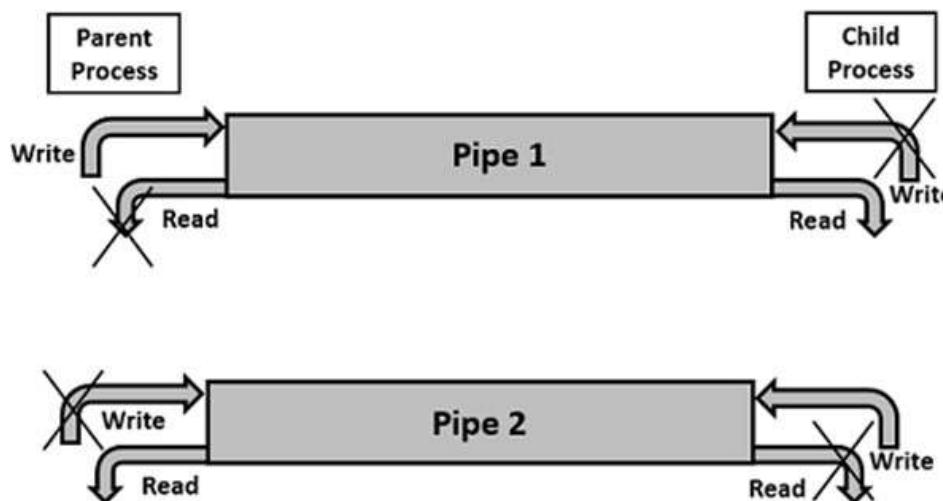
- `ssize_t write(int fd, void *buf, size_t count)`
- to write to the specified file with arguments of the descriptor `fd`, a proper buffer with allocated memory and the size of buffer.
- This call would return the number of bytes written (or zero in case nothing is written) on success and -1 in case of failure.

Message Passing using Ordinary Pipes

- Algorithm
- **Step 1** – Create a pipe.
- **Step 2** – Create a child process.
- **Step 3** – Parent process writes to the pipe.
- **Step 4** – Child process retrieves the message from the pipe and writes it to the standard output.
- **Step 5** – Repeat step 3 and step 4 once again.

Two-way Communication Using Ordinary Pipes

- However, what if both the parent and the child needs to write and read from the pipes simultaneously, the solution is a two-way communication using pipes.
 - Two pipes are required to establish two-way communication.



Two-way Communication Using Ordinary Pipes

- Algorithm
- **Step 1** – Create two pipes. First one is for the parent to write and child to read, say as pipe1. Second one is for the child to write and parent to read, say as pipe2.
- **Step 2** – Create a child process.
- **Step 3** – Close unwanted ends as only one end is needed for each communication.
- **Step 4** – Close unwanted ends in the parent process, read end of pipe1 and write end of pipe2.
- **Step 5** – Close the unwanted ends in the child process, write end of pipe1 and read end of pipe2.
- **Step 6** – Perform the communication as required.

Two-way Communication Using Ordinary Pipes

- Algorithm
- **Step 1** – Create pipe1 for the parent process to write and the child process to read.
- **Step 2** – Create pipe2 for the child process to write and the parent process to read.
- **Step 3** – Close the unwanted ends of the pipe from the parent and child side.
- **Step 4** – Parent process to write a message and child process to read and display on the screen.
- **Step 5** – Child process to write a message and parent process to read and display on the screen.

```
8 #include<stdio.h>
9 #include<unistd.h>
10
11 int main()
12 {
13     int pipefds[2];//two descriptors [0] -> read, [1] -> write
14     int returnstatus;
15     int pid;
16     char writemessages[2][20]={"Hi", "Hello"};
17     char readmessage[20];
18     returnstatus = pipe(pipefds);//This system call would create a pipe
19         for one-way communication i.e., it creates two descriptors, first
20         one is connected to read from the pipe and other one is connected
21         to write into the pipe.
22     if (returnstatus == -1)
23     {
24         printf("Unable to create pipe\n");
25         return 1;
26     }
27     pid = fork();
28
29     // Child process
30     if (pid == 0)
31     {
32         //sleep(2);
33         read(pipefds[0], readmessage, sizeof(readmessage));
34         printf("Child Process - Reading from pipe - Message 1 is %s\n",
35             readmessage);
36         read(pipefds[0], readmessage, sizeof(readmessage));
37         printf("Child Process - Reading from pipe - Message 2 is %s\n",
38             readmessage);
39     }
40     else
41     { //Parent process
42         printf("Parent Process - Writing to pipe - Message 1 is %s\n",
43             writemessages[0]);
44         write(pipefds[1], writemessages[0], sizeof(writemessages[0]));
45         printf("Parent Process - Writing to pipe - Message 2 is %s\n",
46             writemessages[1]);
47         write(pipefds[1], writemessages[1], sizeof(writemessages[1]));
48     }
49 }
50
51 }
```

Pipe1.c

```
8 #include<stdio.h>
9 #include<unistd.h>
10
11 int main()
12 {
13     int pipefds1[2], pipefds2[2];
14     int returnstatus1, returnstatus2;
15     int pid;
16     char pipe1writemessage[20] = "Hi";
17     char pipe2writemessage[20] = "Hello";
18     char readmessage[20];
19     returnstatus1 = pipe(pipefds1);
20
21     if (returnstatus1 == -1)
22     {
23         printf("Unable to create pipe 1 \n");
24         return 1;
25     }
26     returnstatus2 = pipe(pipefds2);
27
28     if (returnstatus2 == -1)
29     {
30         printf("Unable to create pipe 2 \n");
31         return 1;
32     }
33     pid = fork();
34
35     if (pid != 0) // Parent process
36     {
37         close(pipefds1[0]); // Close the unwanted pipe1 read side
38         close(pipefds2[1]); // Close the unwanted pipe2 write side
39         printf("In Parent: Writing to pipe 1 - Message is %s\n",
40               pipe1writemessage);
41         write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));
42         read(pipefds2[0], readmessage, sizeof(readmessage));
43         printf("In Parent: Reading from pipe 2 - Message is %s\n",
44               readmessage);
45     }
46     else
47     { //child process
48         close(pipefds1[1]); // Close the unwanted pipe1 write side
49         close(pipefds2[0]); // Close the unwanted pipe2 read side
50         read(pipefds1[0], readmessage, sizeof(readmessage));
51         printf("In Child: Reading from pipe 1 - Message is %s\n",
52               readmessage);
53         printf("In Child: Writing to pipe 2 - Message is %s\n",
54               pipe2writemessage);
55         write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage));
56     }
57     return 0;
58 }
```

Pipe2.c

Thank You

Any Questions?

Operating Systems (CS3000)

Lecture – 16
(Inter Process Communication - 5)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

IPC using Named Pipe

- Communication between unrelated processes
- Execute client program from one terminal and the server program from another terminal.
- Named Pipe supports bi-directional communication.

Message Passing using Pipes

```
int mkfifo(const char *pathname, mode_t mode)
```

- FIFO special file, which is used for named pipe.
- The file name can be either absolute path or relative path.
- The file mode information is as described as permission
- Return zero on success and -1 in case of failure.

Message Passing using Named Pipes (FIFO)

- Algorithm: server process.
- **STEP1:** Creates a named pipe (using library function mkfifo()) with name “fifofile” in directory, if not created. (Only one process will create the pipe)
- **STEP2:** Opens the named pipe for read purpose.
- **STEP3:** Waits infinitely for a message from the client.
- **STEP4:** Print message received from the client and close the file.
- **STEP5:** Opens the named pipe for write purpose.
- **STEP6:** Accepts string from the user.
- **STEP7:** Sends a message to the client and close the named pipe.
- **STEP8:** Repeats infinitely until the user enters the string “end”.

Message Passing using Named Pipes

- Algorithm: Client process
- **STEP1:** Opens the named pipe for write purpose.
- **STEP2:** Accepts string from the user.
- **STEP3:** Sends a message to the server and close the named pipe.
- **STEP4:** Open the named pipe for read purpose
- **STEP5:** Waits for the message from the server and prints the message.
- **STEP6:** Close the named pipe
- **STEP7:** Repeats infinitely until the user enters the string “end”.

```
//Server Named Pipe
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int fd;
    char * myfifo = "/home/sanjeet/Desktop/Progs/Pipes/myfile7.txt";
    mknod(myfifo, 0666); //myfifo is FIFO path name ; permissions 0666

    char arr1[50], arr2[50];
    //int ch=1;
    while (1)
    {
        fd = open(myfifo, O_WRONLY);
        //Open FIFO for write only keep ref. file descriptor
        fgets(arr2, 50, stdin); //Take input arr2 from user upto 80
        //printf("write file fd %d\n", fd);
        write(fd, arr2, strlen(arr2)+1); //write
        close(fd);
        //sleep(10);
        fd = open(myfifo, O_RDONLY); // Open FIFO for Read only
        read(fd, arr1, sizeof(arr1));
        //printf("read file fd %d\n", fd);
        printf("Server Reading: %s\n", arr1);
        close(fd);
    }
    return 0;
}
```

ServerNP.c

```
//Client Named Pipe
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd1;
    char * myfifo = "/home/sanjeet/Desktop/Progs/Pipes/myfile7.txt";
    mkfifo(myfifo, 0666);
    char str1[50], str2[50];
    while (1)
    {
        //sleep(10);
        fd1 = open(myfifo,0_RDONLY); //open in read only
        //printf("read file fd %d\n", fd1);
        read(fd1, str1, 50);
        printf("Client Reading: %s\n", str1);
        close(fd1);

        fd1 = open(myfifo,0_WRONLY);
        //printf("write file fd %d\n", fd1);
        fgets(str2, 50, stdin);
        write(fd1, str2, strlen(str2)+1);
        close(fd1);
    }
    return 0;
}
```

ClientNP.c

Thank You
Any Questions?



Operating Systems (CS3000)

Lecture – 17
(Inter Process Communication - 4)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Signals

- A signal is an asynchronous event which is delivered to a process.
 - The event can occur at any time (may be unrelated to the execution of the process)
 - An event which is generated to notify a process that some important situation has risen
 - OS -> Process
 - Process -> Process
- Signals are raised by some error conditions
 - Memory segment violations
 - Floating point processor errors
 - Illegal instructions – e.g. user types ctrl-C

Signals

- What Process will do on receipt of Signal?
 - Will stop what its doing and take some action
- Signals are defined in the header file <signal.h> as a macro constant
 - SIGINT
 - SIGFPE
 - SIGKILL
 - SIGUSR1
 - SIGUSR2
- Every signal has a name and an associated number.

Signals

- How to raise a signal?
 - Program
 - System generated
- What happens?
 - Default defined action
- Doesn't want default?
 - User defined action/ handling of signal
- Ignore Signal?
 - Yes
- SIGSTOP/SIGKILL

Signal handling

- On Signal receipt, the process has a choice of action.
 - The process can ignore the signal – Signal is discarded
 - Can specify a handler function
 - Accept the default action for the specific kind of signal
- The program can register a handler function using function such as `signal()` or `sigaction()`.
- If the signal has not been neither handled nor ignored, its default action takes place.

Signal handling

int signal () (int signum, void (*func)(int))

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

void sig_handler(int signum)
{
    printf("I am in sig_handler ()\n");
}

int main()
{
    signal(SIGUSR1,sig_handler);
    printf("I am in main()\n");
    raise(SIGUSR1);
    printf("I am in main() again\n");
    return 0;
}
```

Signal handling

- int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
- To change the signal action

Thank You

Any Questions?

Operating Systems (CS3000)

Lecture – 18
(Threads - 1)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Motivation

- If a man needs 5 days to do a job then how many days will it take for 5 men to do the same job?
- Only 1 day

Agenda

- Basic Concept of Threading
- How thread is created and destroyed.
- How thread is different from Processes.
- How different Operating System support thread in different ways.

```
# include <stdio.h>

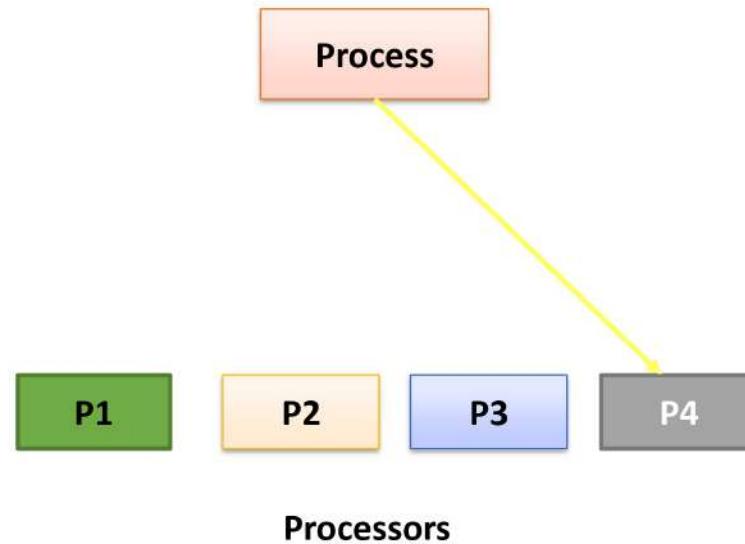
unsigned long sumAll()
{
int i =0;
unsigned long sum=0;

while(i < 100000000)
{
sum +=i;
i++;}
}

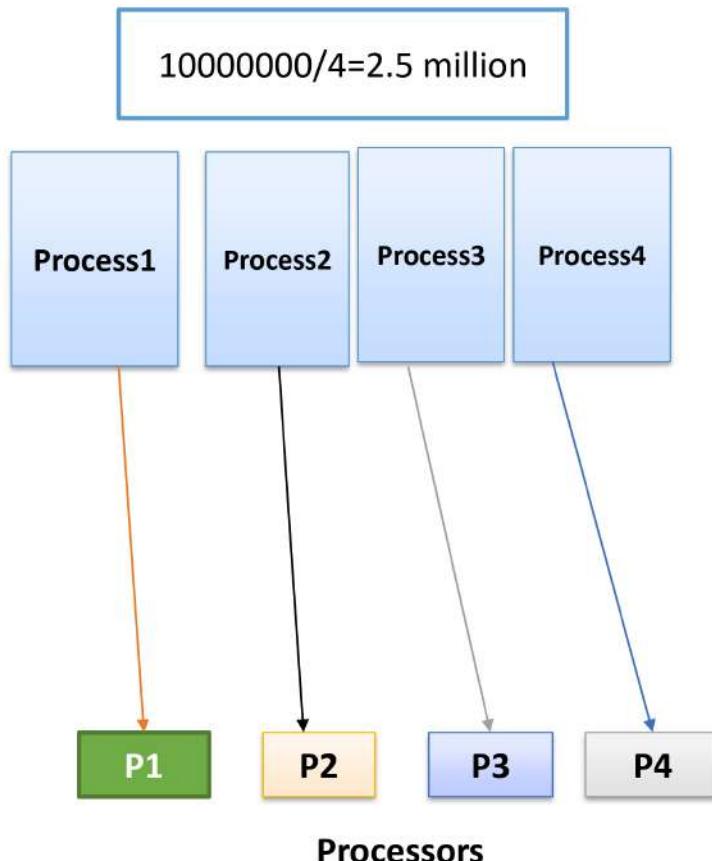
return sum;

int main()
{
unsigned long sum;
srandom(time (NULL));
sum = summAll();
printf("%lu\n", sum);
}
```

Motivation



Motivation – Speedup with Multiple Process



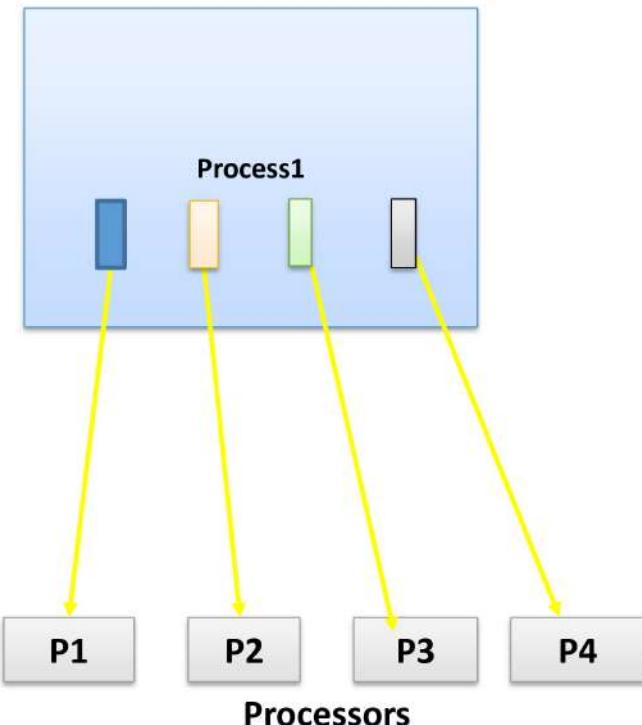
- Create 4 processes, each loop does $1/4^{\text{th}}$ of the work.
- **Properties**
 - Create 4 processes using fork system call
 - Each process is isolated from each other
 - Each process has own memory map
 - Instructions
 - Data
 - Stack

Motivation

- Problems on Creating 4 Processes
- Overhead
 - due to considerable amount of system call.
 - due to large portion of these processes are similar
- Can we reduced this overhead?

Thread

$10000000/4=2.5\text{ million}$

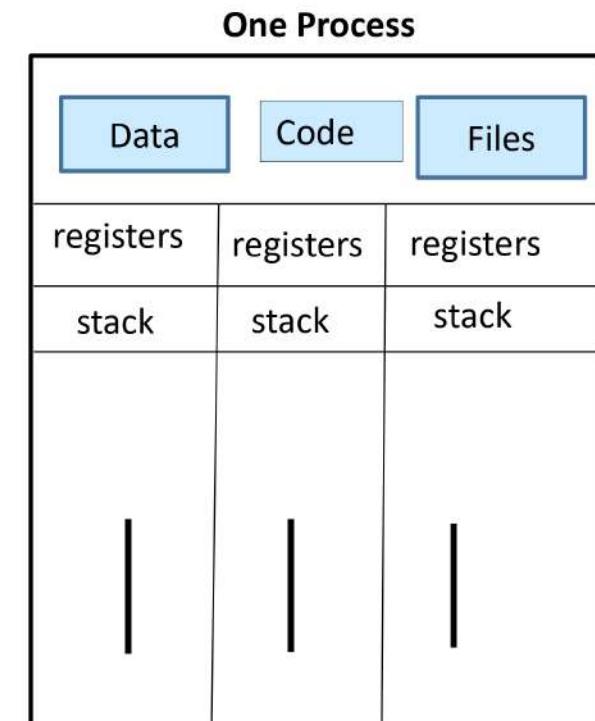


- Properties

- Create 1 process with 4 threads
- 4 threads need to be created (pthread library)
- Each thread is not isolated from each other
- Management of thread with fewer or no system calls
- Each thread has own stack and registers
- Threads share
 - Instructions
 - Data
 - Heap

Thread

- A thread is a basic unit of CPU utilization.
- Separate streams of execution in single process.
- Threads in the process are not isolated from each other.
- Each thread state /thread control block /thread execution context contains
 - Thread id
 - Program counter
 - Registers
 - Stack



Advantages of Threads over Processes

- Lightweight
- Efficient communication between entities
- Efficient context switching

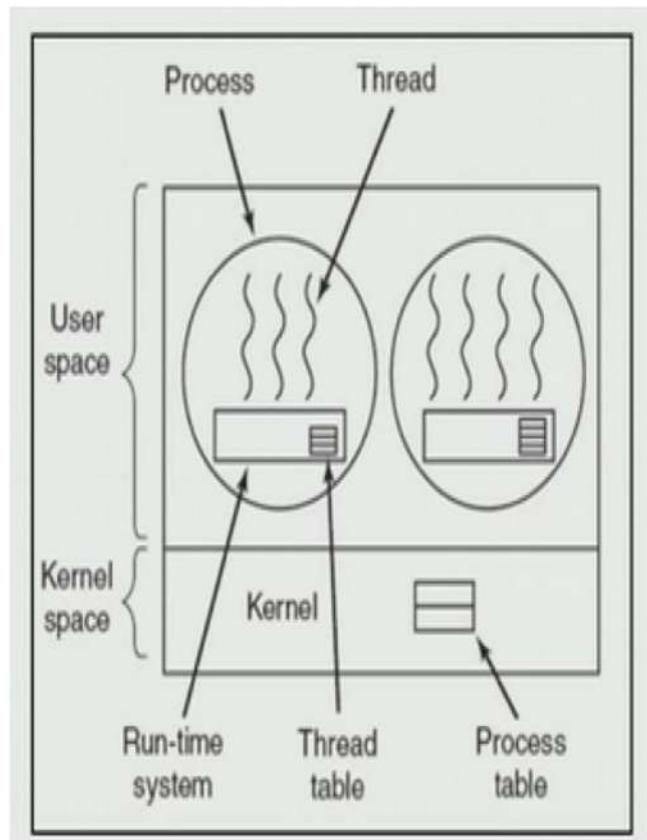
Thread vs Processes

- A thread has no data segment or heap
- A thread cannot live on its own. It needs to be attached to a process.
- There can be more than one thread in a process. Each thread has its own stack.
- If a thread dies, its stack is reclaimed.
- If a process dies, all threads die
- A process has code, heap, stack and other segments.
- We can consider a process has at-least one thread.
- Each process has its own address space.

Who manages Thread?

- There are two Strategies
- User Level Thread
 - Threads are managed by user level thread library.
 - Kernel knows nothing about the threads.
- Kernel Level Thread
 - Threads directly supported by the kernel.

User Level Thread



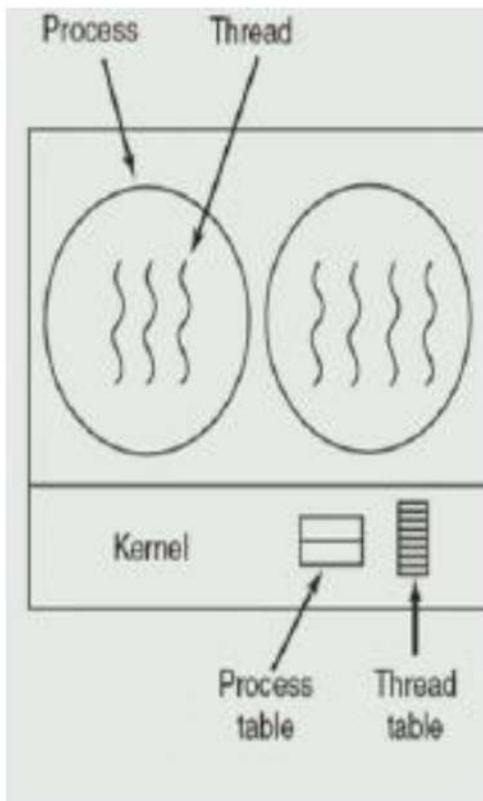
- **Advantages:**

- Fast (really lightweight)
- no system call to manage threads.
- The thread library does everything.
- Can be implemented in an OS that does not support threading.
- Switching is fast. No switch from user to kernel mode.

- **Disadvantages:**

- Lack of coordination between kernel and threads.
- Problem during blocking/wait system calls.
 - If one thread invokes a block system call, all threads need to wait

Kernel Level Thread



- **Advantages**

- Scheduler can decide to give more time to a process having large number of threads than process having small number of threads.

- **Disadvantages**

- The kernel level threads are slow (they involve kernel invocations)
- Overheads in the kernel.

Operating Systems (CS3000)

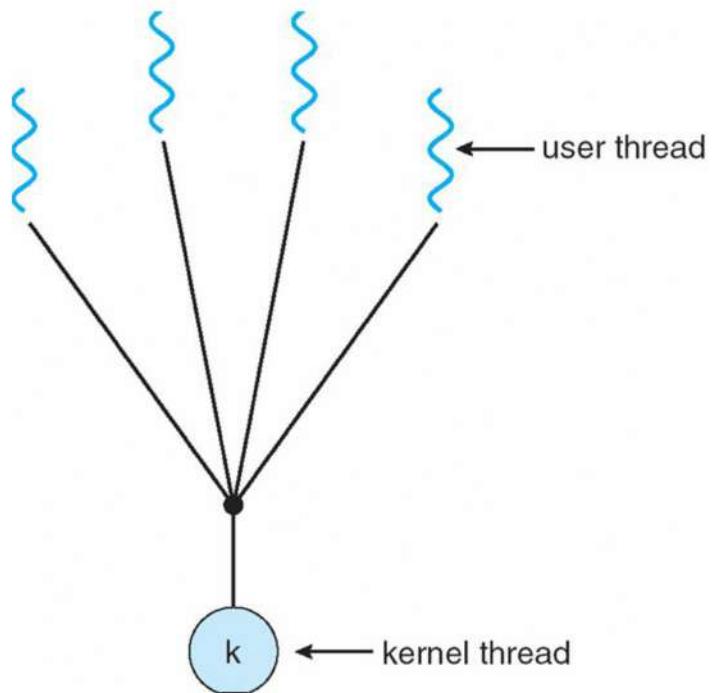
Lecture – 19
(Threads - 2)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

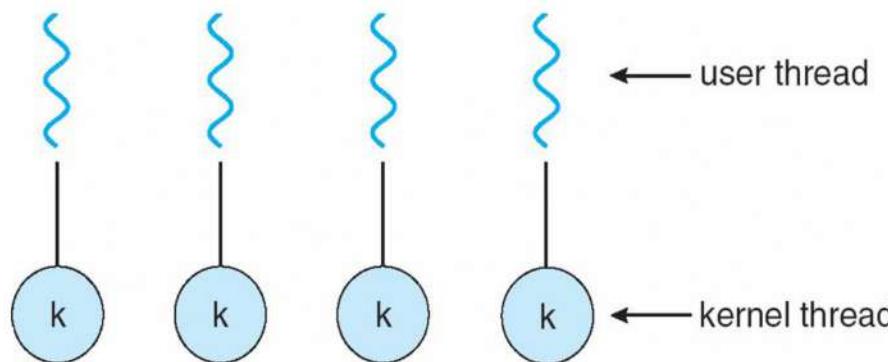
Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Multithreading model - Many-to-One



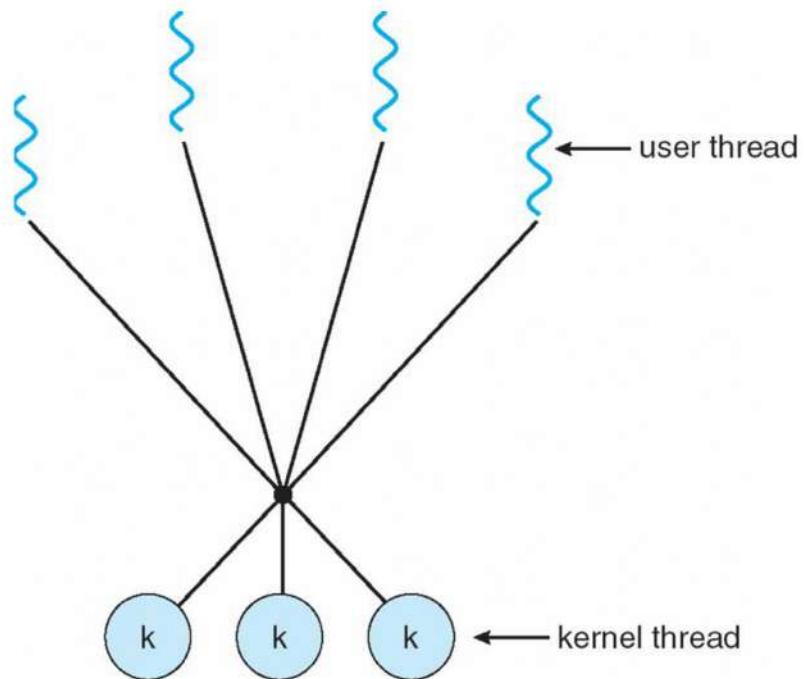
- Many-to-one
 - Many user level thread mapped to one kernel level thread
 - Example: Solaris Green thread (Earlier version of Java)
- **Disadvantage:**
 - Blocking system call by one thread, will block entire process.

Multithreading model - One-to-One



- One-to-one
 - Each user level thread mapped to one kernel level thread
 - Provide concurrency
 - Multiple threads run in parallel on multiprocessors
 - Example: Windows, Linux
- **Disadvantage:-**
 - Creation of a user thread requires to create/link the corresponding kernel thread
 - Overhead of creating kernel threads
 - Restrictions in number of threads supported by the system

Multithreading model - Many-to-Many



- Many-to-Many Model
 - Allows many user level thread to be mapped to **many** kernel level threads
 - Example:- Solaris (prior to version 9)

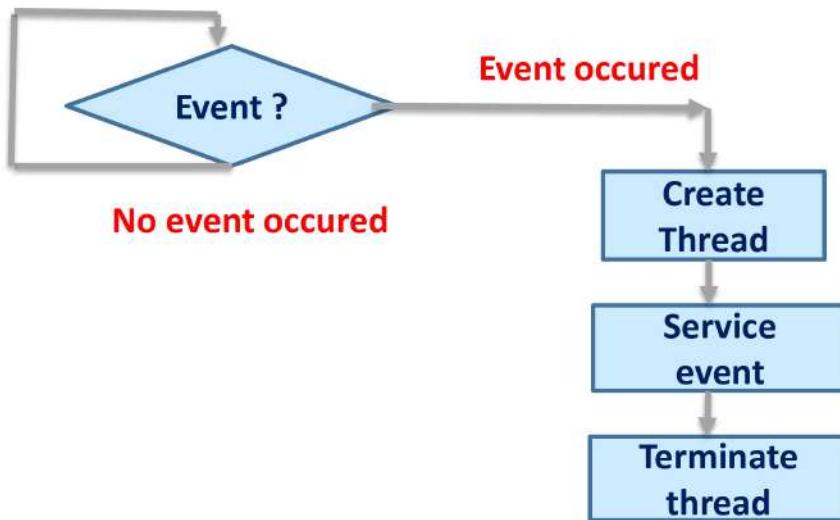
Benefit of Multi-threading

- **Responsiveness**:- It allow a program to continue running even if part of it is blocked.
- **Resource Sharing**:- Threads share the memory and the resources of the process to which they belong.
- **Economy**:- Threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Scalability**:- Threads may be running in parallel on different processors. Basically increases the concurrency.

How to create ULTs?

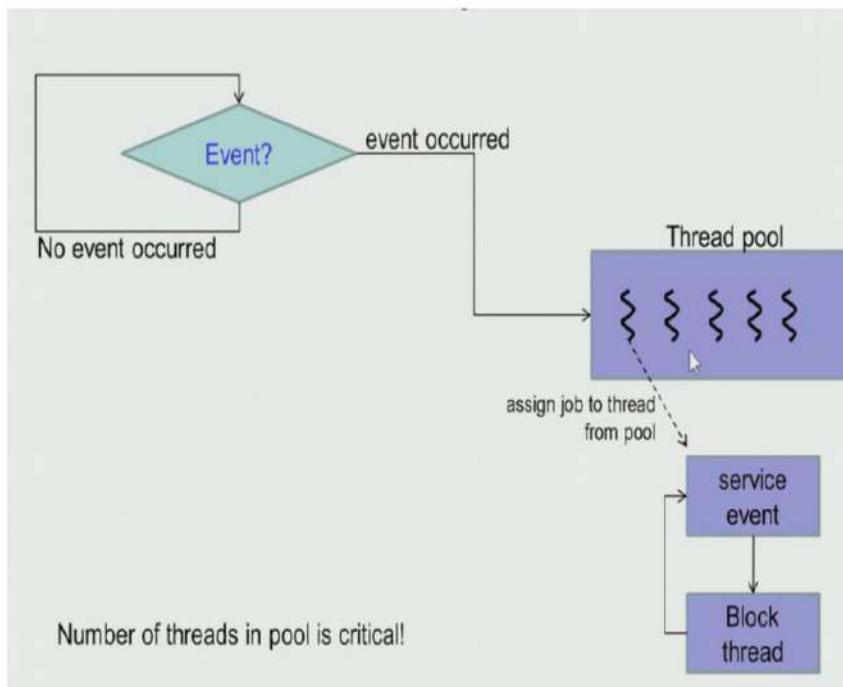
- Using Thread Libraries
 - pthread
 - Win32 thread
 - Java threads
- Provides API for creating and managing threads

Typical usage of Thread



- Whenever an event occurs
 - A thread gets created
 - Service that event
 - Terminate after completion
- Every time you have to create a thread for each event
- Creation and Termination of thread incur overhead

Thread pools



- Create thread pool of 50-100 threads when process starts.
- All threads will reside in blocked state
- Whenever a request comes
 - Wake up one of the thread from the thread pool
 - Service the request
 - Thread gets returned to the thread pool.
- Reduces overhead of creation and termination of a thread
- Once the application will terminate, all the threads of thread pool get terminated.

Threading Issues

- What happens when thread invokes fork?
 - Duplicate all threads?
 - Not easily done, other threads may be running or blocked in a system call or in a critical section.
 - Duplicate only caller thread?
 - More feasible
- Should only the thread terminate or the entire process?
- All the decisions are taken care by the OS designer.

Thread Cancellation Issues

Terminating a thread before it has finished

- Two approaches for the cancellation of thread
- Asynchronous cancellation:-
 - One thread immediately terminates the target thread.
 - Difficulties
 - May not free the resources allocated to a cancelled thread.
 - Inconsistency
 - Because cancellation occurs in middle of updating the data, which is shared by other threads.
- Deferred cancellation:-
 - Allows the target thread to periodically check if it should be cancelled.
 - Difficulties
 - Target thread wait for periodic check unnecessarily.

Thread Scheduling

Operating Systems (CS3000)

Lecture – 20
(Threads - 3)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM
AN INSTITUTE OF NATIONAL PRIORITY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

pthread library

- Create a thread in a process
 - int **pthread_create** (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void*), void *arg);
 - It takes four argument
 - 1st argument- Pointer to Thread_id
 - 2nd argument- specify several properties of the thread (stack size, type of thread, etc.)
 - 3rd argument- pointer to a function where the new thread will begin execution
 - 4th argument- arg which is a pointer to the arguments to the start_routine function (3rd argument)
- Destroy a thread
 - void **pthread_exit** (void *retval);
 - pass a pointer to the return value, in order to pass the return status of the thread
- Join:- wait for a specific thread to complete
 - int **pthread_join** (pthread_t thread, void **retval); //we can keep the return value from thread in 2nd argument
 - the parent thread will wait for it to complete by calling the pthread_join() function.

pthread library

```
void* thread_fn()
{
    printf("Hello I am Thread\n");
    sleep(2);
    printf("Thread Ends\n");
}

int main(int argc, char* argv[])
{
    pthread_t t1, t2;

    if (pthread_create(&t1, NULL, &thread_fn, NULL)){
        return 1;
    }
    if (pthread_create(&t2, NULL, &thread_fn, NULL)){
        return 2;
    }
    if (pthread_join(t1, NULL)){
        return 3;
    }
    if (pthread_join(t2, NULL)){
        return 4;
    }
    return 0;
}
```

pthread library

$0+1+2 + \dots + 99 = 4950$

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

int sum[4];

void *thread_fn (void *arg)
{
    int id =(int) arg;
    int start =id*25;
    int i=0;

    while(i<25)
    {
        sum[id]+=(i+start);
        i++;
    }
    return NULL;
}
```

```
int main(){
    pthread_t t1, t2, t3, t4;
    pthread_create(&t1, NULL, &thread_fn, (void *)0);
    pthread_create(&t2, NULL, &thread_fn, (void *)1);
    pthread_create(&t3, NULL, &thread_fn, (void *)2);
    pthread_create(&t4, NULL, &thread_fn, (void *)3);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    printf("%d\n", sum[0]+sum[1]+sum[2]+sum[3]);
    return 0;
}
```

pthread library

```
void *thread_function(void *arg);

int i,j;

int main(){
    pthread_t a_thread;
    pthread_create(&a_thread, NULL, &thread_function, NULL);

    pthread_join(a_thread, NULL);

    printf("Inside Main Program\n");
    for(j=20;j<25;j++)
    {
        printf("%d\n",j);
        sleep(1);
    }
    return 0;
}

void *thread_function(void *arg){
// the work to be done by the thread is defined in this function
    printf("Inside Thread Function\n");
    for(i=0;i<5;i++)
    {
        printf("%d\n",i);
        sleep(1);
    }
    return NULL;
}
```

pthread library

```
int i,n,j;
int main(){
    char *m="3";
    pthread_t a_thread;
    void *result;

    pthread_create(&a_thread, NULL, &thread_function, (void *)m);

    pthread_join(a_thread, (void **)&result); //return value of thread;

    printf("Thread joined\n");
    for(j=20;j<25;j++){
        printf("%d\n",j);
        sleep(1);
    }
    printf("thread returned %s\n", (char *)result);
    return 0;
}
void *thread_function(void *arg){
    n=atoi(arg);
    for(i=0;i<n;i++){
        printf("%d\n",i);
        sleep(1);
    }
    pthread_exit("Done"); //return value
    return NULL;
}
```

Thank You
Any Questions?



Operating Systems (CS3000)

Lecture – 21

(Threads - Scheduling)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Motivation

- Issue: In multithreaded programs communication between the kernel and the thread library is major concern.
- Applicable for many-to many

Lightweight Process

- An intermediate data structure between user and kernel threads known as lightweight process (LWP)
- To user level thread LWP appears to be *virtual processor* on which the application can schedule a user thread to schedule.
- Each LWP is attached to kernel thread, the kernel thread executes on physical processors.
- Scheduler Activation:
 - Scheme for communicating between the user thread-library and the kernel.

Thread Scheduling

.Process contention Scope (PCS):

- Competition for the CPU takes place among threads belonging to the same process.
- The thread library schedules user-level threads to run on an available LWP.
- Apply for many-to many and many-to-one mode
- PCS is done according to priority

Thread Scheduling

.System Contention Scope (SCS):

- Deciding kernel thread to schedule onto a CPU
- Apply on one to one model

pthread Scheduling

- PTHREAD_SCOPE_PROCESS: Schedules threads using PCS Scheduling
- PTHREAD_SCOPE_SYSTEM: Schedules threads using SCS scheduling
- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- First parameter: attributes set for the thread
- Second parameter: value of PTHREAD_SCOPE_PROCESS or PTHREAD_SCOPE_SYSTEM

pthread Scheduling

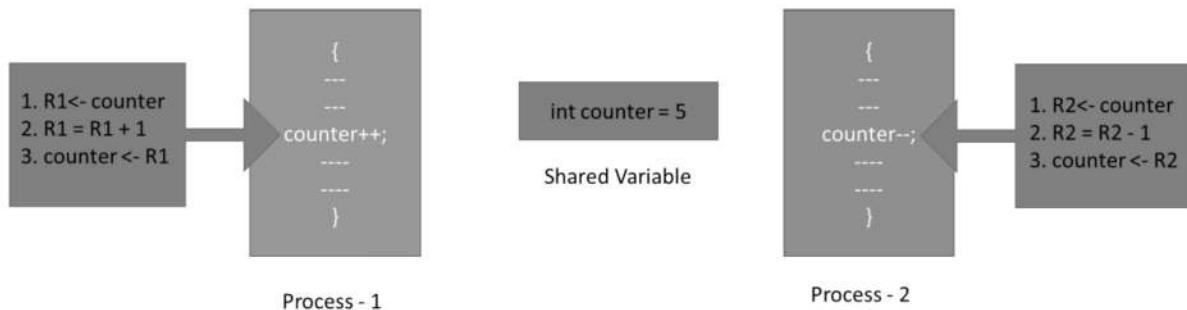
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`
- First parameter: attributes set for the thread
- Second parameter: value of pointer to the int value that is set to the current value of the contention scope.

Thank You
Any Questions?



Process Synchronization Notes

Why Synchronization?



What could happen because of Shared Memory?

- Final Value of Shared Variable depends
 - On the Order of Execution of Instructions
 - On which instruction the process is preempted

Race Condition

- Situation when several processes access and manipulate the same data
- O/P depends on the order in which the accesses to the data took place.

Critical Section

- It is the part of the program where shared resources are accessed.

Synchronization

- Ensure only one process manipulates the shared data at a time

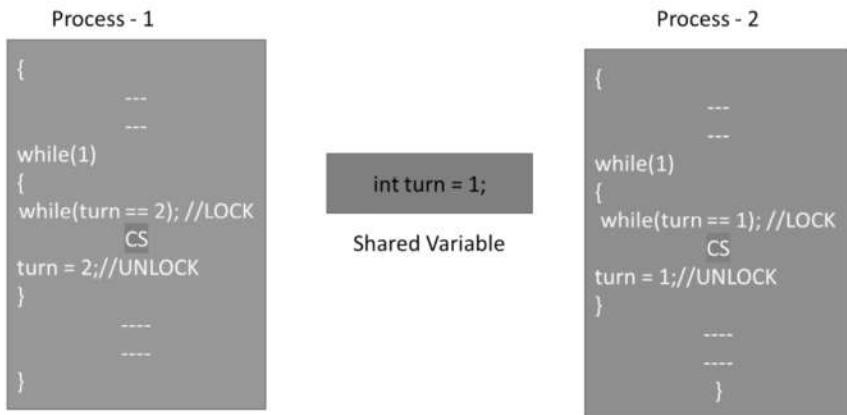
Solution to Critical Section Problem

- Any solution should satisfy the following requirements
 - Mutual Exclusion
 - No more than one process in critical section at a given time
 - Progress
 - When no process is in the critical section, any process that requests enter into the Critical Section must be permitted without any delay
 - No starvation (bounded wait)

- There is an upper bound on the number of times a process enters the critical section, while another is waiting
- Process should not wait infinitely in order to gain access into the Critical Section

Software Based Synchronization Solutions

Software Solution - Basic



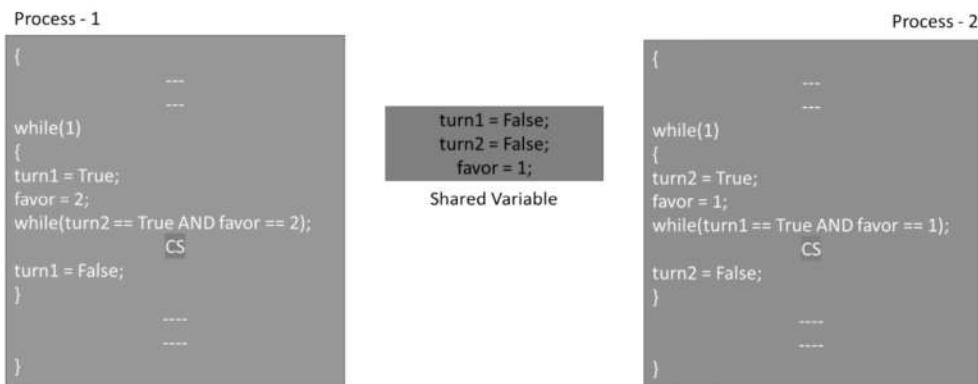
ME Guaranteed

Progress is not Guaranteed

Bounded Wait Guaranteed

Alternate Execution: P1 -> P2 -> P1 -> P2.

Software Solution – Peterson's Solution



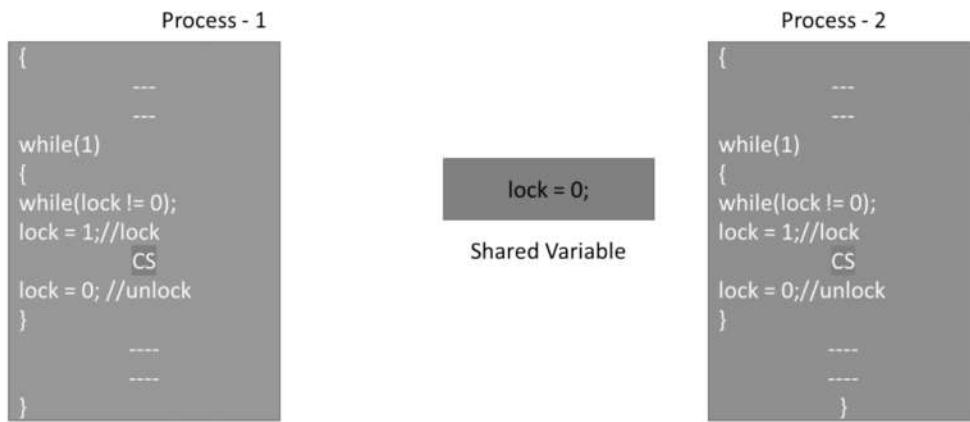
ME Guaranteed

Progress is Guaranteed

Bounded Wait Guaranteed

Peterson's solution works efficiently for two processes

Hardware Based Synchronization Solutions



Why the Previous Software Solution Failed?

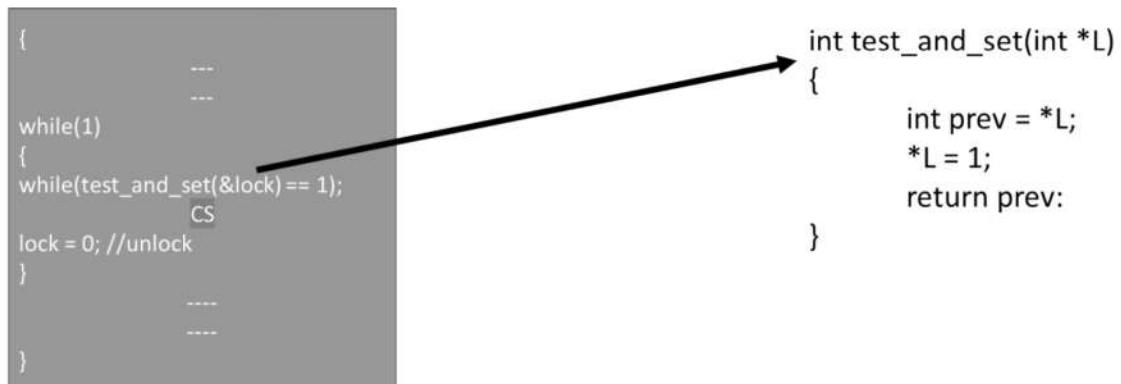
- two set of instructions written in S/W.
- How to ensure No Context Switch between them? (Not Possible. So No ME)
- In some processors, dedicated instructions are there to do this.

Hardware Solution : Test and Set Instruction (TAS)

Software Equivalent Code for TAS Instruction is:

```
int test_and_set(int *L)
{
    int prev = *L;
    *L = 1;
    return prev;
}
```

Solution of Critical Section Problem using Test and Set Instruction



Hardware Solution : xchg Instruction

Software Equivalent Code for xchg Instruction is:

```
int xchg(int *L, int v)
{
    int prev = *L;
    *L = v;
    return prev;
}
```

Solution of Critical Section Problem using xchg Instruction

```
int xchg(int *L, int v)
{
    int prev = *L;
    *L = v;
    return prev;
}

int XCHG(addr, value)
{
    %eax = value;
    xchg %eax, (addr)
}

void release(int *locked)
{
    locked = 0;
}

void acquire(int *locked)
{
    while(1)
    {
        if(XCHG(locked, 1) == 0)
            break;
    }
}
```

Call to

Equivalent Software Code

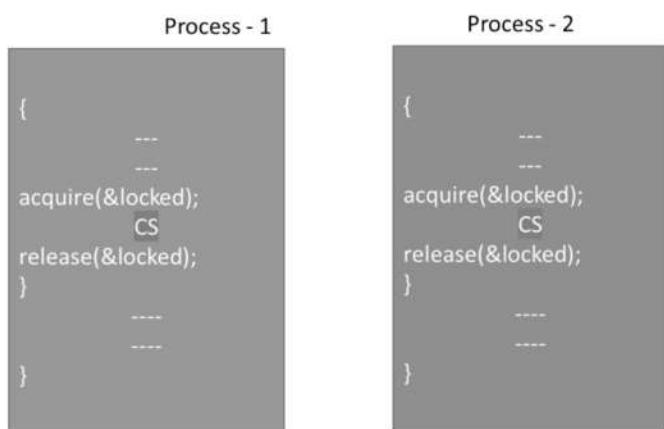
```
{
    ...
    ...
    while(1)
    {
        acquire(&locked);
        CS
        release(&locked);
    }
    ...
}
```

High Level Constructs for solving Critical Section Problems (using Instructions like xchg)

- Spinlocks
- Mutex
- Semaphore
- Monitor

Software APIs for providing synchronization.

High Level Construct: Spinlocks

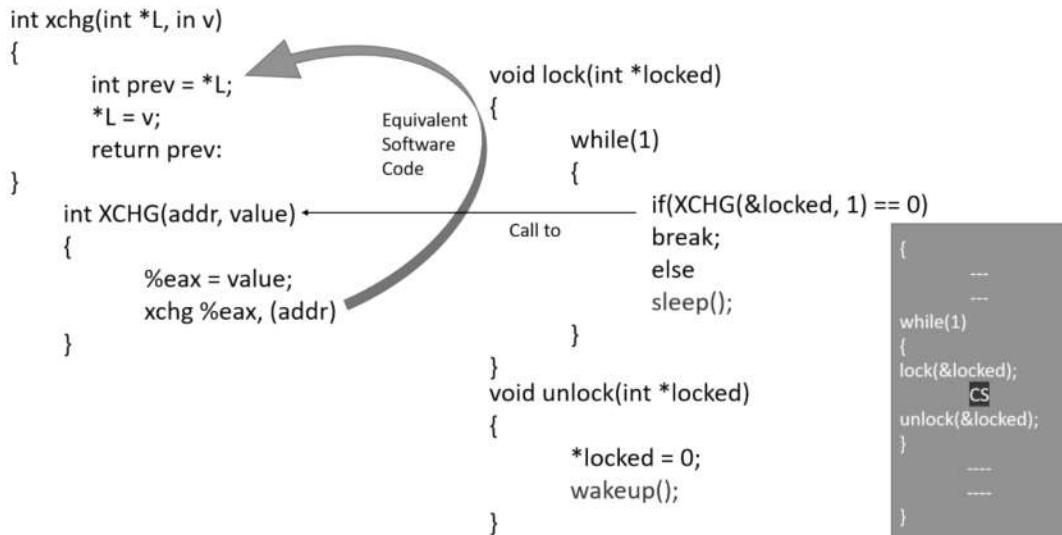


- Two functions
 - `acquire(&locked)`
 - `release(&locked)`
- One process will acquire the lock and the other will wait in a loop repeatedly checking if the lock is available
- The lock becomes available when the former process releases it

Issues with Spinlocks

- Busy Waiting

High Level Construct: Mutex



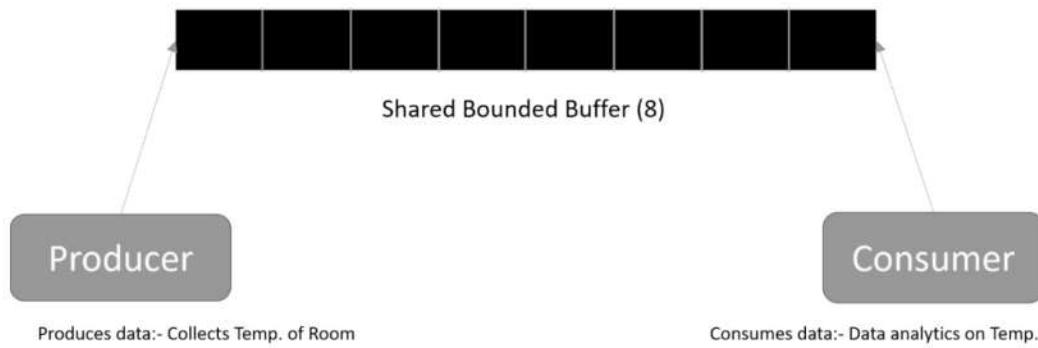
Issue with Mutex

- Thundering Herd Problem

High Level Construct3: Semaphore

- Store the lost wakeup() calls
- Shared Variable (Kernel)
- Atomic functions to store/access the wakeup calls
 - down/wait
 - up/signal
- 2 Types
 - Counting Semaphore (Allows more than one process inside the Critical Section)
 - Binary Semaphore (Allows only one process inside the Critical Section)

Producer Consumer Problem



Producer – Consumer Problem (Solution using Mutex)

```

void producer()
{
    while(1)
    {
        item = produce_item();
        if(count == N)
            sleep(empty);
        lock(mutex);
        insert_item(item);
        count++;
        unlock(mutex);
        if(count == 1)
            wakeup(full);
    }
}

void consumer()
{
    while(1)
    {
        if(count == 0)
            sleep(full);
        lock(mutex);
        item = remove_item();
        count--;
        unlock(mutex);
        if(count == N-1)
            wakeup(empty);
        consume_item(item);
    }
}

```

Issue with Producer – Consumer Problem (Solution using Mutex)

- Lost wakeup()

Producer – Consumer Problem (Solution using Semaphore)

```

buffer[N]
int count = 0;
CSEMAPHORE empty = N, full = 0;
BSEMAPHORE S = 1;

void producer()
{
    while(1)
    {
        1. item = produce_item();
        2. down(empty);
        3. down(S);
        4. insert_item(item);
        5. count++;
        6. up(S);
        7. up(full);
    }
}

void consumer()
{
    while(1)
    {
        1. down(full);
        2. down(S);
        3. item = remove_item();
        4. count--;
        5. up(S);
        6. up(empty);
        7. consume_item(item);
    }
}

```

Operating Systems (CS3000)

Lecture – 30, 31
(Process Synchronization – 11, 12)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Classical Problems of Synchronization?

- Producer Consumer Problem (Bounded-Buffer Problem)
- Readers and Writers Problem
- Dining-Philosophers Problem

Reader Writer Problem

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do not perform any updates
 - Writers – can write
- Problem – allow multiple readers to read at the same time. Only one writer can access the shared data at the same time
- Shared Data
 - Data set
 - Semaphore **rd** initialized to 1
 - Semaphore **wrt** initialized to 1
 - Integer **rcount** initialized to 0

Semaphore Solution

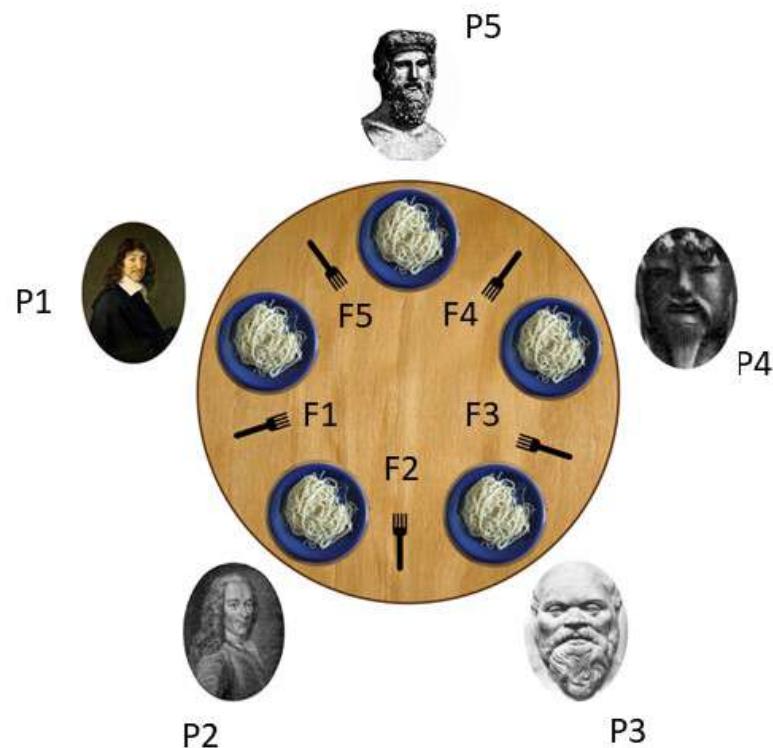
Writer Process

```
while (1){  
    wait(wrt) ;  
    // writing is performed  
    signal(wrt) ;  
}
```

Reader Process

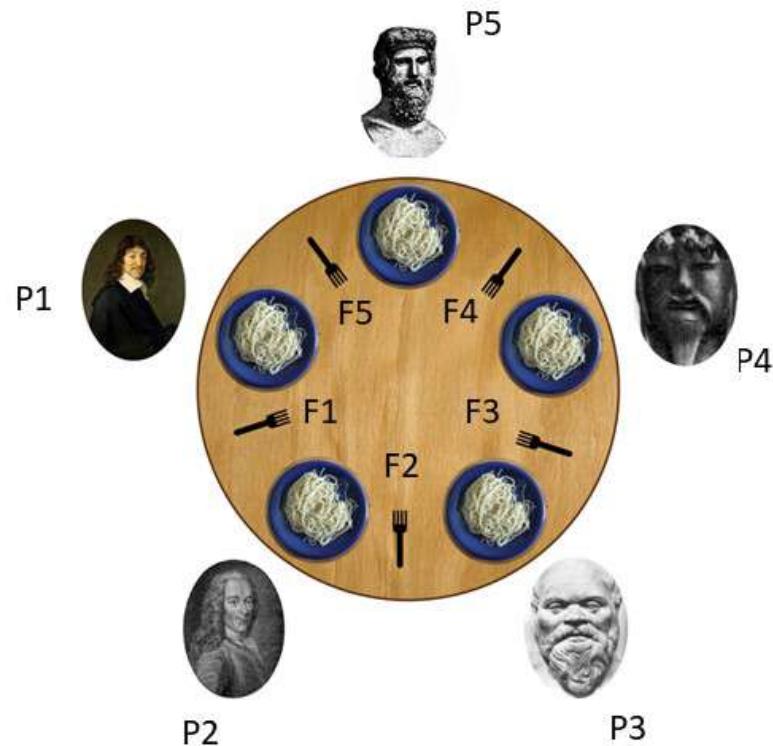
```
while(1) {  
    wait (rd) ;  
    rcount++ ;  
    if (rcount == 1)  
        wait (wrt); //no writer can enter  
    signal (rd);  
    //other readers can enter  
    //reading is performed  
    wait (rd) ;  
    rcount-- ;  
    if (rcount == 0)  
        signal (wrt);  
    signal (rd) ;  
}
```

Dining Philosophers Problem



- Philosophers either **think** or **eat**
 - To eat, a philosopher needs to hold both forks (the one on his left and the one on his right)
 - If the philosopher is not eating, he is thinking
-
- **Problem:**
 - Develop an approach where no philosopher starves

Solution-1



```
#define N 5
Void philosopher(int i){
While(TRUE){
    think(); //for some_time
    take_fork(R_i);
    take_fork(L_i);
    eat();
    put_fork(L_i);
    put_fork(R_i);
}
}
```

Thank You

Any Questions?

Operating Systems (CS3000)

Lecture – 32
(Deadlock - 1)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Shareable vs Non-shareable resources

- Sharable resources
 - can be used by multiple processes during their execution.
 - e.g., CPU, I/O bus.
- Non-sharable resources
 - cannot be used by multiple processes during their execution.
 - e.g., printer



Process of Utilization of a Resource

- **Request:-**
- If the request is available, assigned the resource to the process.
- If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
- **Use:-** The process can operate on the resource.
- **Release:-** The process releases the resource.



Deadlock

- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; and if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other processes.
- This situation is called a **deadlock**.



Deadlock Characterization

Deadlock can arise if four necessary conditions.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0



Methods for Handling Deadlocks

- Prevention
 - Ensure that the system will never enter a deadlock state
- Avoidance
 - Ensure that the system will never enter an unsafe state
- Detection
 - Allow the system to enter a deadlock state and then recover



Deadlock Prevention

- Mutual Exclusion – The mutual-exclusion condition must hold for non-sharable resources.
- Hold and Wait – we must guarantee that whenever a process requests a resource, it does not hold any other resources
 - There is no guarantee that a file that is released by a process will be in the same state when it will get for the next time.



Deadlock Prevention

- No Preemption
 - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted.
 - Preempted resources are added to the list of available resources.
 - A process will be restarted only when it can regain its old resources, and the new ones that it is requesting



Deadlock Prevention

- Circular Wait

- impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. For example:

1. R1
2. R2
3. R3
4. R4

If P_i holds R3 and requests for R2 then it has release R3 first and re-request for R2 then R3



Thank You

Any Questions?



Operating Systems (CS3000)

Lecture – 33
(Deadlock - 2)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Deadlock Avoidance and Safe State

- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a **safe state**
- A system is in a safe state only if there exists a **safe sequence of execution**
- 1 resource type and no. of instances=12, Available = **3 2**

Process	Max	Allocation	Need
P0	10	4 5	6 5
P1	4	2	2
P2	7	3	4

- P0 requests one more resource dynamically. Can it be granted?
 - **<P1,P2,P0> is a safe sequence of execution**



Safe State

- If a system is in safe state \Rightarrow no deadlocks
- If a system is in unsafe state \Rightarrow possibility of deadlock
- Avoidance \Rightarrow ensure that a system will never enter an unsafe state



Avoidance algorithms

- resource-allocation graph (RAG)
- banker's algorithm



RAG for Deadlock avoidance

- $G = (V, E)$
- V: Processes and Resources
- E: Request edge, Allocation edge, Claim edge

Process:



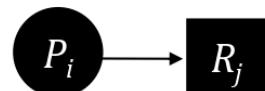
Claim edge:



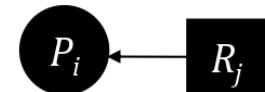
Resource:



Request edge:



Assignment edge:

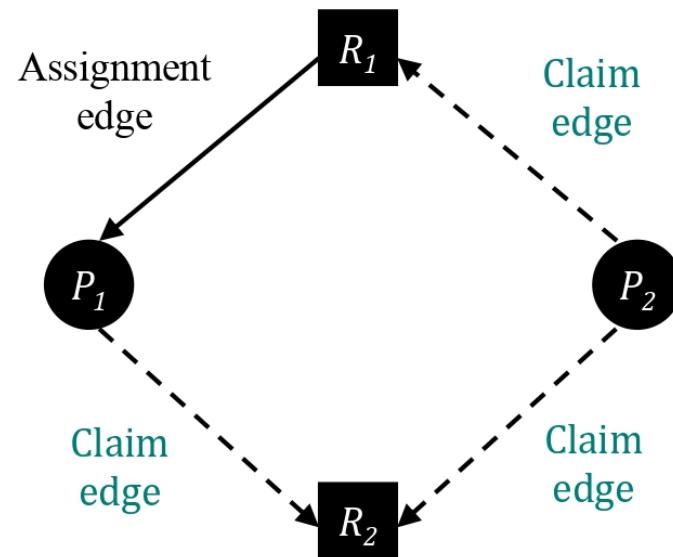


RAG for Deadlock avoidance

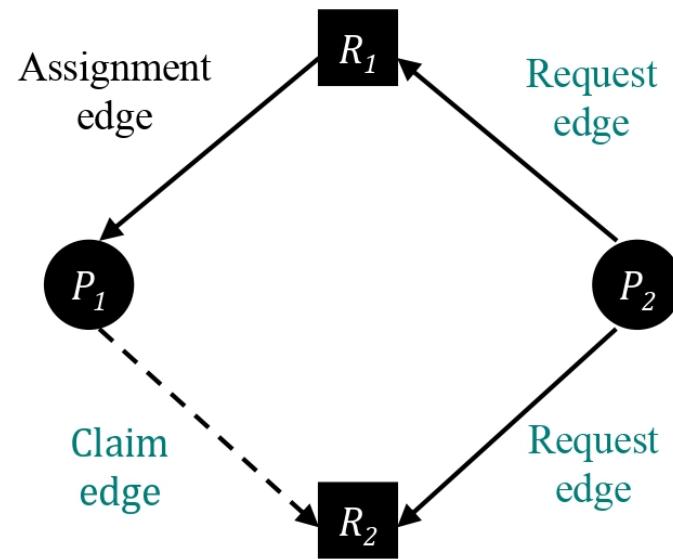
- *Claim edge* $P_i \dashrightarrow R_j$ indicates that process P_i may request resource R_j ; which is represented by a dashed line
- A claim edge converts to a request edge when a process **requests** a resource
- A request edge converts to an assignment edge when the resource is **allocated** to the process



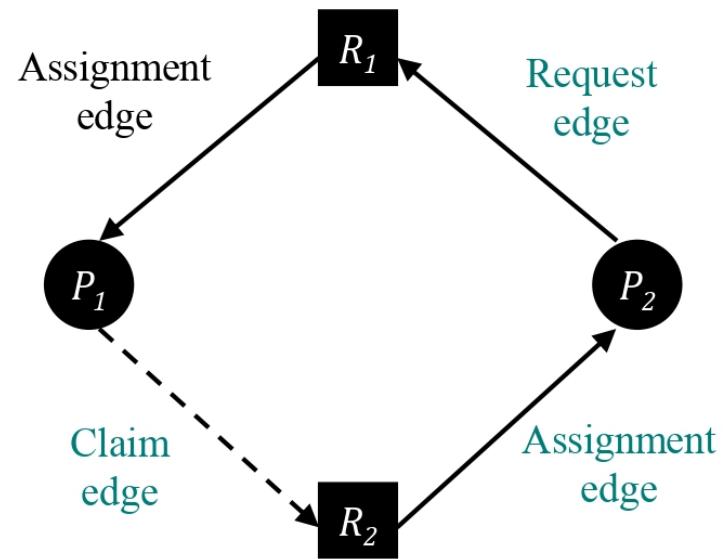
RAG for Deadlock avoidance



RAG for Deadlock avoidance



RAG for Deadlock avoidance



RAG for Deadlock avoidance

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph



Operating Systems (CS3000)

Lecture – 34
(Deadlock - 3)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available:** Vector of length m . If $\text{Available}[j] = k$, there are k instances of resource type R_j available.
- **Max:** $n \times m$ matrix. If $\text{Max}[i,j] = k$, then process P_i may request at most k instances of resource type R_j .
- **Allocation:** $n \times m$ matrix. If $\text{Allocation}[i,j] = k$ then P_i is currently allocated k instances of R_j .
- **Need:** $n \times m$ matrix. If $\text{Need}[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

$$\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$$

Pro cess	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	3	3	2
P1	2	0	0	1	2	2			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			



Banker's Algorithm

- When P_i makes a Request $_i$

Resource Request Algorithm

1. If Request $_i \leq$ Need $_i$ then goto step2
 else error
2. If Request $_i \leq$ Available then goto step3
 else wait
3. Available = Available - Request $_i$
 Allocation $_i$ = Allocation $_i$ + Request $_i$
 Need $_i$ = Need $_i$ - Request $_i$
4. Check if the new state is safe (Safety Algo)
 - If so, grant the request
 - if not, block the process until it is safe to grant the request.



Banker's Algorithm

Safety Algotrihtm

1. Initialize Work = Available

 Initialize Finish[i] = False, for i = 1,2,3,...n

2. Find an i such that:

 Finish[i] == False and Need[i] <= Work

 If no such i exists, go to step 4.

3. Work = Work + Allocation[i]

 Finish[i] = True

 goto step 2

4. if Finish[i] == True for all i, then the system is in a safe state.

5. Return



Banker's Algorithm: Example-1

A B C

Initially Available=[10, 5, 7]

Pr oc ess	Allocation			Need		
	A	B	C	A	B	C
P0	0	1	0	7	4	3
P1	2	0	0	1	2	2
P2	3	0	2	6	0	0
P3	2	1	1	0	1	1
P4	0	0	2	4	3	1

Available = [3 3 2]

P1 makes **Request₁** = (1,0,2)



Banker's Algorithm: Example-2

Pr oc ess	Allocation			Need		
	A	B	C	A	B	C
P0	0	1	0	7	4	3
P1	3	0	2	0	2	0
P2	3	0	2	6	0	0
P3	2	1	1	0	1	1
P4	0	0	2	4	3	1

Available = [2 3 0]

P4 makes **Request**₄ = (3,3,0)



Thank You

Any Questions?



Operating Systems (CS3000)

Lecture – 35
(Deadlock - 4)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Kumar Nayak
Assistant Professor
Department of Computer Sc. and Engg.

Deadlock Detection and Recovery

- An algorithm that examines the state of the system to detect whether a deadlock has occurred
- And an algorithm to recover from the deadlock

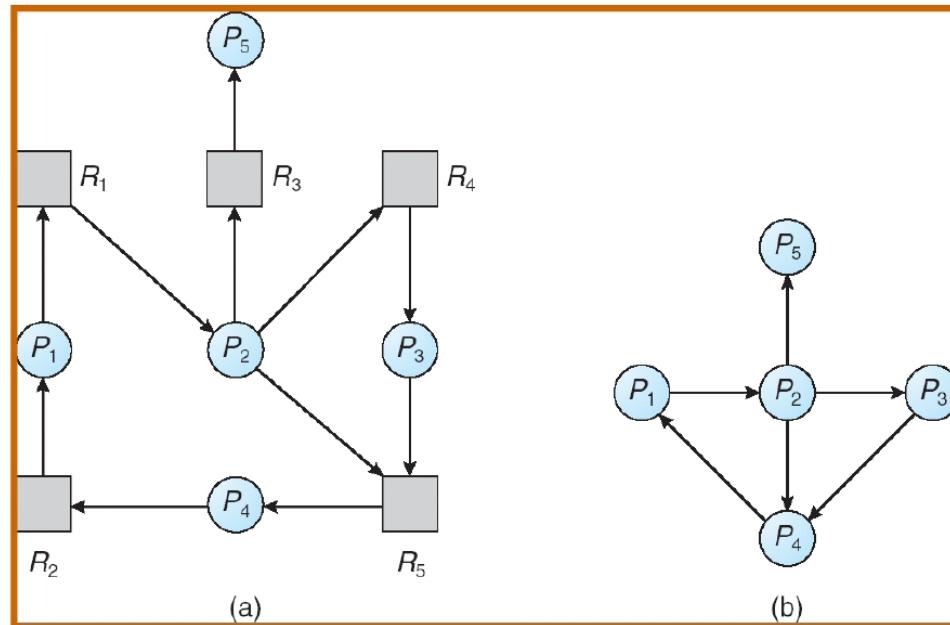


Deadlock Detection

- Requires the creation and maintenance of a wait for graph (WFG)
 - variant of the resource-allocation graph
 - The graph is obtained by **removing** the resource nodes from a RAG and **collapsing** the appropriate edges
 - Consequently; all nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph
 - If there is a cycle, there exists a deadlock



Deadlock Detection



Resource-Allocation Graph

Corresponding wait-for graph



Deadlock Detection

1. Initialize Work = Available
Initialize Finish[i] = False, for i = 1,2,3,..n
2. Find an i such that:
Finish[i] == False and Request[i] <= Work
If no such i exists, go to step 4.
3. Work = Work + Allocation[i]
Finish[i] = True
goto step 2
4. if Finish[i] == true for all i, then the system is not in deadlocked state



Deadlock Detection Algorithm: Example

Sequence $\langle P_0, P_2, P_3, P_4, P_1 \rangle$ results in $Finish[i] == \text{true}$ for all i
⇒ **System is not in deadlocked state**

Pr oc ess	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	1	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			



Deadlock Detection Algorithm: Example

Finish[0] is True but **P1, P2, P3, P4 are deadlocked.**

Pr oc ess	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			



Recovery from Deadlock

- Two Approaches
 - Process termination
 - Resource preemption



Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
 - This approach will break the deadlock, but at great expense
- Abort one process at a time until the deadlock cycle is eliminated
 - This approach incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be re-invoked to determine whether any processes are still deadlocked
- Many factors may affect which process is chosen for termination
 - What is the priority of the process?
 - How long has the process run so far and how much longer will the process need to run before completing its task?
 - How many and what type of resources has the process used?
 - How many more resources does the process need in order to finish its task?
 - How many processes will need to be terminated?



Recovery from Deadlock: Resource Preemption

- With this approach, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- When preemption is required to deal with deadlocks, then three issues need to be addressed:
 - **Selecting a victim** – Which resources and which processes are to be preempted?
 - **Rollback** – If we preempt a resource from a process, what should be done with that process?
 - **Starvation** – How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?



Thank You

Any Questions?



Operating Systems (CS3000)

Lecture – 36
(Memory Management – 1)

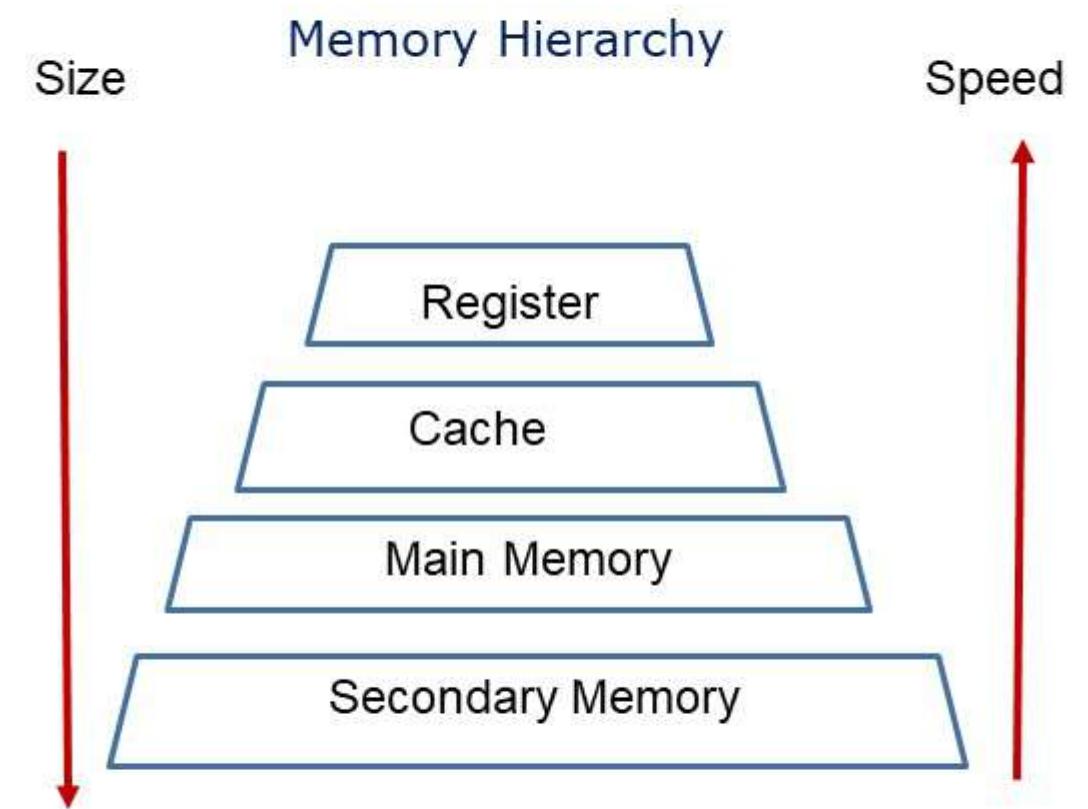


INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,
DESIGN AND MANUFACTURING,
KANCHEEPURAM

Dr. Sanjeet Nayak
Assistant Professor
Department of Computer Sc. and Engg.

What is Memory?

- Physical device which stores information temporarily or permanently.
- **Primary memory** – only large storage media that the CPU can access directly.
- **Secondary storage** – extension of main memory that provides large nonvolatile storage capacity



Primary Memory

Primary Memory-

- Temporary Memory - RAM – (SRAM, DRAM)
- Permanent Memory – ROM - (PROM, EPROM, EEPROM)

Execution of Program (Process)

- ```
#include<stdio.h>
int main(){
char str[]="Hello World\n";
print(str);
}
```



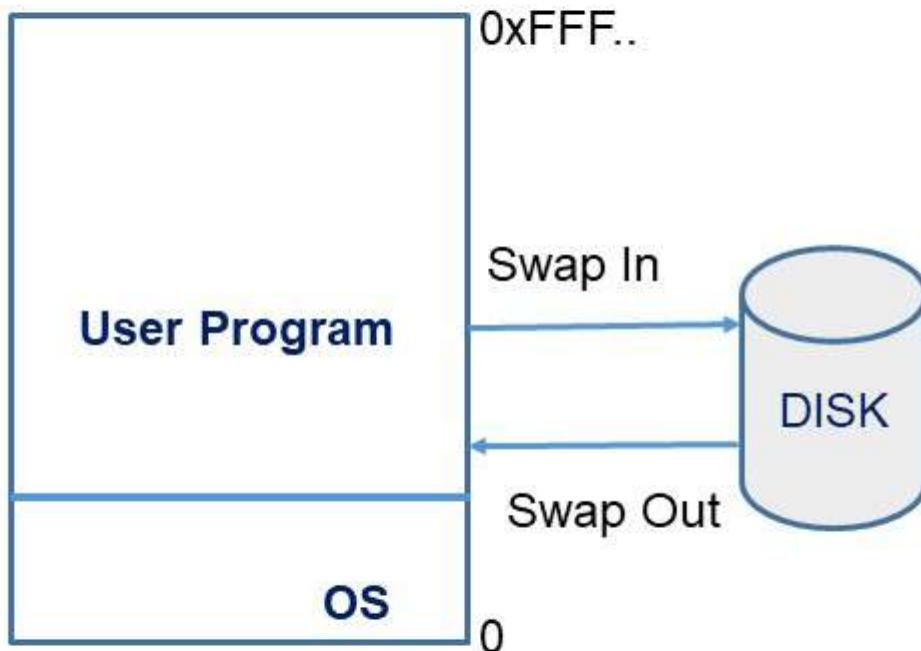
Compile (gcc hello.c)

**Executable  
file**

**Process**

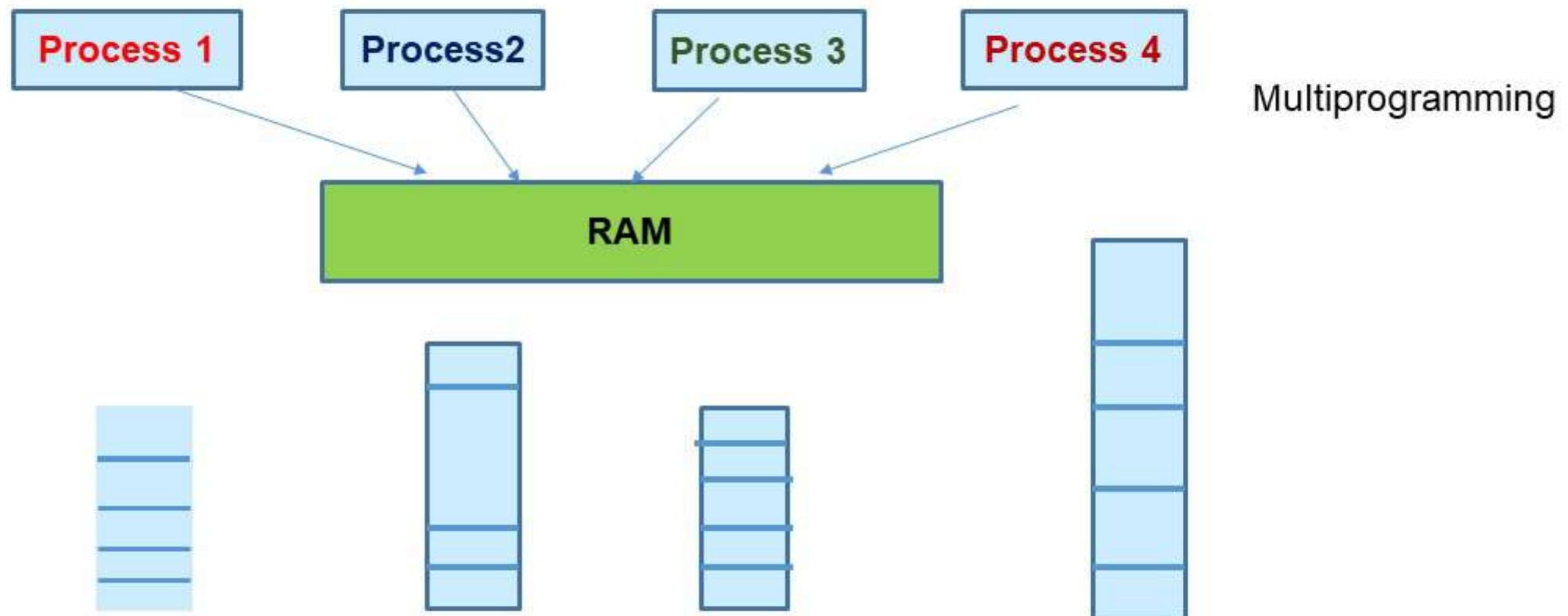
- Process
  - A program in execution
  - Present in the RAM
  - Comprises of
    - Executable Instructions
    - Stack
    - Heap
    - State in the OS (in the kernel)

# Swapping



- Issues with no abstraction?
  - Overwrite
- How we can execute multiple programs?
  - Using Swapping
    - One process occupies RAM at a time
    - When one process completes, another process is allocated RAM

# Multiprogramming

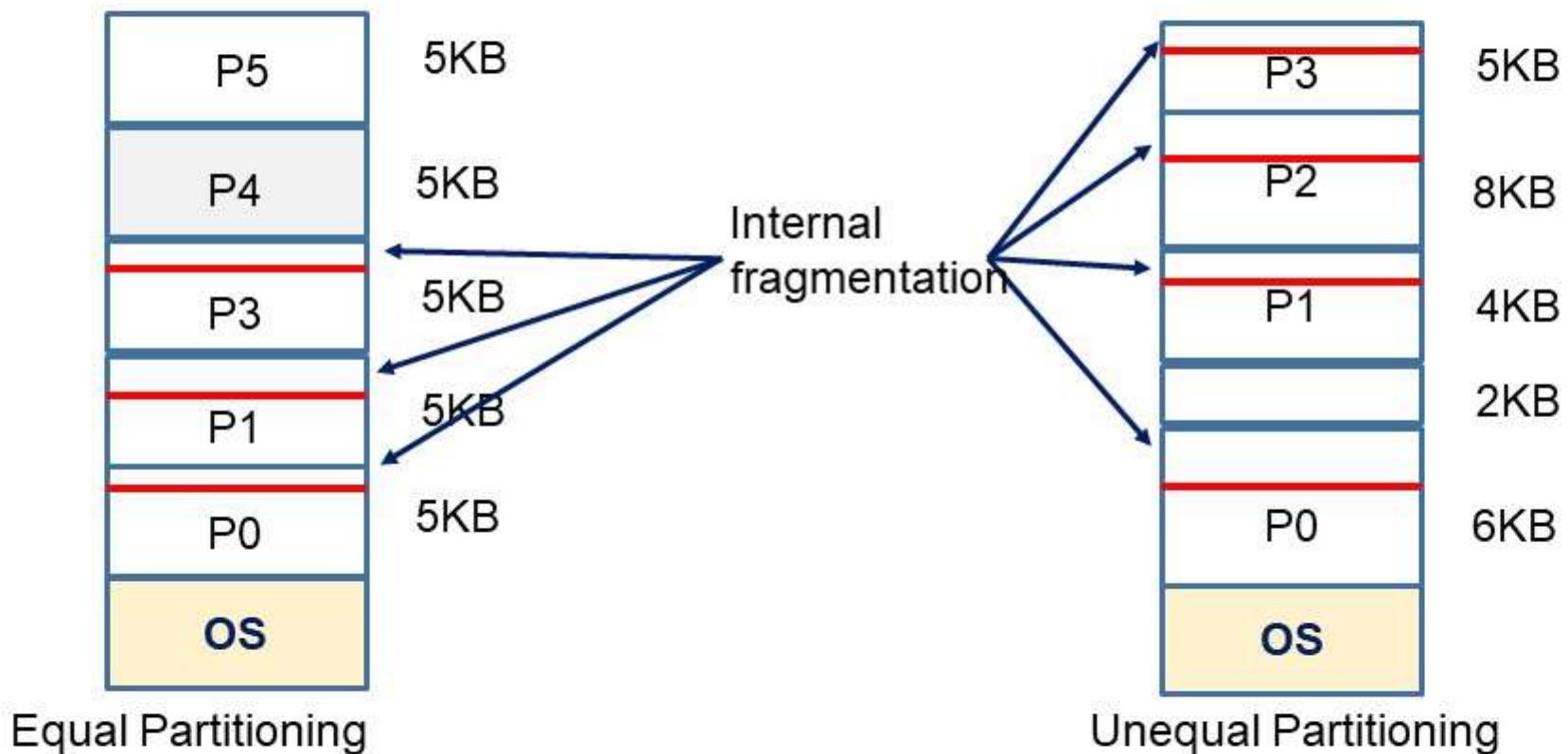


# Memory Management Scheme

- Multiple processes can occupy the RAM simultaneously Using Memory Management Scheme
  - Contiguous
    - Fixed Partition
    - Variable or Dynamic
  - Non-Contiguous (Modern Scheme)
- OS maintains Partition Table in RAM.
  - Allocated partition
  - Free partition (hole)

## Fixed Scheme

- User memory size=25KB, no. of partition=5
- P0=4KB, P1=3KB, P2=6KB, P3=4KB, P4=5KB, P5=5KB

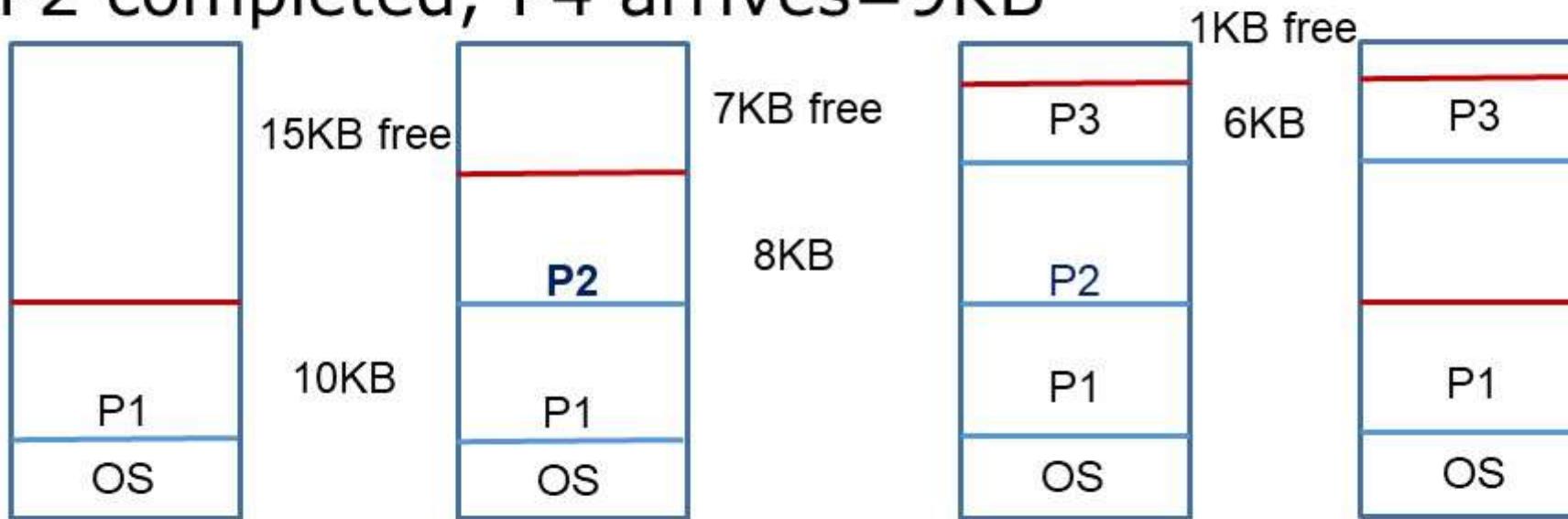


## Fixed Scheme

- The memory is divided into a fixed number of partitions
- The number of partitions is fixed.
- Partition size can be equal or unequal
- Degree of multiprogramming depends on the no. of partitions.
- Issues
  - Internal Fragmentation
  - Limit in process size
  - Degree of multiprogramming is limited

# Variable Scheme

- User memory size=25KB
- P1=10KB, P2=8KB, P3=6KB
- P2 completed, P4 arrives=9KB



# Variable Scheme

- Initially, the memory will be full contiguous free block.
- Whenever a request by a process comes, accordingly the partition will be made.
- Advantages:
  - No internal fragmentation
  - No limitation on the number of processes
  - Size of the process is limited by the available RAM size
- Issues
  - External Fragmentation
  - Solution: compaction

# Memory-Allocation Scheme

Ex: 212KB, 417KB, 112KB, 426KB



# Memory-Allocation Scheme

- First-Fit
  - Allocate the *first* hole that is big enough
- Worst-Fit
  - Allocate the *largest* hole; must also search the entire list
- Best-Fit
  - Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
- Next-Fit
  - Allocate the *next* hole that is big enough from the previous allocation

Thank You  
Any Questions?



# Operating Systems (CS3000)

Lecture – 37  
(Memory Management – 2)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,  
DESIGN AND MANUFACTURING,  
KANCHEPURAM

**Dr. Sanjeet Nayak**  
Assistant Professor  
Department of Computer Sc. and Engg.

# Limitations of Contiguous Memory Management

- Entire process needs to be in RAM
- Allocation needs to be in contiguous memory
- Fragmentation
- Limit the size of the process by RAM-size
- Management of Partitions

Solution:

Go for Non-Contiguous Memory Management Schemes

# Memory Management Unit

- Logical Address: generated by the CPU; also referred to as **virtual address**
- Physical Address: address seen by the memory unit

Memory-Management Unit (MMU)

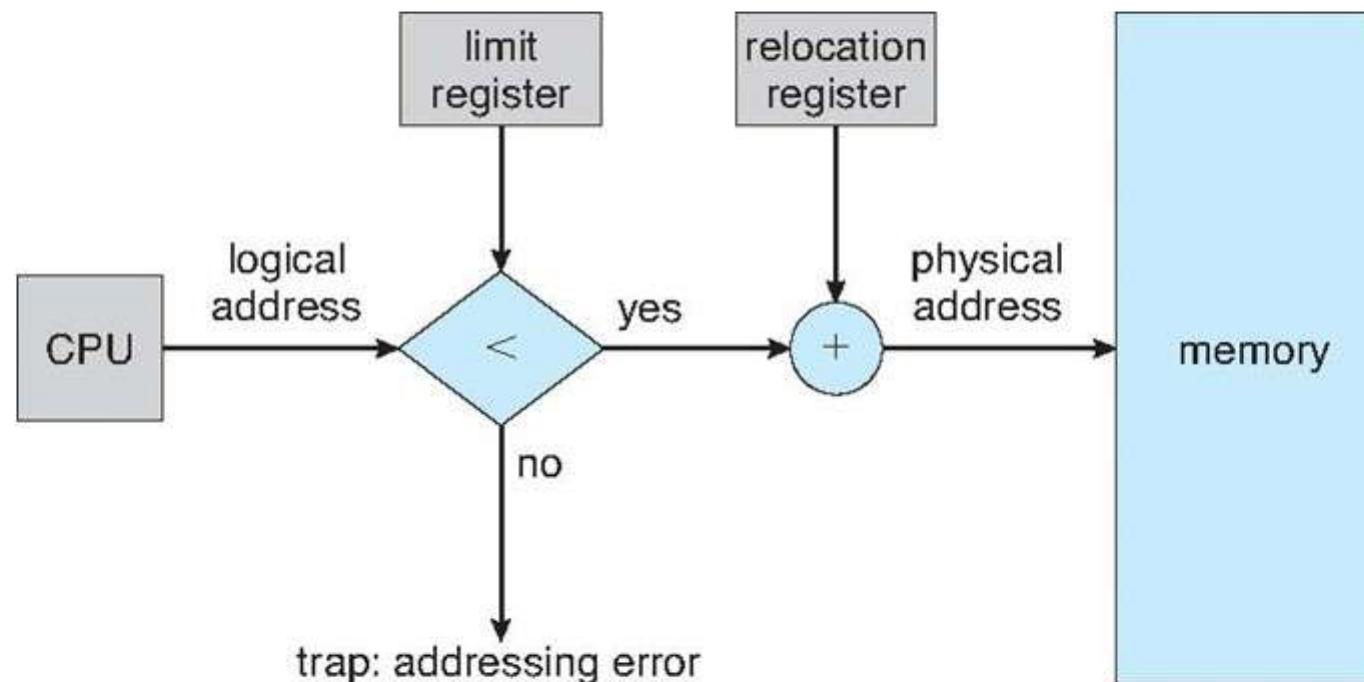
- Hardware device that maps **virtual to physical address**
- In MMU scheme, adds the relocation register value to every address generated by a user process
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

# Memory Management Unit

Assume logical address is 20

Relocation address=100

Limit address= 40

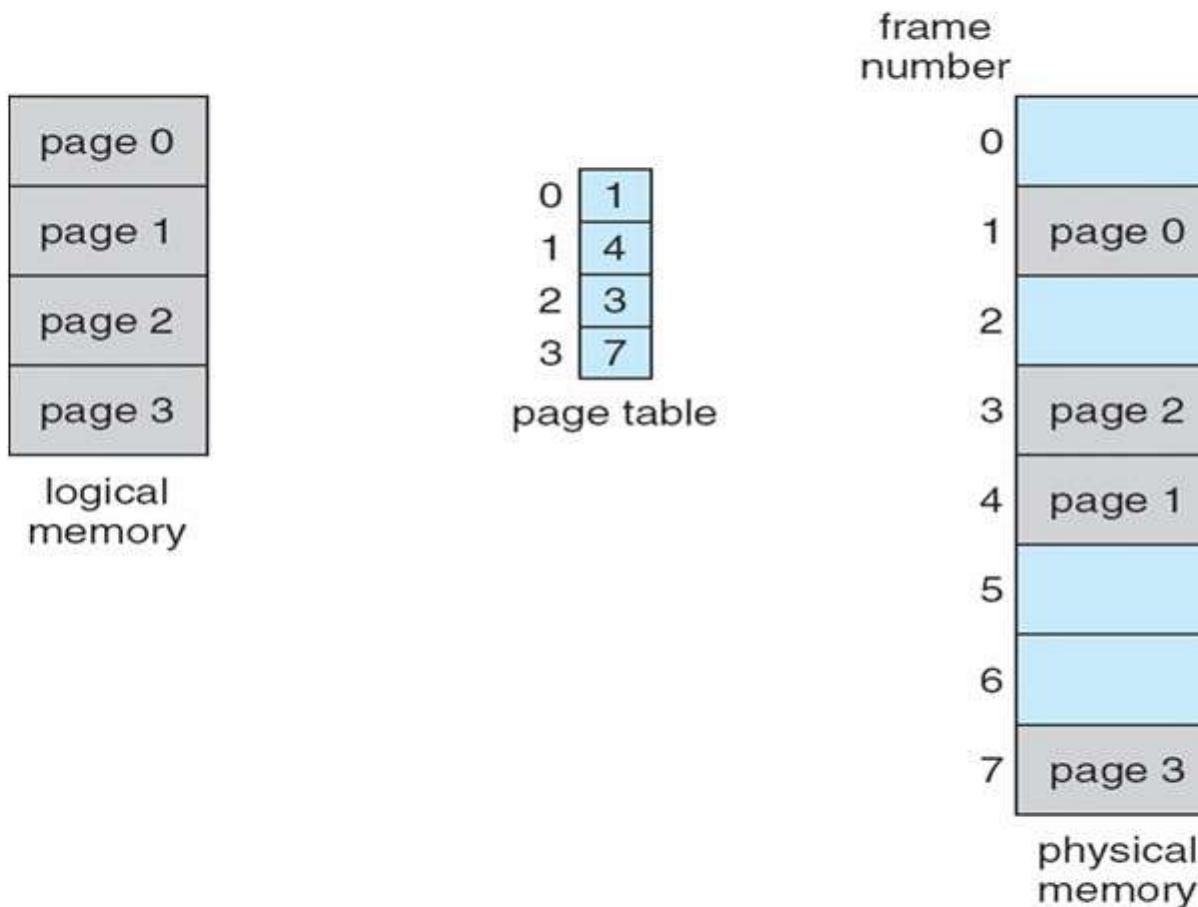


# Modern Memory Management Schemes

## Non-Contiguous Memory Management Scheme

- Virtual Memory
  - Technique that allows only part of the program needs to be in memory for execution
- Segmentation

# Address Translation Scheme



# Example: Address Translation Scheme

|    |   |
|----|---|
| 0  | a |
| 1  | b |
| 2  | c |
| 3  | d |
| 4  | e |
| 5  | f |
| 6  | g |
| 7  | h |
| 8  | i |
| 9  | j |
| 10 | k |
| 11 | l |
| 12 | m |
| 13 | n |
| 14 | o |
| 15 | p |

logical memory

|   |   |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

page table

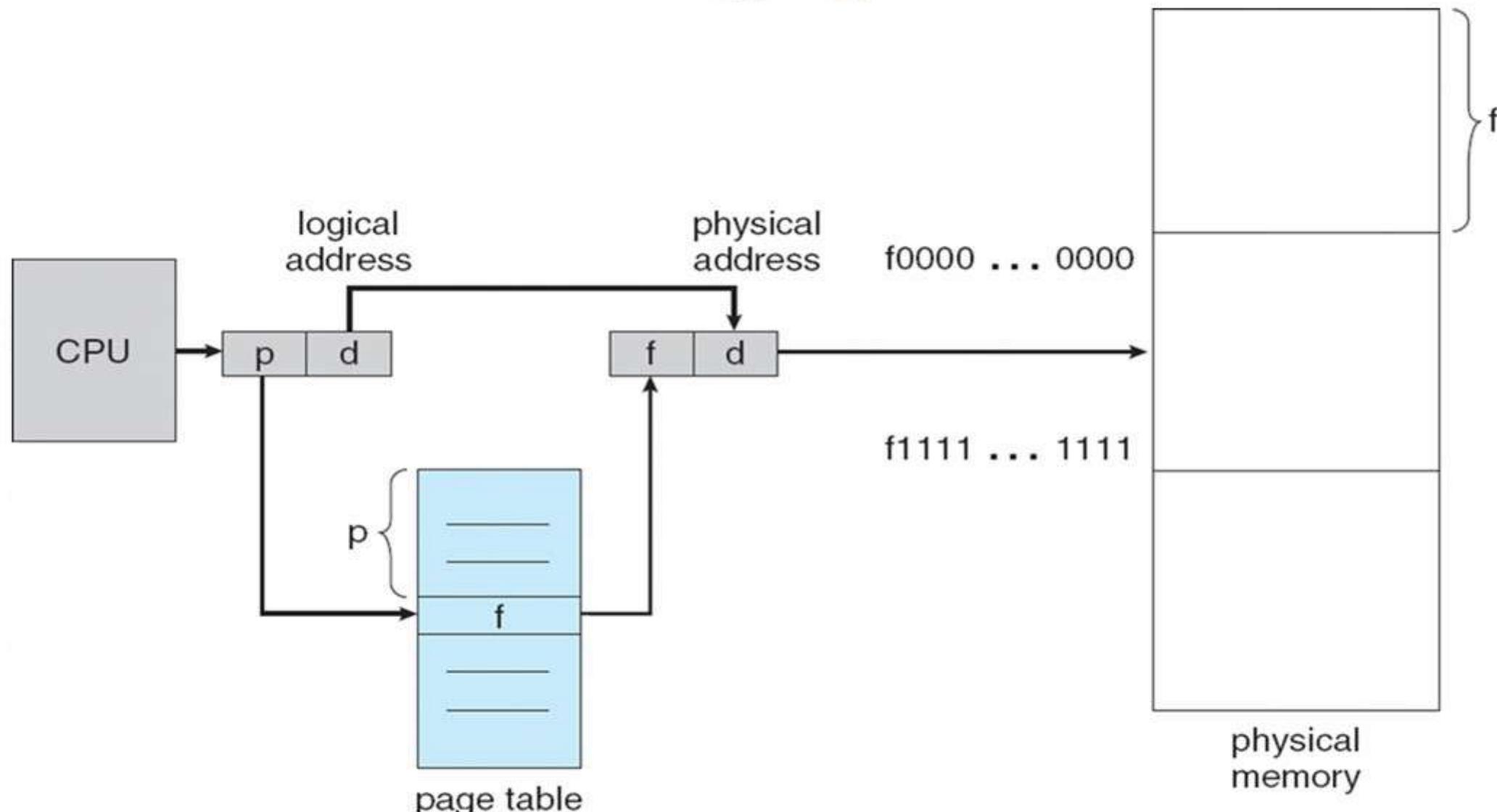
|    |                  |
|----|------------------|
| 0  |                  |
| 4  | i<br>j<br>k<br>l |
| 8  | m<br>n<br>o<br>p |
| 12 |                  |
| 16 |                  |
| 20 | a<br>b<br>c<br>d |
| 24 | e<br>f<br>g<br>h |
| 28 |                  |

physical memory

# Paging

- Paging is a memory-management scheme allows the physical address space of a process to be non-contiguous.
- Divide physical memory into fixed-sized blocks called **frames**
- Divide logical/virtual memory into blocks of same size called **pages**
- To run a program of size  **$n$**  pages, need to find  **$n$**  free frames and load the program
- Set up a page table to translate logical to physical addresses

# Paging



# Address Translation Scheme

Address generated by CPU is divided into:

**Page number (*p*)** – used as an index into a *page table* which contains base/relocation address of each page in physical memory

**Page offset (*d*)** – combined with base address to define the physical memory address that is sent to the memory unit

**Frame number (*f*)** – used to represent frame number which is a base address base address of each page in physical memory

**Page offset (*d*)** – same as previously defined

# Problem 1

Consider the virtual address space is 44KB. Physical address space is 24KB. Page size=4KB. Find the (p, d) in logical and (f, d) in physical address.

## Problem 2

Consider the physical memory space is 64MB and 32-bit virtual address space. Page size is 4KB, what is the approximate size of page table?

- (a) 2 MB (b) 3MB (c) 4 MB (d) 6 MB

## Problem 3

Consider the virtual address as 32 bit.

Page size=4KB. Page table entries of 4 byte. What is the approximate size of page table size?

- (a)2MB (b) 3MB (c) 4 MB (d) 8MB

## Problem 4

Consider the computer system implements 40 bit virtual address and Page size is 16KB,  
what is the approximate size of page table, if each page table entry is 48 bits.

- (a)384MB (b) 48 MB (c) 192 MB (d) 96MB

# Hardware Implementation of Page-table

**Case1:** Implement the page table as a set of dedicated registers

**Problem:** This can be used only when page table is small.

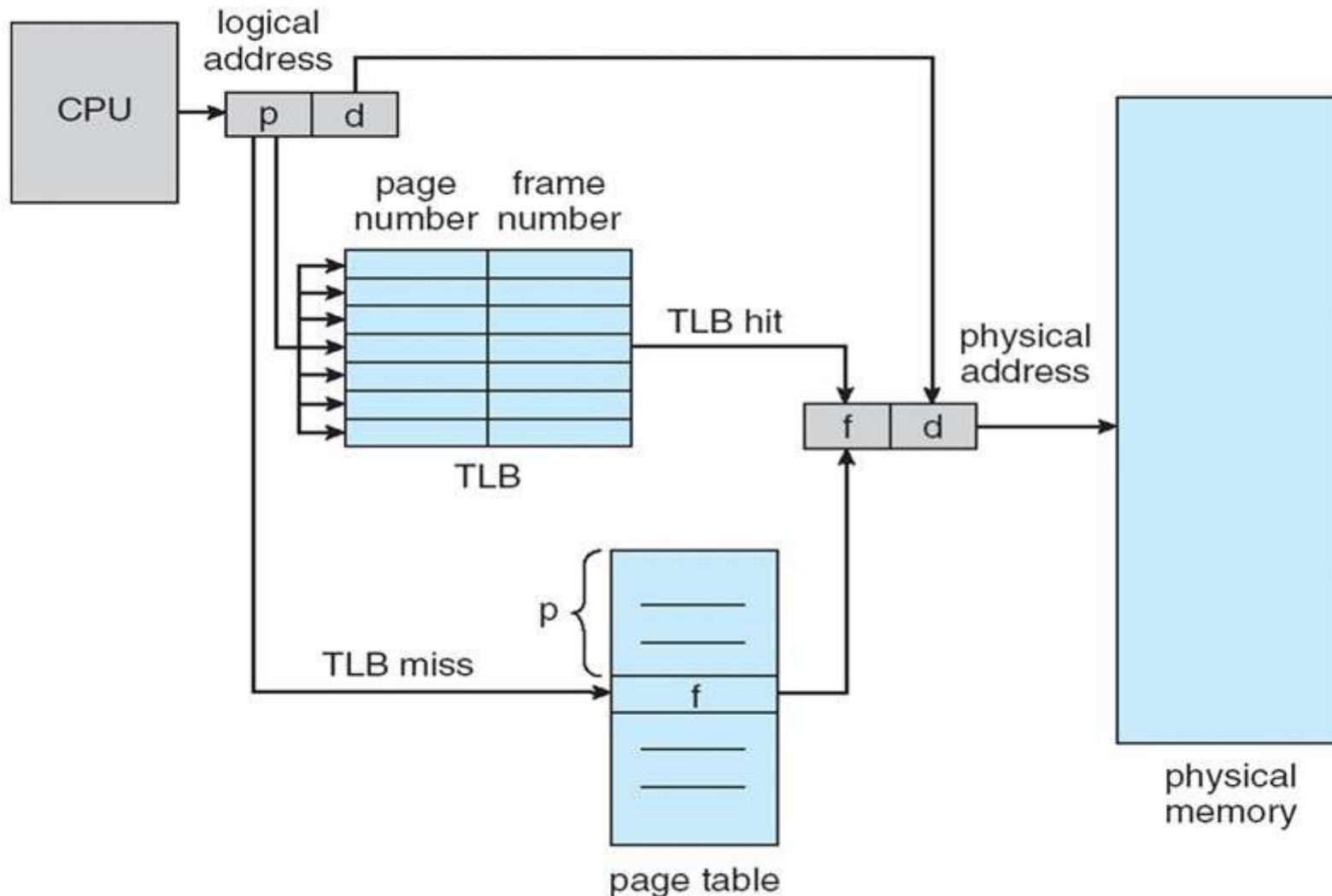
**Case2:-**Keep the page table in main memory and a page table base register (PTBR)

**Problem:-** 2 memory accesses (one for page table and one for actual byte)

## Case 3

- TLB is high-speed cache memory.
- It consists of two parts : **key (page no)** and **value (frame no)**
- Search is fast
  - Few of the page-table entries
  - When logical addresses is generated by the CPU, its page number is presented to the TLB.
  - If page number is found, its frame number is easily available and used to access the memory.
  - If not found in the TLB, a memory reference to the page table is made.
  - After accessing, we add the page number and frame number to the TLB.

# Paging With TLB



# Effective Access Time

- TLB Access Time =  $\varepsilon$  time unit
- Assume memory access time is  $t$  microsecond
- Hit ratio ( $\alpha$ ) – percentage of times that a page number is found in TLB;
- Effective Access Time (EAT)  
$$EAT = \alpha(\varepsilon + t) + (1 - \alpha)(\varepsilon + t + t)$$

## Problem 1

Consider that 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time.

Assume it takes 20 nanoseconds to search the TLB and 100 nanoseconds to access memory.

Find the effective memory-access time.

$$\begin{aligned}\text{Sol:-EAT} &= \text{TLB hit (TLB AT + Memory AT)} + (1-\text{TLB hit}) (\text{TLB AT} + 2*\text{MAT}) \\ &= 0.8(20+100)+0.2(20+200) \\ &= 96+44 \\ &= 140\text{ns}\end{aligned}$$

## Problem 2

If effective memory access time is given as 160ns. We assume 90-percent hit ratio means that we find the desired page number in the TLB 90 percent of the time. 100 nanoseconds to access memory.

Find the TLB access time.

$$\text{Sol:- EAT} = \text{TLB hit (TLB AT} + \text{Memory AT}) + (1-\text{TLB hit}) (\text{TLB AT} + 2 * \text{MAT})$$

$$160 = 0.9(T+100) + 0.1(T+200)$$

$$160 = 1T + 90 + 20$$

$$T = 50\text{ns (TLB AT)}$$

Thank You  
Any Questions?



# Operating Systems (CS3000)

Lecture – 38  
(Memory Management – 3)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,  
DESIGN AND MANUFACTURING,  
KANCHEPURAM

**Dr. Sanjeet Nayak**  
Assistant Professor  
Department of Computer Sc. and Engg.

# Shared Pages Concept

- An advantage of paging is the possibility of sharing common code.
- This is particularly important in time-sharing environment.

## Shared code

- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes

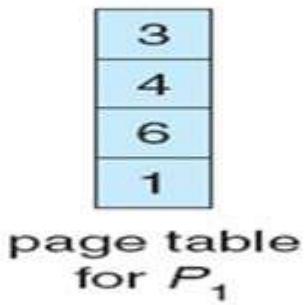
## Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space

# Example

|        |
|--------|
| ed 1   |
| ed 2   |
| ed 3   |
| data 1 |

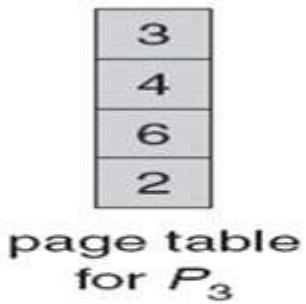
|        |
|--------|
| ed 1   |
| ed 2   |
| ed 3   |
| data 3 |



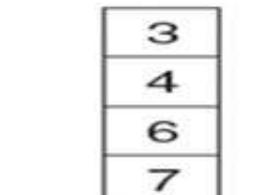
page table  
for  $P_1$

|        |
|--------|
| ed 1   |
| ed 2   |
| ed 3   |
| data 2 |

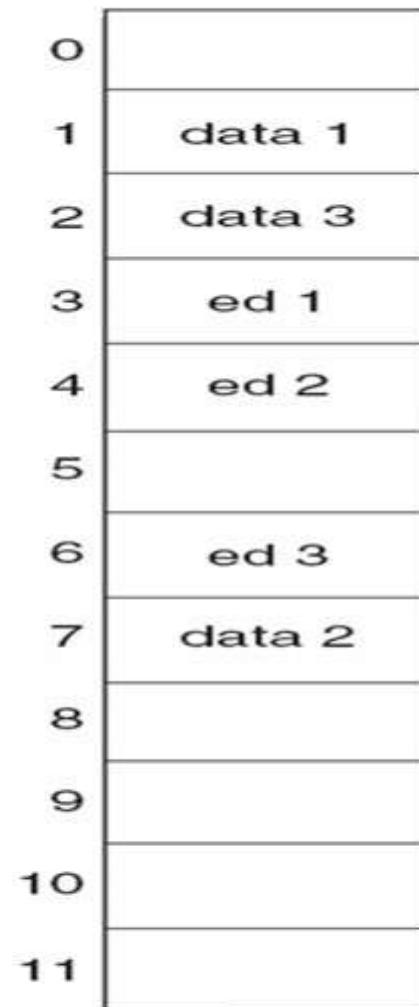
process  $P_2$



page table  
for  $P_3$



page table  
for  $P_2$



# Shared Pages Concept

- Assume 40 users using text editors
- Text editor size= 150 KB
- Data size=20 KB
- Non-Shared ~~Memory~~ Pages  
Total Size=6800KB
- Shared ~~Memory~~ Pages  
Total Size=950KB

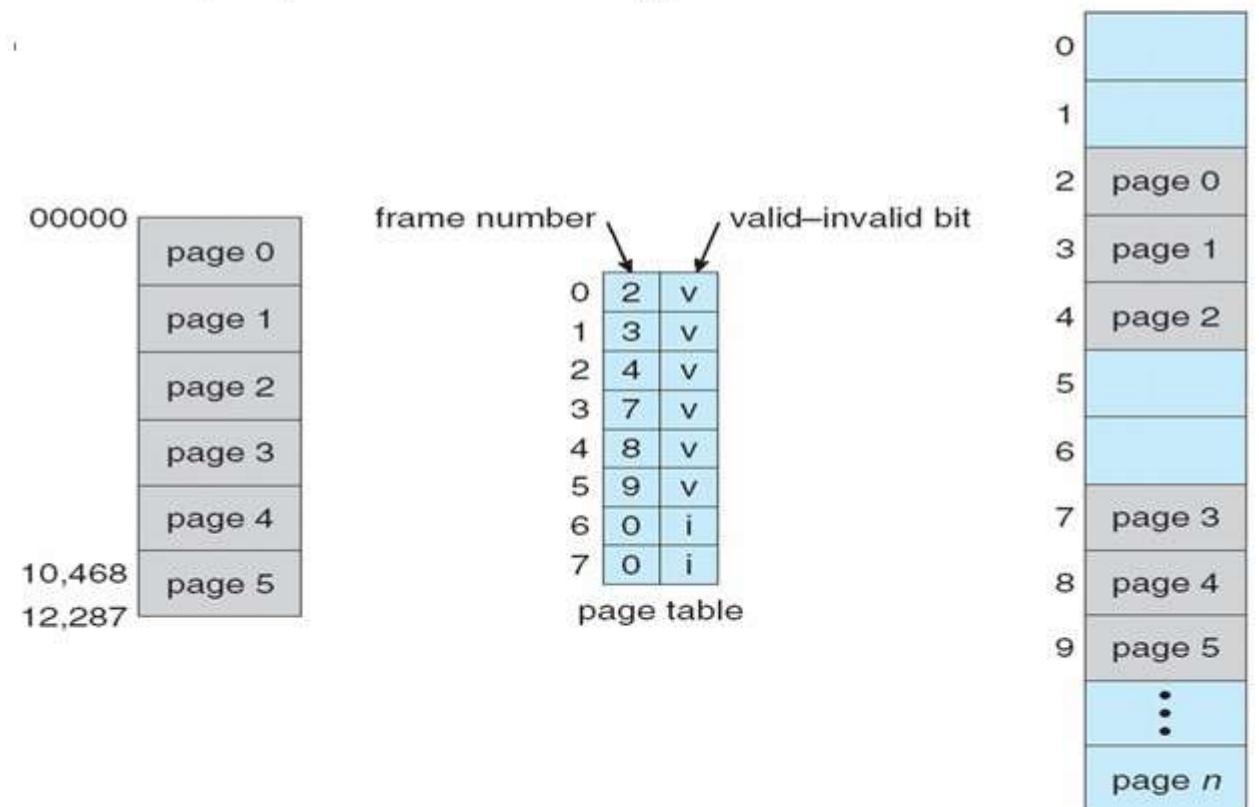
# Demand Paging

- Bring a page into memory only when it is needed
  - Demanded during program execution

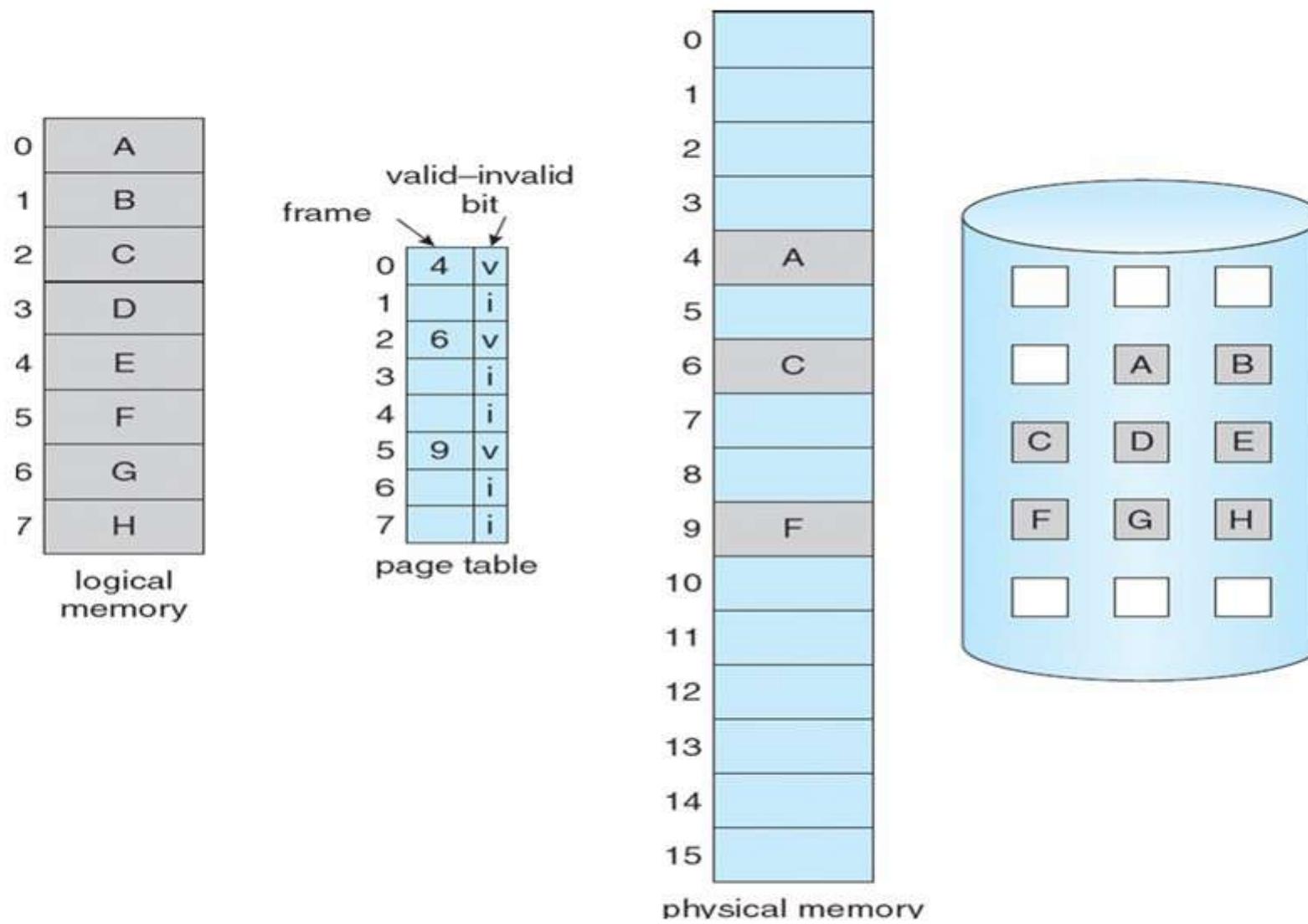
# Memory Protection Using bit

**Valid-invalid** bit attached to each entry in the page table:

- “valid” indicates that the associated page is in the physical memory, and is thus a legal page
- “invalid” indicates that the page is not in the physical memory/not valid – not in logical address space of the process.



# Page Table When Some Pages Are Not in Main Memory



# Page Fault

If there is a reference to a page, first reference to that page will trap to operating system: **page fault**

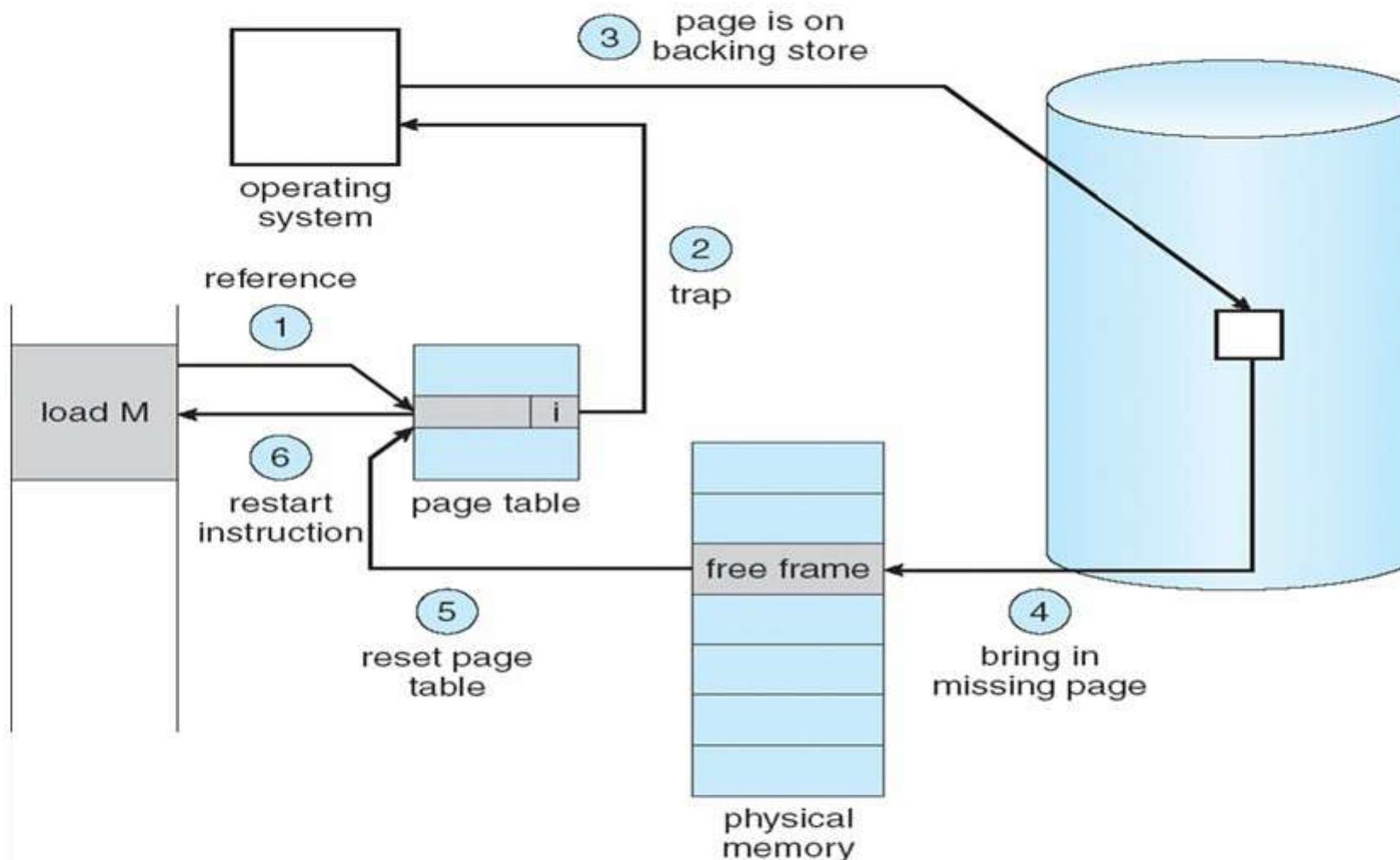
1. OS looks at Page table to decide:

    Invalid reference  $\Rightarrow$  abort

    Just not in memory

1. Get empty frame (Swap Out some frames from main memory)
2. Swap In page into frame
3. Modify page table
4. Set the valid bit
5. Restart the instruction that caused the page fault

# Steps in Handling a Page Fault



# Page Fault

## Page Fault Rate $0 \leq p \leq 1.0$

if  $p = 0$ , no page faults

if  $p = 1$ , every reference is a fault

## Effective Access Time (EAT)

$$\begin{aligned} EAT = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & \quad + \text{swap page out} \\ & \quad + \text{swap page in} \\ & \quad + \text{restart instruction overhead}) \end{aligned}$$

# Page Fault

Memory access time = 200 nanoseconds = 0.2 microsecond

Average page-fault service time = 8 milliseconds

$$\begin{aligned} \text{EAT} &= (1 - p) \times 0.2 + p \text{ (8 milliseconds)} \\ &= (1 - p) \times 0.2 + p \times 8,000 \text{ [In microseconds]} \\ &= (1 - p) 0.2 + p \times 8000 \end{aligned}$$

If one access out of 1,000 causes a page fault [p],

then

$$\text{EAT} = 8.2 \text{ microseconds}$$

This is a slowdown by a factor of 41 [8200/200]

$$0.2 * 1000 = 200$$

$$8.2 * 1000 = 8200$$

## Problem 1

Consider a system which has page fault service time =100ns.

Page fault rate is 65%.

Average Main memory access time is 1ns. [Assume AMAT is time required to access Page Table as well as the desired data from Main Memory]

What is effective memory access time?

Sol:

$$\begin{aligned} \text{EAT} &= 0.65 \times 100 + 0.35 \times 1 \\ &= 65.35 \text{ ns} \end{aligned}$$

## Problem 2

Let the page fault service time be 10ms in a computer with average memory access time being 20ns. [Assume AMAT is time required to access PT as well as the desired data from Main Memory]

If one page fault is generated for every  $10^6$  memory accesses, what is the effective access time (approx.) for the memory?

Solution:

$$\begin{aligned} EAT &= \left(\frac{1}{10^6}\right) \times (10^7) + \left(1 - \frac{1}{10^6}\right) 20 \\ &= 30 \text{ ns} \end{aligned}$$

## Problem 3

Assume TLB hit 80%.

TLB Access Time = 2ns

Memory Access Time = 10ns [Assume  $M_{AT}$  is time required to access PT as well as the desired data from Main Memory]

Page fault rate = 10%

Page fault time = 200ns

what is the effective access time (approx.) for the memory?

Solution:  $TLB_{Hit} (TLB_{AT} + M_{AT}) + (1-TLB_{Hit})[(1-PF_T)(TLB_{AT}+2*M_{AT})+(PF)(TLB_{AT}+PFST)]$   
EAT =

Thank You  
Any Questions?



# Operating Systems (CS3000)

Lecture – 39  
(Memory Management – 4)



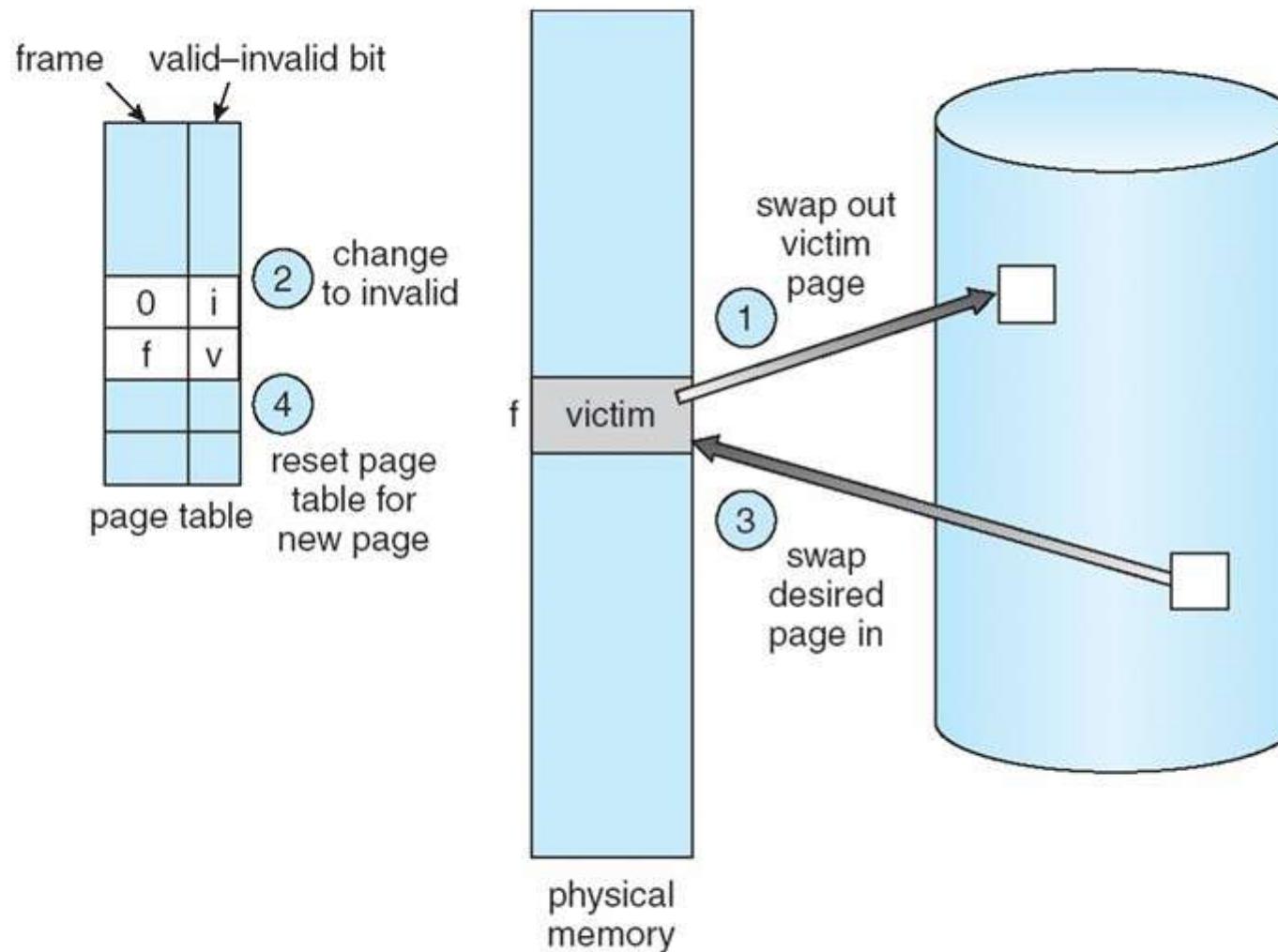
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,  
DESIGN AND MANUFACTURING,  
KANCHEPURAM

**Dr. Sanjeet Nayak**  
Assistant Professor  
Department of Computer Sc. and Engg.

# Page Replacement

- Page replacement – find some page in memory, but not really in use, swap it out
  - Algorithm -
  - performance – want an algorithm that will result in a minimum number of page faults
- The same page may be brought into memory several times
- **Algorithm Structure**
  1. Find the location of the desired page on the disk
  2. Find a free frame:
    - If there is a free frame, use it
    - If there is no free frame, use a **page replacement algorithm** to select a **victim** frame
  3. Bring the desired page into the (newly) free frame; update the page tables
  4. Continue the process

# Page Replacement



# Frame Allocation Schemes

- Each process needs *minimum* number of pages
- Two major allocation schemes
  - **Equal allocation** – For example, if there are 100 frames and 5 processes, give each process 20 frames.
  - **Proportional allocation** – Allocate according to the size of process

$s_i$  = size of process  $p_i$

$$S = \sum s_i$$

$m$  = total number of frames

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Frame Size – 1 Byte  
→ MM Size – 64 Byte

# Allocation Schemes

- **Priority allocation**
  - Use a proportional allocation scheme using priorities rather than size
  - If process  $P_i$  generates a page fault,
    - select for replacement one of its frames
    - select for replacement a frame from a process with lower priority number

## Replacement Schemes

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement** – each process selects from only its own set of allocated frames
  - Ex – FIFO, Optimal, LRU

# FIFO-Page Replacement

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- 3 frames (3 pages can be in memory at a time per process)

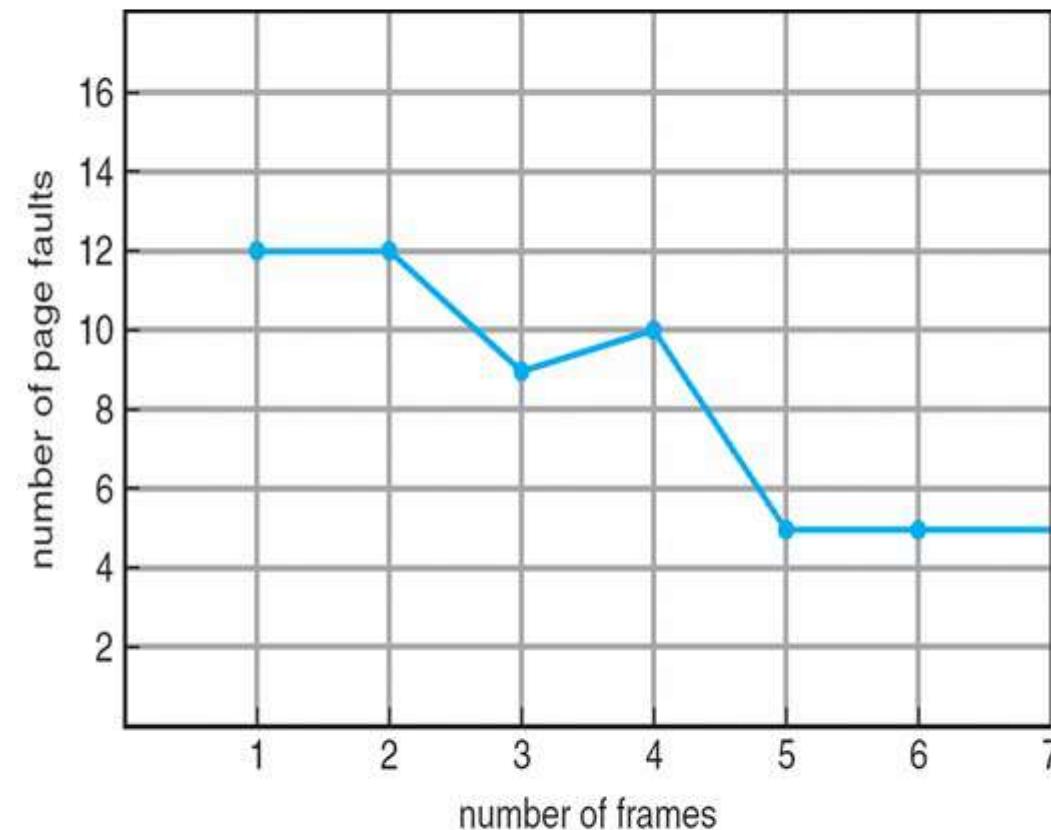
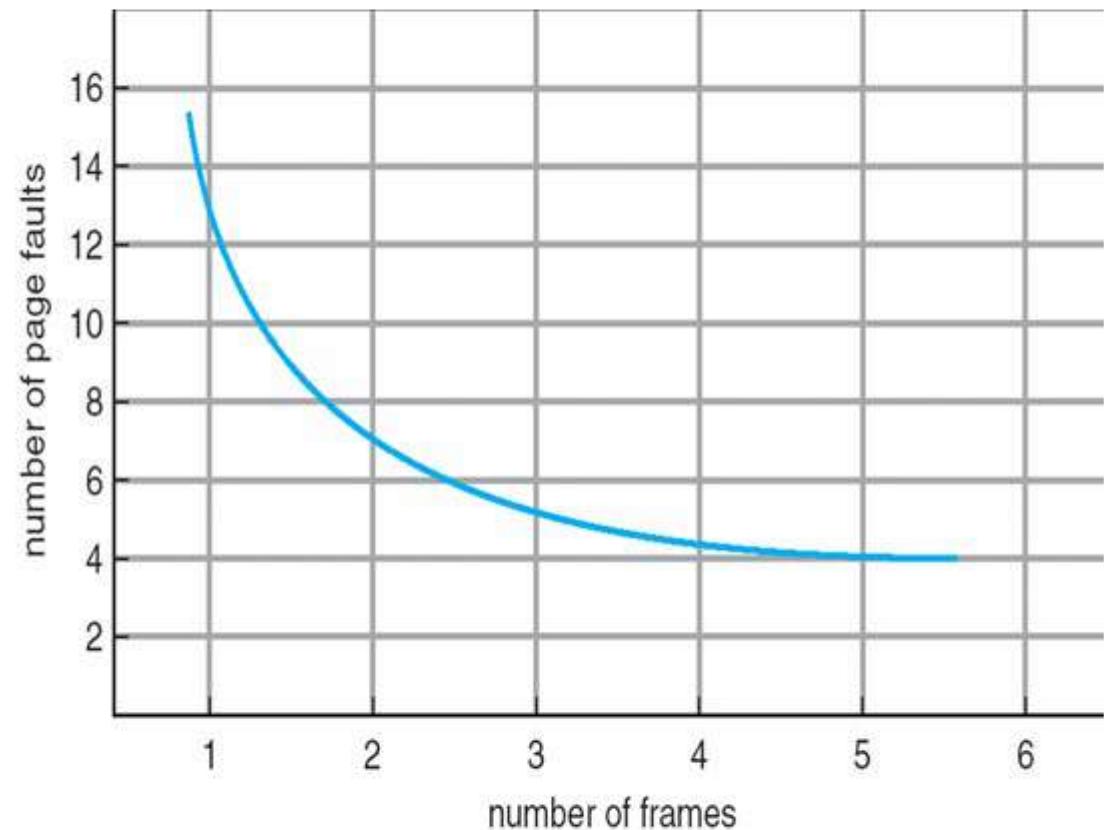
Number of fault= 9

- 4 frames

Number of fault= 10

**Belady's Anomaly:** more frames  $\Rightarrow$  more page faults

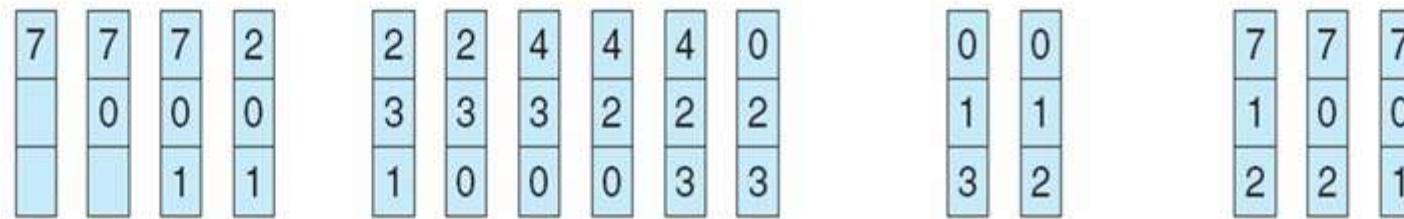
# Normal vs Belady's Anomaly Graph



# FIFO-Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

## Problem

1. A system uses FIFO policy for page replacement. It has 4 page frames with no pages loaded to begin with.

The system first accesses 100 distinct pages in some order and then accesses the same 100 pages but now in the reverse order.

How many page faults will occur?

Solution : 196

# Optimal Page Replacement

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- Replace page that will not be used for longest period of time
- 3 frames

Number of page fault=7

- 4 frames

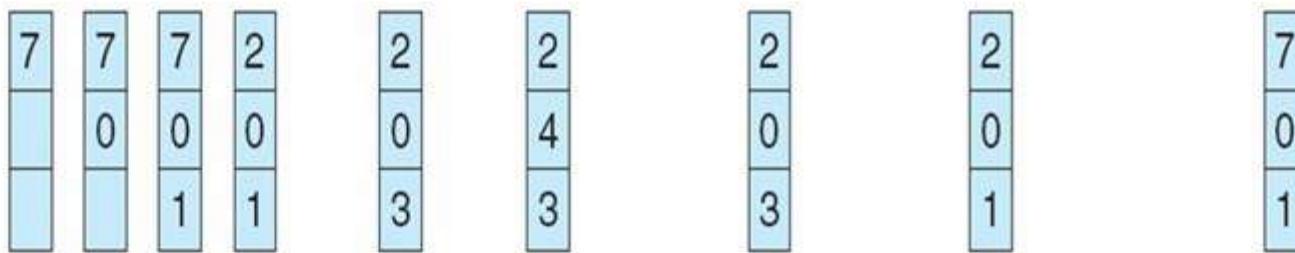
Number of page fault=6

Used for measuring how well your algorithm performs

# Optimal Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

# LRU Page Replacement

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

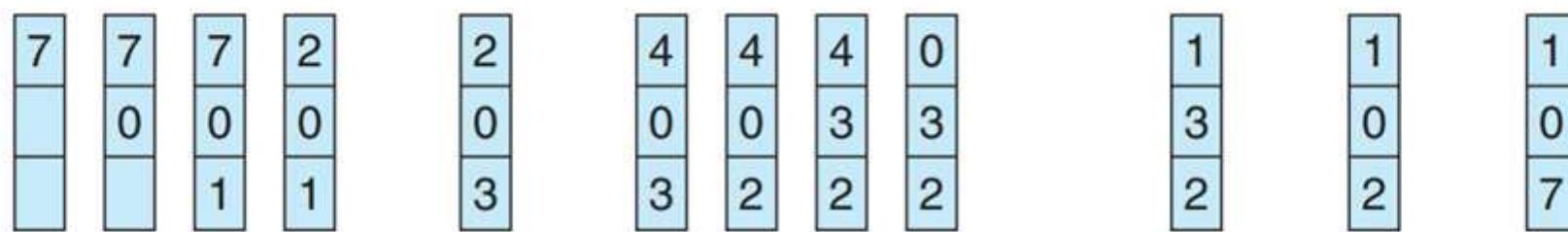
3 frames

4 frames

# LRU Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



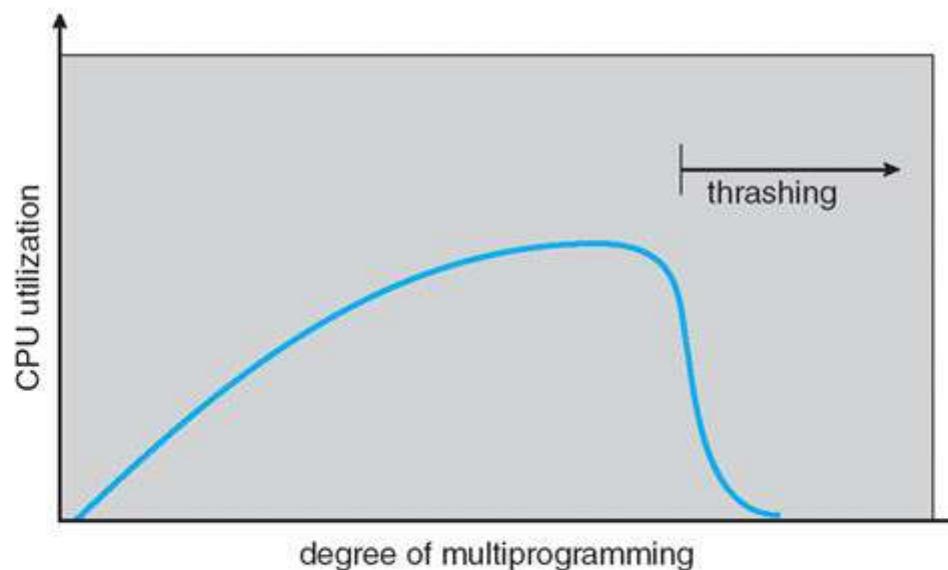
page frames

# Thrashing

If a process does not have “enough” pages, the page-fault rate is very high. This leads to:

- low CPU utilization
- OS thinks that it needs to increase the degree of multiprogramming another process added to the system

**Thrashing** = a process is busy swapping pages in and out



Thank You  
Any Questions?



# Operating Systems (CS3000)

Lecture – 40  
(Memory Management – 5)



INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,  
DESIGN AND MANUFACTURING,  
KANCHEEPURAM

**Dr. Sanjeet Kumar Nayak**  
Assistant Professor  
Department of Computer Sc. and Engg.

## Structure of the Page Table

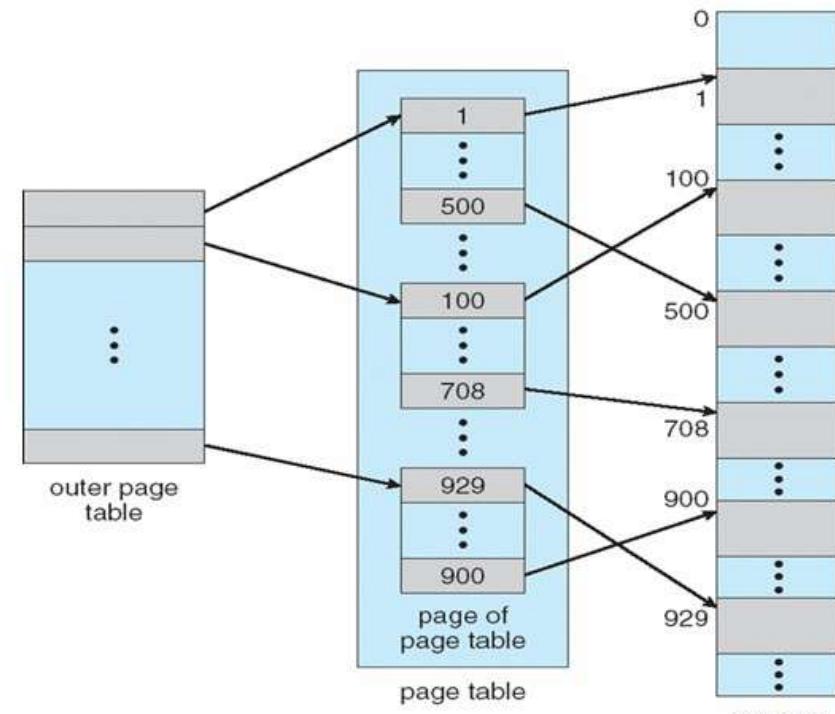
- Many modern computer systems support a large logical address space, in such an environment, the page table itself becomes excessively large

Types of Page table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

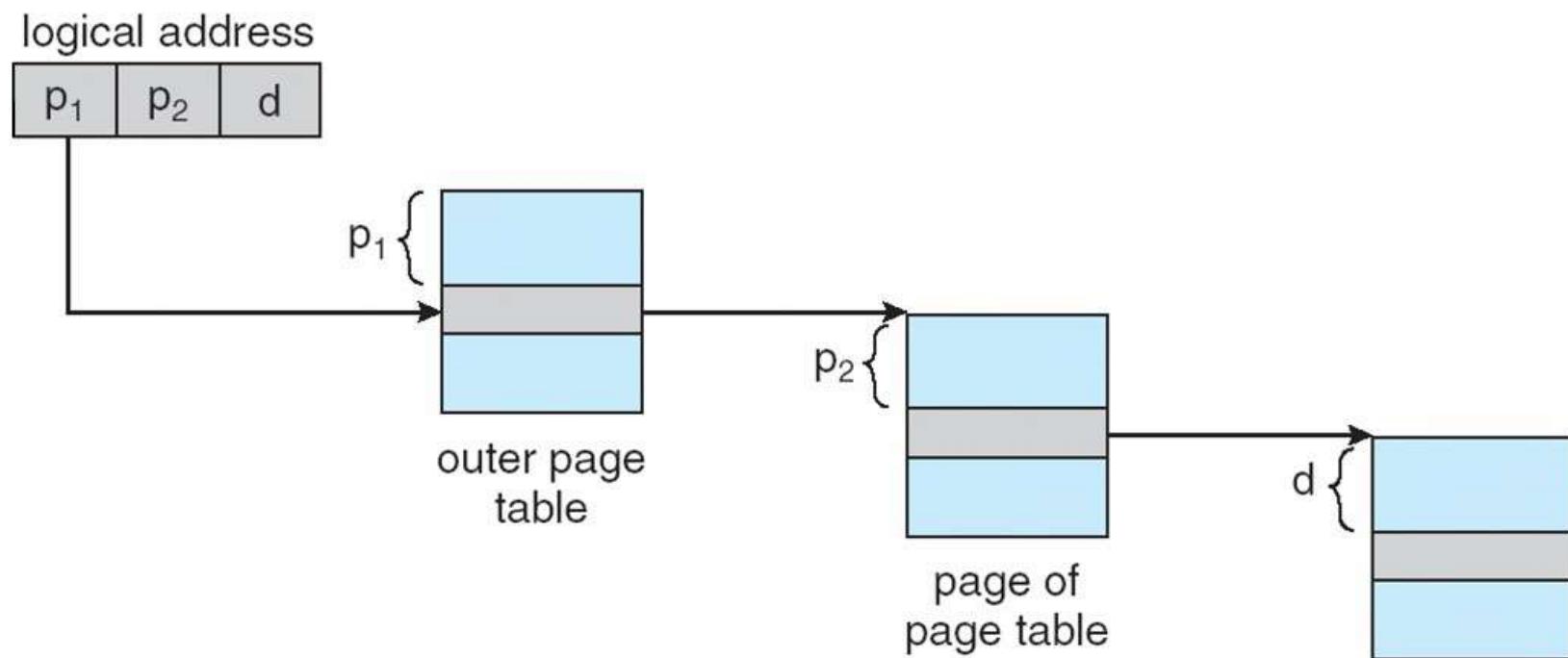
## 2-Level Page Table

- Break up the logical address space into multiple page tables
- 2-level paging, in which page table itself is also paged.
- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset



PT entry = 1 Byte

## 2-Level Page Table



## Problems 1

Consider a three level paging scheme with a TLB. Assume no page fault occurs. It takes 20 ns to search the TLB and 100 ns to access the physical memory. If TLB hit ratio is 80%, the effective memory access time is \_\_\_\_\_ ns.

Solution:

$$\text{EAT} = \text{TLB hit}(\text{TLB acc. Time} + \text{Memory acc. time}) + \text{TLB miss}(\text{TLB acc. Time} + (\text{L}+1) \text{ Memory access time})$$

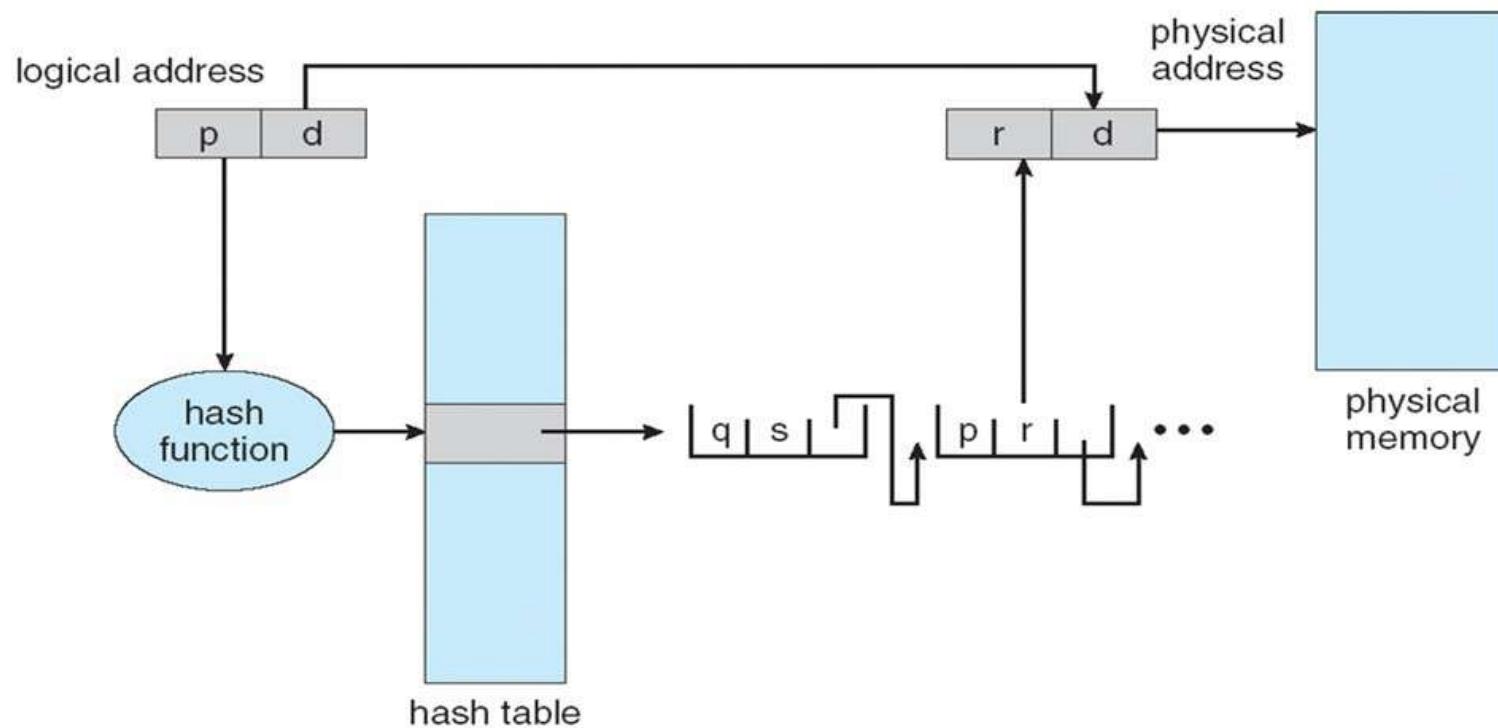
$$\begin{aligned} &= 0.8(120) + 0.2(420) \\ &= 96 + 84 \\ &= 180 \text{ ns} \end{aligned}$$

## Hashed Page Table

Used for handling address spaces larger than 32 bits (or large address space)

- Hash value is the virtual page number.
- This page table contains a chain of elements hashing to the same location
- Steps for hashed paging
  - The virtual page number in the virtual address is hashed into the hash table.
  - The virtual page number is compared with field 1 in the first element in the linked list.
  - If match, corresponding page frame is used to form the physical address.
  - If no match, subsequent entries in the linked list are searched for a matching virtual page number.

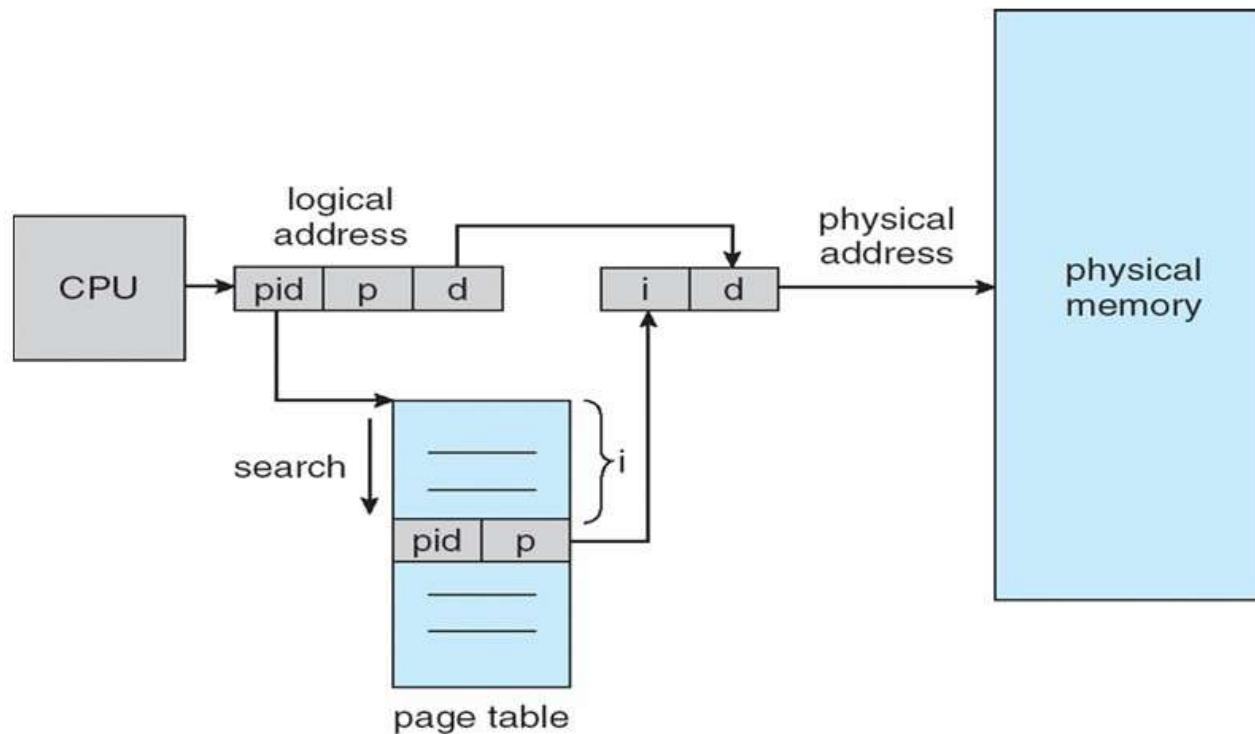
# Hashed page table



## Inverted Page Table

- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

# Inverted page table



## Problem 1

In a 64-bit machine, with 2 GB RAM, and 8 KB page size, how many entries will be there in the page table if it is inverted?

Sol:- PAS = $2\text{GB} = 2^{31}$

Page size= 8KB=  $2^{13}$

Number of entries in page table=  $2^{31}/2^{13}= 2^{28}$ .

## Problem 2

Consider a system with logical address of 34 bits and physical address of 39 bits. Page size is 16KB. The memory is byte addressable. Page-table entry size is 8 bytes. Calculate the page table size in conventional paging and inverted paging.

Solution: LAS=  $2^{34}$

PAS=  $2^{39}$

Page size=  $2^{14}$

Number of pages=  $(2^{34})/(2^{14})= 2^{20}$

Number of frames=  $(2^{39})/(2^{14})=2^{25}$

Conventional page table size= no of pages x entry size=  $2^{20} \times 8= 4\text{MB}$

Inverted page table size= no of frames x entry size=  $2^{25} \times 8= 128\text{MB}$

Thank You  
Any Questions?



# Operating Systems (CS3000)

Lecture – 40  
(Memory Management – 5)



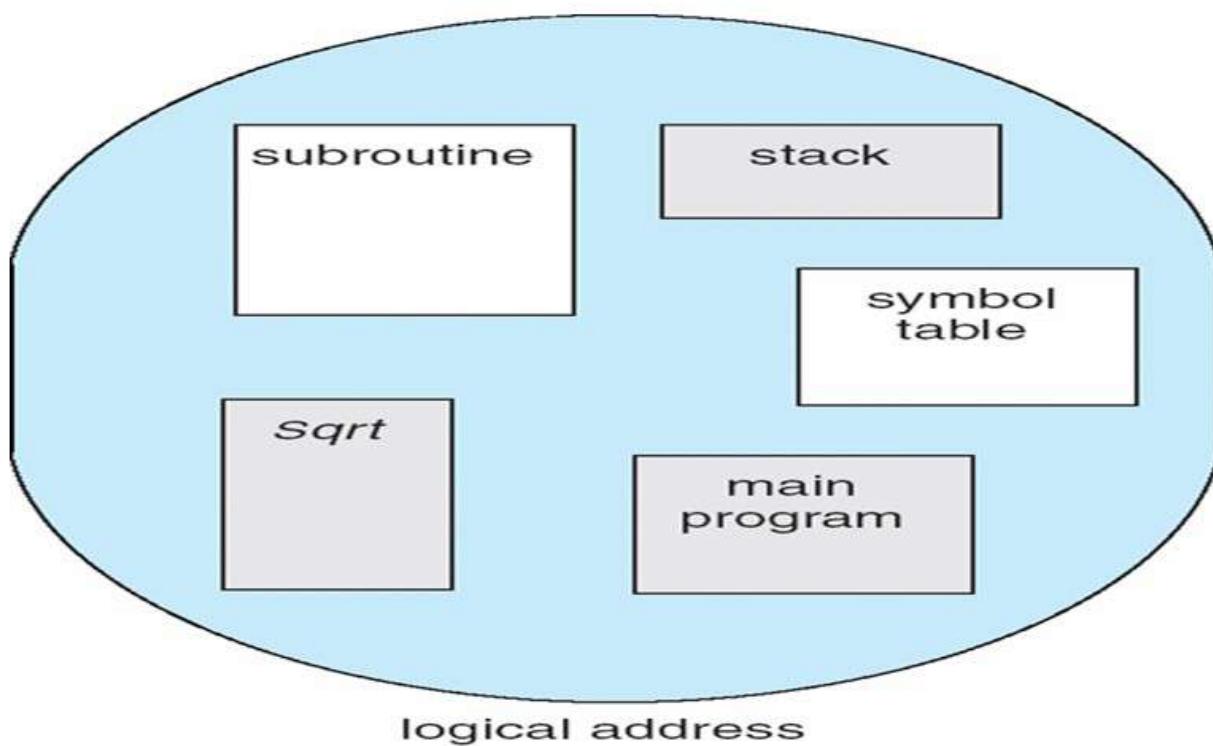
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,  
DESIGN AND MANUFACTURING,  
KANCHEEPURAM

**Dr. Sanjeet Nayak**  
Assistant Professor  
Department of Computer Sc. and Engg.

# Segmentation

- Non-contiguous memory allocation techniques like paging.
- Unlike paging in segmentation, the processes are not divided into fixed-size pages.
- Processes are divided into several modules called **segments** which improve the visualization for the users.
- Segments are of different sizes.
- Segmentation supports programmer's view of memory.

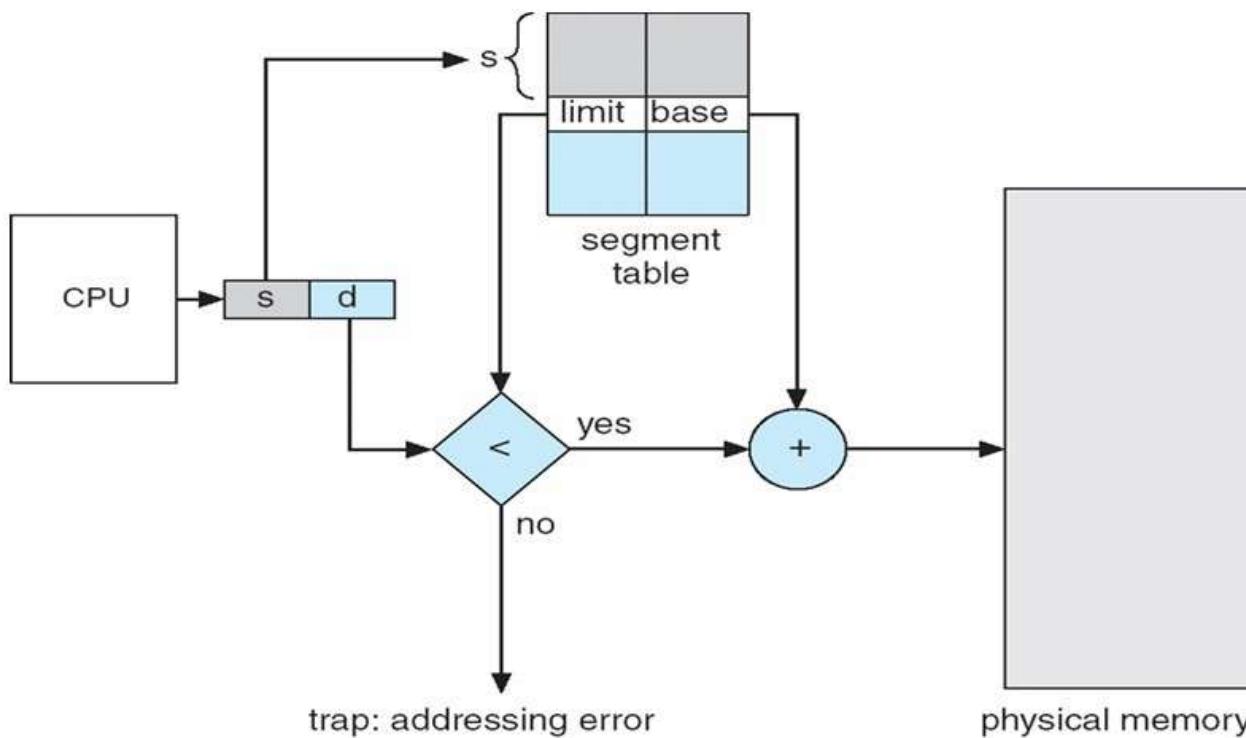
# User Views of a program



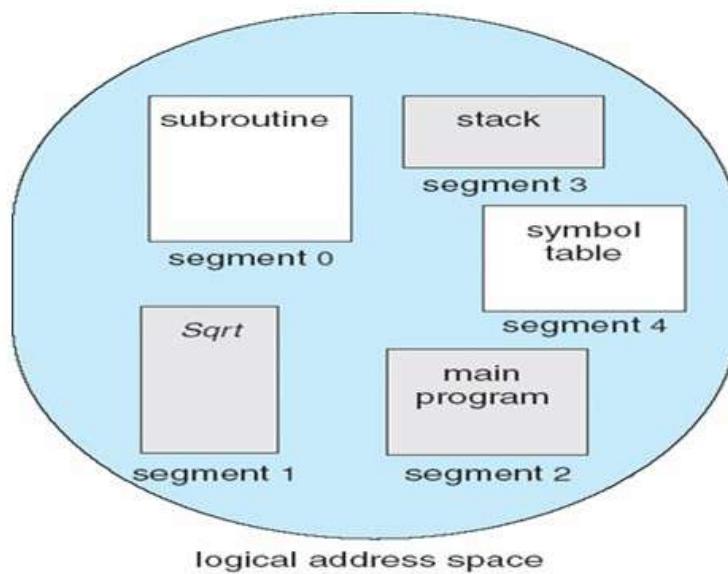
# Segmentation Architecture

- Logical address consists of a two tuple:  
 $\langle \text{segment-number}, \text{offset} \rangle$
- Segment table – maps two-dimensional physical addresses; each table entry has:  
base – contains the starting physical address where the segments reside in memory  
limit – specifies the length of the segment
- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program;  
segment number **s** is legal if **s < STLR**

# Segmentation Hardware

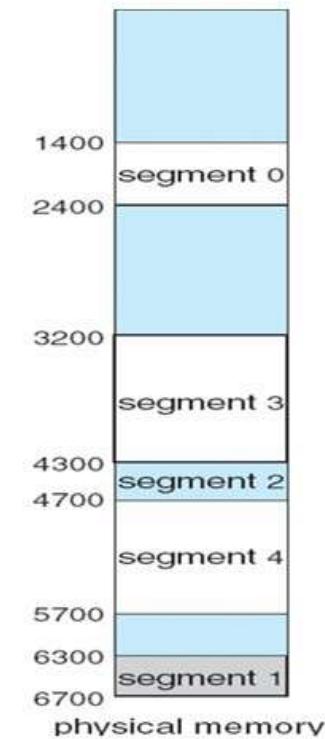


# Example of Segmentation



|   | limit | base |
|---|-------|------|
| 0 | 1000  | 1400 |
| 1 | 400   | 6300 |
| 2 | 400   | 4300 |
| 3 | 1100  | 3200 |
| 4 | 1000  | 4700 |

segment table



Thank You  
Any Questions?



# Operating Systems (CS3000)

Lecture – 41  
(File and Disk Management – 1)

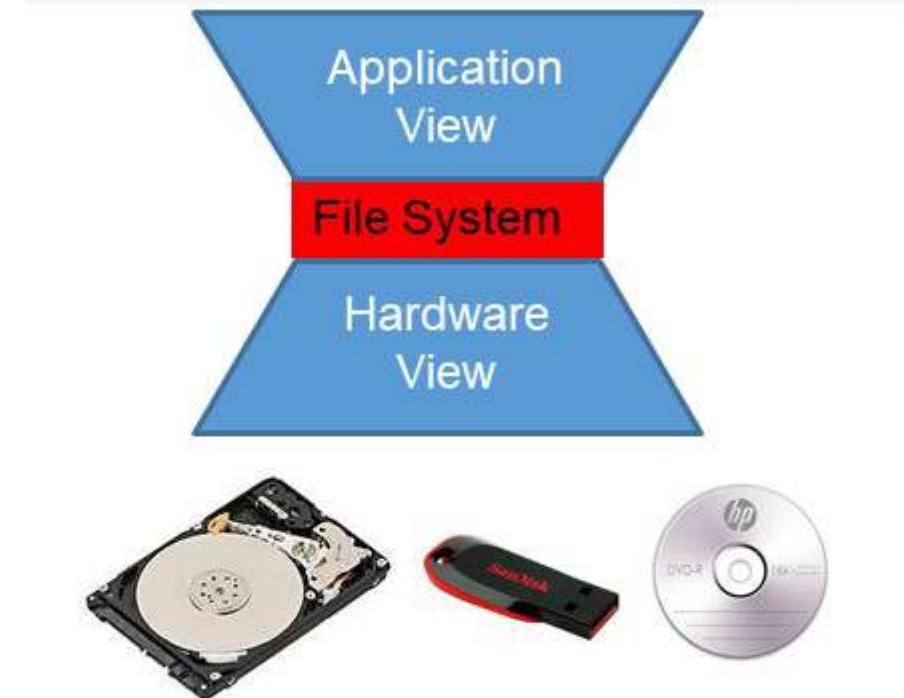
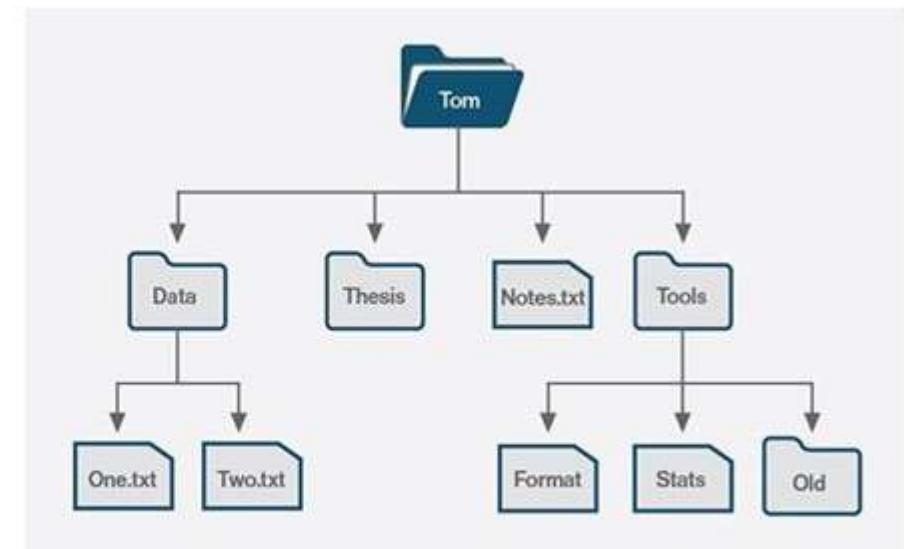


INDIAN INSTITUTE OF INFORMATION TECHNOLOGY,  
DESIGN AND MANUFACTURING,  
KANCHEEPURAM

**Dr. Sanjeet Nayak**  
Assistant Professor  
Department of Computer Sc. and Engg.

# File Systems

- Provides mechanism for storage of and access to programs
  - Logical view of stored information
    - Collection of Files
    - Directory Structure
1. User's Interface to the File Systems
  2. Internal Data Structure and algorithms used by OS to implement File System's interface
  3. Lowest Level of File Systems



# File Systems

- File Attributes
  - Name
  - Identifier
  - Type
  - Location
  - Size
  - Protection
  - Time, date and user identification

# File Systems

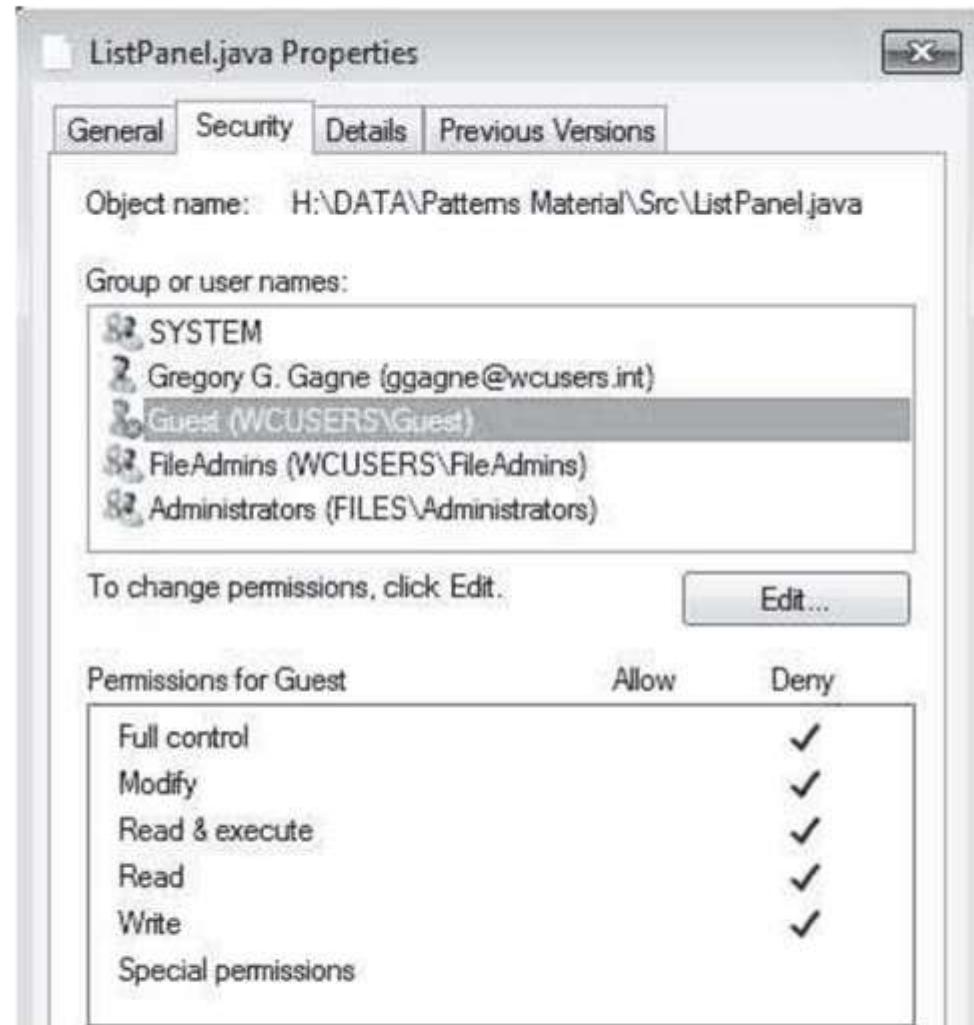
- File Operations
  - Create
  - Read
  - Reposition : seek()
  - Write
  - Delete

# File Systems

- File types
  - executable - .exe
  - Object - .obj
  - source code - .c
  - Batch - .sh
  - Markup - .xml
  - word processor - .docx
  - Library - .lib, .dll
  - Multimedia - .mpeg, .mp4

# File Systems

- File Access Permissions
  - Owner
  - Group
  - Universe
- Types of access
  - Read
  - Write
  - Execute

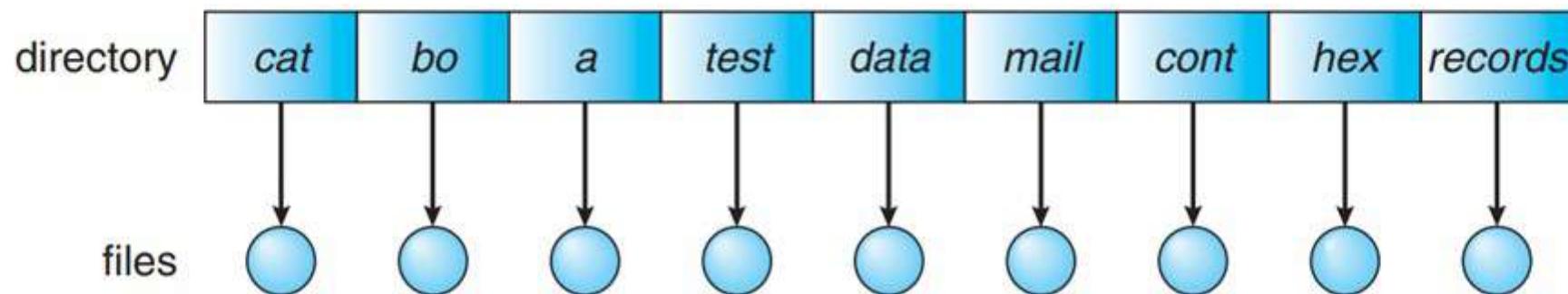


# File Systems

- Directory Structure
  - Single level
  - Two level
  - Tree structured
  - Acyclic graph
  - General graph

# File Systems

- Single level directory

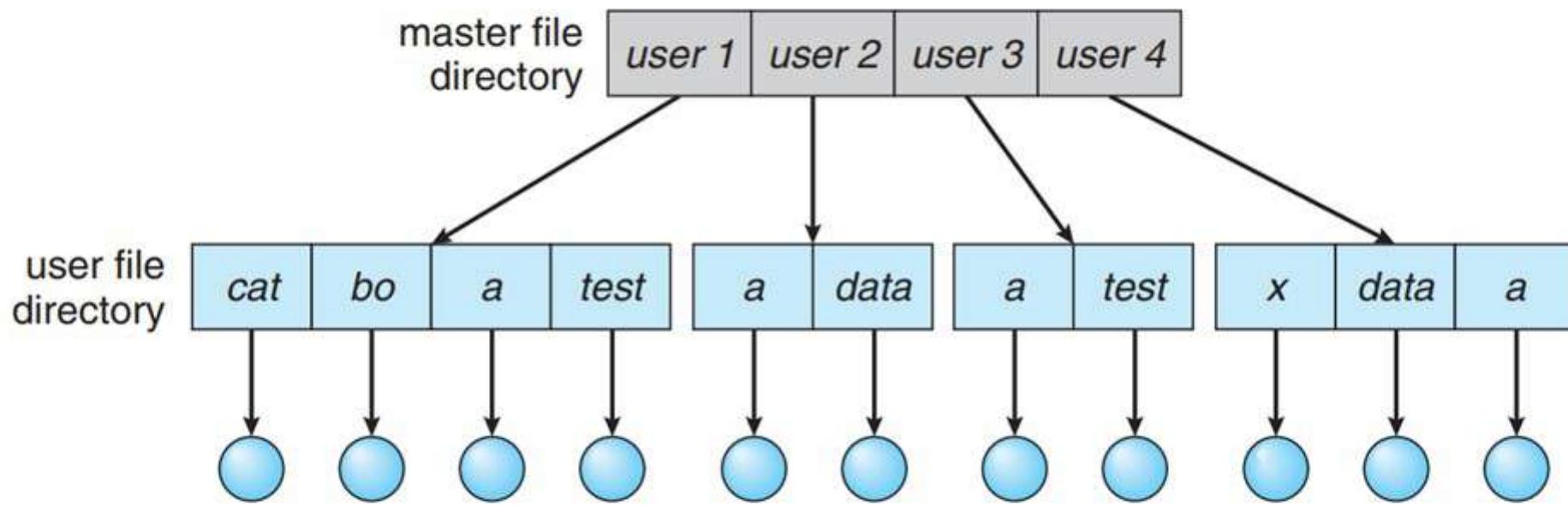


All files in single directory

Problem if no of users and files increases

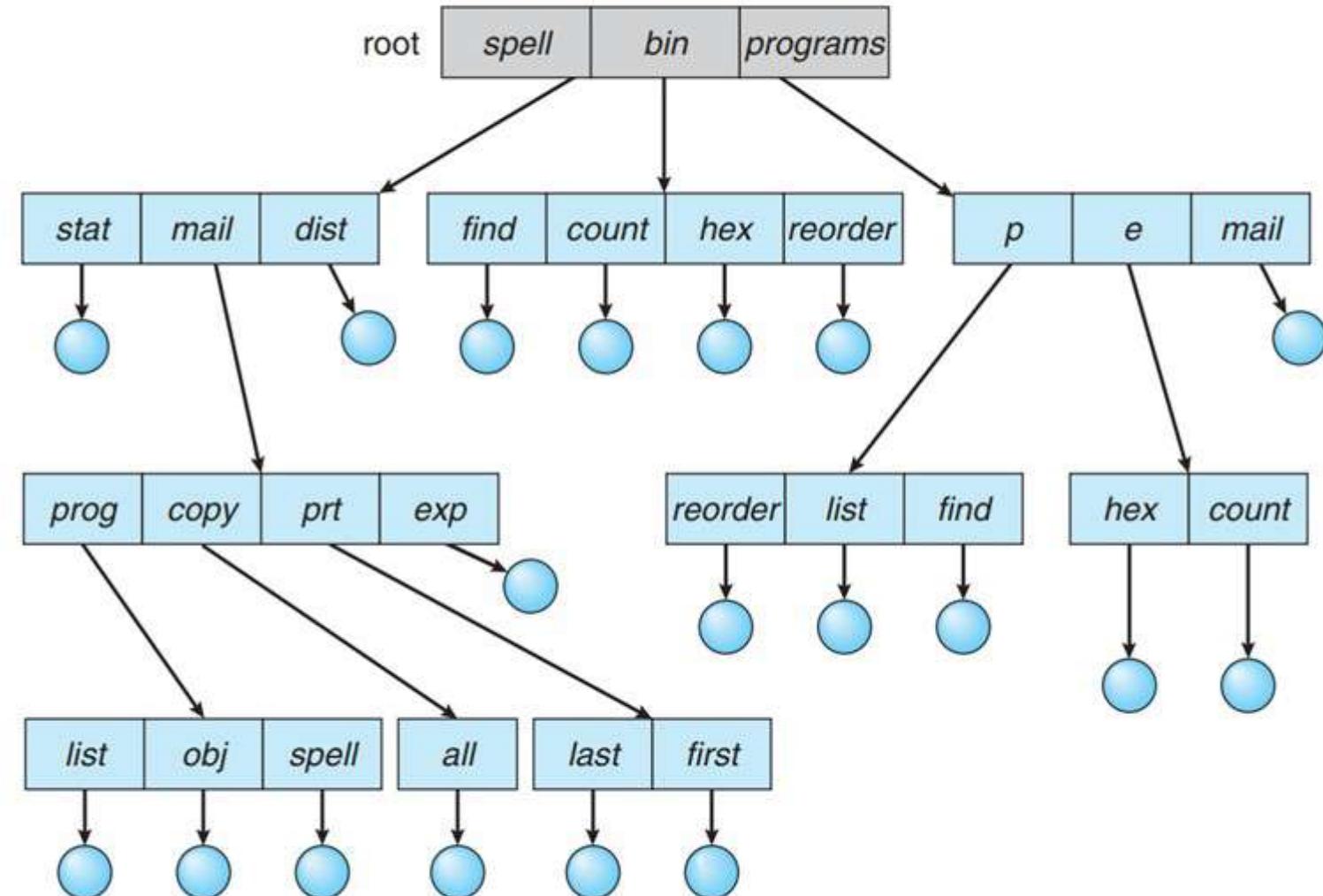
# File Systems

- Two level directory
  - UFD
  - MFD



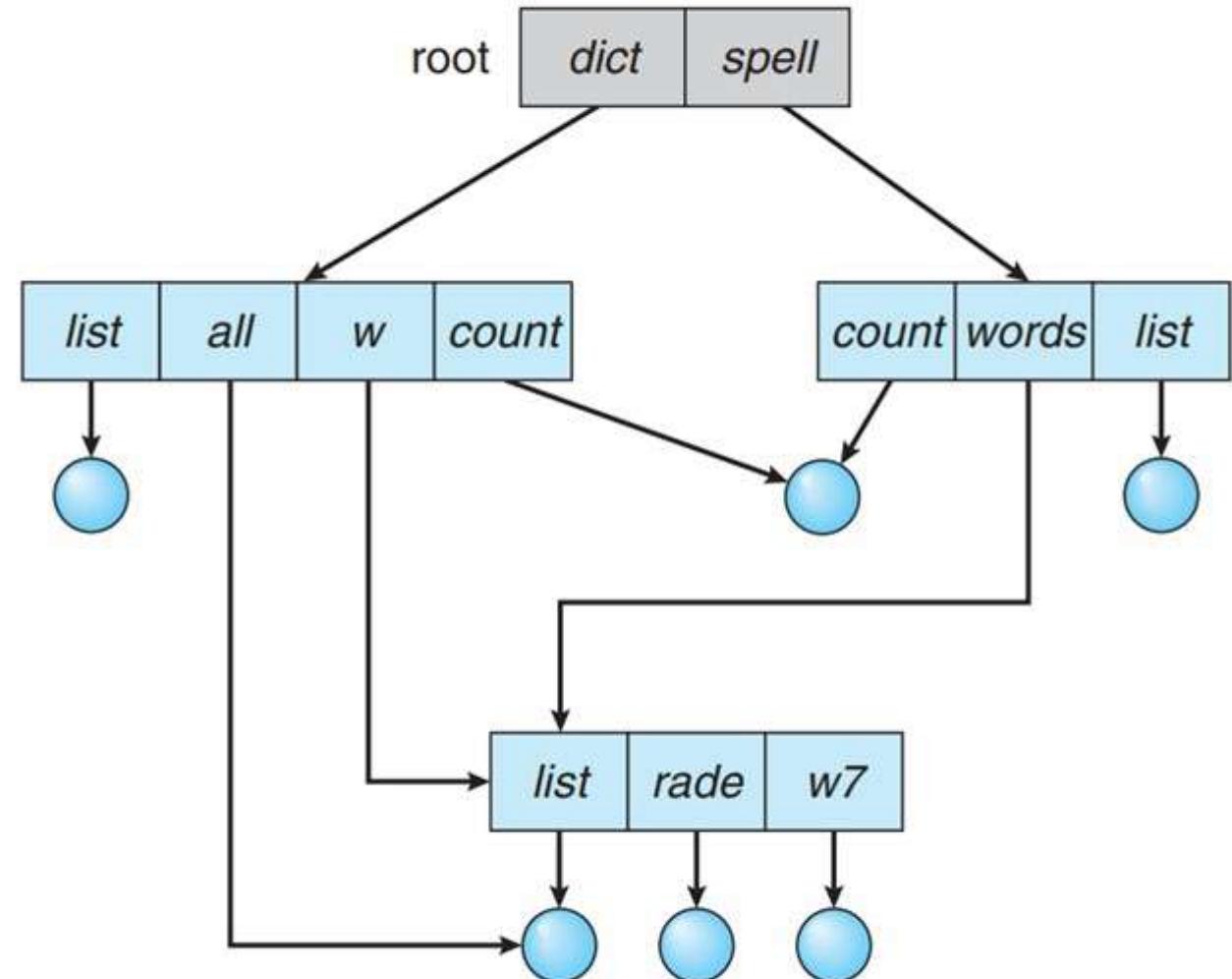
# File Systems

- Tree structured directory
    - Users create their own subdirectories
    - One bit : File/Directory
    - System call : Create/Delete Directory



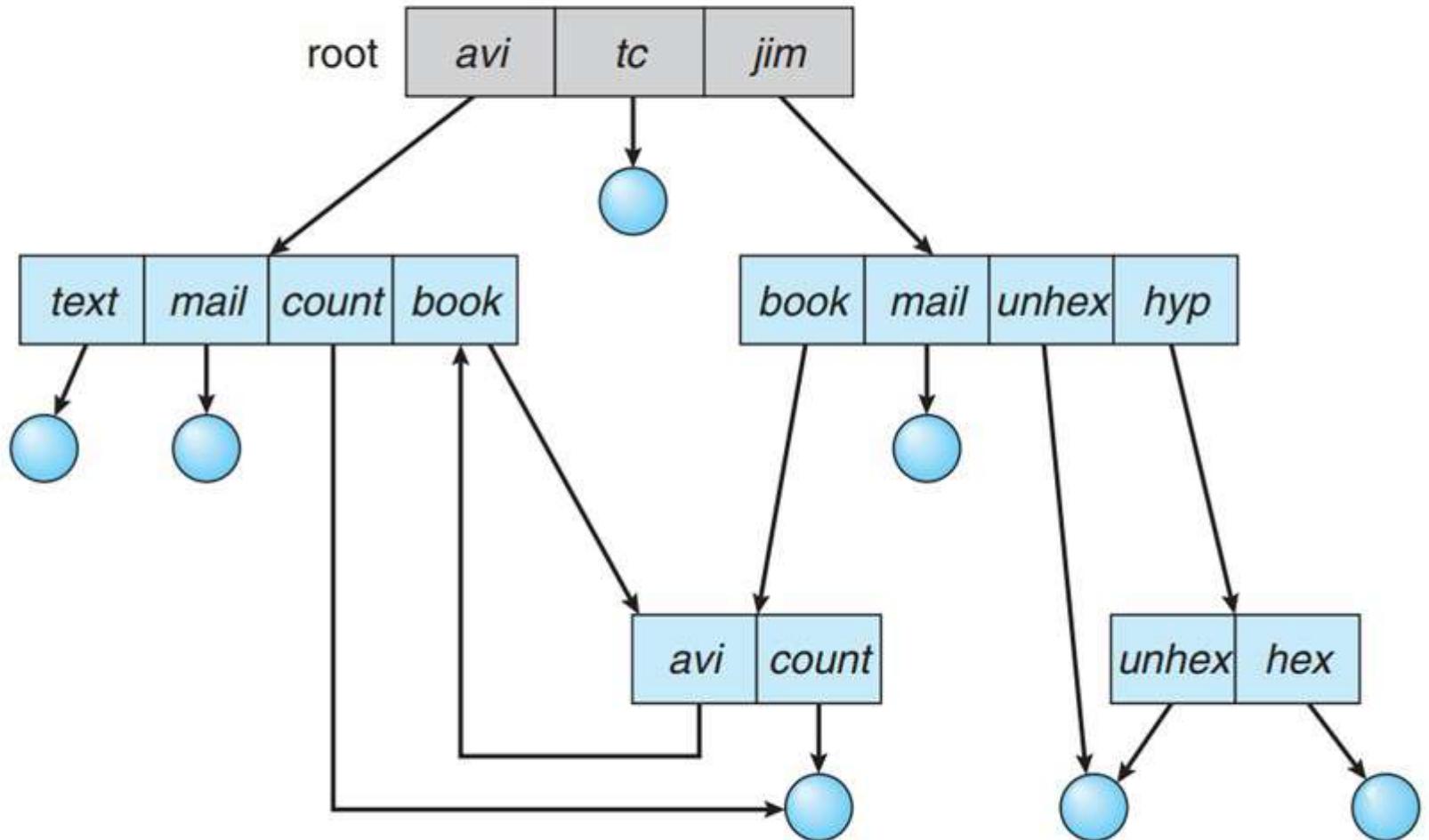
# File Systems

- Acyclic graph directory
  - Sharing of common sub directory is possible



# File Systems

- General graph directory



# File Systems

- File System Mounting
  - File system must be mounted before it can be available to process
  - OS is provided with the name of the device and the mount point
  - /home/john

# File System

- File System Implementation
  - boot control block
  - volume control block
  - directory structure
  - per-file FCB

file permissions

file dates (create, access, write)

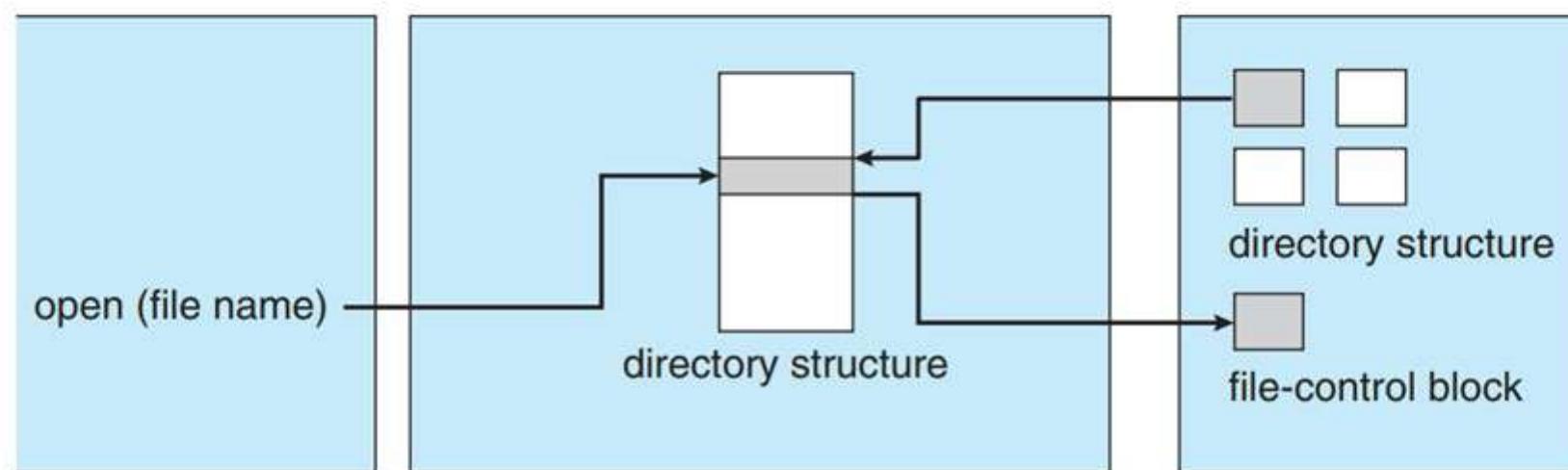
file owner, group, ACL

file size

file data blocks or pointers to file data blocks

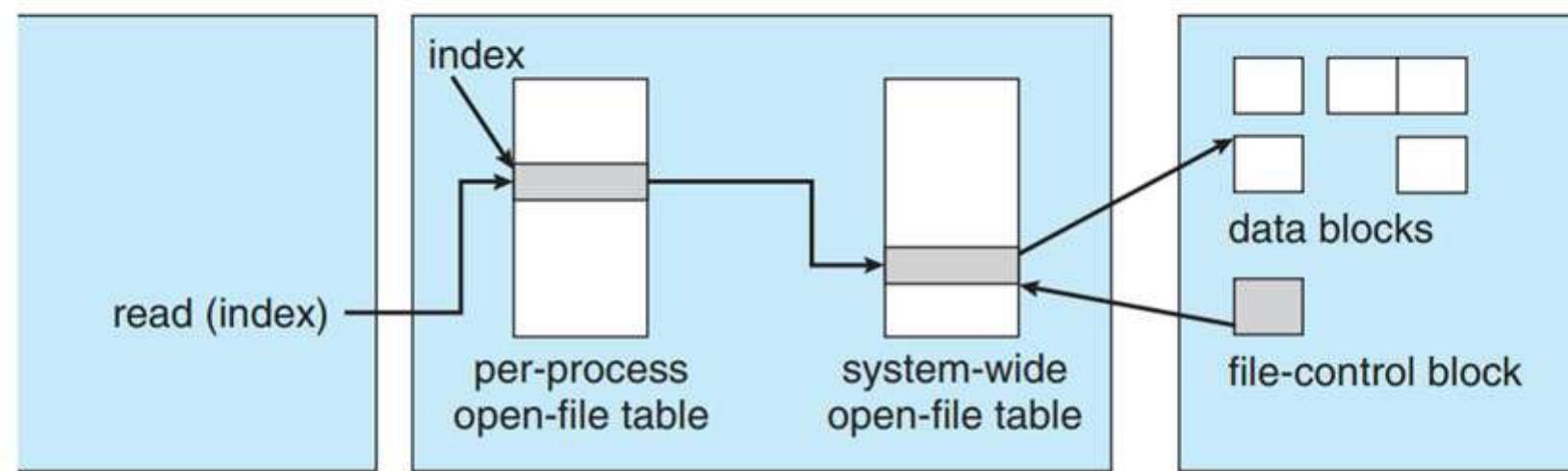
# File System

- File System Implementation (File Open)



# File System

- File System Implementation (File Read)

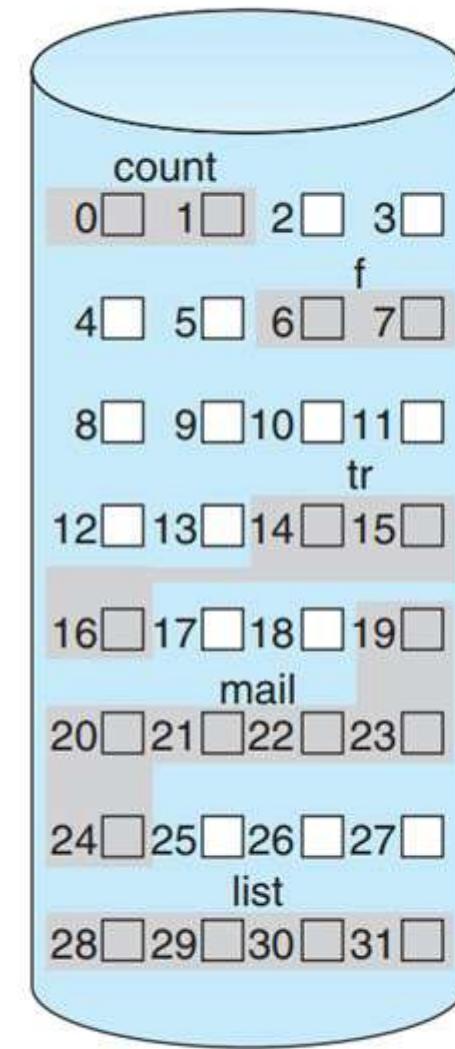


# File Systems

- File Allocation Methods
  - Contiguous Allocation
  - Linked Allocation
  - Indexed Allocation

# File Systems

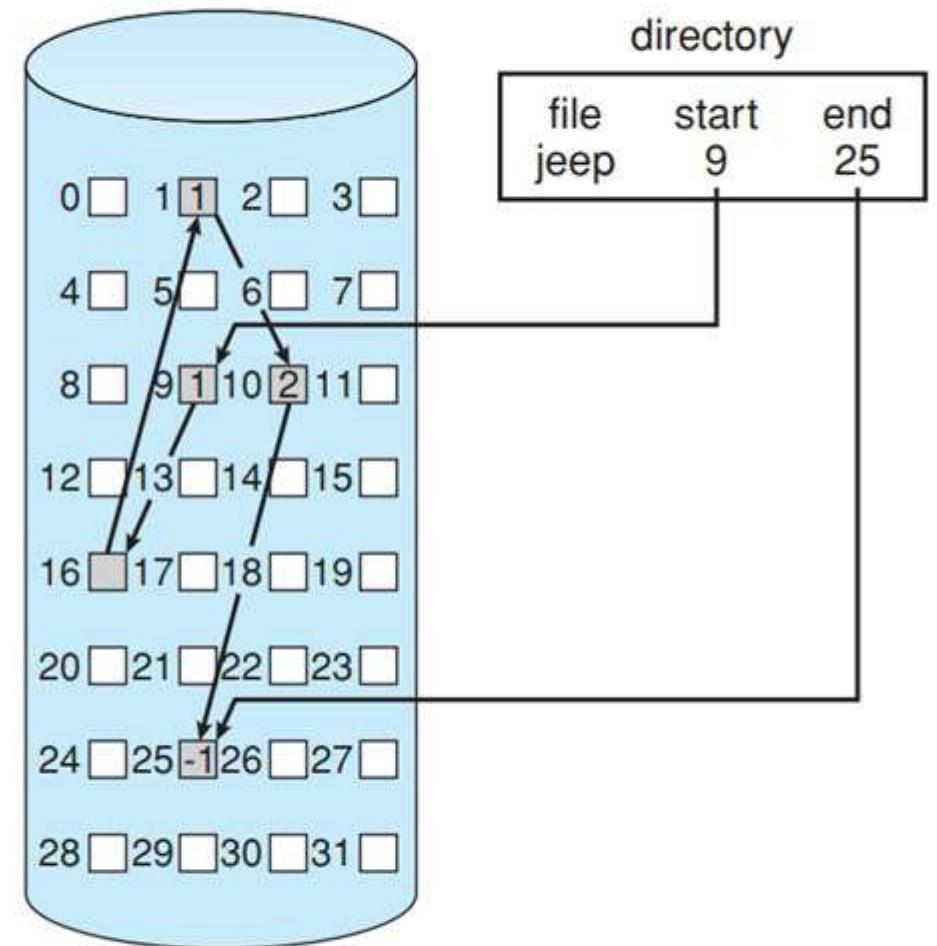
- Contiguous Allocation
  - Generally, No R/W head movement
  - Accessing File is Easy



| directory |       |        |
|-----------|-------|--------|
| file      | start | length |
| count     | 0     | 2      |
| tr        | 14    | 3      |
| mail      | 19    | 6      |
| list      | 28    | 4      |
| f         | 6     | 2      |

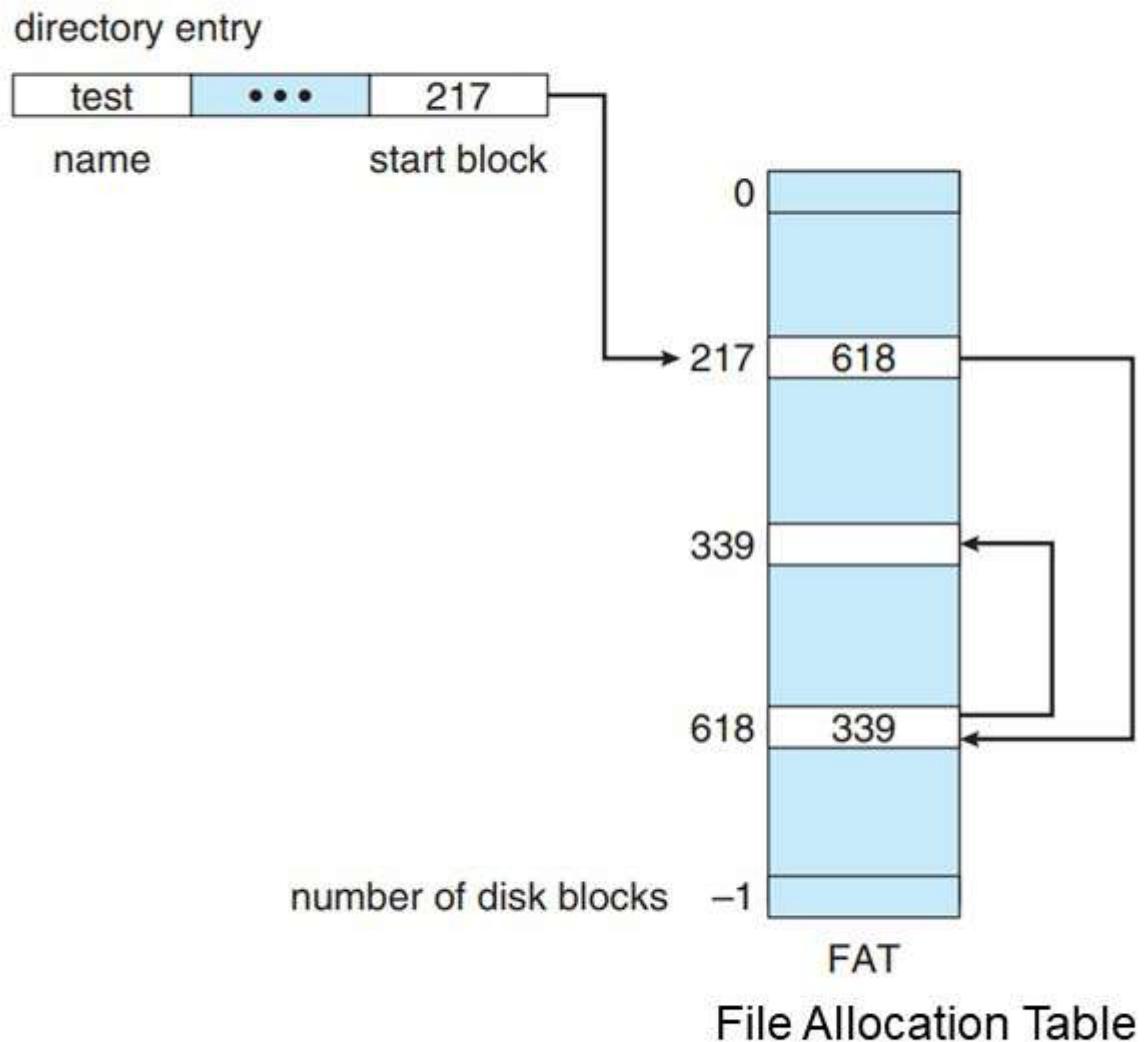
# File Systems

- Linked Allocation
  - Ex: FAT



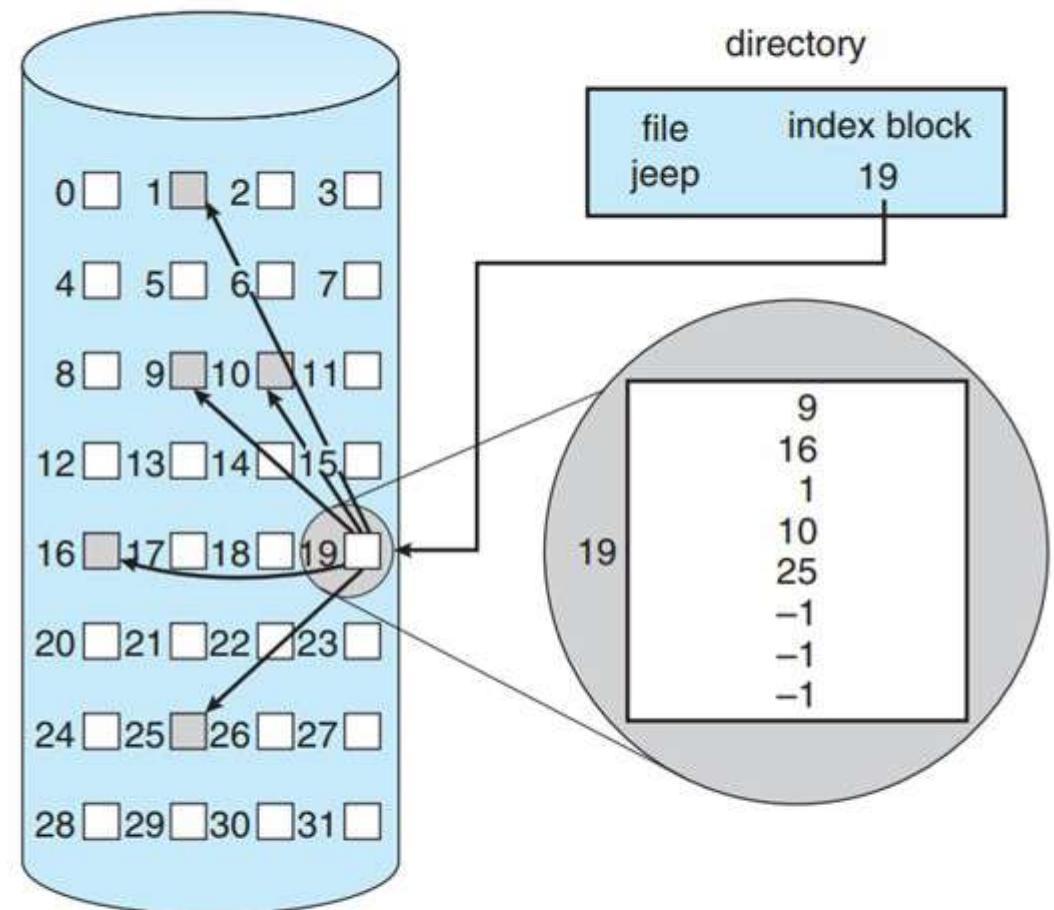
# File Systems

- Indexed Allocation
  - Index Block: All pointers to one location



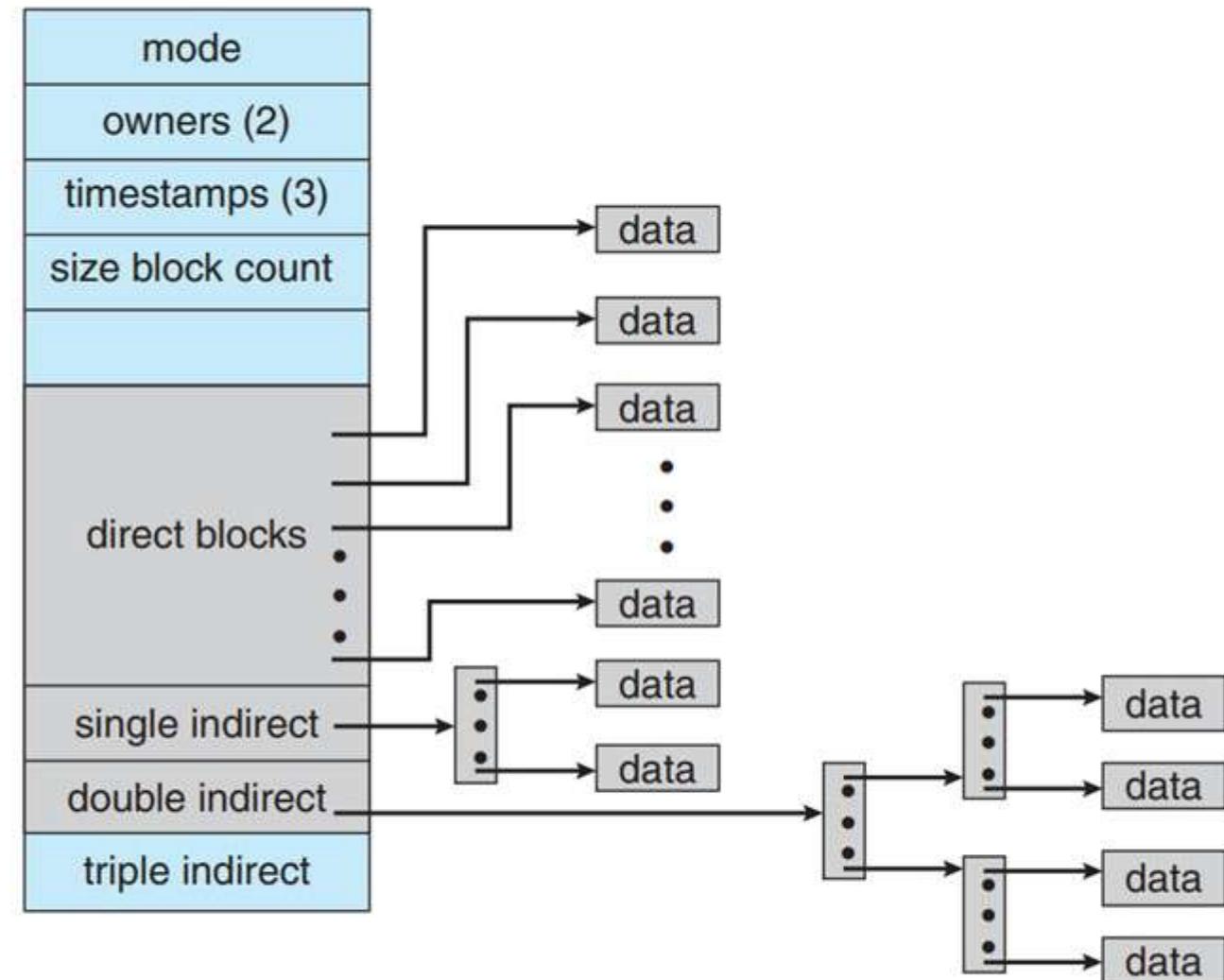
# File Systems

- Indexed Allocation



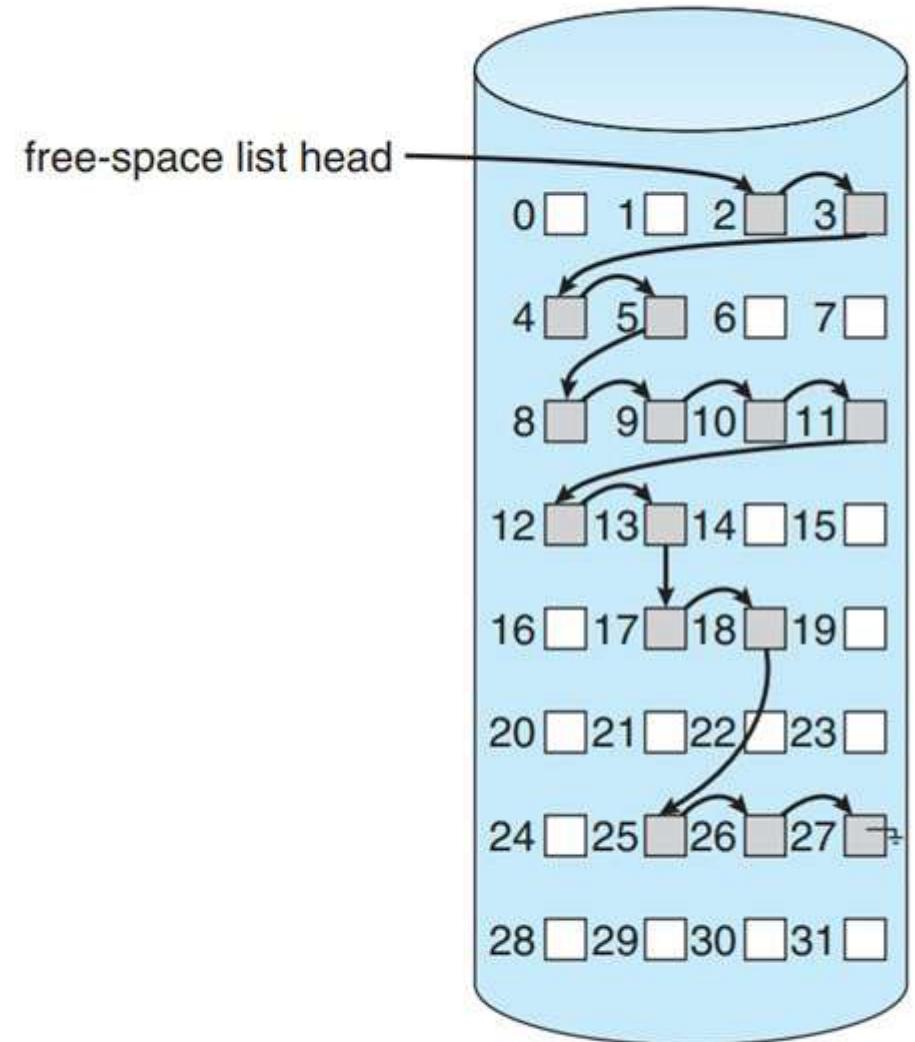
# File Systems

- Multi level Index
- UNIX inode



# File Systems

- Free Space Management
  - Bit Vector
    - 00111100111110001100000011100000 ...
  - Linked List



Thank You  
Any Questions?

