DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIALS



CS 204 OPERATING SYSTEMS

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the University of A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.
5.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

CO1	**Identify** the significance of operating system in computing devices and exemplify the communication between application programs and hardware devices through system calls.
CO2	**Apply knowledge** of Process on Inter Process Communication.
CO3	**Compare and illustrate** Synchronization process and Classical Synchronization Problems.
CO4	**Analyze** CPU Scheduling to illustrate various process scheduling algorithms.
CO5	**Learn** the concept of Memory Management.
CO6	**Appreciate** the need of access control and protection in an operating system. To apply and illustrate appropriate memory and file management schemes and various disk scheduling algorithms

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
CO1	3	-	-	-	-			-				2
CO2	3	2	-	3	2			-				3
CO3	3	3	-	2	2			-				2
CO4	3	2	3	3	2			-				2
CO5	3	2	-	-	-			-				3
CO6	3	2	2	2	3			3				3

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

MAPPING OF COURSE OUTCOMES WITH PROGRAM SPECIFIC OUTCOMES

CO'S	PSO1	PSO2	PSO3
CO1	2	3	-
CO2	3	3	-
CO3	3	3	-
CO4	3	3	-
CO5	2	3	-
CO6	3	3	3
CO1	2.67	3	3

SYLLABUS

Course code	Course Name	L-T-P -Credits	Year of Introduction
CS204	Operating Systems	3-1-0-4	2016

Pre-requisite: CS205 Data structures

Course Objectives

1. To impart fundamental understanding of the purpose, structure, functions of operating system.
2. To impart the key design issues of an operating system

Syllabus

Basic concepts of Operating System, its structure, Process management, inter-process communication, process synchronization, CPU Scheduling, deadlocks, Memory Management, swapping, segmentation, paging, Storage Management - disk scheduling, RAID, File System Interface-implementation. Protection.

Expected outcome

Students will be able to:

1. identify the significance of operating system in computing devices.
2. exemplify the communication between application programs and hardware devices through system calls.
3. compare and illustrate various process scheduling algorithms.
4. apply appropriate memory and file management schemes.
5. illustrate various disk scheduling algorithms.
6. appreciate the need of access control and protection in an operating system.

Text Book:

1. Abraham Silberschatz, Peter B Galvin, Greg Gagne, Operating System Concepts, 9/e, Wiley India, 2015.

References:

1. Garry Nutt, Operating Systems: 3/e, Pearson Education, 2004
2. Bhatt P. C. P., An Introduction to Operating Systems: Concepts and Practice, 3/e, Prentice Hall of India, 2010.
3. William Stallings, Operating Systems: Internals and Design Principles, Pearson, Global Edition, 2015.
4. Andrew S Tanenbaum, Herbert Bos, Modern Operating Systems, Pearson, 4/e, 2015.
5. Madnick S. and J. Donovan, Operating Systems, McGraw Hill, 2001.
6. Hanson P. B., Operating System Principle, Prentice Hall of India, 2001.
7. Deitel H. M., An Introduction to Operating System Principles, Addison-Wesley, 1990.

Course Plan

Module	Contents	Hours	Sem. Exam marks

I	**Introduction**: Functions of an operating system. Single processor, multiprocessor and clustered systems – overview. Kernel Data Structures – Operating Systems used in different computing environments. **Operating System Interfaces and implementation** - User Interfaces, System Calls – examples. Operating System implementation - approaches. Operating System Structure – Monolithic, Layered, Micro-kernel, Modular. System Boot process.	7	15%
II	**Process Management**: Process Concept – Processes-States – Process Control Block – Threads. Scheduling – Queues – Schedulers – Context Switching. Process Creation and Termination. **Inter Process Communication**: Shared Memory, Message Passing, Pipes.	9	15%
	FIRST INTERNAL EXAMINATION		
III	**Process Synchronization**: Critical Section-Peterson's solution. Synchronization – Locks, Semaphores, Monitors, Classical Problems – Producer Consumer, Dining Philosophers and Readers-Writers Problems	9	15%
IV	**CPU Scheduling** – Scheduling Criteria – Scheduling Algorithms. **Deadlocks** – Conditions, Modeling using graphs. Handling – Prevention – Avoidance – Detection-Recovery.	8	15%
	SECOND INTERNAL EXAMINATION		
V	**Memory Management**: Main Memory – Swapping – Contiguous Memory allocation – Segmentation – Paging – Demand paging	9	20%
VI	**Storage Management:** *Overview of mass storage structure- disks and tapes. Disk structure – accessing disks.* Disk scheduling and management. Swap Space. **File System Interface**: File Concepts – Attributes – operations – types – structure – access methods. File system mounting. Protection. File system implementation. Directory implementation – allocation methods. Free space Management. **Protection**– Goals, Principles, Domain. Access Matrix.	10	20%
	END SEMESTER EXAM		

Question Paper Pattern:

1. There will be *five* parts in the question paper – A, B, C, D, E
2. Part A
   a. Total marks : 12
   b. *Four* questions each having *3* marks, uniformly covering module I and II; All *four* questions have to be answered.
3. Part B
   a. Total marks : 18
   b. *Three* questions each having *9* marks, uniformly covering module I and II; T*wo* questions have to be answered. Each question can have a maximum of three subparts
4. Part C
   a. Total marks : 12
   b. *Four* questions each having *3* marks, uniformly covering module III and IV; All *four* questions have to be answered.
5. Part D
   a. Total marks : 18
   b. *Three* questions each having *9* marks, uniformly covering module III and IV; T*wo* questions have to be answered. Each question can have a maximum of three subparts
6. Part E
   a. Total Marks: 40
   b. *Six* questions each carrying 10 marks, uniformly covering modules V and VI; *four* questions have to be answered.
   c. A question can have a maximum of three sub-parts.
7. There should be at least 60% analytical/numerical/design questions.

APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY
KTUNOTES.IN
Estd. 2014

QUESTION BANK

MODULE I				
Q:NO:	**QUESTIONS**	**CO**	**KL**	**PAGE NO:**
1	Explain various types of operating systems	CO1	K2	1
2	Explain about the purpose of operating system	CO1	K2	21
3	What you mean by buffering, explain in detail	CO1	K2	32
4	What is a resource, why it is called the resource manager	CO1	K2	09
5	Define process, What are the different components that the process consists of	CO1	K2	15
7	What is device management	CO1	K2	14
8	What are different parts of device	CO1	K1	16
9	Explain different types of process modes	CO1	K2	23
10	What are different types of I/O strategies	CO1	K2	25
11	Explain direct I/O polling.	CO1	K2	26
12	Explain about direct memory access	CO1	K2	25
13	Explain interrupt driven I/O	CO1	K2	27
14	Explain device class characteristics	CO1	K2	35
15	What you mean by an object	CO1	K2	13
MODULE II				
1	Define process	CO2	K1	44
2	With the help of an appropriate diagram explain various process states	CO2	K2	44
3	Differentiate process and resource abstraction	CO2	K3	46
4	Explain process hierarchy in detail	CO2	K2	48
5	What you mean by process address space	CO2	K5	53

6	Explain Race condition	CO2	K2	56
6	What is thread, explain thread data and thread status	CO1	K2	10
8	Describe interprocess communication in detail	CO2	K2	69
9	Explain PCB in detail	CO2	K2	45
	MODULE III			
1	What is process synchronization?	CO3	K2	55
2	Define the method for implementing semaphore with example.	CO3	K2	60
3	Define the role of semaphore mechanisms in OS.	CO3	K2	60
4	Explain the producer consumer problem with example.	CO3	K2	68
5	What is context switch	CO2	K1	74
6	With an example explain dining philosophers problem	CO2	K2	71
7	What are the classic problems of synchronization explain in detail	CO2	K2	66
8	With an example explain critical section problem	CO2	K5	57
9	With an example explain readers writers problem	CO2	K5	68
	MODULE IV			
1	Differentiate between pre-emptive and on pre-emptive scheduling.	CO4	K3	76
2	What are the disadvantages of Shortest Job first algorithm? What is the strategy to overcome it?	CO4	K2	80
3	What is pre-emptive scheduling? Give one example of pre-emptive scheduling & its Disadvantages.	CO4	K2	76
4	What are the necessary conditions that cause deadlock in a system?	CO4	K3	86
5	What is circular wait? Explain	CO4	K2	88
6	Explain the term Resource allocation graph.	CO4	K2	88

<table>
<tr>
<td>7</td>
<td>Find the average waiting time and average turnaround time for the processes given in the table below using:- i) SRT scheduling algorithm ii) Priority scheduling algorith
<table>
<tr><th>Process</th><th>Arrival Time (ms)</th><th>CPU Burst Time (ms)</th><th>Priority</th></tr>
<tr><td>P1</td><td>0</td><td>5</td><td>3</td></tr>
<tr><td>P2</td><td>2</td><td>4</td><td>1</td></tr>
<tr><td>P3</td><td>3</td><td>1</td><td>2</td></tr>
<tr><td>P4</td><td>5</td><td>2</td><td>4</td></tr>
</table>
</td>
<td>CO4</td><td>K6</td><td>76</td>
</tr>
<tr>
<td>8</td>
<td>Consider a system with four processes P1, P2, P3, P4, and three types of resources R1, R2, R3. The maximum no: of instances of resources of each type are 5, 7 and 7 respectively. What will be the order of processing of jobs if the allocated matrix and the maximum claim
<table>
<tr><th></th><th colspan="3">Allocated Matrix</th><th colspan="3">Maximum Claim</th></tr>
<tr><td>P1</td><td>2</td><td>1</td><td>3</td><td>3</td><td>5</td><td>6</td></tr>
<tr><td>P2</td><td>0</td><td>0</td><td>1</td><td>1</td><td>3</td><td>4</td></tr>
<tr><td>P3</td><td>1</td><td>2</td><td>1</td><td>1</td><td>4</td><td>3</td></tr>
<tr><td>P4</td><td>1</td><td>1</td><td>0</td><td>2</td><td>3</td><td>1</td></tr>
</table>
</td>
<td>CO4</td><td>K6</td><td>76</td>
</tr>
<tr>
<td>9</td>
<td>Consider the following snapshot of a system with five processes P1, P2, P3, P4, P5 and four resources A,B,C,D. Using Bankers Algorithm check whether the system is in safe state or not.
<table>
<tr><th></th><th colspan="4">Allocation</th><th colspan="4">Max</th></tr>
<tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>A</th><th>B</th><th>C</th><th>D</th></tr>
<tr><td>P1</td><td>1</td><td>0</td><td>2</td><td>2</td><td>3</td><td>2</td><td>5</td><td>2</td></tr>
<tr><td>P2</td><td>0</td><td>2</td><td>1</td><td>2</td><td>3</td><td>4</td><td>1</td><td>2</td></tr>
<tr><td>P3</td><td>2</td><td>4</td><td>5</td><td>0</td><td>2</td><td>7</td><td>7</td><td>3</td></tr>
<tr><td>P4</td><td>3</td><td>0</td><td>0</td><td>0</td><td>5</td><td>5</td><td>0</td><td>7</td></tr>
<tr><td>P5</td><td>4</td><td>2</td><td>1</td><td>3</td><td>6</td><td>2</td><td>1</td><td>4</td></tr>
</table>
<table>
<tr><th colspan="4">Available</th></tr>
<tr><th>A</th><th>B</th><th>C</th><th>D</th></tr>
<tr><td>3</td><td>0</td><td>0</td><td>1</td></tr>
</table>
</td>
<td>CO4</td><td>K6</td><td>99</td>
</tr>
<tr>
<td>10</td>
<td>1. Consider the following set of processes that arrive at time 0with the length of the CPU burst time given in milliseconds. Schedule the process using Round Robin Scheduling Algorithm and the time slice is 2ms.
<table>
<tr><th>Process</th><th>Burst Time</th></tr>
<tr><td>P1</td><td>24</td></tr>
<tr><td>P2</td><td>3</td></tr>
<tr><td>P3</td><td>3</td></tr>
</table>
</td>
<td>CO4</td><td>K6</td><td>84</td>
</tr>
</table>

	MODULE V			
1	What are the sequence of memory address generated by a running program, explain	CO5	K2	113
2	Differentiate logical and physical address space	CO5	K4	114
3	Analyze the advantages of dynamic loading	CO5	K2	117
4	Compare compile time, link time and load time	CO5	K3	118
5	List out various memory allocation strategies	CO5	K1	123
6	Discuss the various Memory partition strategies	CO5	K3	124
7	Compare best fit and worst fit replacement strategies	CO5	K2	127
8	Explain the concept of paging in detail	CO5	K2	135
9	What you mean by TLB explain using appropriate figures	CO5	K3	139
10	Differentiate static and dynamic paging algorithms	CO5	K4	141
11	Using appropriate figures explain demand paging	CO5	K3	143
12	Explain Bedlys anomaly	CO5	K2	147
13	Compare FIFO and optimal page replacement algorithms	CO5	K4	147
	MODULE VI			
1	List out the file attributes	CO6	K2	165
2	What are the various file operations, explain in detail	CO6	K2	166
3	Differentiate sequential and direct file access methods	CO6	K4	169
4	List out various operations in a byte stream file	CO6	K1	171
5	Using appropriate figures explain file OPEN(), CLOSE() operations	CO6	K3	175
6	Explain in detail about block management	CO6	K2	178
7	Explain in detail about various storage abstractions	CO6	K2	181
8	What are the various methods used to protect files, Explain	CO6	K2	183

APPENDIX 1

CONTENT BEYOND THE SYLLABUS

S:NO;	TOPIC	PAGE NO:
1	Windows 3X.	222
2	Windows 9X	222
3	Windows NT	224
4	Windows XP	225
6	Windows 8	226
7	Windows 10	227

MODULE NOTES

Module I :

Operating System Strategies :

OS Strategy – to share resources.

↳ multiprogramming.

Six different types of OS :-

Batch systems : Handle collections, or batches of jobs.

Job is a predefined sequence of cmds (such as compile a pgm or execute a pgm), pgms & data.

— The job is self-sufficient in the sense that it contains all the pgms & data required for ex? without human intervention.

— first to use multiprogramming – allowed the OS to execute a few jobs ~~contin~~ concurrently.

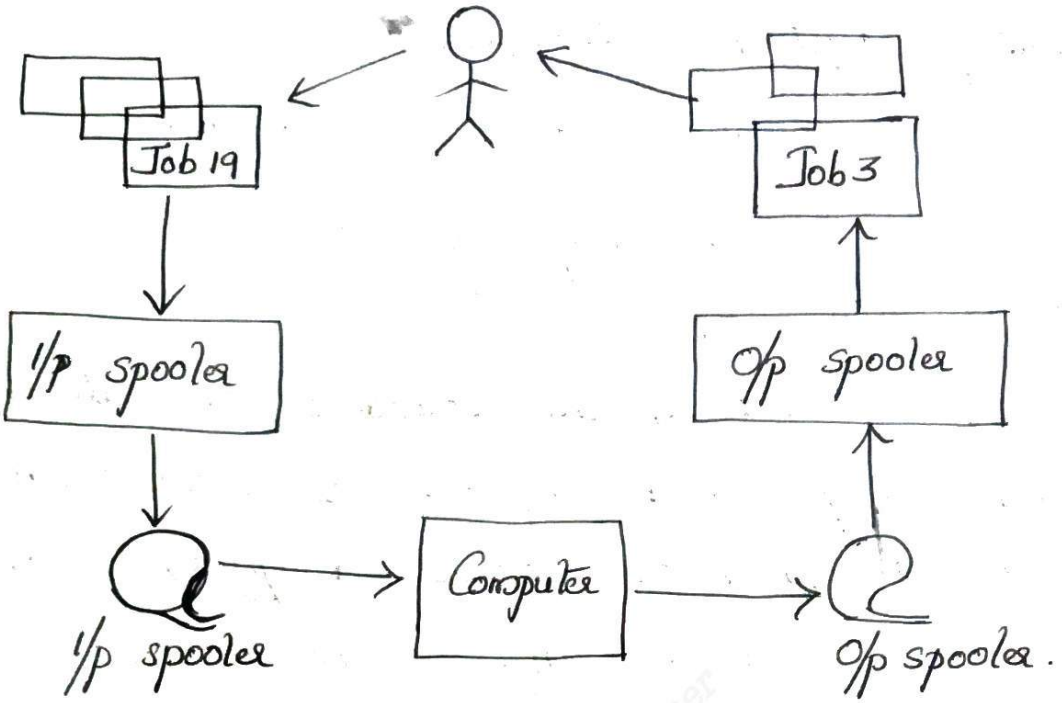
Batch Systems

— A batch processing system services individual jobs from a predefined collection of jobs. Each batch job is specified by a predefined list of OS cmds (copy a file or print a file) called job ctl specification.

Batch Processing System :-

i/p spooler groups jobs into a batch, then sends to the computer. As the computer completes jobs, it writes each job's o/p to the o/p spool.

The o/p spool is printed & the job results are returned to the end user.



Batch Processing Technology :-

Spooling : reads each job & saves it in the current batch of jobs. The jobs were saved onto magnetic tape & the tape was mounted on the main computer. The main computer then read jobs from the i/p batch tape, executed them, & wrote the results to an o/p spool tape.

The O.S uses scheduling policies to determine the order to execute the jobs.

Medium term scheduling : Once the OS selects a job from the batch, it allocates memory to the job. After the job is allocated memory, it can begin to compete for the processor. When the

processor becomes available, the processor scheduler selects a job currently loaded in the memory & allocates the processor to it.

A job can use the processor only while it is loaded in the memory. When a job completes exn, its mem. is released & the o/p for the job is copied to an o/p spool.

In some batch s/ms, the OS preemptively releases the memory allocated to a job, this means that the job can no longer compete for the processor. Such s/ms are called swaping s/ms.

Scanned by Easy Scanner

Personal Computers & Workstations:

Workstation is a special computer designed for technical or scientific applications. Workstations generally come with a large, high resolution graphics screen; Most workstations also have a mass storage device such as a disk drive, but a special type of workstation called a diskless workstation.

Workstations lies b/w personal computers & minicomputers

Personal Computer (IBM): is a general-purpose computer whose size, capabilities and original sale price makes it useful for individuals, & which is intended to be operated directly.

A workstation is a high-end personal computer designed for technical or scientific applications. Workstations were configured with enough resources that it made sense to use timesharing technology. The ROM based Operating systems provided a few routines to control the personal computer's devices. Soon, ROM based s/ms were eshanced by additional as s/w usually to manage files that could be loaded from a disk into the computer's readable/writable random access memory (RAM).

MS-DOS, Windows

Workstation h/w was originally more flexible and faster than personal computer h/w. They typically incorporated more resources than a personal computer

Embedded Systems

An embedded s/m is a computer that is a component in some more complex s/m. It is dedicated to the support of its host s/m. Eg. of it include computers that ctrl the flood gates of a dam, control the cooling process of a nuclear reactor, guide a missile, ctrl a point of sale terminal, etc.

Embedded systems do not have a human user. Instead, the s/m's user is a set of sensors & actuators.

Real time computing is based on the idea that the "user" (a collection of h/w sensor/actuators) is required to receive a guaranteed response time for any request to perform work. For eg:, a sensor detects an increase in a reactor's core temp., so the embedded s/m must send a s/l to the actuator to enable the cooler within a small fixed amt of time.

Soft real time & Hard real time

Soft real time : partial credit can be awarded if the deadline is not ~~as~~ met. ie; if it fails to meet the deadline, the s/m should continue to provide service rather than abandoning the service request due to failure.

Hard real time: There is no use ~~of meeting~~ if they are not meeting deadline

Sometimes an embedded s/m has only a single purpose: to run a single application pgm. If there is only one appl^n in the embedded s/m, it may not be necessary to implement resource isolation & sharing among concurrent processes. The main purpose of the OS will then be to provide h/w resource abstraction. As a result, it reduces the amount of m/c resource used by the OS, leaving more for the application.

Contemporary embedded s/ms are often concerned with the amt of power the processor is consuming at any given moment.

Small Communicating Computers:

Advances in chip technology & the ubiquity of the internet have been the catalyst for a revolution in computing devices. Many of these devices are used for specific tasks - (from portable MP3 players to internet capable mobile phones.

SCC requirements encourage the evolution of OS technology. There has been a steady move toward thread-based computing in modern O.S.

B/c thread-based computing uses fewer s/m resources than does process-based computing, SCCs are designed around thread computation rather than process computation.

SCCs contain little or limited amounts of device storage.

SCCs operate in situations where resource mgmt is more challenging than in conventional s/ms. For eg: an SCC in a mobile computer is required to conserve as much battery power as possible, since the amt of battery power can sustain s/m op^n is a limiting factor on the overall utility of the computer.

In mobile computing envt, a n/w conn^n may not be available at certain times during the m/c's op^n. Alternatively the characteristics of the n/w may vary with time — some times wireless s/l is strong & bandwidth is high, while at other times, the s/l is weak & the bandwidth is lower. SCC resource mgmt strategies need to be able to adapt to dynamic resource availability.

Networks

- point to point & bit serial comm^n media.
- Routing n/w. & path.

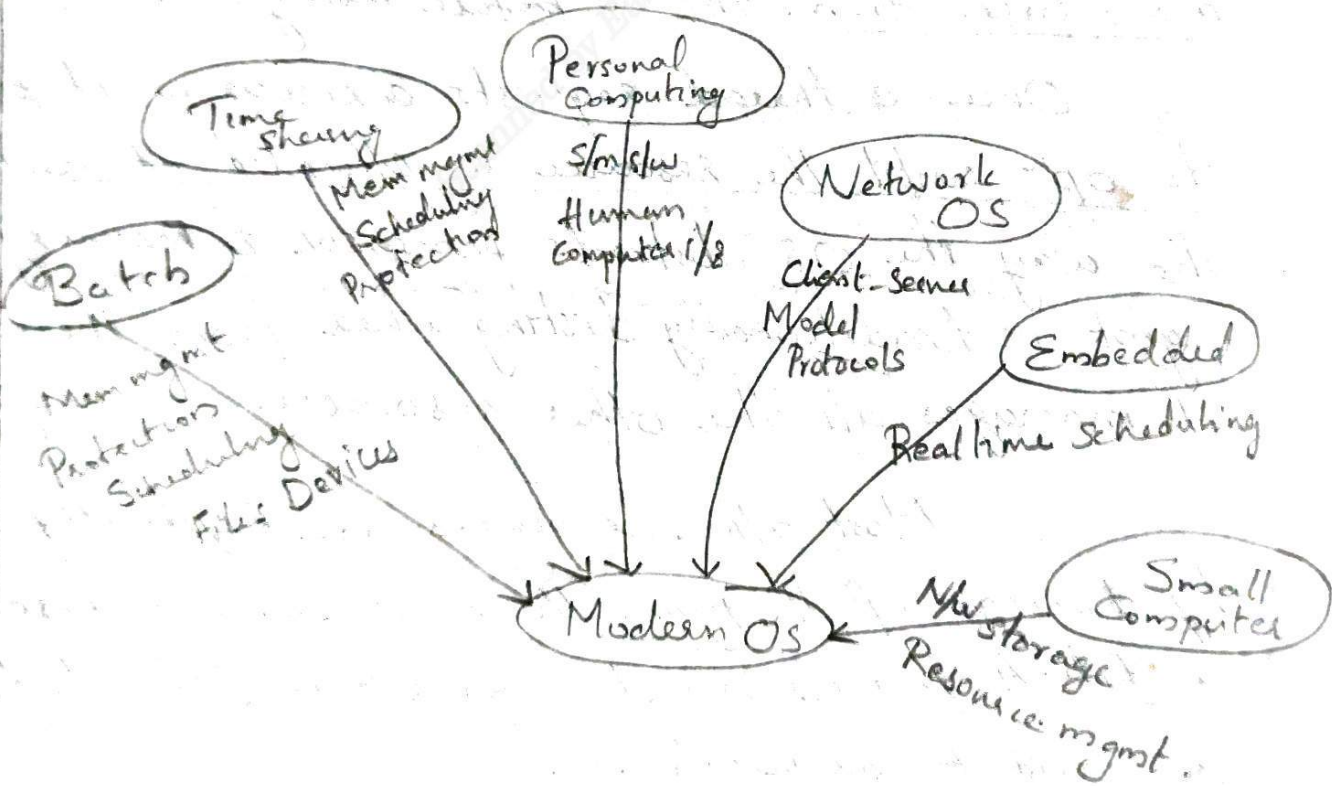
Each m/c must be able to forward inf^n to other m/cs so that all m/cs collectively implement a logical n/w that behaves as if it were fully connected.

LANs. – cost effective technology (1Mbps)

Wifi ✓

Web browsers: ✓

N/w protocols: IP, Transmission Ctrl Protocol (TCP)



Resources

All components (physical & logical) of the abstract m/c that are needed to execute a pgm are called resources. When a process executes, it requires some of the computer's resources. at least the processor & memory. It also needs to be able to store & retrieve inf^n to/from the computer's i/p /o/p devices.

The OS is responsible for managing its resources.

- When a process is executing, it must request a resource from the OS before using it.

- Once a thread requests a resource, it suspends its op^n until the resource is allocated it to it.

The way the OS abstracts processor & memory resource mgmt is traditionally distinguished from the way it manages all the other resources.

Most other resources are explicitly reqstd by ₹ action of the pgm. The most common resource is a file. A process must request a file before it can read it or write it.

open( ) - s/m call.

File is a named, linear stream of bytes of inf^n that are kept on a device.

If a process requests a resource that is unavailable, the thread normally discontinues ex^n & remains suspended until the resource becomes available. Every OS provides access to a no. of resources in addition to files, including the processor, memory keyboards & displays.

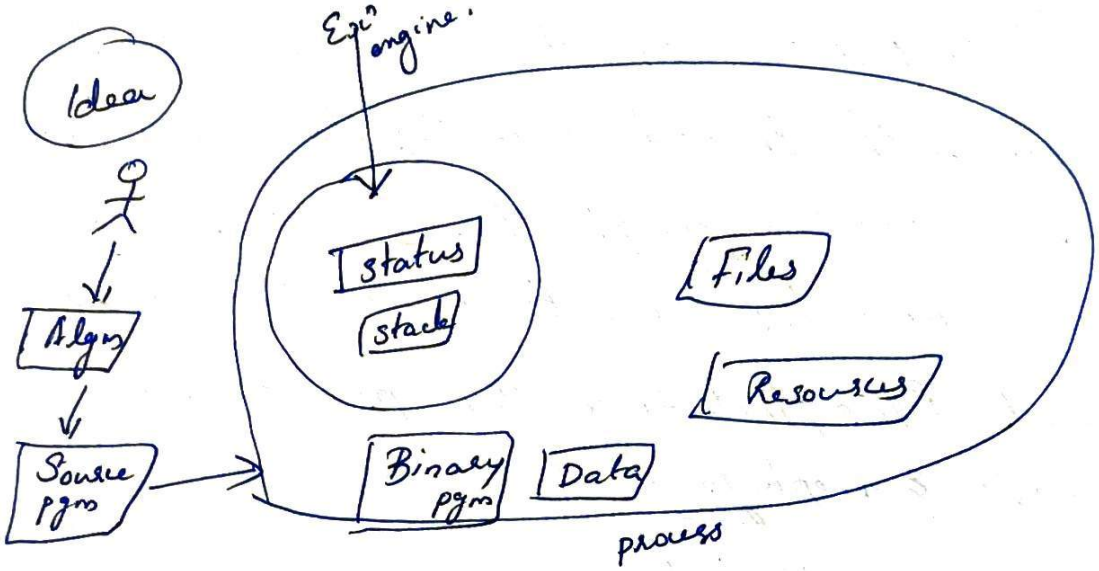
> Processes & Threads

A computation is a combination of a process, atleast one thread and a collection of resources.

The process component is made up of

* a binary pgm to be executed. ✓ obj code
* Data on which the pgm will execute (obtained either from a file or interactively from the user of the process.
* Resources: required for ex^n, including files, that contain data on which the executing program will work.

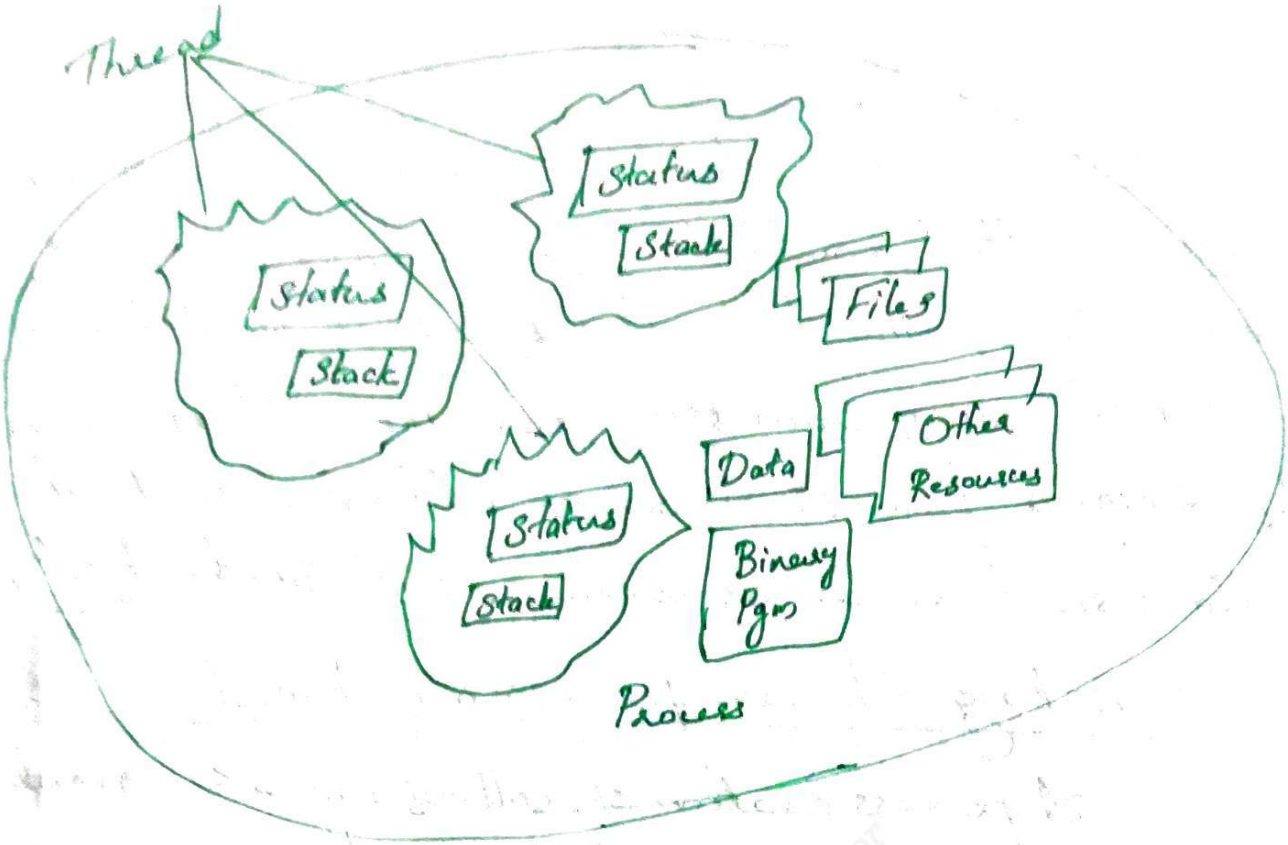
The process is an abstract m/c frame work in which computation can take place and that the active element of ex^n is provided by the ex^n engine. In a single threaded computation, there is only one ex^n engine in the process.



The process component is able to contain multiple exn engines. Each exn engine component is called a thread. ¶

→ Thread data that is private to the thread. This data is usually allocated on a thread specific stack. Each thread may also have its own, private data workspace

→ The thread status, which is an OS data structure that holds all of the properties that are unique to thread. For eg: the status includes the address of the next instruction to be executed by the thread, data about whether or not the thread is blked waiting for a resource, inf^n about which resource it is waiting for if the thread is blocked & so on.



Creating Processes & Threads

Bootstrap loading - loads OS into memory.

Fork(), Join() & Quit()

Fork(): ~~executes on the same~~ results in the creation of a second, "child process. The child process begins executing the same pgm as the original process but at the stmt with the specified label. The process executing fork() continues exn at the next inst?. Once the new process has been created, the original process & the new coexist & proceed concurrently.

Quit(): is used by the process to terminate itself. The command destroys the calling process.

Join (const): is used to logically merge 2 or more processes into a single process.

Process Creation

Address space : set of m/c components ie; mem addresses that could be referenced by the exe engine. Elements of the stack, status, data & pgm blks were referenced using addresses from the address space. By giving a process a set of addresses that it could use to read or write a predefined object,

Creating Modern processes & threads

A process creation s/m call defines a new modern process, The child process will have its own address space pgm, data, files & other resources.

The parent will define how those components are to be configured when it creates the child process.

Objects

An objt was a model of an autonomous entity to represent the op^n of those units in the simulated s/m. A simulation pgm can be thought of as a pgm that manages a large no. of individual units of computation, each of which performs small amt of computation at a time & each of which is closely correlated with sibling units of computation.

A "model of a process" is used to define objts as an alternative unit of computation. Once an object is created, other objts send it msgs. It responds by performing computation on its internal data & by sending other msgs either back to the original sender or to other objts.

An objt's behaviour is defined by its class def", the objt-oriented pgmer designs a s/m by defining a set of classes & describing when objts should be instantiated from the class def"s.

Device Management

— allocation, isolation & sharing of devices according to policies chosen by the designer/s/m administrator.

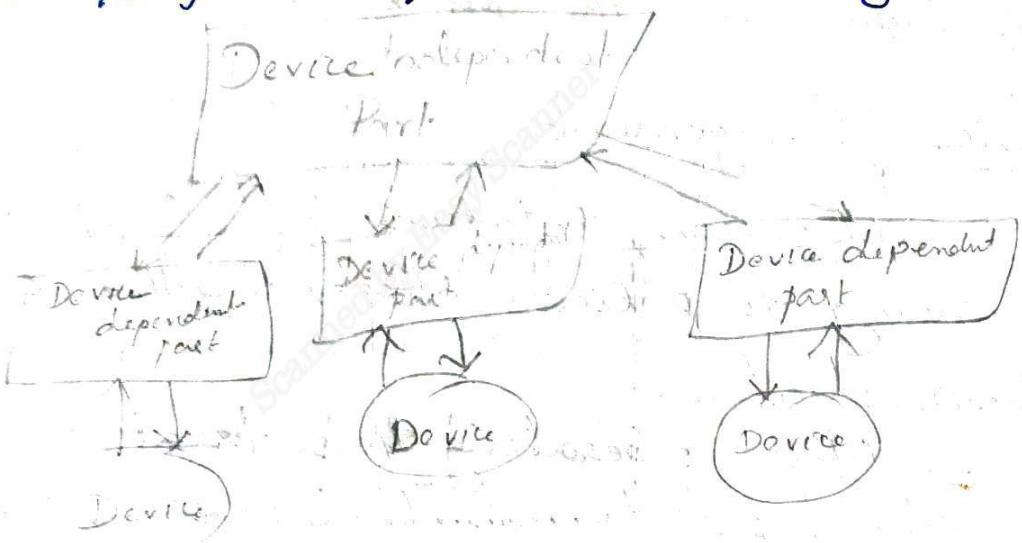
— Most operating s/ms treat all devices such as disks, tapes, terminals & printers in the same general manner, but they provide special mgmt approaches for the processor & memory.

There are device dependent & device-independent parts:

The dependent parts, called device drivers, implement the aspects of device mgmt that are unique to each device type. For eg: a device driver for a keyboard is constructed explicitly to sense

keystrokes from a keyboard device as the corres. keys are pressed. A device mngr for a display screen is constructed to write characters or graphics to a video display.

The device mngr is composed of a device independent part & collection of device dependent parts. The device dependent parts export services to the device independent part, which provides a unified i/f for all diff. device types.



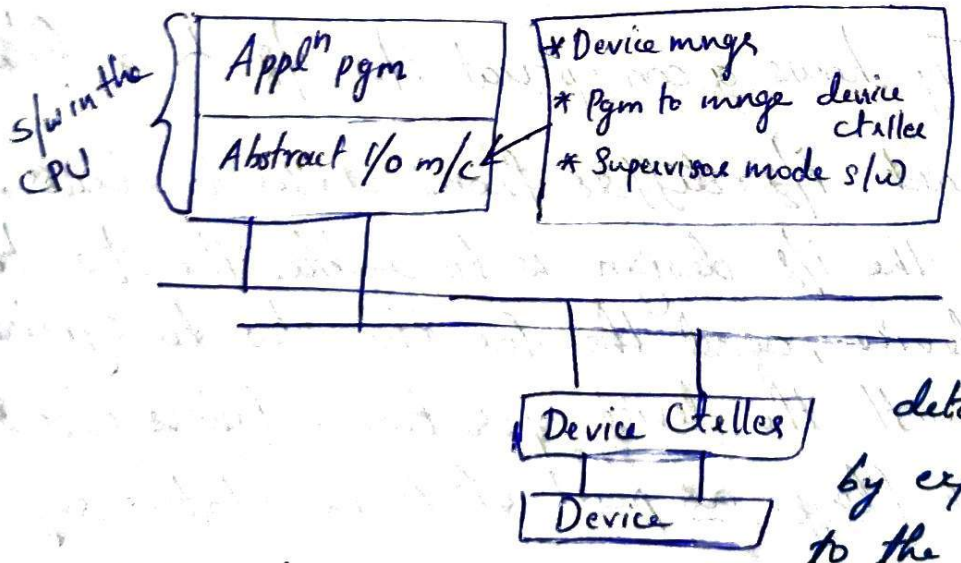
The independent part of the device mngr defines a general s/w envt. in which a device dependent driver can execute. For eg:, the independent part includes the s/m call i/f & a mechanism to forward the calls to the correct device driver. The independent part of the device mgmt s/m is a relatively small part of the device mngr.

I/O Devices

→ I/O devices are attached to the computer bus. An i/p device transfers data from a mechanism such as keyboard, mouse, or microphone into CPU registers. The CPU can then store the data into the primary mem. It fetches inf^n from the primary memory into its registers, & then writes the inf^n over the bus to an o/p device such as a computer screen, speaker or printer.

Each I/O device consists of a ctrller subassembly to ctrl the detailed op^n of the device & the phy. device itself.

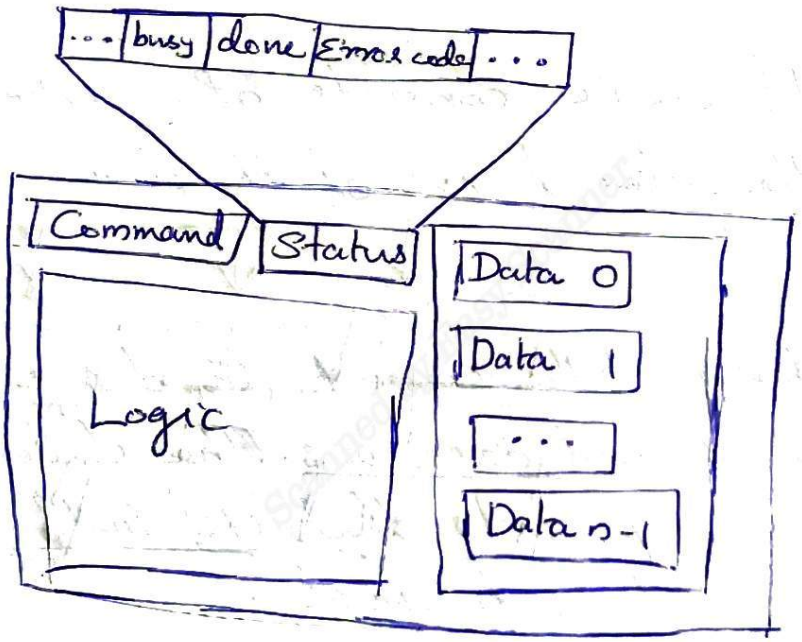
The device ctrller connects the device to the computer's address and data bus. The ctrller provides provides a set of components that CPU inst^{n}s can manipulate to cause the device to operate.



The OS device driver s/w manges the device h/w by interacting with the device's ctrller. Device drivers hide the details of ctrller's mngmt by exporting a common i/f to the appl^n s/w. The device ctrller i/f is a h/w h/w i/f. Its details are device specific.

Device Ctllers

A device may require constant attention a its operation. If s/w were to ctl the device directl it would need to continuously monitor the device's detailed operating status during the operation. The attention may simply to observe status, provide de ends, & correct for minor errors.



busy	done	
0	0	Idle
0	1	finishe
1	0	work in
1	1	(unde

Fig shows a conceptual s/w i/f to a h/w ctller. The exact i/f differs from ctller to ctller. The goal of the i/f design is to enable the s/w to operate the device, and ~~the~~ for the s/w to synchr its behavior with the device operation. Ctllers effect incorporate two flags as part of their status reg. busy & done.

→ If both flags are set to 0 (~~flase~~false), the s/w can place a cmd in the cmd reg. to activate the device. After s/w has put data in one or more of the data regs. for o/p opⁿ, the device is available for use.

→ The presence of a new I/o command causes the ctrller to set the busy flag to TRUE & to begin the opⁿ. An o/p opⁿ causes the data in the data reg. to be written to the device, and an i/p opⁿ causes a read opⁿ to be sent to the device. The process can detect the status of the opⁿ by checking the status reg.

→ When I/o operation has completed, the ctrller clears the busy flag and sets the done flag. On completion of the read opⁿ, data will have been copied into the data reg. from the device. On the completion of a write opⁿ, data will have been copied from the ctrller data reg to the device.

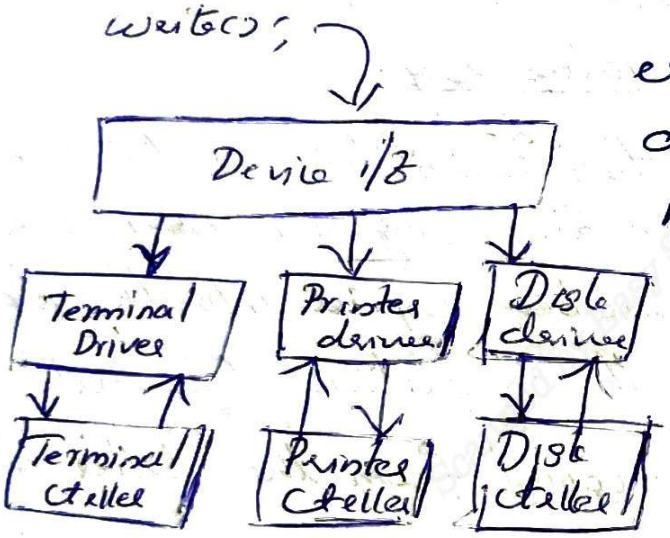
→ If the device flags have both been set to FALSE after a write opⁿ, it is safe to write new data to the ctrller data reg. When the s/w reads the data from the ctrller, the ctrller clears the done reg. to indicate the device is again ready for use.

If the ctrller encountered an unrecoverable error, then when it terminated, it would set the error code field in the status register.

DMA, Mem. mapped I/O.

Device Manager Abstraction

Devices have ctrllers that provide an i/f to the s/w. A device driver is a collection of fns. that abstract the opn of a particular device ctrller. For eg: even though a printer device driver encapsulates the knowledge required to ctrl a printer & a disk driver specializes in disk mgmt, the same general driver i/f invokes their services.

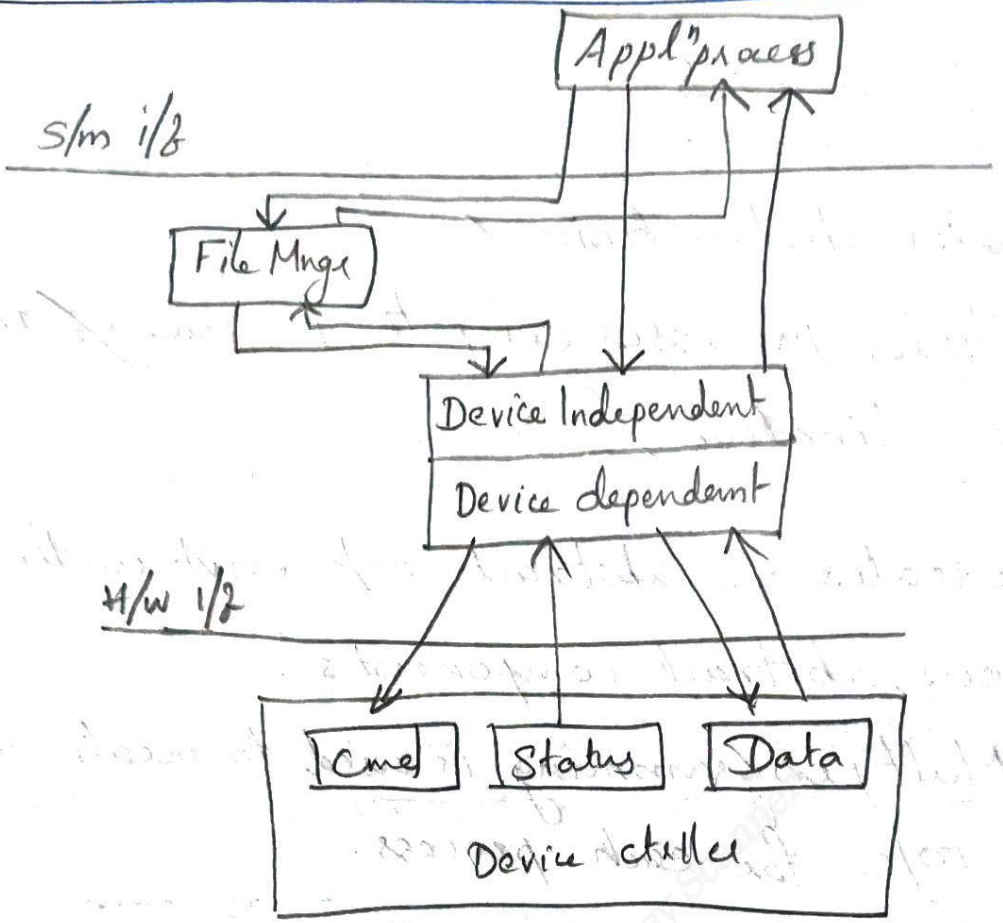


Diff. devices have diff. ctrllers, and diff. ctrllers have diff. i/fs. Each device driver must be written to use the unique device ctrller & to export a common set of fns to be used by appln pgms.

Device Manager

The device mngr infrastructure is the part of the OS that houses the collection of device drivers. This infrastructure makes it possible for the OS to export the common device i/f as s/m calls. Then it routes calls on the generic i/f to specific device driver fns. The device mngr infrastructure & coll^n of device drivers constitute device mngr.

Device Management Organization



The device mngr is constructed of device-dependent and device-independent parts. The infrastructure is the independent part, & the drivers are the dependent parts of the device mngr. The device driver is partitioned into one part that initiates the opⁿ & a matching device interrupt handler to handle the completion of the opⁿ. The infrastructure is the device independent of the device mngr.

Operating Systems Organization

Purpose of an O.S

→ creates abstractions ✓

→ Multiple processes compete for use of processor

→ Coordination.

- creates an abstract m/c envt with multiple autonomous, abstract components.

- Multiprogramming is used to create an abstract m/c for each process.

— Coordinate the use of the components acc. to the policies of the m/c's administrator.

- scheduler decides when & which processes get to use the processor.

Basic Requirements an OS must satisfy:

→ Device Mgmt ✓

→ Process, Thread & Resource Mgmt

→ Mem. mgmt

→ File Mgmt.

Device Mgmt: OS mngs the allocation, isolation, sharing of devices acc. to policies chosen by the designer or s/m admin.

Resource Manager. ✓

OS Implementation Considerations & Mechanisms.

Considerations:

→ Performance:

OS provides simplified pgming i/fs. & manages resource allocation & sharing.

Efficiency vs. Convenience

- How much do these abstractions slow down the ex^n of a pgm.
- What is the performance cost of using files rather than cmds to directly manipulate storage device.
- How much contribution to fnality of the s/m is desired versus the impact to performance.

→ Exclusive use of Resources

Multipgmed s/ms support multiple processes & threads that need to share resources.

Design principles:

- do not to interfere with other processes.
- Cannot use a resource unless it is specifically authorized by

- Allow both exclusive ctl to a process/shared among a set of processes.

- Protection Mechanisms & Security Policies
  * Defines strategies for managing resources.
    eg: File protection.
  * Trusted versus untrusted s/w & access barriers.

Processor Modes

The key h/w element for implementation of trusted s/w

* Mode Bit (Supervisor or user)
* Supervisor (privileged, protected)
  processor can execute every inst^n
* User
  processor can execute only a subset of all inst^{n}s.

I/O Processor Overlap within an Application.

Application programmers expect serial exn semantics ~~of I/O~~ for an individual thread.

```
...
read(dev_1, "%d", x);
y=f(x);
...
```

```
.....
startRead(dev-1, "%d", x);
.....
while(stillreading());
y=f(x);
.....
```

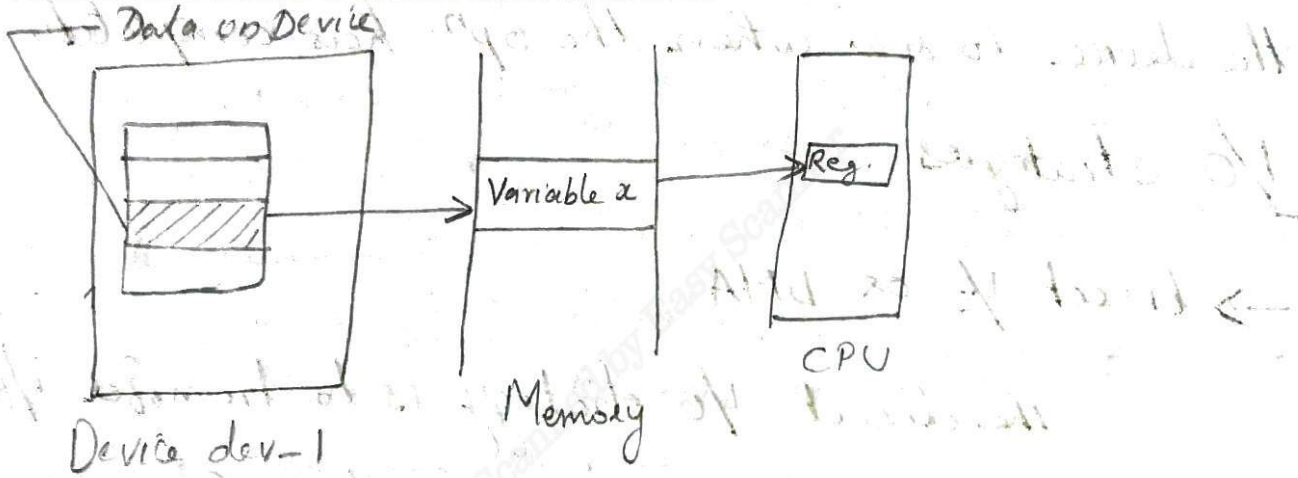


Fig shows the situation while the read() s/m call is being executed. The read() fn in the device driver has started the dev-1 device, but the opn has not completed. If a process were to execute the assignment statement, y=f(x), at that moment, then the f(x) function would be executed using an old value of x, not the new one being read from the device. To prevent this situation, the OS blocks the process until it completes the read() call.

I/O Processor Overlap Across Threads

There are two pre-requisites for a thread to be able to overlap its CPU & I/O operation.

→ The computation performed by the thread must be arranged so that while the I/O op^n is taking place, the thread has other work to do.

→ The pgmming lang. & OS must provide tools to allow the thread to start an I/O op^n, & then to poll the device to see when the op^n has completed.

I/O Strategies

→ direct I/O or DMA

The direct I/O strategy is to transfer i/p data from the device to a CPU reg. & then from the CPU reg. to a primary mem. location. Illy the CPU copies o/p from the primary mem. cell into a reg. & then from the reg. to the device.

DMA I/O : where the data flows directly b/w the primary mem. & device device.

2 dimensions define 4 diff. I/O strategies:

- → Direct I/O with polling
- → DMA I/O with polling
- → Direct I/O with interrupts
- → DMA I/O with interrupts

Scanned by Easy Scanner

Direct I/O with Polling

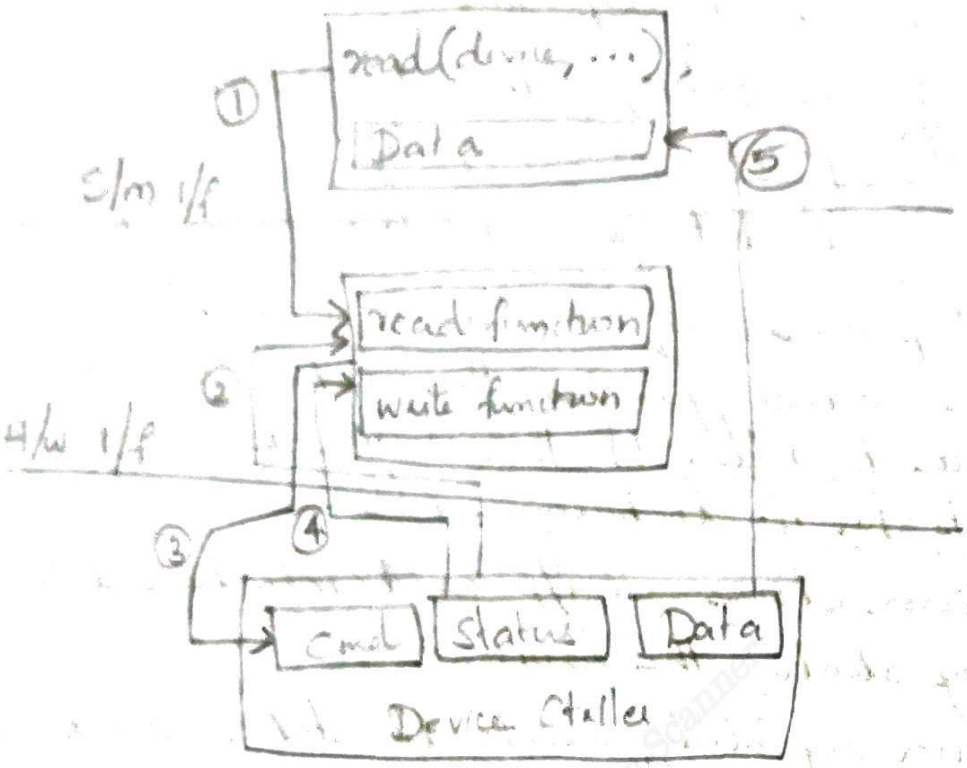
Direct I/O refers to the method of accomplishing I/O whereby the CPU is responsible for determining when the I/O op^n has completed, & then transferring the data b/w the m/c's primary mem. & device stellar data reg. The steps required to accomplish a read op^n.

1. The appl^n process requests a read op^n. ✓
2. The device driver queries the status reg. to determine if the device is idle. If the device is busy, the driver waits for it to become idle.
3. The driver stores an i/p cmd into the stellar's cmd reg. thereby starting the device.
4. The driver repeatedly reads the status reg while waiting for the device to complete its op^n.
5. The driver copies the contents of the stellar's data reg.s into the user process's space.

The steps to perform an o/p op^n are as follows.

1. The application process requests a write op^n.
2. The device driver queries the status reg. to determine if the device is idle. If the device is busy, the driver waits for it to become idle.
3. The driver copies data from user space mem. to the stellar's data register.
4. The driver stores an o/p cmd into the cmd reg. thereby starting the device.
5. The driver repeatedly reads the status reg while waiting for the device to complete its op^n.

Polling I/O Read Operation



Interrupt Driven I/O

The motivation for incorporating interrupts into the comp. h/w is to eliminate the need for the device driver s/w to constantly poll the controller status register. Instead, the device ctrller "automatically" notifies the device mngr when the op" has completed.

- Device mngmt functionality is partitioned into 4 diff. parts:
  - * "Top half" of the driver that initiates the op"
  - * Device status table
  - * Interrupt handler
  - * Device Handler.

The follo. are the steps for performing an i/p instⁿ in a s/m by using interrupts.

1. The applⁿ process requests a read opⁿ.

2. The top half of the device driver queries the status reg. to determine if the device is idle. If the device is busy, the driver waits for the device to become idle.

3. The driver stores an i/p cmd into the cntlr's cmd reg., thereby starting the device.

4. When the top half of the device driver completes its work, it saves infⁿ regarding the opⁿ that it began in the device status table. This table contains an entry for each device in the s/m. The top half of the driver writes into the entry for the device it is using infⁿ such as the return address of the original call & special parametres for the I/O opⁿ. The CPU then can be used by another pgm, so the device mngr invokes the scheduler part of the process mngr. It then terminates.

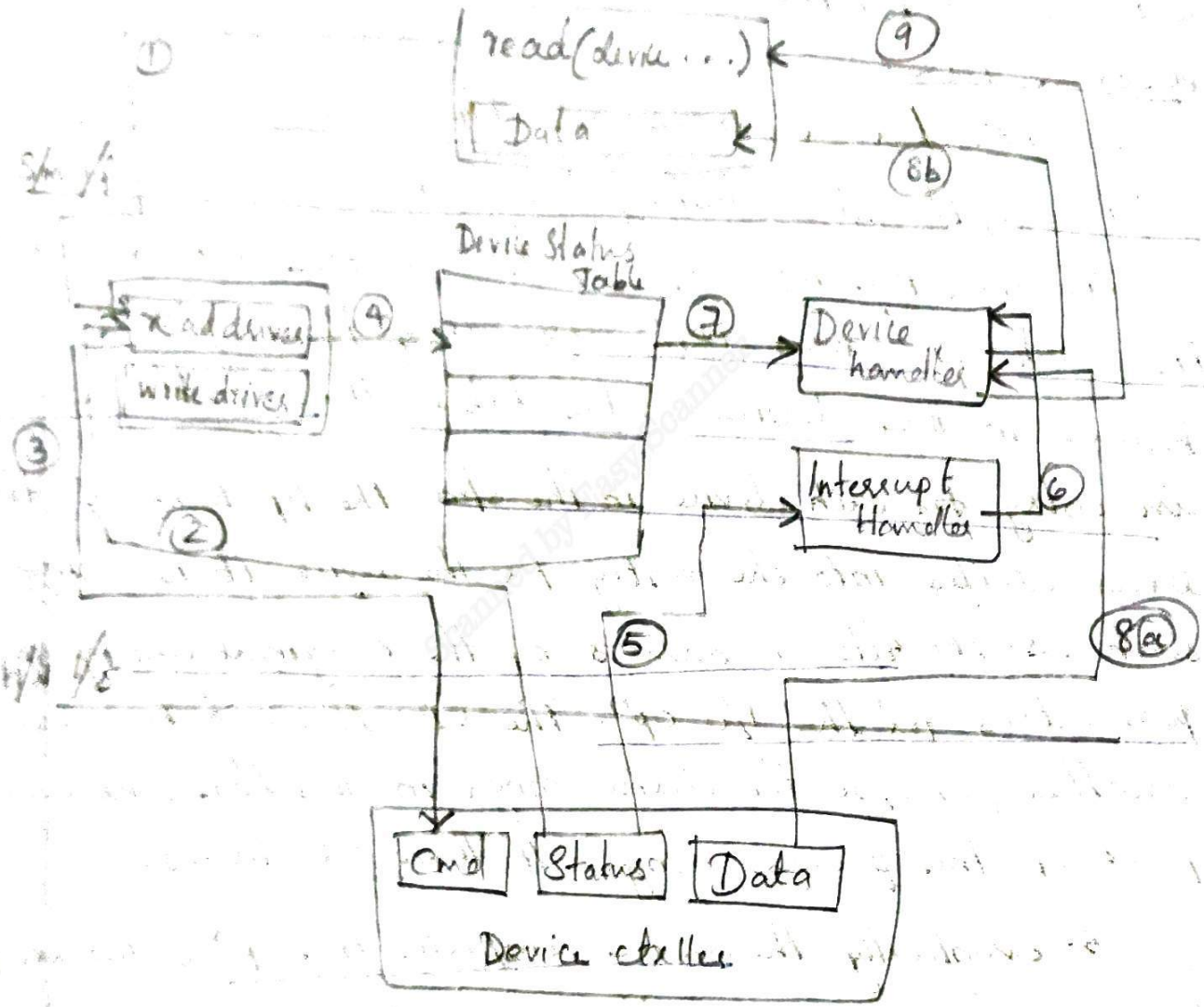
5. Eventually the device completes the opⁿ & interrupts the CPU, thereby causing the interrupt handler to run.

6. The interrupt handler determines which device caused the interrupt. It then branches to the device handler for that device.

8. The device handler copies the cont.s of the cntlr's data regs. into the user process's space.

7. The device handler retrieves the pending I/O status info from the device status table.

$ 9. The device handler - behaving as "bottom half" of the device drivers invoked by the appln process - returns ctl to the appln process.



Polling Versus Interrupt-Driven I/O Performance

The time to execute a thread can be broken down into:

$time_{compute}$: The time spent on computation.

$time_{device}$: The time spent on I/O operations

$time_{overhead}$: The time the process spends determining when each I/O op^n is complete.

Total time to execute the computation is

$$time_{total} = time_{compute} + time_{device} + time_{overhead}.$$

If an I/O device mngr uses (polling) to determine when the op^n is complete, we have

$time_{overhead} = time_{polling}$ = the accumulated amount of time after a device completes an op^n but before the polling loop has determined that the completion has occurred.

In a s/m with (interrupts), $time_{overhead}$ is calculated as follows

$$time_{overhead} = time_{handler} + time_{ready}$$

where $time_{handler}$ is the accumulated time required to execute the interrupt handler & device handler & $time_{ready}$ is the accumulated time the process waits to use the CPU after it has completed its I/O but another thread is using the CPU

Polling is normally superior from the viewpt. of the individual process, since normally

$$time_{polling} < time_{handler} + time_{ready}$$

Suppose 3 threads are to be executed on a s/ms. Thread1 requires time total, to complete, thread2 requires $time_{total_2}$ to complete, Thread3 — $time_{total_3}$.

In a polling s/ms, thread1 might run to completion before thread2 is started & thread2 would run to completion before thread3 is started.

$$time_{total\text{-}p} = time_{total_1} + time_{total_2} + time_{total_3}$$

In a s/m with interrupts, multiprogramming can make good use of the CPU by threads 2 & 3 when thread1 is conducting I/O.

$$time_{device_1} \leq time_{compute_2}$$

$$time_{device_2} \leq time_{compute_3}$$

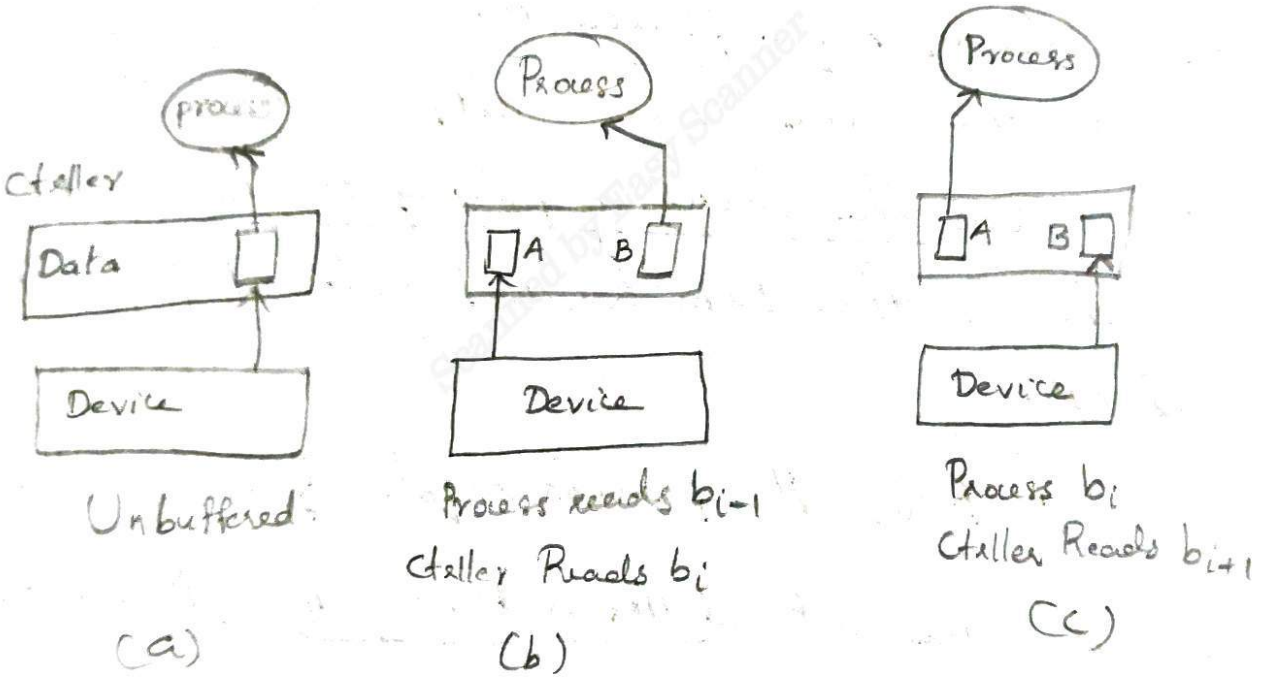
$$time_{device_3} \leq time_{compute_1}$$ meaning the total time to execute the three threads in a s/ms with interrupts

$$time_{total\text{-}i} = time_{compute_1} + time_{compute_2} + time_{compute_3} + time_{overhead}.$$

> Buffering

Buffering is employed by a device mngr to keep I/O devices busy during times when a process is not requiring I/O op's thus overlapping the device and CPU operation. Input buffering is the technique of having the i/p device copy inf^n into the primary memory before the process requests it.

Output buffering is the technique of saving inf^n in memory & then writing it to the device while the process continues execution.



Consider a simple character device ctaller that reads a single byte from a modem for each i/p op^n. The normal mode of op^n for the ctaller is for one byte data register to contain the last character read following the driver's read op^n. When an appl^n pgm next calls the device driver read fn, the driver passes a cmd to the ctaller. The ctaller, in turn

instructs the device to i/p the next byte & place it in the data reg. The process calling for the byte waits for the I/O opⁿ to complete & then retrieves the char. from the data reg.

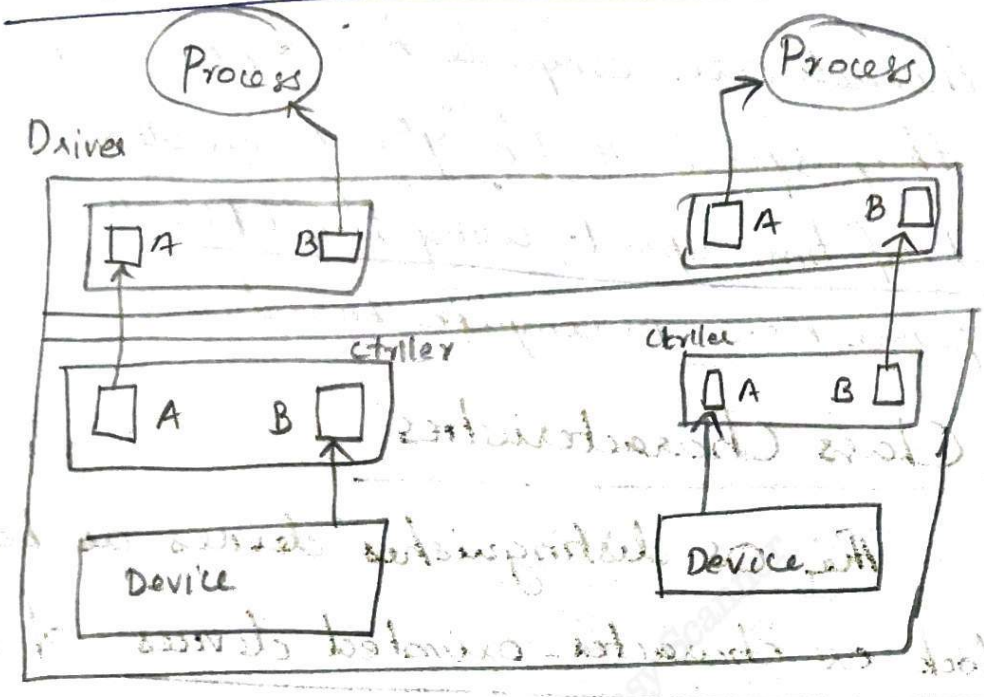
In fig b & c, a h/w buffer has been added to the steller. This can substantially decrease the amt of time the process has to wait for a char. if the steller reads it ahead of time.

In fig (b), the next char. to be read by the process has already been placed into data reg. B by the steller. The device is currently reading the next char from the device & placing it into data reg. A, even though the pgm has not yet called for the 2nd read opⁿ.

In fig(c) the process reqsts the char from the second read into steller data reg. A, so the device is started on a read opⁿ to fill buffer B. The device read for characters i+1 will be overlapped with the CPU exⁿ that uses character i.

The h/w buffering technique can also be applied at the steller-driver level. This is called double buffering, since there are 2 s/w buffers.

One buffer is for the driver or caller to store data while waiting for it to be retrieved by higher levels of hierarchy. The other buffer is to store data from the lower level module (the caller).



The no. of buffers is extended from 2 to n.

The data producer (the device caller in read op's, the CPU in write op"s) is writing into buffer i while the data "consumer" (the caller in write op"s, the CPU in read op"s) is reading from buffer j.

Performance & Buffers

→ The first prerequisite is that the device driver must know enough about the way inf" is read from the device that it can predict what data will be read in near future.

→ The second prerequisite is that the thread behavior be conducive to buffering.

Some processes are I/O bound, meaning that their overall time to execute is dominated by the time to perform cumulative I/O op^{n}s. A thread that copies one file to another is an eg. of an I/O bound thread.

Other threads are compute bound, meaning that the time they spend in I/O op^{n}s is small compared to the amt of time spent using the CPU. A thread computes prime no.s is compute bound.

Device Class Characteristics

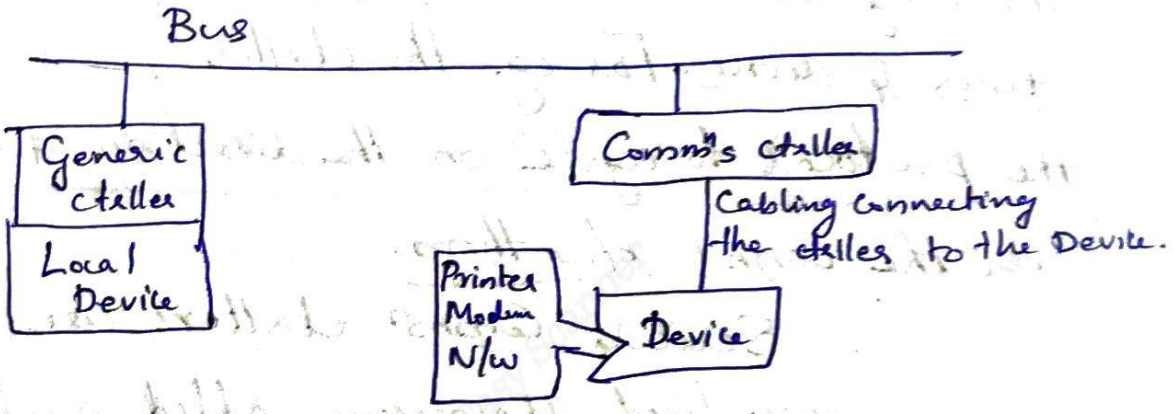
The OS distinguishes devices as being either block or character-oriented devices. A single I/O op^n on a character-oriented device reads or writes one byte.

A block oriented device reads or writes a fixed no. of bytes in one op^n. Many of the char. oriented devices use serial or parallel comm^n ports. These devices are often connected to the device ctllers with a cable.

Storage devices are usually blk-oriented devices, so they are usually integrated with the ctrllr as a single component.

Communication Devices

- transmit inf^n b/w a computer & a device using a medium such as public broadcast, coaxial cable etc. While n/ws dominate in large-scale comp.-comp. installations, simple comm^n devices are still heavily used for connecting terminals, printers, scanners, modems etc.



Communication devices are character devices that transfer bytes of inf^n b/w a comp & a remote device. To use the device, the driver manipulates the comm's ctaller, which in turn manipulates the device using a ctaller-device protocol. The ctaller & device must agree on the i/f as well as a protocol for using the i/f. Thus to connect a modem to a computer using serial comm's port the computer must include a serial comm's ctaller that can be attached to the modem using an external cable. Once the comm^n is made, the ctaller & the modem must exchange electronic s/ls acc. to a common protocol.

Communication devices often have a ctrller that can be independent of the device. This requires that there be a ctrller_device protocol in addition to the driver_controller protocol. The RS-232 is an eg: of a ctrller_device protocol that specifies the connections b/w the ctrller and device as well as the way that electronic s/ls exchanged b/w the ctrller & device. For eg: the ctrller & device use the protocol to agree on the direction of data flow over the conn^n b/w them.

Serial Comm^n ctrllers are implemented using a specialised µprocessor, called a Universal Asynch. Receiver Transmitter (UART). which includes onboard ROM & RAM. — physical i/f through which inf^n transfers in or out one bit at a time.

Parallel Communication ports are used to connect a printer to a computer, thus the devices can be char. streams, bitmaps or other comp-printer protocols. — several bits at a time.

The Universal Serial Bus (USB)

⇒ external port that would act like a bidirectional 1.5 to 12 Mbps.

The USB computer port can be directly connected directly to a device. USB also provides power to the device via the USB port.

The idea is to provide ext. sockets on a comp. that support the of USB protocol. The

Firewire: alternative to USB

- one or more external connectors to allow users to connect a digital cameras or other devices directly on to the serial bus.
- The high speed possible with firewire make it better than USB for streaming media appl^{n}s.

Sequentially Accessed Storage Devices

Storage devices implement persistent storage in the comp. meaning that inf^{n} placed in a storage is preserved after the comp. is turned off.

Seq. storage devices physically store the blks on a recording medium in a linear sequence.

eg: Magnetic Tape

→ is the sequential storage medium. Tapes may be any length, although common length range from 600 to 2400 feet. A collection of bytes is packed densely on the tape to form a phy. record containing one blk of data. Physical records are separated by an inter-record gap. When the driver issues an I/O op^{n}, the tape drive must accelerate the tape across the read/write head so that it can read/write the tracks.

The inf^{n} stored on a tape is physically accessed by moving the tape forward & backward

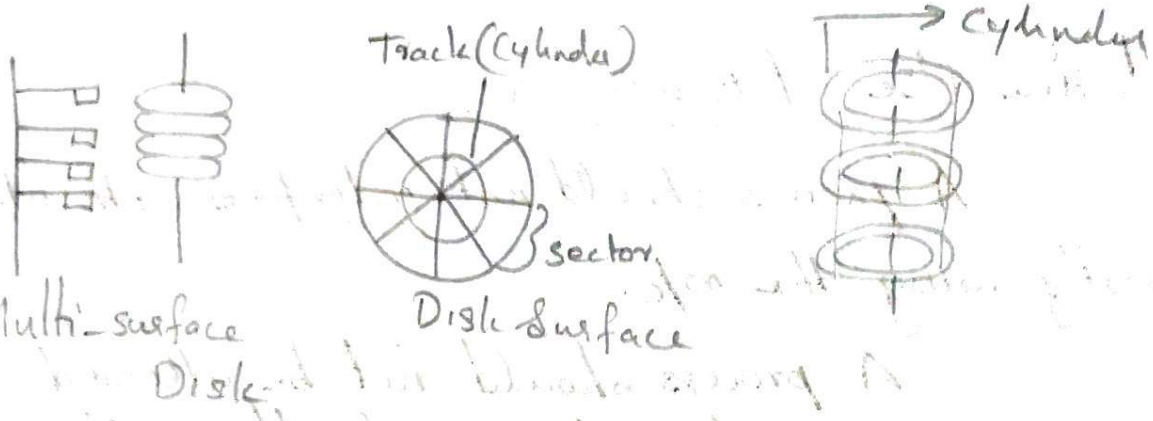
across the read/write head. Thus data can be accessed only at reasonable rates if the process reading the tape intends to read all of inf^n in order it is stored on the tape.

Randomly Accessed Storage Devices

—allow a driver to access the blks on the device in an arbitrary order. Rotating disks are the most prevalent form of randomly accessed storage device. These disks employ a block read/write i/f b/w the ctller & the device. The blk orientation is propagated to the driver/ctller i/f. Disks do not ordinarily directly support sequential access, since contiguous blks on the device do not necessarily hold logically contiguous blks of inf^n.

Eg: Magnetic Disks

—are organized as one or more physical disk platters, each having one or two storage surfaces. Each surface is logically devided into several sectors, which are defined as angular portions of the disk circle, like slices of a pie. The surface of each disk is organized into concentric rings, each called a track that passes through each sector.



When the head has been positioned over the appropriate track, the device generally has to wait for the proper sector to rotate under the head before it can access the inf^n. This rotational delay is called disk latency time.

Seek time : the time taken to access the blk of inf^n.

General Implementation Considerations

The OS is a collection of algorithms & data structures. There are two recurring issues in the s/w design:

* Performance:

The OS must be implemented so that it is as efficient as possible in its use of m/c resources, maximizing the availability of resources for the use by applications.

— rapid inf^n processing than manual techniques.

— O.S provide simplified pgmming i/fs and a mechanism to manage resource sharing.

— Every design issue in an OS must be evaluated w.r. to its contribution to the fnality of the s/m & its impact on the computer's performance.

Exclusive Use of Resources

A process should not ~~interfere with others~~ currently using the s/c.

A process should not be allowed to use a resource unless it is explicitly authorised to do so. A process must be able to determine that it has exclusive ctrl of a resource before accessing it

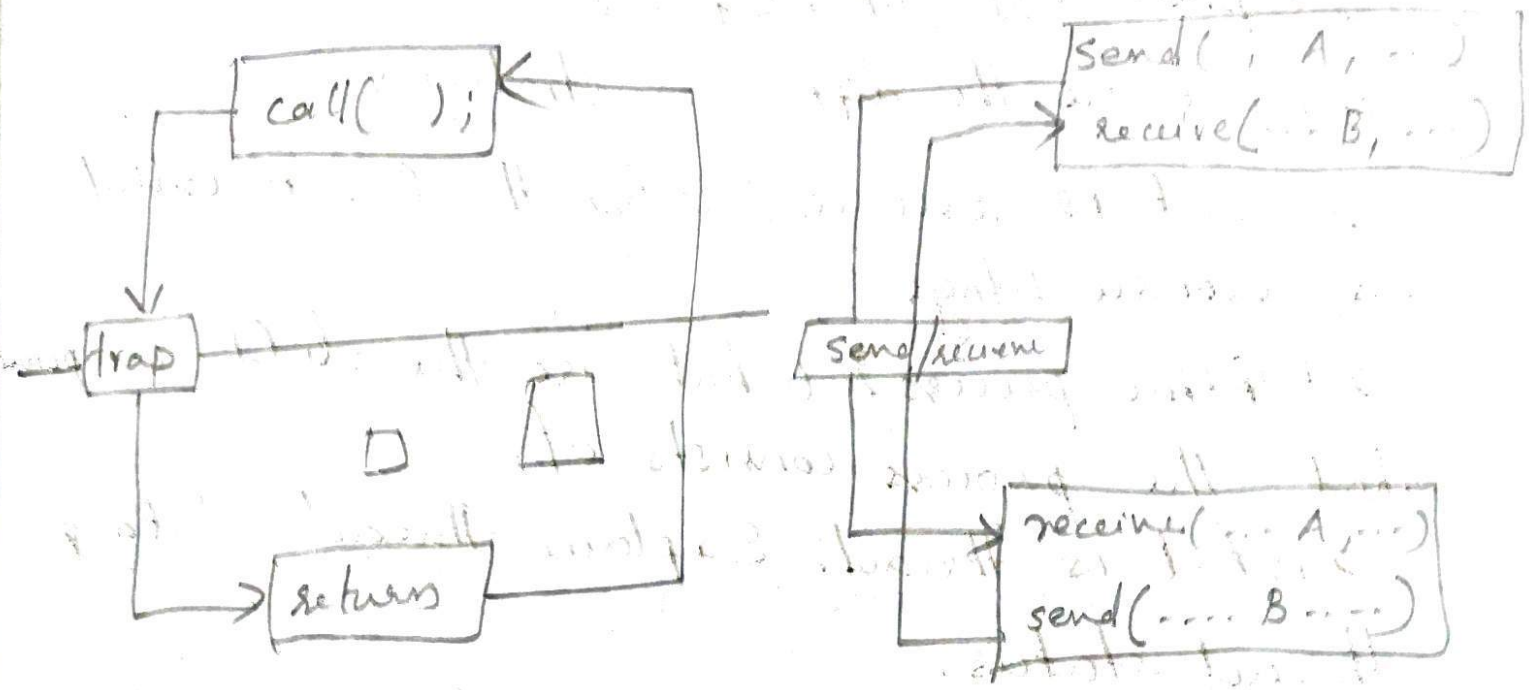
Protection mechanisms are tools that the OS provides to implement security policy chosen by the s/m's administrator.

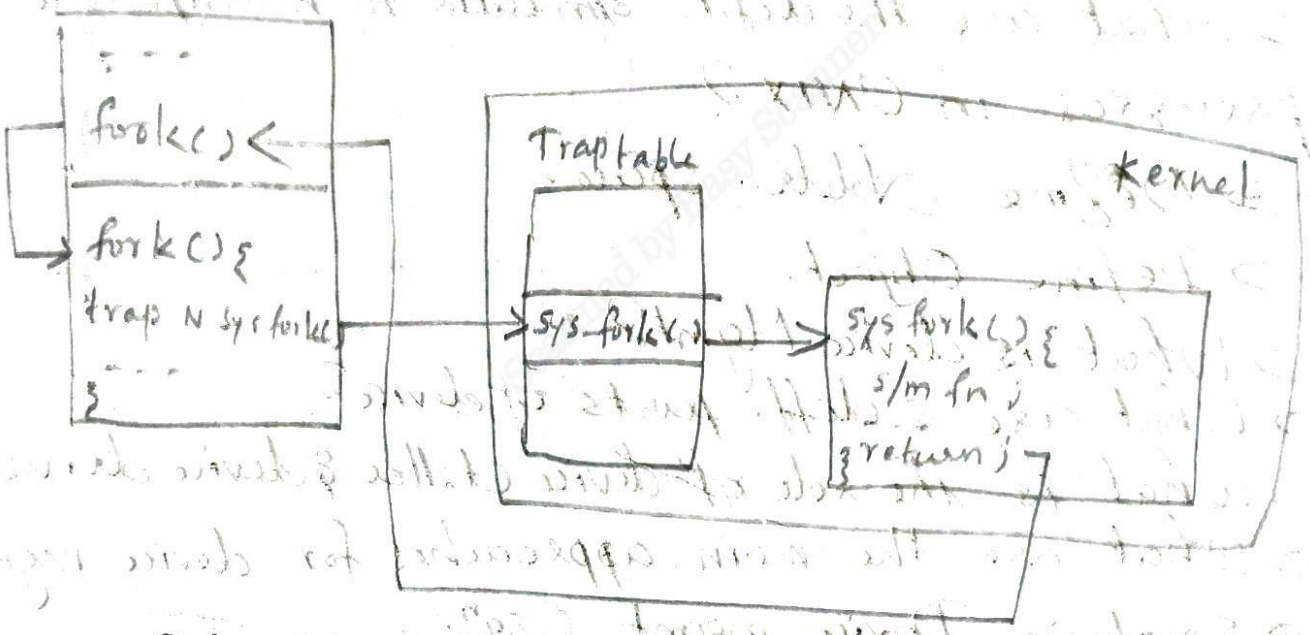
A security policy defines the s/c-specific strategy for managing access to resources. For eg:, a policy may specify that only one process at a time can have file open for writing, but that many processes open the file simultaneously for reading.

Trusted s/w: has been carefully written & debugged at the time OS is created & updated.

Untrusted s/w: has not been subjected to such careful analysis, & the overall opⁿ of the s/c should & not depend on the correct opⁿ of untrusted s/w.

Procedure Call & Msg Passing





fork() is a stub fn. When the appl^n pgm calls the stub, all the stub does is execute the trap inst^n with the correct operand, switching the processor to supervisor mode, & then branching indirectly through the trap table to the entry pt. of the fn that is to be invoked. When

Questions

→ Explain diff. types Of O.S.
→ What are the diff O.S strategies.
→ What is resource? Why the O.S is called as Resource Mngs.?
→ Define process? What are the diff. components that the process consists of.
→ What is thread? Explain thread data & thread status.
→ What are the diff. s/m calls to manipulate processes in UNIX?
→ Define Addr. space.
→ Define Object.
→ What is device Mgmt?
→ What are 2 diff. parts of device?
→ What is the role of device Cteller & device drivers?
→ What are the main approaches for device mgmt?
→ Explain Device mgmt Org".
→ Explain O.S Organization.
→ What are the O.S implementation consideration.
→ What are the diff. processor modes.?
→ What are the diff. I/O strategies.
→ Explain direct I/O Polling
→ Explain DMA
→ Explain Interrupt driven I/O
→ What is buffering? Explain Performance & buffering
→ Explain Device Class characteristics?

Module II

PROCESS MANAGEMENT

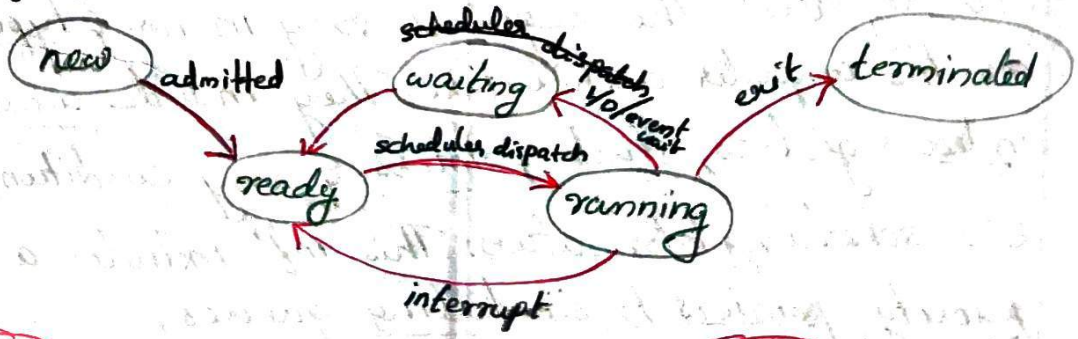
The Process:

A process is a programs in execution. It also includes the current activity, as represented by the value of the pgm counter & the conts. of the processor's registers.

A process generally includes the process stack which contains temporary data (such as fn parameters, returns addresses, & local variables) & a data section, which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during run time.

Process State Diagram

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.



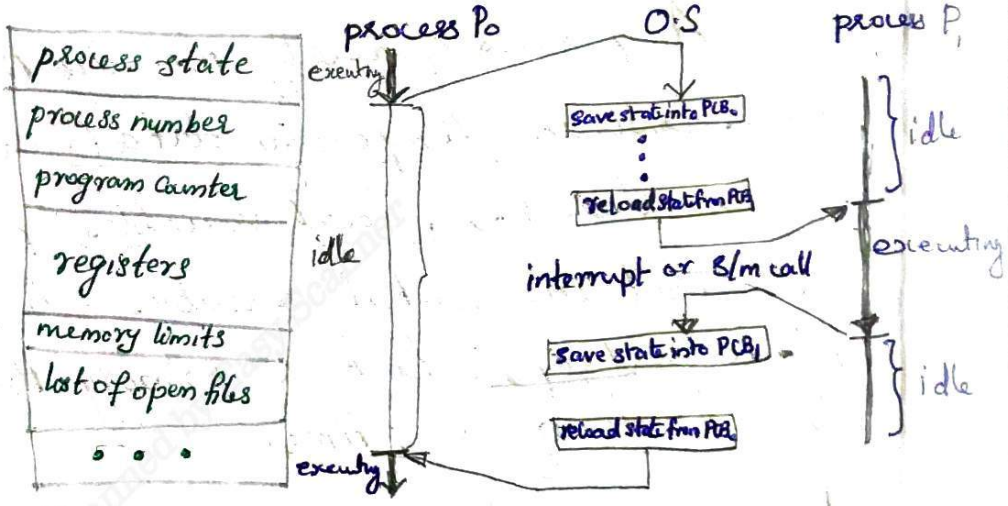
New: The process is being created. Ready: Process is waiting to be assigned to a processor.
Running: Inst"s are being executed

Waiting: The process is waiting for some event to occur (I/O or reception of a s/l).

Terminated: The process has finished execution.

Process Control Block

Each process is represented in the O.S by a process control block (PCB) also called a task control block



Process state: The state may be new, ready, running, waiting, halted & so on.

Program Counter: The counter indicates the address of the next instruction to be executed for this process.

CPU Registers: The registers vary in no. & type, depending on the computer architecture. They include accumulators, index registers, stack pointers & any condition-code inf".

CPU-scheduling Information: This inf" includes a process priority, pointers to scheduling queues, & any other scheduling parameters.

Memory-Mgmt Inf^n: This inf^n may include such inf^n as the value of the base & limit registers, the page tables or segment tables depending on the mem. s/m used by the O.S.

Accounting Inf^n: This inf^n includes the amnt of CPU & real time used, time limits, acc. numbers, job or process numbers.

I/O status inf^n: This inf^n includes the list of I/O devices allocated to the process, a list of open files, & so on.

Process & Resource Abstraction

Process Abstraction:

The abstraction provided by the OS of a running program is something we will call a process. A process can be summarised by taking an inventory of the diff. pieces of the s/m. To understand what constitutes a process, we thus have to understand its m/c state:- what a pgm can read/update when it is running.

One obvious component of m/c state that comprises a process is its memory. Instructions lie in memory; the data that running pgm reads & writes in memory. Thus the mem. that the process can address is part of the process.

There are some particular regs. in m/c state. Pgm Counter/Inst^n Pointer tells us which inst^n of the pgm is currently being executed

xcopy

Stack pointer & associated with frame pointer are used to manage the stack for fn parameters, local variables & returns addresses.

Resource Abstraction:

H/w abstraction are sets of routines in s/w that emulate some platform-specific details, giving pgms direct access to the h/w resources.

The often allow pgmmers to write device independ high performance appl'n's by providing standard Op. Systems (OS) calls to h/w.

Each type of CPU has a specific inst'n set Archi (ISA). The ISA represents the primitive op's of the m/c that are available for use by assembly pgmmers & compiler writers.

One of the main fns of a compiler is to allow a pgmmer to write an pgm in a high-level lang without having to care about CPU specific inst's. Then it is the job of the compiler to generate a CPU specific executable. The same type of abstration is made in OS, but OS APIs now represent the primitive op's of the m/c rather than an ISA. This allows a pgmmer to use OS-level op's in their pgms while still remaining portable over a variety of diff platforms.

Many early computer s/ms do not have any form of h/w abstraction. This means that anyone writing a pgm for such a s/ms must know how each h/w device communicates. This becomes a significant challenge to s/w developers, who must know how every device available works to ensure its compatible. With h/w abstraction, the pgm instead tells the operating s/m what the device should do & the O.S gives the instⁿ. This means that pgmmers don't need to know how individual devices work, & their pgms are compatible with any device.

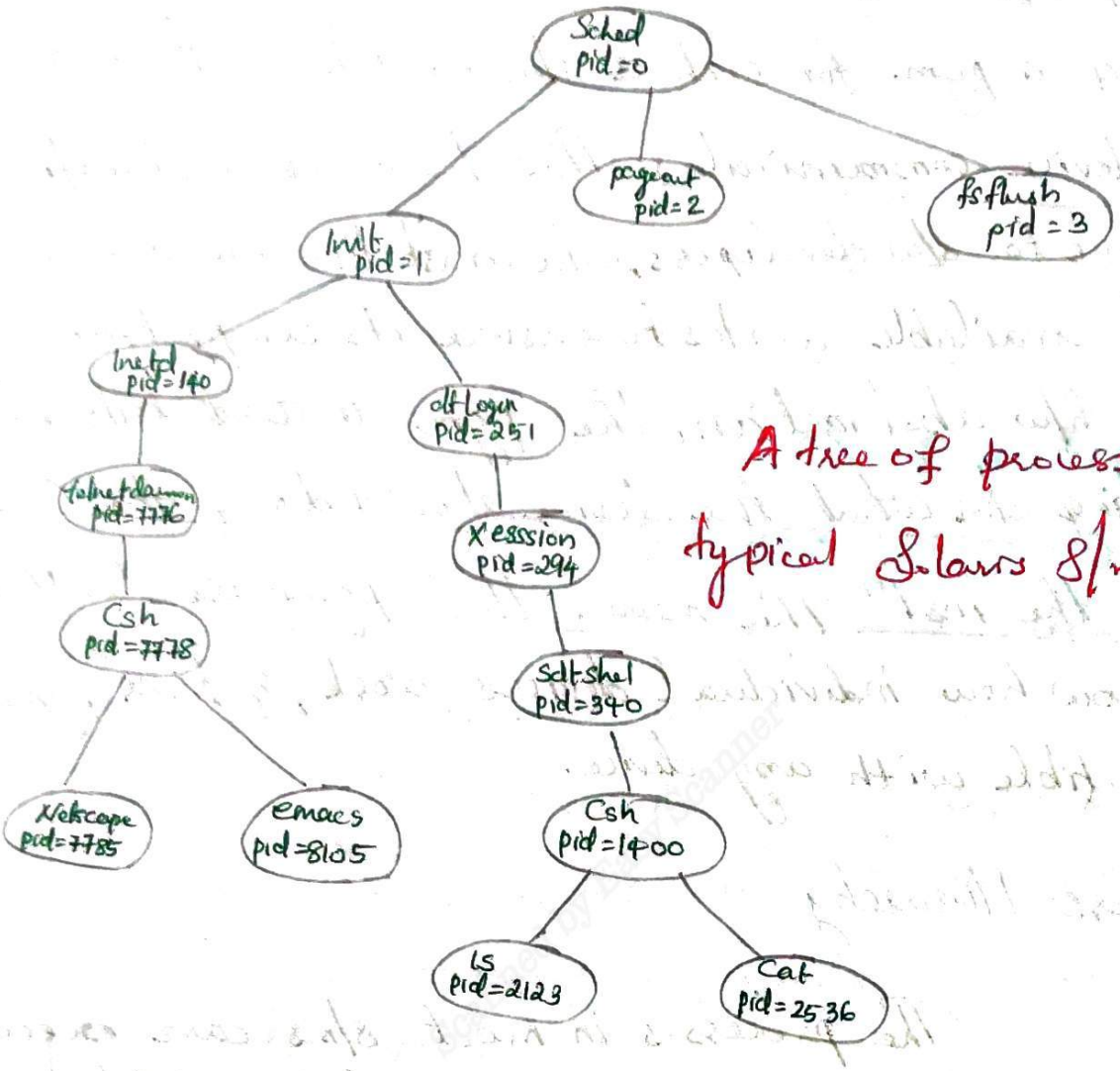
Process Hierarchy

The processes in most s/ms can execute concurrently, & they may be created & deleted dynamically. These s/ms must provide a mechanism for process creation & termination.

Process Creation:

A process may create several new processes, via create process s/m call, during the course of exⁿ. The creating process is called a parent process, & the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.

process identifier (pid): to identify processes.



A tree of processes on a typical Solaris 8/ms.

In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from O.S or it may be constrained to a subset of resources of the parent process. The parent may have to partition its resources among its children or it may be able to share some resources among several of its children.

When a process creates a new process, 2 possibilities exist in terms of exn.

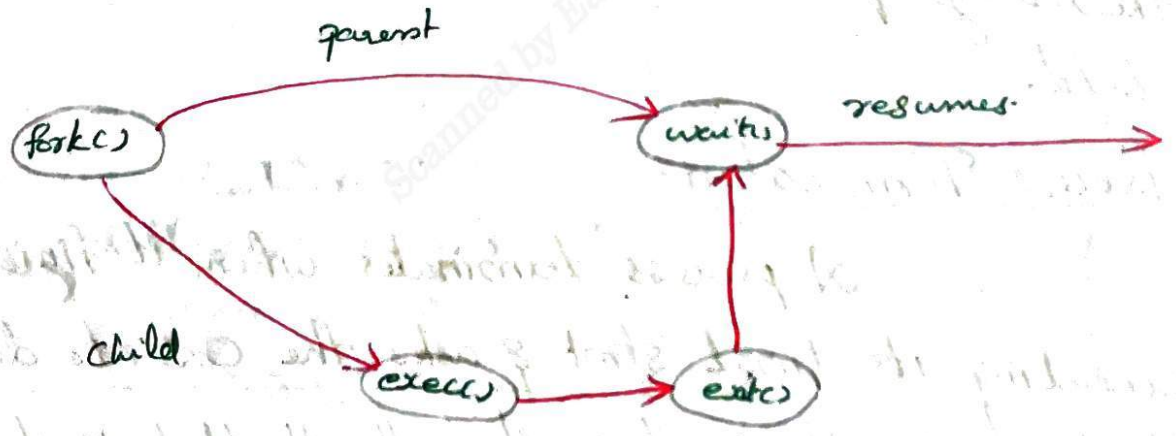
1. The parent continues to execute concurrently with its childrens.

2. The parent waits until some or all of its children have terminated.

There are also 2 possibilities in terms of address space of the new process.

1. The child process is a duplicate of the parent process (it has same pgm & data as parent)

2. The child process has a new pgm loaded into it



Each process is identified by its process identifier which is a unique integer. A new process is created by the fork() s/m call. The new process consists of a copy of the addr. space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes continue exn at the instn after the fork(), with one diff.

The return code for the fork() is zero for child process, where as non zero identifier of the child is returned to the parent.

exec() s/m call is used after a fork() s/m call by one of the two processes to replace the process's memory space with a new pgm. The exec() s/m call loads a binary file into memory & starts its exn. The parent can then create more children or if it has nothing else to do while the child runs it can issue a wait() s/m call to move itself off the ready queue until the termination of the child.

Process Termination

A process terminates when it finishes executing its final stmt & asks the O.S to delete it by using the exit() s/m call. At that point, the process may return a status value to its parent process (via wait() s/m call). All the resources of the process - including physical and virtual memory, open files, & I/O buffers - are deallocated by the O.S.

A process can cause the termination of another process via an appropriate s/m call eg: Terminate process(). Such s/m call can be invoked only

by the parent of the process that is to be terminated. Otherwise users could arbitrarily kill each other's jobs. A parent needs to know the identities of its children. When one process creates a new process, the identity of the newly created process is pass to the parent.

A parent may terminate the exn of its children for a variety of reasons.

→ The child has exceeded its usage of some of the resources that it has been allocated.

→ The task assigned to the child is no longer required.

→ The parent is exiting and the O.S doesn't allow a child to continue if its parent terminates.

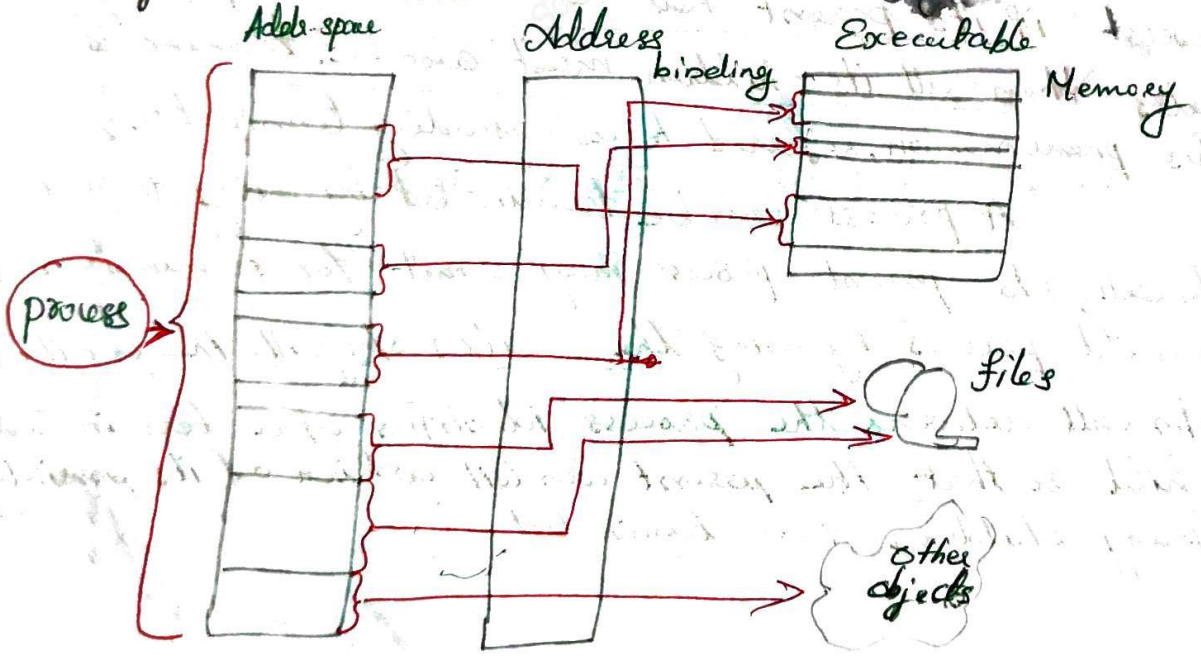
Some s/ms do not allow a child to exist if its parent has terminated. If a process terminates then all its children must also be terminated. This phenomenon, referred to as cascading termination;

A process can be terminated by using the exit() s/m call; its parent process may wait for termination of a child process by using the wait() s/m call. The wait() s/m call returns the process identifier of a terminated child so that the parent can tell which of its possibly many children has terminated.

The Process Adress Space

The process address space is the coll^n of adelresses that a thread can reference. These addresses refer to an executable memory location. they can also associated with other abstract m/c elmts.

For eg: some O.S allow a pgmmer to read & write the conts. of a file using addresses from the process address space. This is accomplished by opening the file, then binding each byte in the file to an addr. in the addr. space. Memory mapped resources can also have their i/p bound to a coll^n of addresses in the addr. space, there by allowing the components of the resource to be referenced using adelresse.



→ The process address space is a large collection of contiguous addresses usually of bytes. Any resource that can be referenced as a collection of bytes (exe mem) can have each of its byte-addressable elmts bound to an address in the address space. This allows a process/thread to access inf^n by loading/storing inf^n from/to byte address.

The address space provides a uniform mechanism by which ~~provides~~ process can reference bytes in all memory-mapped resources. If all resources could be memory mapped, then a process could interact with all resources using the address space.

Each resource mngr is responsible for binding addresses with addressable elmts of the resource. For eg: the memory resource mngr associates blocks of exe. mem. with blks of addresses in the addr. space.

For mem. mapped resources, the OS uses the addr. space to ctrl which processes have access to which resources. If the OS doesn't bind a memory mapped resource's i/p into the addr. space, then the process cannot reference the resource. The addr. space is an imp. part of the OS's mechanism for protecting resources from unauthorised access.

Process Synchronization

or

Cooperating Process: A cooperating process is one that can affect or be affected by other processes executing in the s/m. Cooperating processes can either directly share a logical address space or be allowed to share data only through files or msgs. Concurrent access to shared data may result in data inconsistency.

Bounded Buffer

An integer variable counter initialised to 0.

- Counter is incremented every time we add a new item to the buffer & is decremented every time we remove one item from the buffer. The code for producer.

```
while(true)
{ /* produce an item in nextProduced */
  while(counter == Buffer_size)
    ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in+1) % Buffer_size;
  counter++;
}
```

in=0
in=(0+1)%5 = 1

5

we can show that the value of counter may be incorrect as follows.

$reg_1 = counter$, $reg_1 = reg_1 + 1$, $counter = reg_1$

where reg_1 is a local CPU reg. counter-- is implemented as $reg_2 = counter$, $reg_2 = reg_2 - 1$, $counter = reg_2$

Initial counter = 5

The code for the consumer process can be

```
while (true)
{ while (counter == 0)
    ; /* do nothing */
  nextConsumed = buffer[out];
  out = (out+1) % buffer size;
  counter--;
}
```

T_0 : producer reg_1 = counter {$reg_1 = 5$}
T_1 : producer $reg_1 = reg_1 + 1$ {$reg_1 = 6$}
T_2 : consumer reg_2 = counter {$reg_2 = 5$}
T_3 : consumer $reg_2 = reg_2 - 1$ {$reg_2 = 4$}
T_4 : producer counter = reg_1 {counter = 6}
T_5 : consumer counter = reg_2 {counter = 4}

; incorrect state counter = 4
recording that there are 4 full buffers, when in fact there are 5 full buffers.

Both producer & consumer routines are correct separately, they may not function correctly when executed concurrently.

→ the value of the variable counter = 5

→ the producer & consumer process es execute the stmts "counter++" & "counter--" concurrently.

→ following the exn of these stmts, the value of the variable counter may be 4, 5, or 6.

Race Condition : A situation where several processes access & manipulate the same data concurrently and the outcome of the exn depends on the particular order in which the access takes place, is called race condition.

To guard against the race condition above we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized.

The Critical-Section Problem

A s/m consisting of n processes $\{P_0, P_1 \ldots P_{n-1}\}$. Each process has a segment of code, called a critical section, in which process may be changing common variables, updating table, writing a file & so on. The imp. feature of the s/m is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

ie; no two processes are executing in their critical sections at the same time.

The critical_section pblm is to design a protocol that the processes can use to cooperate.

Each process must request permission to enter its critical section. The section of code implementing this rqst is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

A solution to the critical-section pblm must satisfy the follo. 3 requirements:

1. Mutual exclusion: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

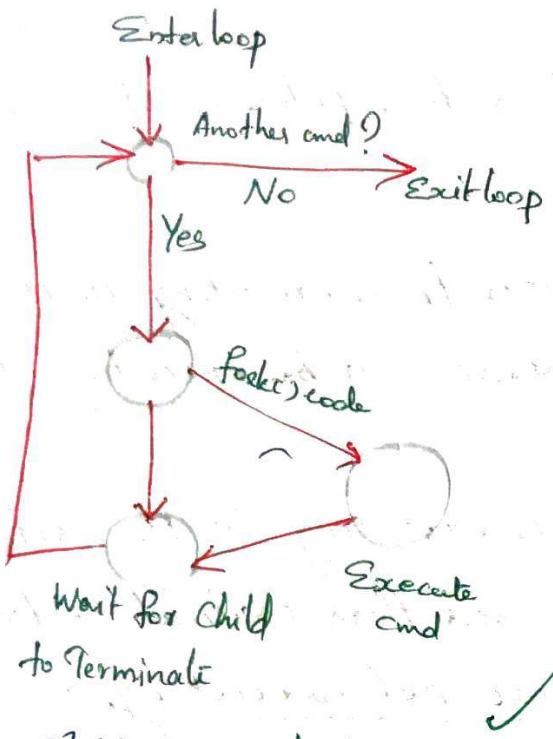
2. Progress: If no process is executing in its critical section & some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, this selection can't be postponed indefinitely.

3. Bounded Waiting: There exists a bound, limit on the no. of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section & before that reqst is granted.

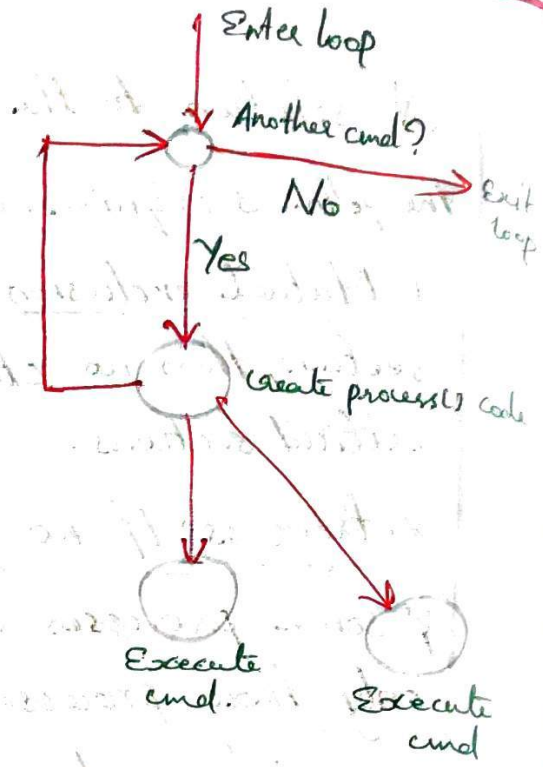
Cooperating Process Contd.

Command Execution:

Part(a): The parent process synchronizes with the child process activity by waiting for each child to terminate before starting the next one. Part(b) is the ctrl flow pattern the Windows code segment. In this case, the parent creates a child, but does not synchronize with that child again. Instead, the parent creates the next child to



(a) UNIX Shell



(b) Windows Cmd Launch

fig(a) → Parent pgm creates a child process to execute a cmd, then waits for the child to terminate before reading the next cmd. There is a concurrency b/w the parent & atmost one child at a time.

fig(b): The parent process creates a process to execute a command, then immediately goes back to the top of the loop to create another processes that execute the cmds will be created to run at the same time. There is concurrency among the parent & all of the child processes.

Semaphores

A semaphore S is an integer variable that, apart from initialization, is accessed only through 2 std atomic operations: wait() & signal().

The wait() opⁿ was originally termed P.

The signal() " " V

The defⁿ of wait() is as follows:

P

```
wait(S)
{ while S<=0
    ; // no op
  S--
}
```

The defⁿ of signal() is as follows: (V)

```
Signal(S)
{ S++;
}
```

All the modifications to the integer value of the semaphore in the wait() & signal() opⁿs must be executed indivisibly. Ie; when one process modifies the semaphore value. In addition, in the case of wait(S), the testing of the integer value of S (S≤0) & its possible modification (S--), must also be executed without interruption.

The P opⁿ is intended to indivisibly test an integer variable and to blk the calling process if the variable is not true.

The V opⁿ indivisibly signals a blocked process to allow it to resume opⁿ.

Binary & Counting Semaphore

→ The value of counting semaphore can range over an unrestricted domain.

→ The value of a binary semaphore can range only b/w 0 & 1. Binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

We can use binary semaphores to deal with the critical-section pblm for multiple processes. The n processes share a semaphore, mutex, initialised to 1. Each Pi process Pi is organised as

```
do
{
  waiting(mutex);
    // crit section
  signal(mutex);
    remainder section
}while(TRUE);
```

Counting semaphores can be used to ctrl access to a given resource consisting of a finite no. of instances. The semaphore is initialised to the no. of resources available.

Each process that wishes to use a resource performs a wait() opr on the semaphore (thereby decrementing the count.)

When a process releases a resource, it performs a signal() op^n (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that processes that wish to use a resource will blk until the count becomes greater than 0.

For eg: consider 2 concurrently running processes: P_1 with a stmt S_1 and P_2 with a stmt S_2. Suppose we require that S_2 be executed only after S_1 has completed.

```
S1;
signal(synch);
```

```
synch = 0;
S1
signal(synch);
```

in process P_2

```
wait(synch)
S2;
```

```
if synch>=0
    wait(synch);
S2;
```

in process P_2. synch is initialised to 0, P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after stmt S_1 has been executed.

Busy waiting

While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. The continual looping is clearly a pblm in a real multi pgmming s/m, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of ~~sepe~~ semaphore is also called spinlock.

The advantage of spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful.

To overcome the need for busy waiting, we can modify the defn of the wait & signal semaphore op's. When a process executes the wait opn and finds that the semaphore value is not +ve, it must wait. However, rather than busy waiting, the process can block itself. The block opn places a process into a waiting queue associated with the semaphore, & the state of the process is switched to the waiting state. The ctrl is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore, S should be restarted when some other process executes a signal opn. The process is restarted by a wakeup opn, which changes the process from the waiting state to the ready state. The process is then placed in ready queue.

we define semaphore as a 'C' struct.
Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore, it is added to the list of processes. A signal op

removes one process from the list of waiting processes and awakens that process.

The wait semaphore op^n can now be defined as

```
void wait(semaphore S)
{ S.value--;
  if(S.value<0)
    { add this process to S.L; spinLock(SL).
      block();
    }
}
```

The signal semaphore op^n can now be defined as

```
void signal(semaphore S)
{ S.value++;
  if(S.value<=0)
    { remove process P from S.L;
      wakeup(P);
    }
}
```

The block operation suspends the process that invokes it. The wakeup(P) op^n resumes the exc^n of a blocked process P. These 2 op^{n}s are provided by the O.S as basic s/m calls.

Deadlocks & Starvation

The implementation of a semaphore with a waiting queue may result in a situation where 2 or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. When such state is reached, these processes are said to be deadlocked.

A s/m consisting of 2 processes, P_0 & P_1 each accessing 2 semaphores, S & Q, set 2 to the value 1

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

P_0 & P_1 are deadlocked.

Another pblm related to deadlocks is indefinite blocking or starvation, a situation where processes wait indefinitely within the semaphore. Indefinite blking may occur if we add & remove processes from the list of associated with a semaphore in LIFO order.

→ Suppose that P_0 executes wait(S), & then P_1 executes wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q).

Ily S.

Classic Problems of Synchronization

* The Bounded-Buffer Problem.

— The pool consists of n buffers, each capable of holding 1 item. The mutex binary semaphore provides mutual exclusion for accesses to the buffer pool & is initialised to the value 1. The empty & full counting semaph. ores count the no. of empty & full buffers respectively. The semaphore empty is initialised to the value n. the semaphore full is initialised to the value 0.

```
do
{ ...
   produce an item in next p
   ...
   wait(empty);
   wait(mutex);
   ...
   add next p to buffer
   ...
   signal(mutex);
   signal(full)
} while(1);
```

Restructure of producer process.

```
do{
  wait(full);
  wait(mutex);
  ...
  remove item from buffer to next c
  ...
  signal(mutex);
  signal(empty);
  ...
  consume the item in next c.
}while(1);
```

Suppose a s/m incorporates 2 single threaded processes one of which produces inf^n & another that uses inf^n (consumer process). The 2 processes communicate by having the producer obtain an empty buffer from an empty buffer pool, fill it with inf^n & place it in a pool of full buffers. The consumer obtains inf^n by picking up a buffer from the full buffer pool, copying the inf^n from the buffer, and placing the buffer into the empty buffer pool for recycling. The producer & consumer use a fixed, finite no, N of buffers to pass an arbitrary amt of inf^n b/w them.

The empty & full semaphores illustrate a new type of semaphore, called a general semaphore is also called counting semaphore. Whereas a binary semaphore takes on only the values 0 & 1, the counting semaphore takes on values from 0 to N for N-buffer pblms.

In the eg^l counting semaphore kep

the a count of the no. of empty & full buffers. They also are used to synchronize the process op^n by blocking the producer when there are no empty buffers and blocking the consumer when there are no full buffers.

The buffers are a block of mem. logically split into N parts. Each buffer must contain space for links to associate the buffer with other empty or full buffers & space for the data itself. Since the producer and consumer each manipulate these links, the code for buffer pool manipulation must be treated as a critical section.

The mutex semaphore protects access to the two buffer pools so that only one process takes or puts a buffer at a time. The mutex semaphore is used to protect the critical section relating to manipulating buffers. The P(empty) op^n blocks the producer if there are no empty buffers. Ully the P(full) op^n blks the consumer if there are no full buffers.

The Readers-Writers Problems

A data object is to be shared among several concurrent processes. Some of these processes may want only to read the cont. of the shared object, whereas others may want to update the shared object. Those processes that are interested in only reading as readers & to the rest as writers.

2 readers can access shared objt simultaneously.
2 writers cannot ".

Here we require that the writers have exclusive access to the shared objt. This synch. pblm is referred to as the readers-writers problems.

The readers-writers pblm has several variations,

first readers-writers pblm: no reader will be kept waiting unless a writer has already obtained perm^n to use the shared objt. (no reader should wait for other readers to finish simply b/c a writing is waiting.) → writers may starve.

second ~~writers~~: readers-writers pblms: once a writer is ready, that the writer performs its write as soon as possible. (if a writer is waiting to access the objt no new readers may start reading) → readers may starve.

The structure of a writer process:

```
wait(wrt);
    writing is performed
    ---
signal(wrt);
```

Solution to the first readers-writers pblm, the reader processes share the follo. data structures:

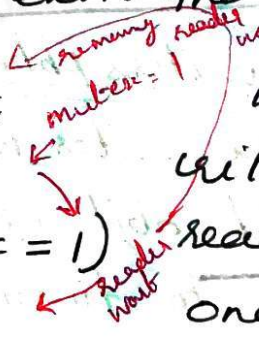
```
semaphore mutex, wrt;
int readcount;
```

wrt - mutual exclusion semaphore for writers

→ mutex & wrt are initialised to 1.

→ readcount = 0. initialized

⇒ mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the obj. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section.




```
wait(mutex);
readcount++;
if (readcount == 1)
    wait(wrt);
signal(mutex);
    ...
    reading performed
    ...
wait(mutex);
readcount--;
if(readcount == 0)
    signal(wrt);
signal(mutex);
```

If a writer is in critical section and n readers are waiting, then one reader is queued on wrt, and n-1 reader are queued on mutex.

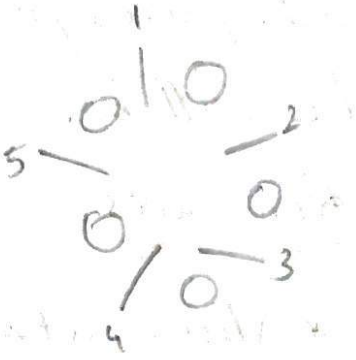
When a writer executes signal(wrt), we may resume the exn of either the waiting readers or a single waiting writer.

The Dining Philosophers Problem

```
do
{ wait(chopstick[i]);
  wait(chopstick[(i+1)%5]);
  ...
  eat
  ...
  signal(chopstick[i]);
  signal(chopstick[(i+1)%5]);
  ...
  think
  ...
} while(1);
```

chopstick[1]

chopstick[2%5] = 2



→ It is considered a classic synch. pblm., no

→ It is a simple representation of the need to allocate several resources among several processes in deadlock- and starvation free manner.

One simple sol^n is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a wait op^n on that semaphore; she releases her chopsticks by executing the signal op^n on the appropriate semaphores.

The shared data are semaphore chopstick[5]; where all elmts of chopstick are initialised to 1.

Although this sol^n guarantees that no two neighbours are eating simultaneously, it nevertheless must be rejected b/c it has the possibility of creating a deadlock.

Suppose that all five philosophers become hungry simultaneously, & each grabs her left chopstick. All the elmts of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Q3

Remedies

* Allow atmost 4 philosophers to be sitting simultaneously at the table.

* Allow a philosopher to pick up her chopsticks only if both chopsticks are available.

* An odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosophee.

CPU Scheduling

Multipgmming: is to have some process running at all times so as to maximize CPU utilization.

Time sharing: is to switch the CPU among processes so frequently that users can interact with each pgm while it is running.

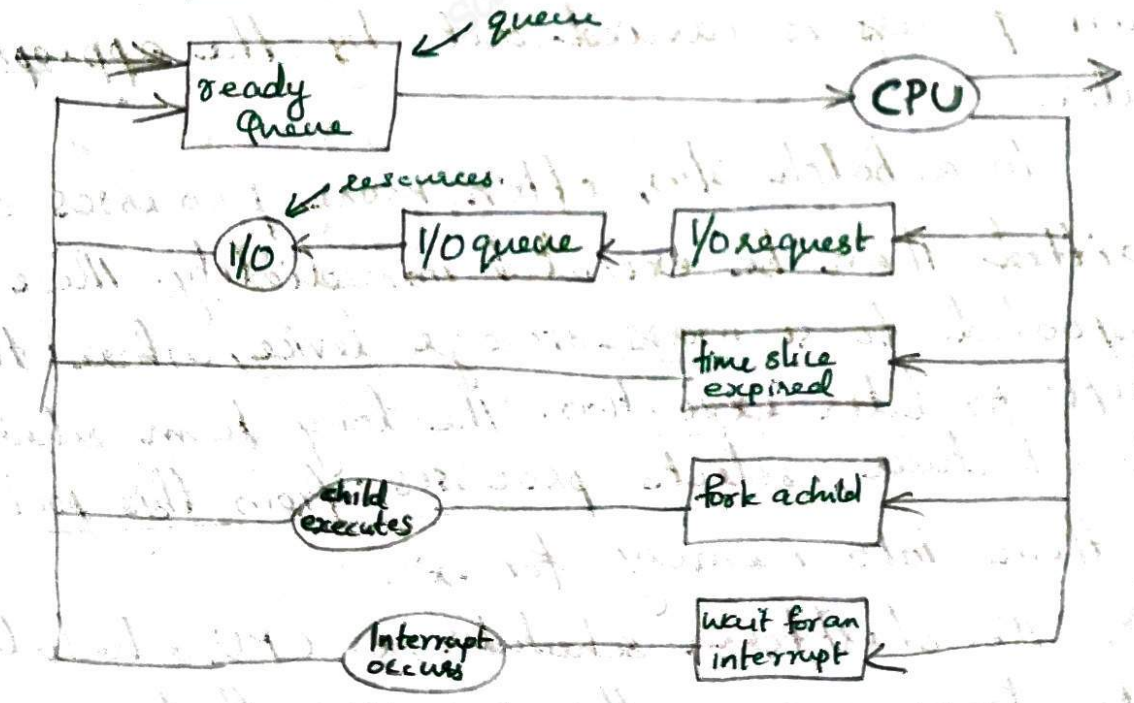
Uniprocessor s/ms: can have only one running process. If more process exist, the rest must wait until the CPU is free and can be rescheduled.

Scheduling Queues:

As processes enter the s/m, they are put into a job queue. This queue consists of all processes in the s/m.

The processes that are residing in m/m and are ready & waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready queue header contains pointers to the first and final PCBs in the list. Each PCB to include a pointer field that points to the next PCB in the ready queue.

The list of processes waiting for a particular I/O device is called a device queue.



Queueing diagram representation of process scheduling

The primary distinction b/w these 2 schedulers is the frequency of their ex^n. The short-term scheduler must select a new process for the CPU frequently.

The long term scheduler executes much less frequently. There may be minutes b/w the creation of new processes in the s/m. The long term scheduler controls the degree of multipgmming (no. of processes in mem.).

An I/O bound process spends more of its time ~~using~~ doing I/O than it spends doing computations. A CPU-bound process generates I/O requests infrequently using more of its time doing computation than an I/O bound process uses.

Context Switch

Switching the CPU to another process requires saving the state of the old process & loading the saved state for the new process. The context of process is represented in the PCB of a process; it includes the value of the CPU registers, the process state & mem. mgmt info^n.

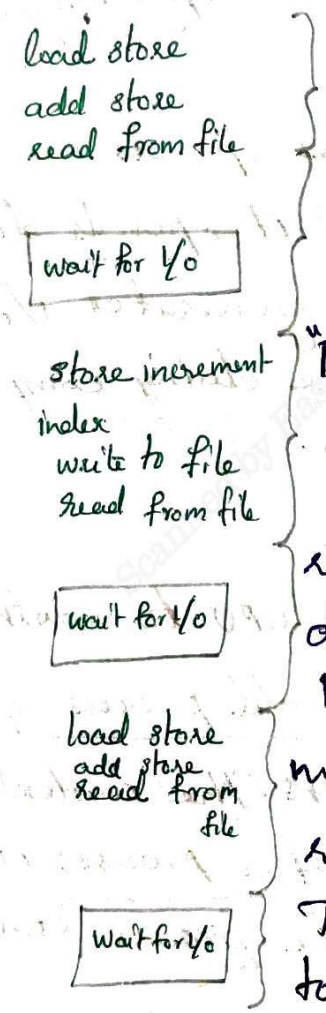
When a context switch occurs, the kernel saves the context of the old process in its PCB & loads the saved context of the new process scheduled to run.

Context-switch time is pure overhead, b/c the s/m does no useful work while switching.

CPU – I/O Burst Cycle

Process exⁿ consists of a cycle of CPU exⁿ & I/O wait. Processes alternate b/w these 2 states.

→ Process exⁿ begins with a CPU burst that is followed by an I/O burst, then another I/O burst & so on.



Alternating sequence of CPU & I/O bursts.

"Device Queue": continuation of queues.

In the case of an I/O rqst, may be to a dedicated tape drive or a disk. Since the s/m has many processes, the disk may be busy with the I/O rqst of some other processes. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O Device is called a device queue.

Preemptive Scheduling Module IV

CPU scheduling decisions may take place under the follo. 4 circumstances:

1. When a process switches from the running state to the waiting state (for eg: I/O rqst, or invocation of wait for the termination of one of the child processes.)
2. When a process switches from the running state to the ready state (for eg: when interrupt occurs)
3. When a process switches from the waiting state to the ready state (for eg: completion of I/O).
4. When a process terminates.

When scheduling takes place only under circumstances 1 & 4, we say the scheduling scheme is nonpreemptive. Otherwise scheduling scheme is preemptive.

Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

Dispatcher

The dispatcher is the module that gives ctrl of the CPU to the process selected by the short_term scheduler. This fn involves:

* Switching context
* Switching to user mode
* Jumping to the proper location in the user pgm to restart that pgm.

The time it takes for the dispatcher to stop one process & start another running is known as the dispatch latency.

Scheduling Criteria

Many criteria have been suggested for comparing CPU-scheduling algms.

* CPU Utilization: ✓
* Throughput: No. of processes completed / time unit
* Turnaround: The interval from the time of submission of a process to the time of completion.
* Waiting time: the sum of the periods spent waiting in the ready queue.
* Response time: the time from the submission of a request until the first response is produced.

Scheduling Algorithms

CPU scheduling deals with the pblm of deciding which of the processes in the ready queue is to be allocated the CPU.

FCFS – First Come, First Serve Scheduling: (Nonpreemptive)

→ the process that requests the CPU first is allocated the CPU first.

eg: Consider the follo. set of processes that arrive at time 0 with the length of the CPU-burst time given in ms.

Process	Burst time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P_1, P_2, P_3 and are served in FCFS order: Gantt Chart.



The waiting time → for P_1 = 0

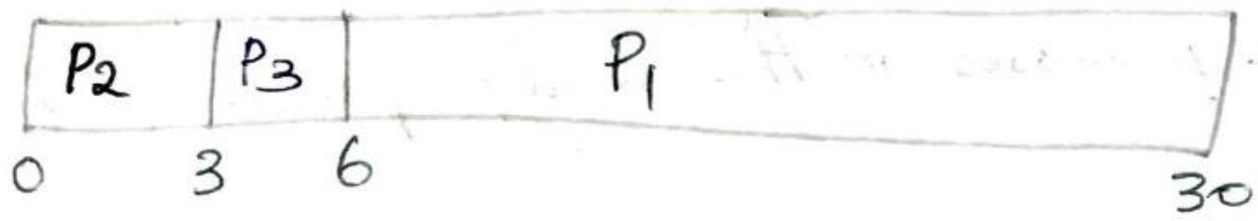
" P_2 = 24

" P_3 = 27

Average waiting time = (0+24+27)/3 = 17 ms.

Scanned by TapScanner

If the processes arrive in the order P_2 P_3 P_1 however Gantt chart:



Avg waiting time $= (6+3+0)/3 = \underline{3 \text{ ms.}}$

Thus the avg waiting time under a FCFS policy is not minimal.

Assume we have one CPU-bound process & many I/O bound processes. The CPU bound process will get CPU & hold it. During this time, all the other processes will finish their I/O & move into the ready queue, the I/O devices are idle. Eventually the CPU bound process finishes its CPU burst & moves to an I/O device.

All I/O bound processes, which have very short CPU bursts, execute quickly & move back to I/O queues. At this point CPU sits idle. The CPU bound process will then move back to the ready queue & be allocated the CPU. Again all the ~~CPU~~ I/O processes end up waiting in the ready queue until the CPU process is done.

There is a convoy effect, as all the other processes wait for the one big process to get off CPU. This effect results in lower CPU &

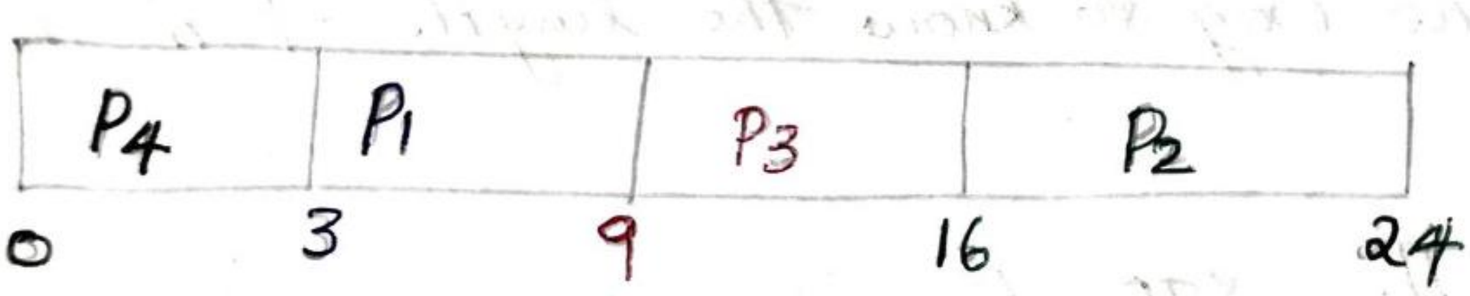
Scanned by TapScanner

device utilization than might be possible if the shorter processes were allowed to go first.

Shortest Job First Scheduling

When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

Process	Burst time.
P_1	6
P_2	8
P_3	7
P_4	3



The waiting time for $P_1 = 3 + 6 = 9$

" $P_2 = 16 + 8 = 24$

" $P_3 = 9 + 7 = 16$

" $P_4 = 0 + 3 = 3$

Avg waiting time $= (3 + 16 + 9 + 0)/4$

$= \underline{7 \text{ ms}}$

The SJF scheduling algm is optimal, in that it gives the minimum avg waiting time for a given set of processes.

Scanned by TapScanner

By moving a short process before a long one, the waiting time of short process decreases more than it increases the waiting time of the long process. Consequently, the avg waiting time decreases.

The real difficulty with the SJF algm is knowing the length of the next CPU request. For long term scheduling in a batch s/m, we can use as the length the process time limit that user specifies when he submits the job. SF SJF scheduling is used frequently in long-term scheduling.

Although the SJF algm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst.

The SJF algm may be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is executing. The new process may have shorter next CPU burst that than what is left of the currently executing process.

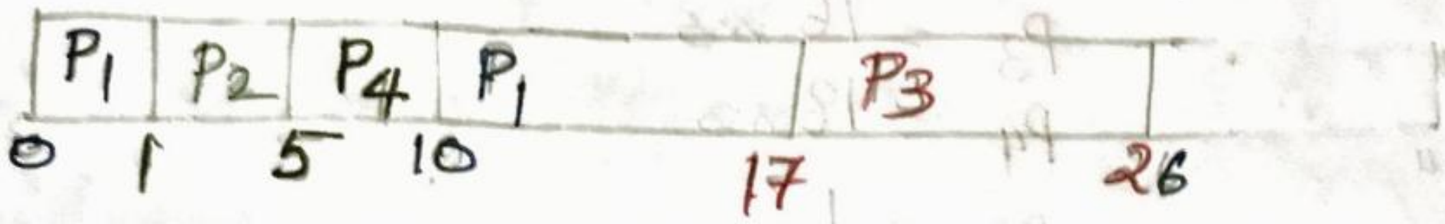
A preemptive SJF algm will preempt the currently executing process, whereas nonpreemptive SJF algm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is called shortest_remaining_time_first.

Scanned by TapScanner

eg:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Preemptive SJF Scheduling.



Waiting time for $P_1 = (10-1) = 9$ ms

" $P_2 = (1-1) = 0$ ms

" $P_3 = (17-2) = 15$ ms

" $P_4 = (5-3) = 2$ ms

Avg waiting time $= (9+0+15+2)/4$

$= 26/4 = \underline{6.5 \text{ ms}}$

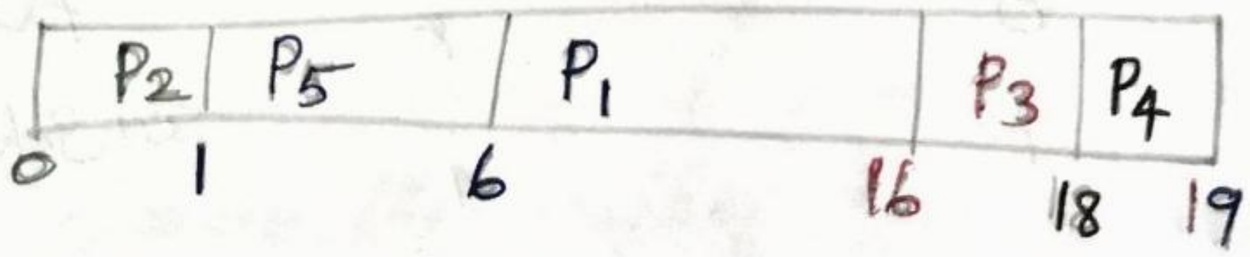
Priority Scheduling:

A priority is associated with each proce and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order.

Eg,

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

Scanned by TapScanner



Waiting time for P_1 = 6 ms

" P_2 = 0 ms

" P_3 = 16 ms

" P_4 = 18 ms

" P_5 = 1 ms.

Avg waiting time = $(6+16+0+18+1)/5 = 8.2$ ms

Priority scheduling can be either preemptive or non preemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algm will preempt CPU if the priority of the newly arrived process is higher than the priority of the running process. A non preemptive priority scheduling algm will simply put the new process at the head of the ready queue.

Priority scheduling – indefinite blocking or starvation.

A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algm can leave some low priority processes indefinitely. In a heavily loaded comp. system a steady stream of higher priority processes can prevent a low priority process ever getting the CPU

Scanned by TapScanner

A solution to the pblm of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the s/m for a long time. For eg: if priorities range from low to high, we could increase the priority of a waiting process by 1 every 15 minutes.

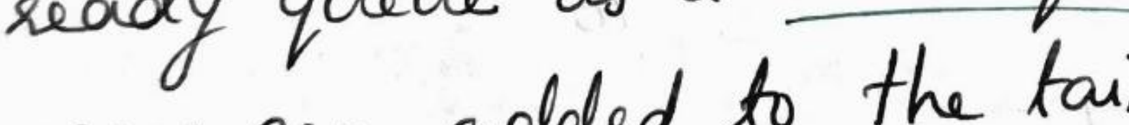
Round Robin Scheduling

—is designed for time sharing s/ms. A small unit of time, called a time quantum or time slice, is defined. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of upto 1 time quantum. To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the queue.

Eg:

Process	Burst time
P_1	24
P_2	3
P_3	3

$t = 4$ ms.



waiting time for $P_1 = \cancel{0+10=10ms}$ $0+(10-4) = 6$

$P_2 = 4$ ms

$P_3 = 7$ ms.

Avg waiting time $= \frac{(6+4+7)}{3} = 5.66$ ms

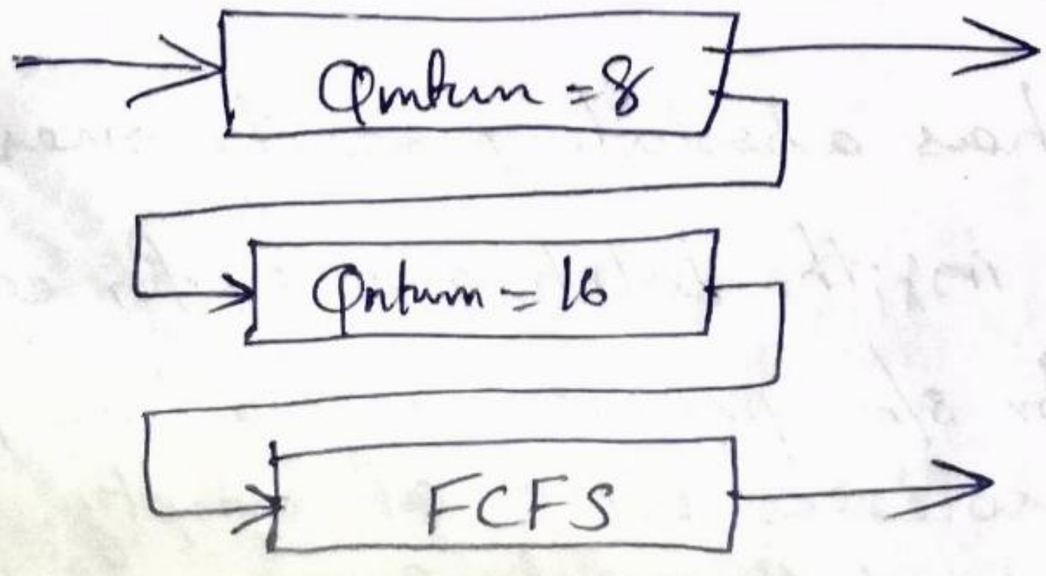
Scanned by TapScanner

Multilevel Feedback Queue Scheduling

— allows a process to move b/w queues. The idea is to separate processes with diff. CPU-burst characteristics. If a processor uses too much CPU time, it will be moved to lower priority queue. This scheme leaves I/O bound and interactive processes in the higher-priority queues. Ily a process that waits too long in lower priority queue may be moved to a higher priority queue. This form of aging prevents starvation.

A multilevel feedback queue scheduler is defined by the following parameters:

* The no. of queues. * The scheduling algm for each queue.
* The method used to determine when to upgrade a process to a higher priority queue.
* The method used to determine when to demote a process to a lower-priority queue.
* The method used to determine which queue a process will enter that process needs service.



Scanned by TapScanner

DEADLOCKS

A process requests resources, if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, b/c the resources they have requested are held by other waiting processes. This solution is called a situation deadlock.

→ Prevention (4 conditions)
- Hold & wait
- Circular wait
- Preemption

→ Avoidance
- Safe State flow diagram (Not fig 10.13
- Safe, Unsafe, & Deadlock state
- Bankers Algm

→ Detection & Recovery:
- Serially reusable Resources.
- Reusable resource Graph
- Consumable Resources
- " Graph Model
- Recovery

Scanned by TapScanner

Under normal mode of operation, a process may utilize a resource in only the following sequence:

* Request: If the request cannot be granted immediately (for eg: the resource is being used by another process), then the requesting process must wait until it can acquire the resource.

* Use: The process can operate on the resource (for eg:, if the resource is a printer, the process can print on the printer)

* Release: The process releases the resource.

The Request & Release of resources can be accomplished through the wait & signal op's on semaphores. A s/m table records whether each resource is free or allocated and if a resource is allocated to which process.

To illustrate a deadlock state,

eg: a s/m with 3 tape drives, each of three processes holds one of these tape drives. If each process now requests another tape drive, the three processes will be in a dead lock state.

Scanned by TapScanner

The features that characterize Deadlock.
if 4 conditions hold situations hold simultaneously.

1. Mutual Exclusion: At least one resource must be held in a nonshareable mode; only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold & wait: A process must be holding at least one resource & waiting to acquire additional resources that are currently being held by other processes.

3. No preemption: Resources cannot be preempted that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular Wait: A set $\{P_0, P_1, \ldots P_n\}$ of waiting process must exist such that P_0 is waiting for a resource that is held by $P_2, \ldots, P_{n-1}$ 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.

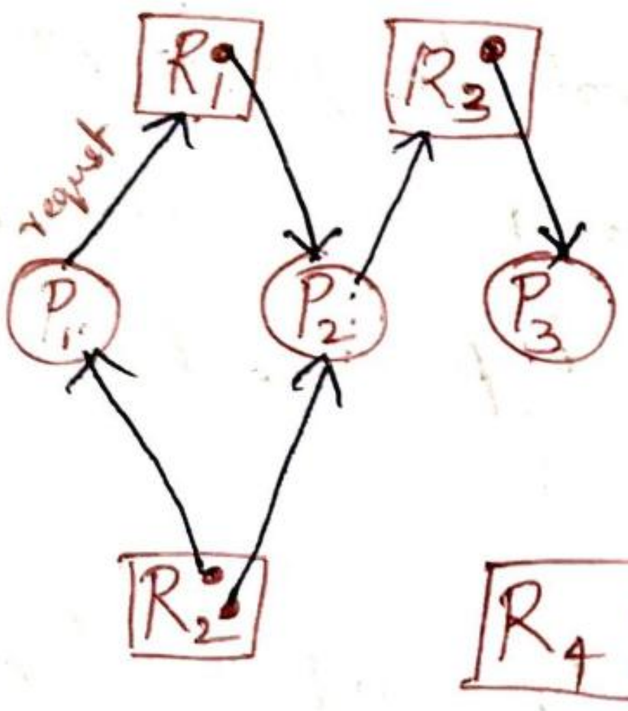
Resource Allocation Graph:

Deadlocks can be described in terms of directed Resource Allocation Graph. This graph consists of a set of vertices (V) & a set of edges (E). The set of vertices V partitioned into two diff. types of nodes $P = \{P_1, P_2 \cdots P_n\}$, the set consisting of all active processes in the s/m and $R = \{R_1 R_2 \cdots R_n\}$, the

Scanned by TapScanner

set consisting of all resource types in the s/m.

A directed edge from process P_i to resource R_j is denoted by $P_i \longrightarrow R_j$, it signifies that process P_i requested an instance of resource type R_j & its process P_i is denoted by $R_j \longrightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i. A directed edge $P_i \longrightarrow R_j$ is called a request edge a directed edge $R_j \longrightarrow R_i$ is called an assignment edge.



$P = \{P_1, P_2, P_3\}$ $R = \{R_1, R_2, R_3, R_4\}$

$E = \{P_1 \to R_1, P_2 \to R_3, R_1 \to P_2, R_2 \to P_2, R_2 \longrightarrow P_1, R_3 \longrightarrow P_3\}$

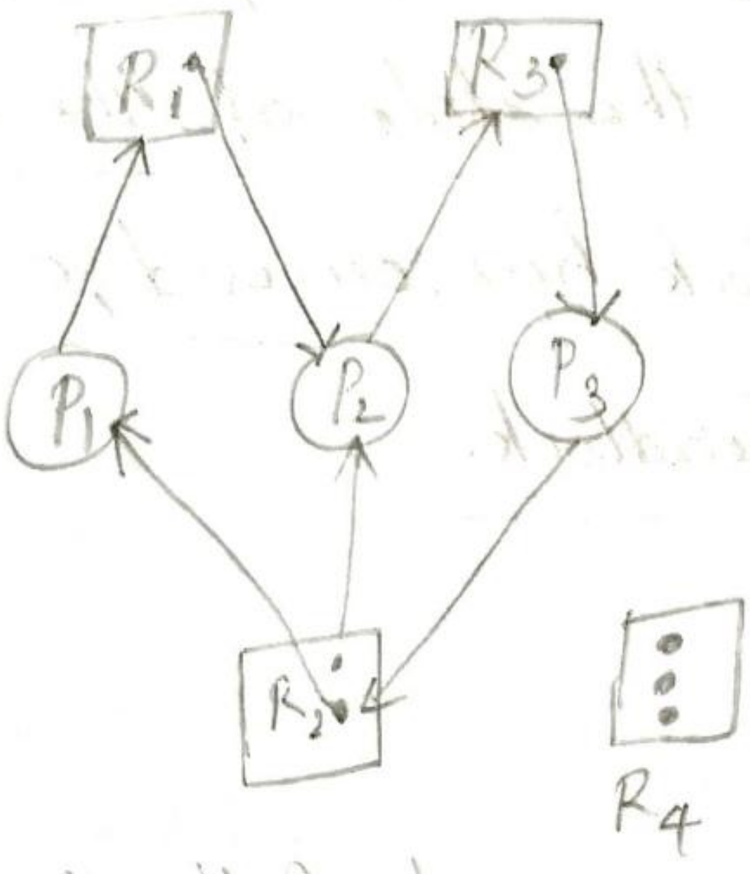
Process states:

→ Process P_1 is holding an instance of resource type R_2, and is waiting for an instance of resource type R_1.

→ Process P_2 is holding an instance of R_1 and R_2 and is waiting for an instance of resource type R_3.

→ Process P_3 is holding an instance of R_3.

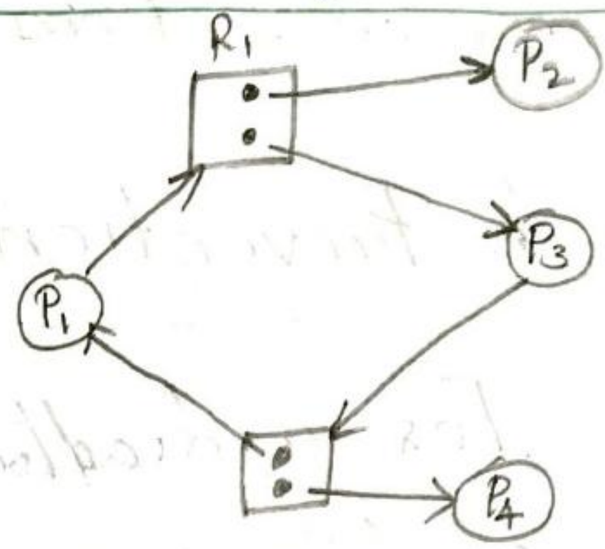
Scanned by TapScanner



Resource Allocation Graph with a deadlock.

$P_2 \rightarrow R_3$

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$



$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

If a resource-allocation graph does not have a cycle, then the s/m is not in a deadlock state. If there is a cycle, then the s/m may or may not be in a deadlock state.

Methods for Handling Deadlocks

Deadlock Prevention: is a set of methods for ensuring that atleast one of the necessary conditions cannot hold.

Deadlock Avoidance: requires that the operating s/m be given in advance additional inf" which resources a process will request and use during its lifetime. With this additional knowledge, we can decide for each rqst whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the s/m must consider the resources currently available the resources currently allocated to each process, and the future requests & releases of each process.

Scanned by TapScanner

Deadlock Detection & Recovery: The s/m provides an algorithm that examines the state of the s/m to determine whether a deadlock has occured, and an algm to recover from the deadlock.

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that atleast one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

* Mutual Exclusion

The mutual exclusion must hold for nonsharable resources. For eg:, a printer cannot be simultaneously shared by several processes. ~~Sharable~~ Sharable resources do not require mutually exclusive access, and thus cannot be involved in a deadlock. Read only files are a good example of a sharable resource. If several processes attempt to open a read only file at the same time, they can be granted simultaneous access to the file.

* Hold & Wait

To ensure that the hold & wait cond^n never occurs in the s/m, we must guarantee that, whenever a process requests a resource, it doesn't hold any other resources.

Scanned by TapScanner

Protocol 1: requires each process to request and be allocated all its resources before it begins execution.

Protool 2: allows a process to request resources only when the process has none.

Before it can request any additional resources however, it must release all the resources that it is currently allocated.

Proto 1: eg: a process - copies from tape to disk, sort & print:

- requests tape, disk & printer. initially. even printer is needed at the end.

proto 2: requests tape & disk, copies, & releases & then only requests printer.

2 disadvantages:

→ Resource utilization - low ✓

- many resources may be allocated but unused for long time.

→ Starvation: A process that needs several resources may have to wait indefinitely. b/c at least one of the resources that it needs is always allocated to some other process.

* No Preemption:

If a process is holding some resources & requests another resource that cannot be immediately

Scanned by TapScanner

allocated to it (ie; the process must wait) then all resources currently being held are preempted.

These resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting.

If a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

*Circular Wait:

One way to ensure that this condn never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing order.

Let $R = \{R_1, R_2 \ldots R_m\}$ be the set of resource types. We assign to each resource type a unique integer no., which allows us to compare 2 resources and to determine whether one precedes another in our ordering. For eg: if the set of resource types R includes tape drives, disk drives & printers then the fn F might be defined as follows:

Scanned by TapScanner

F(tape drive) = 1

F(disk drive) = 5

F(printer) = 12.

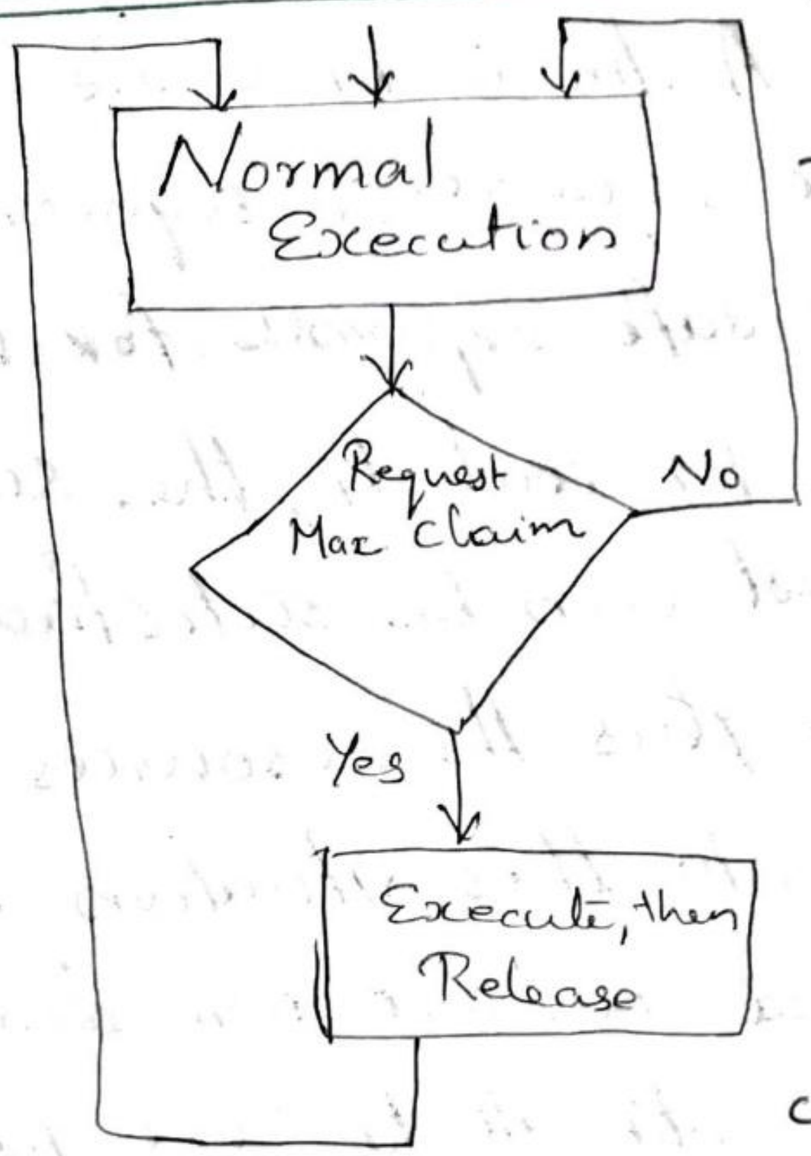
{P0, P1 ... Pn}, Pi waiting Ri held by Pi+1.
Modulo arithmetic Pn waiting for Rn held by P0
Pi+1 holding Ri requesting resource Ri+1
F(Ri) < F(Ri+1) for all i
ie; F(R0) < F(R1) ... < F(Rn) < F(R0)
transitivity F(Rn) < F(R0)

To prevent deadlocks; Each process can request resources only in an increasing order of enumeration.

ie; A process can initially request any no. of instances of a resource type, say Ri. After that, the process can request instances of resource type Rj iff F(Rj) > F(Ri)

If several instances of same resource type are needed, a single request for all of them must be issued.

Deadlock Avoidance



Each request requires that the o/m consider the resources currently availab the resources currently allo to each process and the future requests and releas of each process, to decide whether the current requ can be satisfied or must wait to avoid future deadlo

→ Each process declare the max. no. of resources of ea type that ~~may be requested for each~~ it may need. Given a priori info about the max. no. of resources of

Scanned by TapScanner

type that may be requested for each process, it is possible to construct an algm that ensures that the s/m will never enter deadlock state. This algm defines the deadlock-avoidance approach.

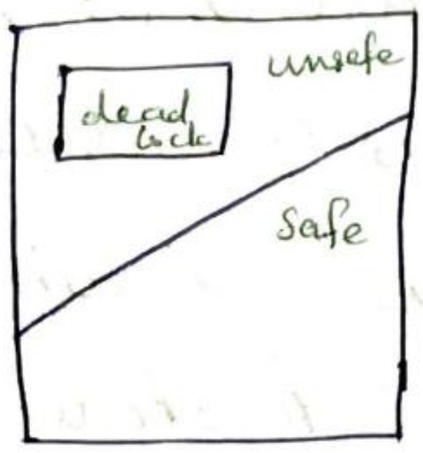
A deadlock-avoidance algm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The resource allocation state is defined by the no. of available and allocated resources, and the maximum demands of the processes.

Safe State

A state is safe if the s/m can allocate resources to each process in some order and still avoid a deadlock. A s/m is in a safe state only if there exists a safe sequence. A sequence of processes $\langle P_1, P_2, \ldots P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the current available resources plus the resources held by all the P_j, with $j<i$. In this situation the resources that process P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources & soon. If

Scanned by TapScanner

no such sequence exists, then the s/m state is said to be unsafe.



safe state $\not\Rightarrow$ no deadlock

unsafe state $\Rightarrow$ deadlock but not all unsfestate.

Eg: 12 magnetic tapes drives & 3 processes - P_0, P_1 & P_2.

P_0 requires 10 tape drives.

P_1 " 4 " . Total Rqmnt = 23

P_2 " 9 " .

Suppose at time t_0, P_0 is holding 5 tape drives.

P_1 " 2 "

P_2 " 2 " Total held = 9

12 - 9 = (3)

	Max. Needs	current needs
P_0	10	5
P_1	4	2
P_2	9	~~7~~

At time t_0, the s/m is in safe state. The sequence $\langle P_1, P_0, P_2 \rangle$

$\langle P_0, P_1, P_2 \rangle$ at t_0

only 3 tapes are available

P_0 needs 10 tape drives. — It cannot be satisfied.

So not ~~its~~ safe sequence.

$\langle P_1, P_0, P_2 \rangle$

P_1 — holding 2 & needs 2. It can be satisfied & returns 4 then total (4+1) = 5 available.

P_0 — holding 5 & needs 5. It can be satisfied & returns 10 — Then total 10 available

P_2 — holding 2 can be satisfied & returns 9

Scanned by TapScanner

Given the concept of a safe state, we can define avoidance algms that ensure that the s/m will never deadlock. The idea is to ensure that the s/m will ~~never deadlock~~. always remain in a safe state. Initially, the s/m is in a safe state. Whenever the process requests a resource that is currently available, the s/m must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the s/m in a safe state.

If a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would be without a deadlock-avoidance algm.

Resource-Allocation Graph Algorithm for single instance of resource

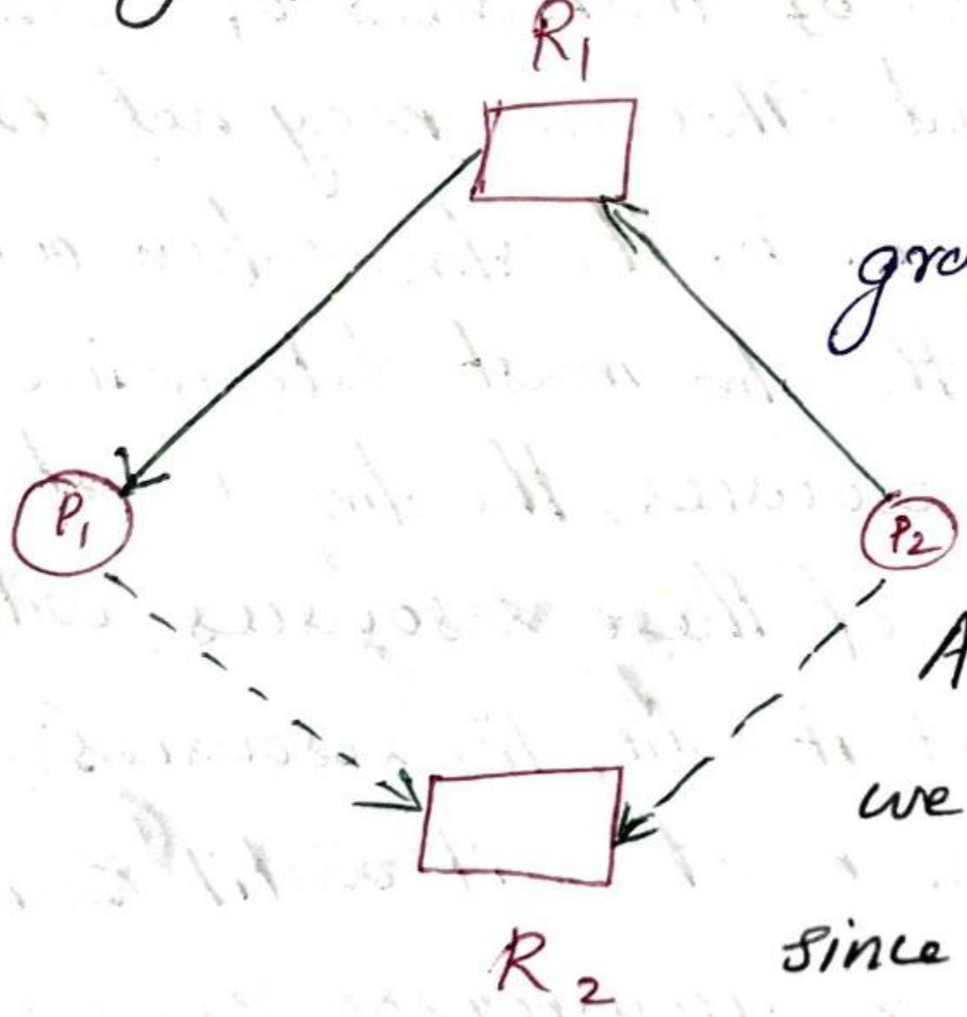
→ request edge, assignment edge & claim edge.

A claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in future. This edge resembles a request edge in direction, but is represented by a dashed line. When a process P_i requests resource R_j, the claim edge $P_i \rightarrow R_j$ is converted to a request edge. Illy, when a resource R_j is released by P_i, the assignment edge $R_j \rightarrow P_i$

Scanned by TapScanner

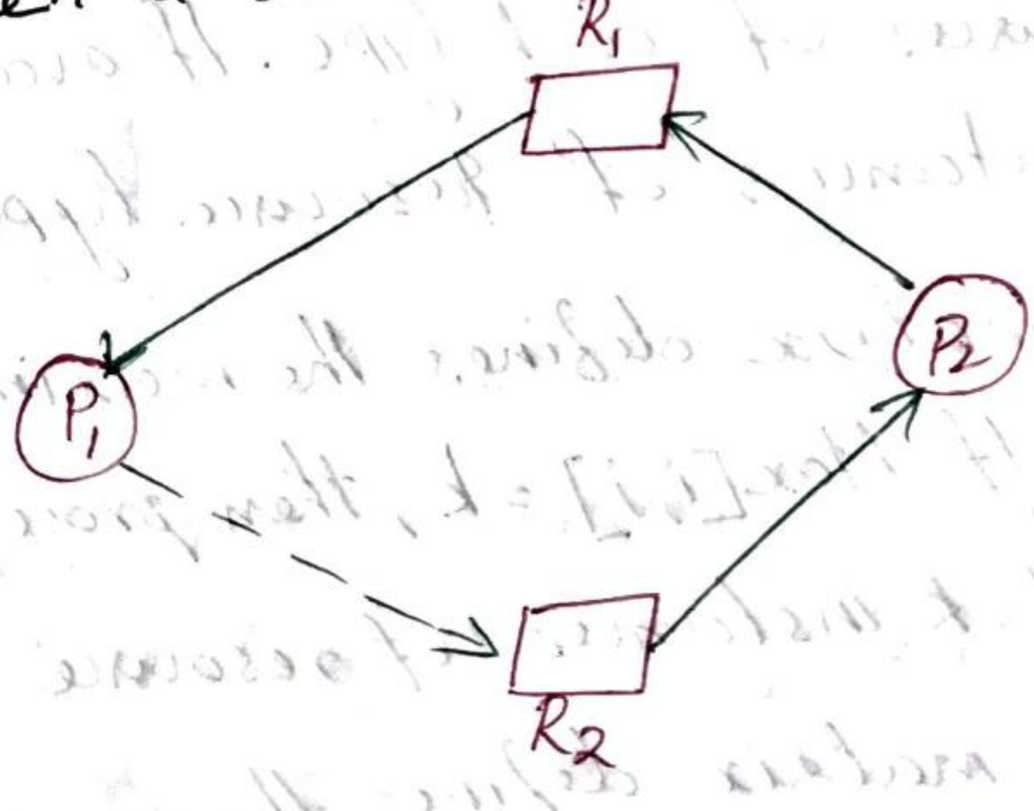
is reconverted to a claim edge $P_i \rightarrow R_j$.

Suppose that process P_i requests resource R_j. The request can be granted only if converting the rqst edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ doesn't result in the formation of a cycle in the resource allocation graph.



Resource Allocation graph for deadlock avoidance.

Suppose P_2 requests R_2. Although R_2 is currently free, we cannot allocate it to P_2, since this action will create a cycle in the graph. A cycle indicates that the s/m is in an unsafe state. If P_1 requests R_2 and P_2 requests R_1 then a deadlock will occur.



An unsafe state in a resource-allocation graph.

Scanned by TapScanner

Banker's Algorithm

- Applicable to a s/m with multiple instances of each resource type.

When a new process enters the s/m, it must declare the maxx. no. of instances of each resource type that it may need. This no. may not exceed the total no. of resources in the s/m. When a user reqts a set of resources, the s/m must determine whether the allocation of these resources, the s/m must determine whether the allocation of these resources will leave the s/m in safe state. If it will, the resources are allocated, otherwise, the process must wait until some other process releases enough resources.

Data structures used:

Available: A vector length m indicates the no. of available resources of each type. If available[i] = k there are k instances of resource type R_j available

Max: An n x m matrix defines the maximum demand of each process. If Max[i,j] = k, then process P_i may request at most k instances of resource type R_j.

Allocation: An n x m matrix defines the no. of resources of each type currently allocated to each process. If Allocation[i,j] = k, then process P_i is currently allocated k instances of resource type R_j.

Scanned by TapScanner

Need: An n x m matrix indicates the remaining resource need of process. If Need[i, j] = k, then process P_i may need k more instances of resource type R_j to complete its task. Note that Need[i, j] = Max[i, j] − Allocation[i, j].

→ Safety Algorithm

The algm for finding out whether or not a s/m is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work := Available & Finish[i] := false for i = 1, 2, ... n.
2. Find an i such that both
   a. Finish[i] = false
   b. $Need_i \leq Work$. If no such i exists, go to step 4.
3. Work := Work + Allocation.
   Finish[i] := true
   go to step 2
4. If Finish[i] = true for all i, then the s/m is in a safe state.

→ Resource-Request Algorithm

Let $Request_i$ be the request vector for process P_i. If $Request_i[j] = k$, then process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the follo. actions are taken.

1. If $Request_i \leq Need_i$ go to step 2. Otherwise raise an error condition, since the process has exceeded its max. claim.

Scanned by TapScanner

2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since the resources are not available.

3. Have the s/m pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$Available := Available - Request_i$

$Allocation_i := Allocation_i + Request_i$

$Need_i = Need_i - Request_i$

Example

A s/m with 5 processes P_0 through P_4 and 3 resource types A, B, C. Resource type (A) has (10) instances, resource type (B) has (5) instances, and resource type (C) has (7) instances. At time T_0, the following snapshot of the s/m.

	Allocation (A B C)	Max (A B C)	Available (A B C)
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

$\Rightarrow Need = Max - Allocation.$

	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Scanned by TapScanner

Safe sequence

$\langle P_1, P_3, P_4, P_2, P_0 \rangle$

satisfies safety criteria.

Now process P_1 requests

$Request_1 = (1,0,2)$

- first check $Request_1 \leq Available$

$(1,0,2) \leq (3,3,2)$

new state

	Allocⁿ	Need	Available
	ABC	ABC	ABC
P_0	010	743	230
P_1	(200+102) 302	020	
P_2	302	600	
P_3	211	011	
P_4	002	431	



P_0 1) Work = Available = 332
finish = false

2) finish[0] = false → true
Need$_0$ = 743 ≤ 332 (work) → false
goto step 4

4) finish[i] ≠ true
not in safe state.

1) work = Available = 332, finish = false;

2) finish$_1$ = false & Need$_1$ ≤ 122 ≤ 332
true

3) Work = 332 + 200 = 532
finish ≠ true ; goto 2

P_2 2) finish$_2$ = false — false
Need = 600 ≤ 532 → false

4) not in safe state.

P_3 2) finish$_3$ = false
Need$_3$ = 011 ≤ 532
work = 532 + 211 = 743

P_4 2) finish$_4$ = false
Need$_4$ = 431 ≤ 743.

3) Work = 743 + 002 = 745

P_0 2) finish[0] = false
Need$_0$ = 743 ≤ 745

Scanned by TapScanner

Deadlock Detection

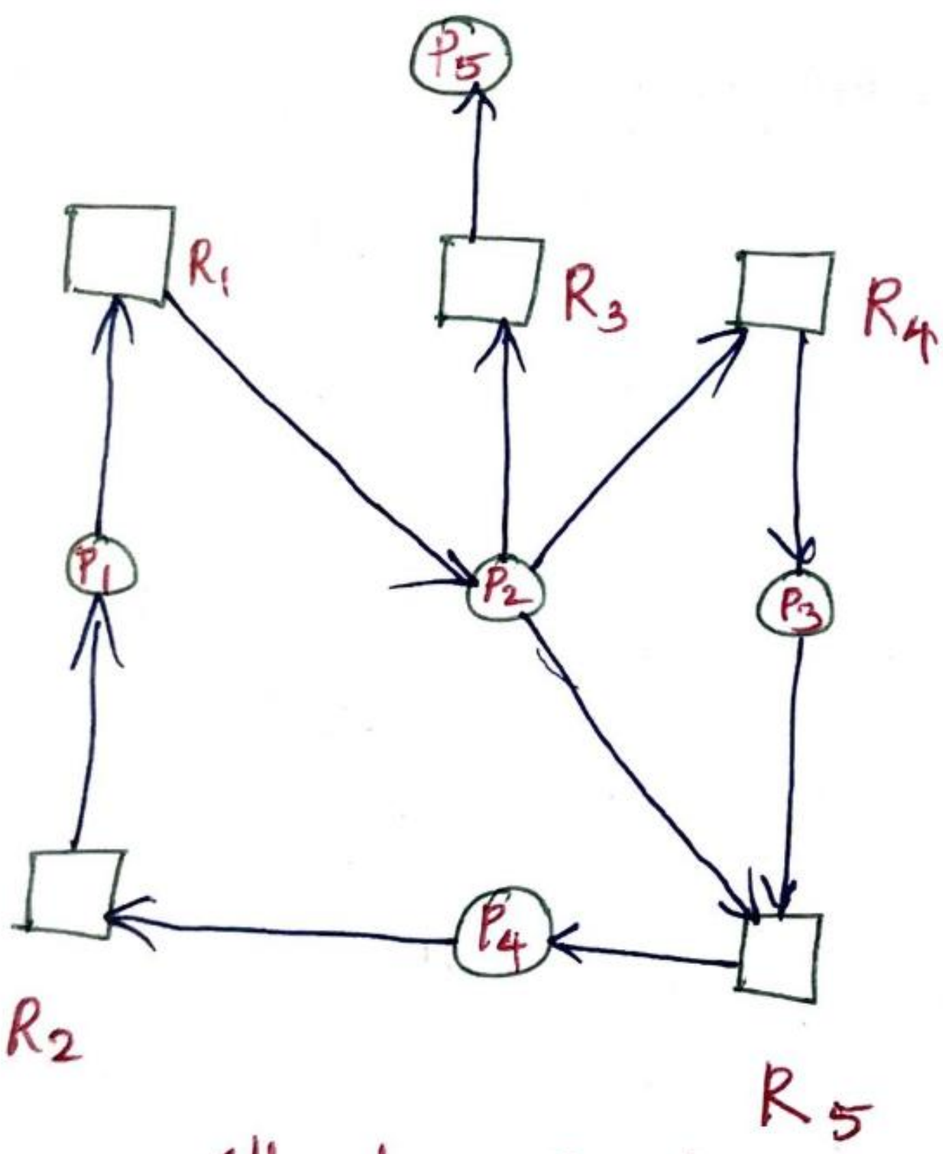
→ An algm that ~~determines~~ examines the state of the s/m to determine whether a deadlock has occured.

→ An algm to recover from the deadlock.

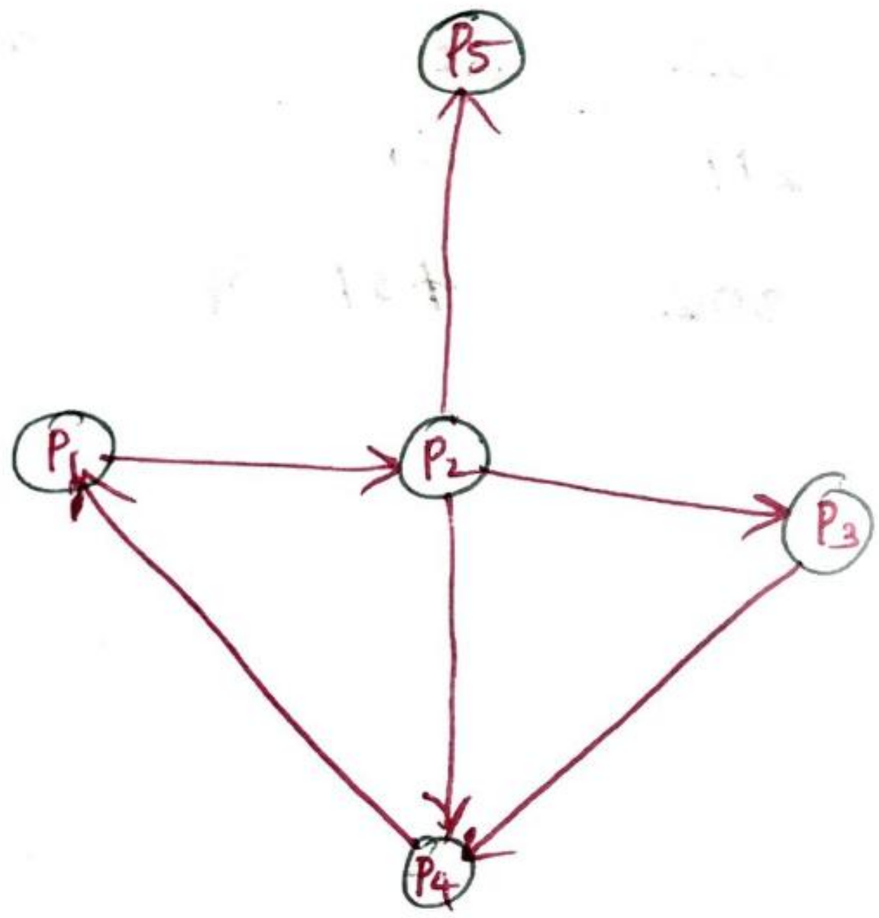
> Single Instance of Each Resource Type

Deadlock detection algm uses a variant of resource allocation graph called wait for graph.

Wait for graph



Resource Allocation Graph.



wait for graph.

→ An edge from P_i to P_j in a wait_for graph implies P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait for graph iff the corres. resource allocation graph contains 2 edges $P_i \rightarrow R_q$ & $R_q \rightarrow P_j$ for some resource R_q.

Scanned by TapScanner

Deadlock Detection & Recovery

Detection & Recovery algms ignore the distinction b/w safe & unsafe states; the resource mngrs are allowed to allocate units whenever they are available. If processes seem to be blocked on resources for an inordinately long time, the detection algm is executed to determine whether the current state is a deadlock. Detection algms make no predictions about states that could be reached from the current state.

Resources have been defined as "anything a process needs to proceed". This need could be a block of primary memory, a file or exclusive access to a device. Such resources are called serially reusable resources b/c a process requests the OS to allocate units of the resource exclusively to the process & the process will later release the units for another process to ~~use it~~ use.

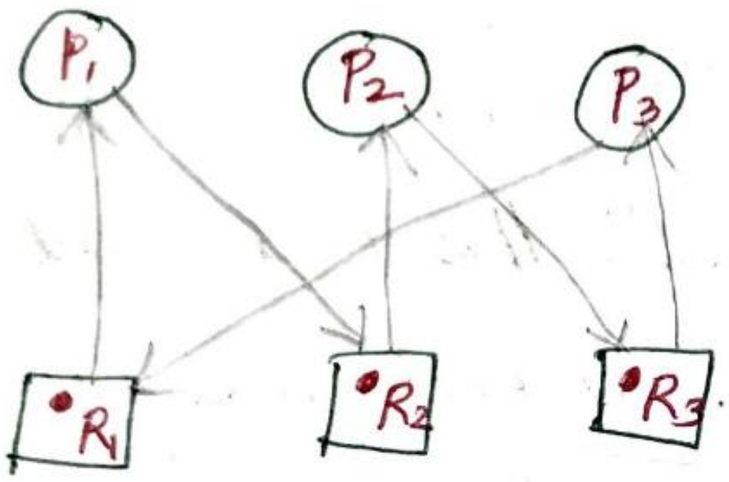
Processes can also use consumable resources. For eg:, when a process blocks on an opⁿ that is attempting to read the next character from keyboard, then the character is needed by the process in order to proceed, so it is a resource. However once the process acquires the character resource, it will never "release" it.

Scanned by TapScanner

Serially Reusable Resources

Serially reusable resources represent traditional h/w resources and their abstractions.

* The no. of units of resource, C_j is constant.
* Each unit of R_j is either available or allocated to one process.
* A unit of R_j can be released only if it was previously acquired.



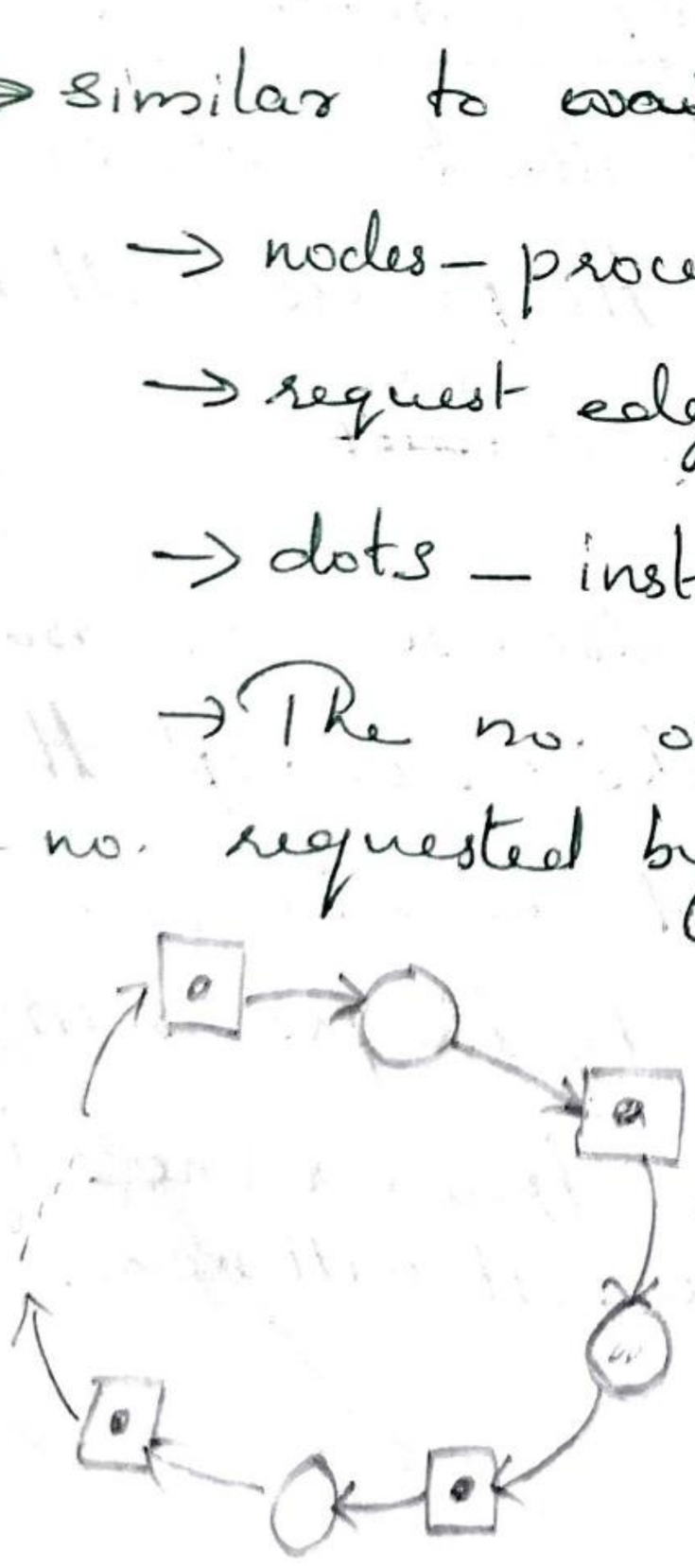
→ similar to await Resource Allocation Graph

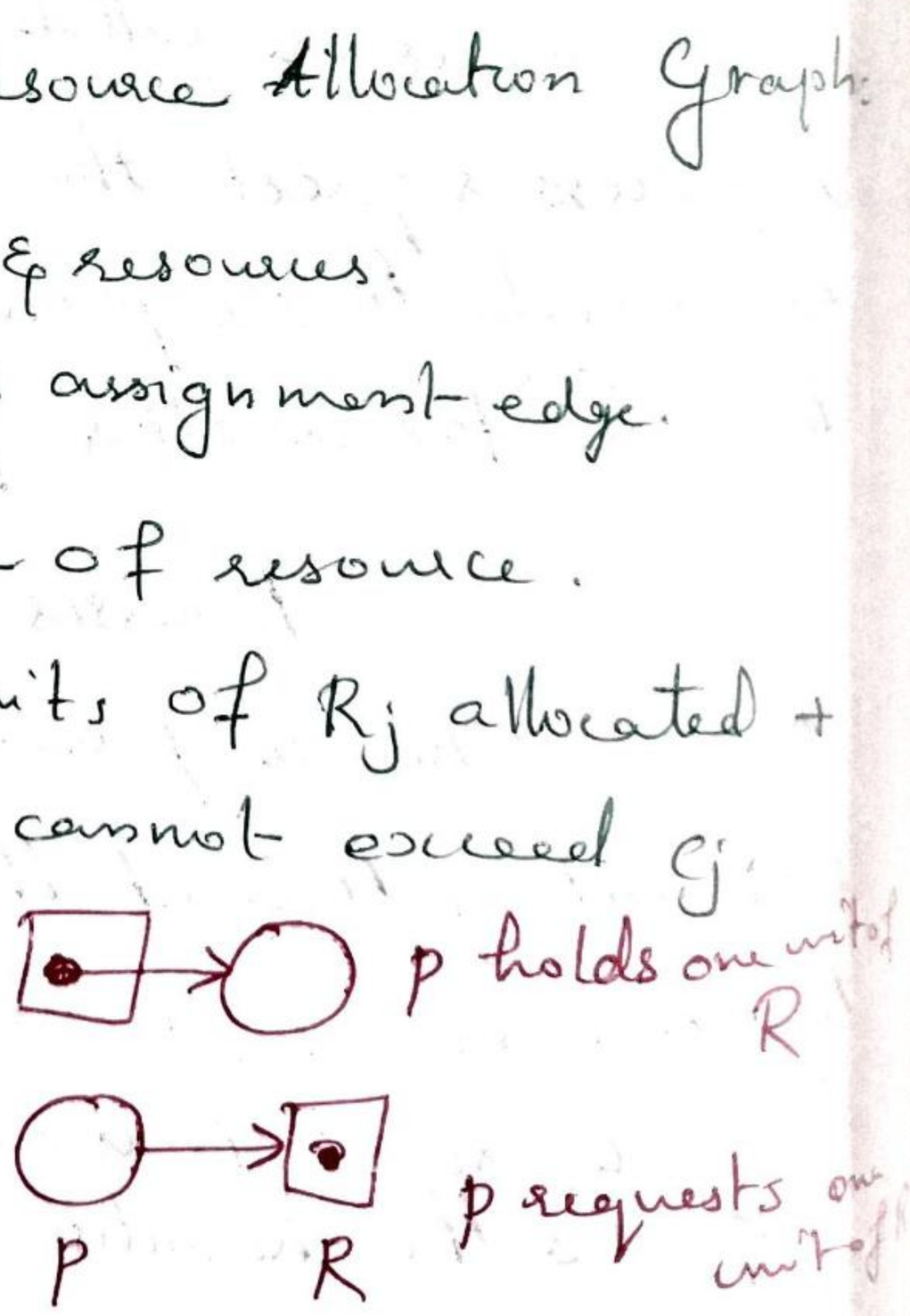
→ nodes – processes & resources.

→ request edge & assignment edge.

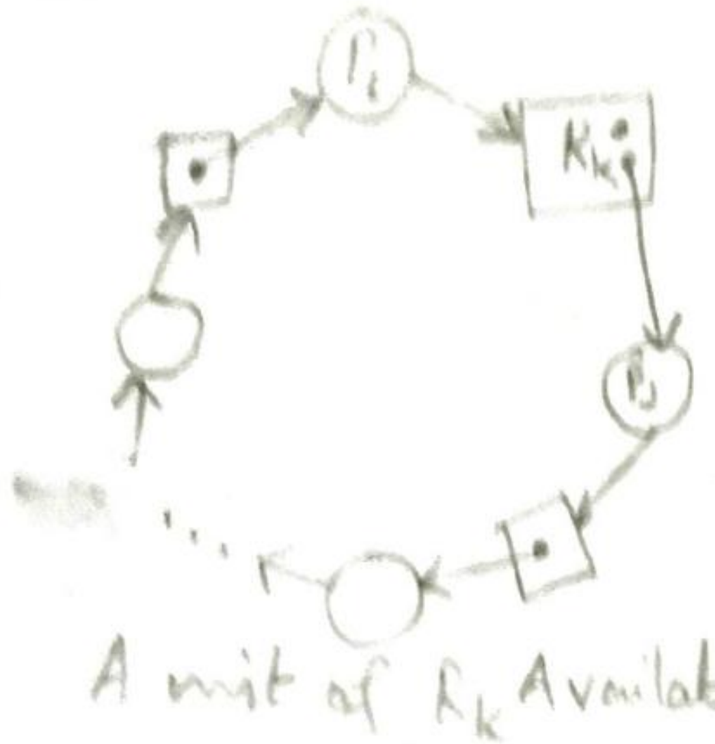
→ dots – instance of resource.

→ The no. of units of R_j allocated + the no. requested by P_i cannot exceed C_j.



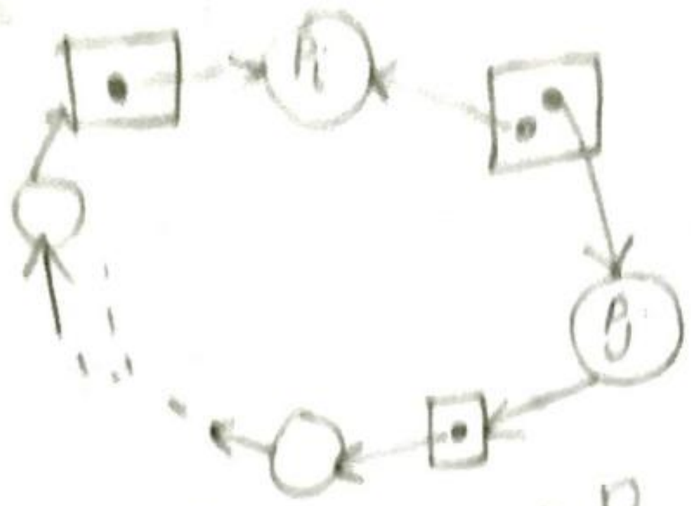


Scanned by TapScanner



A unit of R_k Available.

P_i is requesting a unit of R_k.



All units of R_k allocated

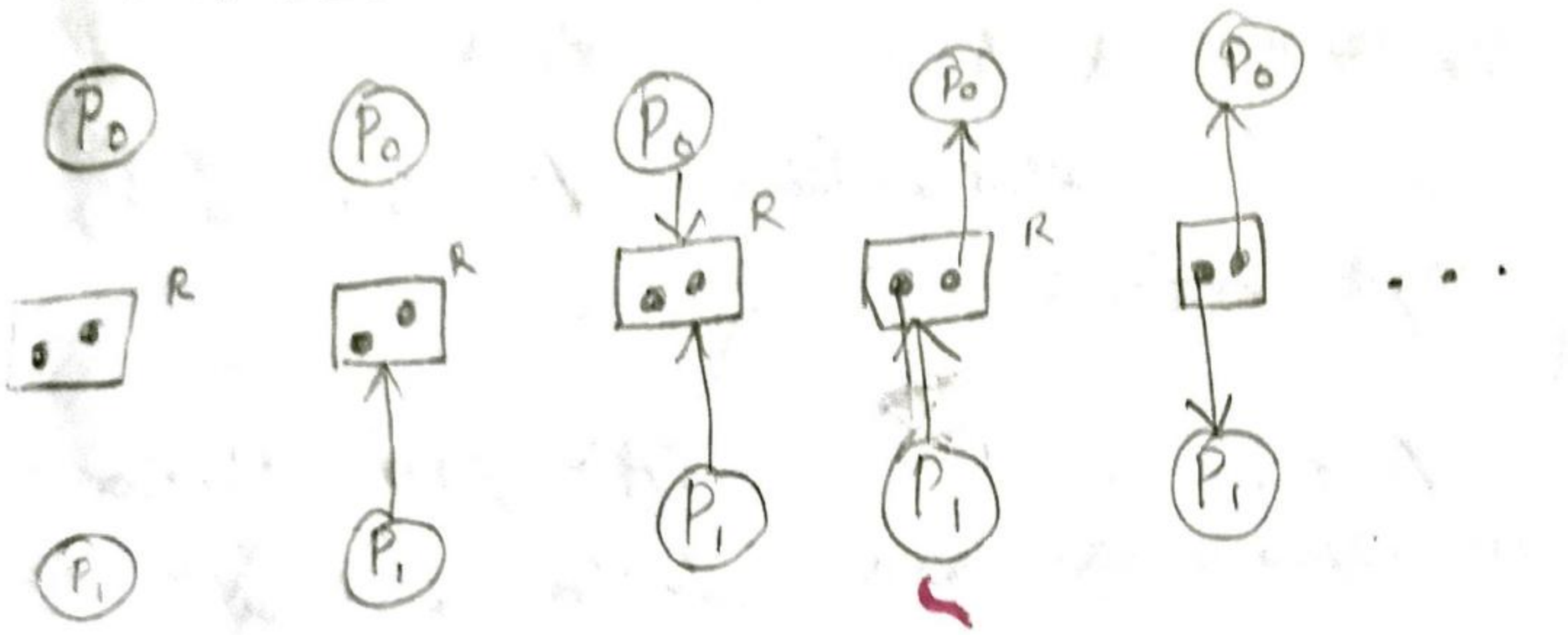
The available unit of R_k has been allocated to P_i

S/m state transition: A state transition occurs whenever one of three events occurs:

→ Any allocated resource is released through the deallocn event

→ A new resource is rqsted via the reqst event.

→ A resource is allocated to a process with the allocate event.



$S_{00} \xrightarrow{r_1} S_{01} \xrightarrow{r_0} S_{11} \xrightarrow{q_0} S_{21} \xrightarrow{q_1} S_{22} \cdots \rightarrow S_{33}$

Scanned by TapScanner

Recovery from Deadlock

There are 2 options:

→ abort one or more processes.

→ preempt some resources.

1. Process Termination:

* Abort all ~~processes~~ deadlocked processes

This method clearly will break the deadlock cycle, but at a great expense.

* Abort one process at a time till the deadlock~~ed~~ cycle is eliminated.

1. What the priority of process is
2. How long computation.
3. How many & what type of resources the process has used.
4. How many more resources need.
5. How many processes will need to be terminated

2. Resource Preemption:

* Selecting a victime
* Roll back
* Starvation.

Scanned by TapScanner

Interprocess Communication

In situations, where the threads are in different processes — there is no shared address space — the OS must assist the threads in sharing inf^n.

IPC — way for a thread in one process to share inf^n with threads in other processes — even processes that exist on a diff. m/c.

⟶ The sending m/c's OS will copy the inf^n from sender's address space to a comm^n device. The device will transmit the inf^n to a comm^n device into the address space of the receiving process.

⟶ The OS on the receiving process's m/c can copy the inf^n from its comm^n device into the address space of the receiving process.

* Pipe Model

⟶ FIFO implemented in the kernel.

⟶ read end & write end.

write() — function to have the OS place data into the pipe.

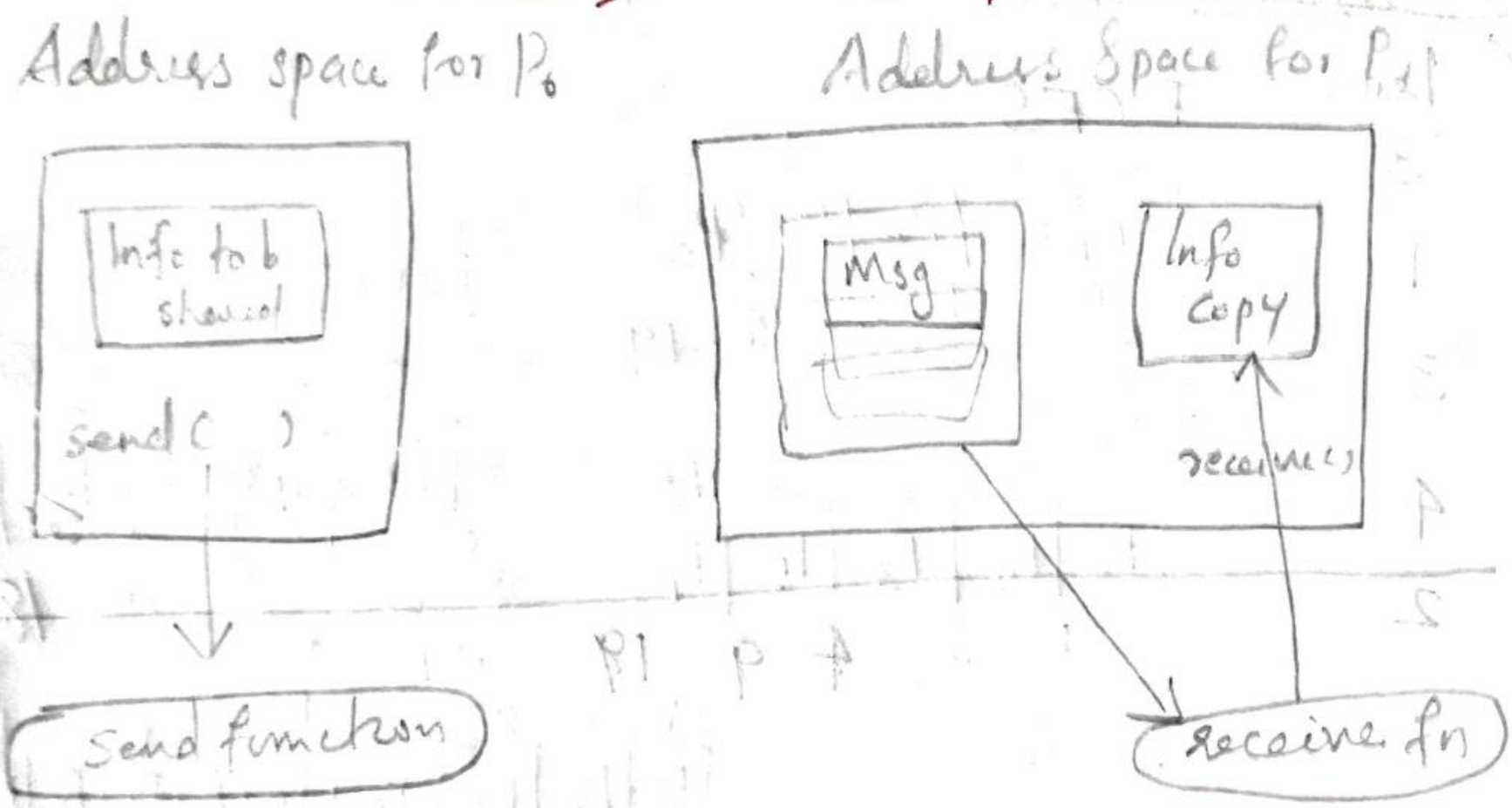
read() — fn. to remove data from the pipe.

Scanned by TapScanner

Named pipes: process obtains a pipe end by using a string that is analogous to a file name, but which is bound to a pipe. This allows any set of processes to exchange info using a "public pipe" whose end names are filenames.

Message Passing Mechanisms.

↳ Mail boxes.
↳ Send() & Receive()



The OS can store incoming msgs in the recepient's mailbox buffer prior to copying them into the receiver's address space.

A mailbox can be used to prevent info from spontaneously appearing a recipient's addr. space. When the OS receives a msg for a particular process, it stores the msg in the addr. space in the mail box area.

Scanned by TapScanner

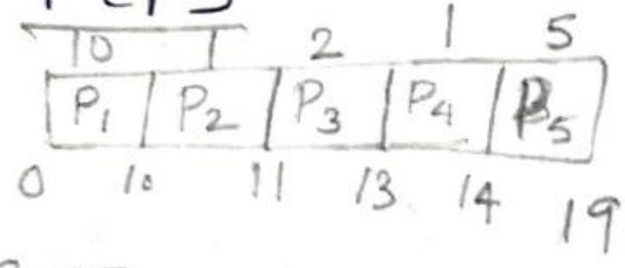
Message Protocols

For the msg to have meaning to the receiving process, there must be agreement b/w the sending & receiving processes regarding the format used to store inf^n by the sending process & interpret inf^n by the receiving process.

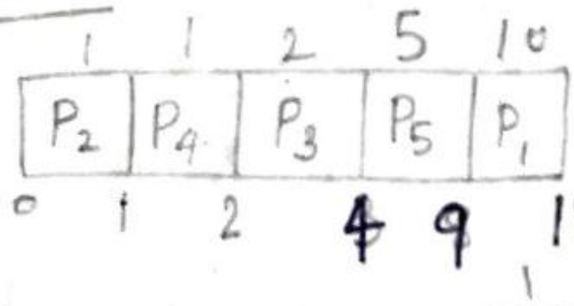
~~to there must be a protocol~~

	Btime	Prio.
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

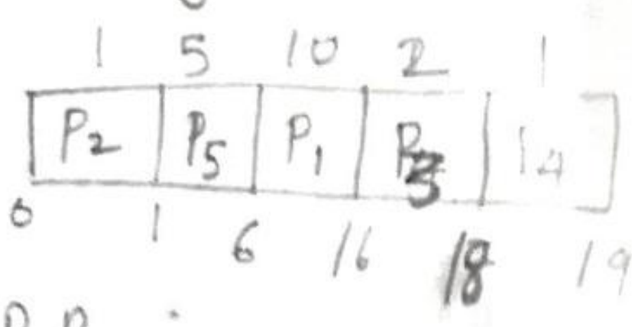
FCFS



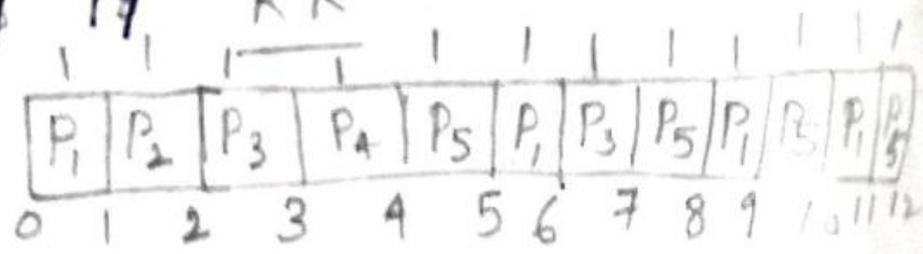
SJF

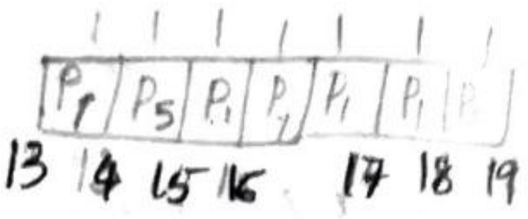


Priority



RR





FCFS

Waiting time — $P_1 - 0$, $P_2 - 10$, $P_3 - 11$, $P_4 - 13$, $P_5 - 14$

Avg — 9.6

TAT — $P_1 - 10$, $P_2 - 11$, $P_3 - 13$, $P_4 - 14$, $P_5 - 19$

13.4

SJF

$P_1 - 9$, $P_2 - 0$, $P_3 - 2$, $P_4 - 1$, $P_5 - 4$

3.8

$P_1 - 19$, $P_2 - 1$, $P_3 - 4$, $P_4 - 2$, $P_5 - 9$

6.6 7

Priority

$P_1 - 6$, $P_2 - 0$, $P_3 - 16$, $P_4 - 18$, $P_5 - 1$

8.2

$P_1 - 16$, $P_2 - 1$, $P_3 - 18$, $P_4 - 19$, $P_5 - 6$

12

R-R

~~$P_1 - 0+5+8+$~~

$P_1 - 0+4+2+1+1+1 = 9$

$P_2 - 1+$

$P_3 - 2+3 = 5$

$P_4 - 3$

$P_5 - 4+2+1+1+1 = 9$

27/5 = 5

TAT = 9.2

Scanned by TapScanner

MODULE - III

Memory Management

The memory system includes all parts of the computer that store information.

Primary Memory: (executable memory) holds inf^n while it is being used by the CPU. Primary Mem. can be referenced one byte at a time, has relatively fast access time, & is usually volatile. (it loses its conts. when the comp. crashes or is turned off.)

Secondary Memory: is referenced as block of bytes, has relatively slow access time, & is persistent (non-volatile)

Modern memory mngrs automatically transfer inf^n back & forth blw the primary & 2° memories using virtual memory technology.

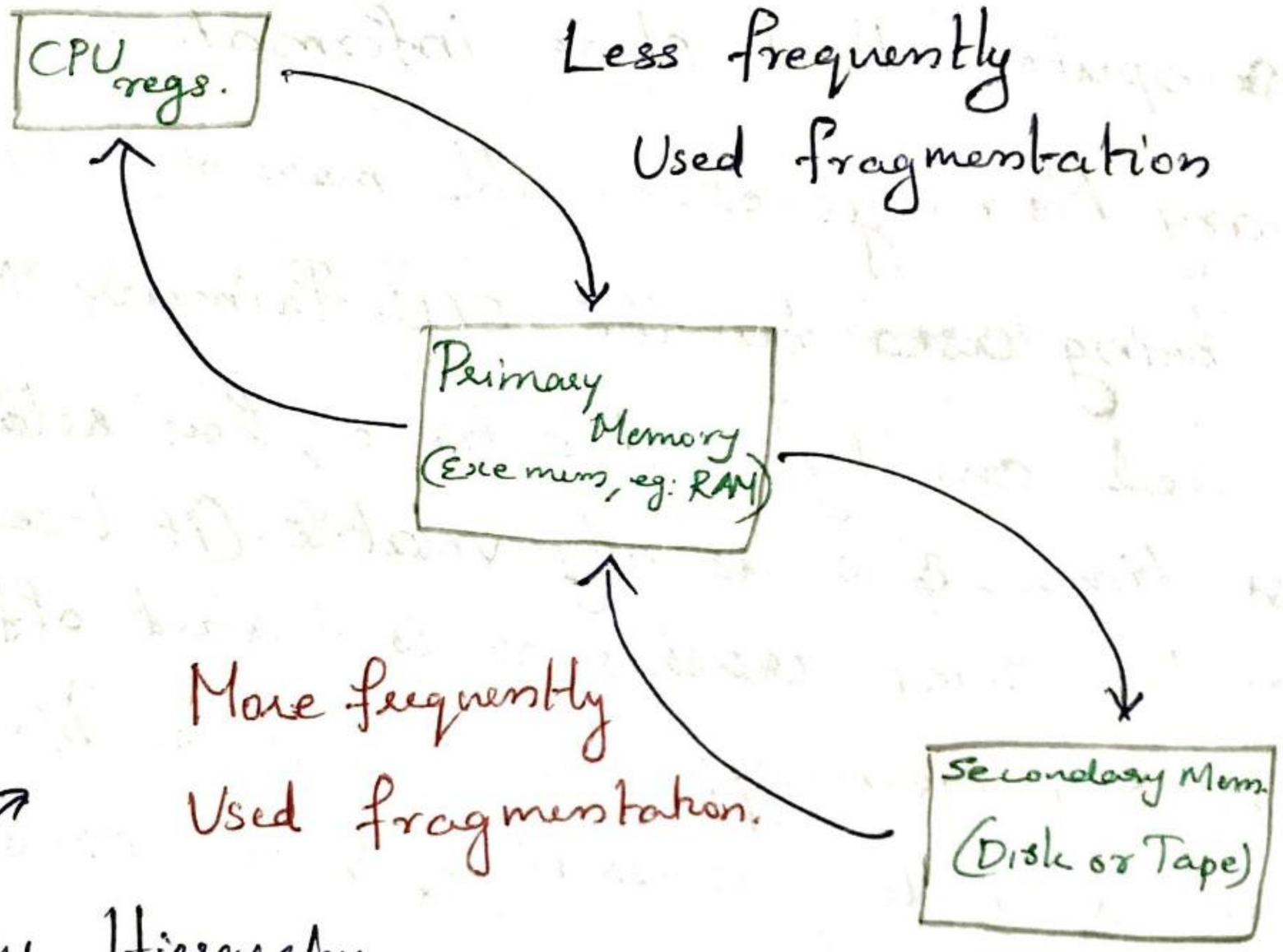
Storage Hierarchy:

The highest level is CPU reg. memory, the middle level is the primary memory & the lowest level is secondary memory (storage devices)

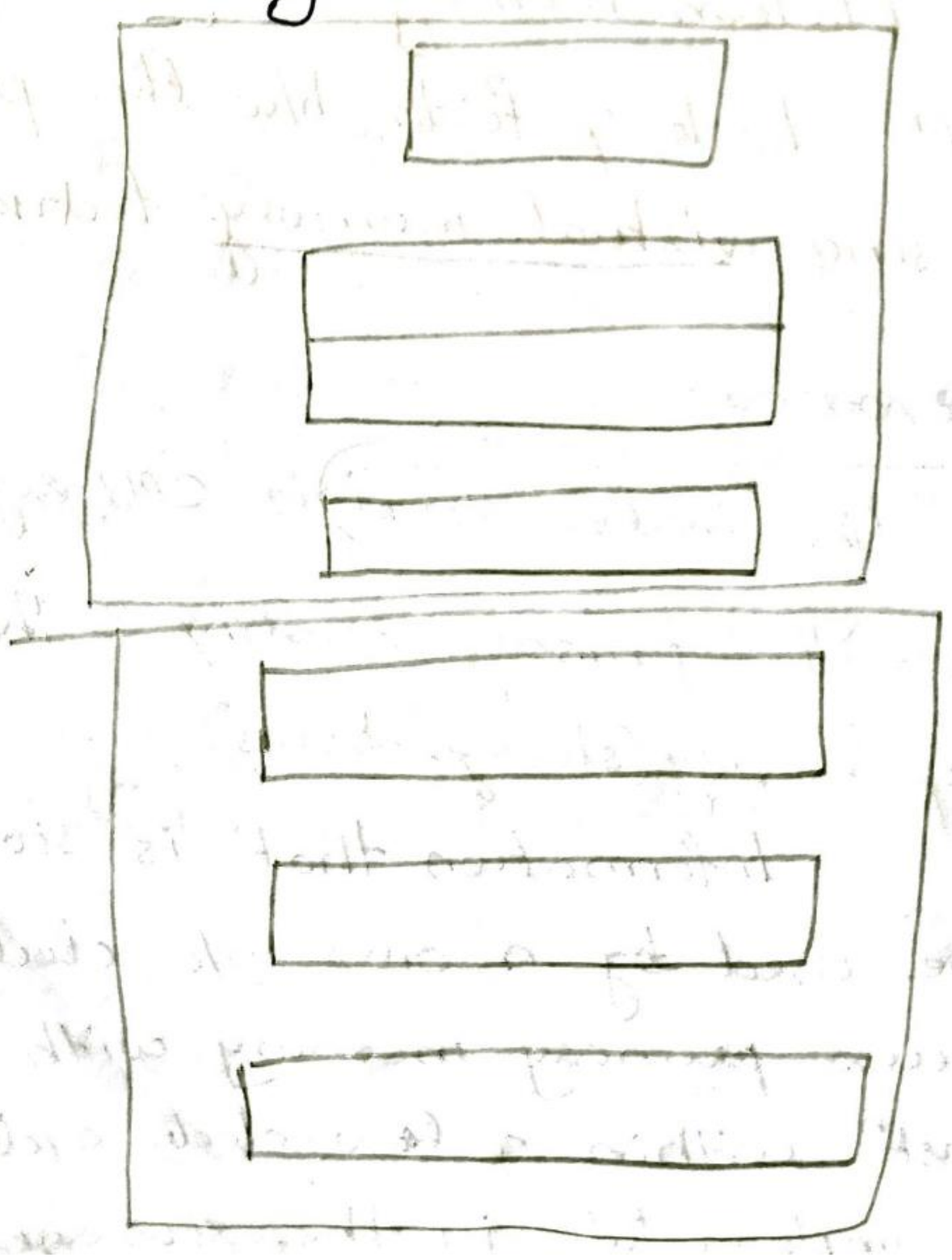
Information that is stored in a CPU reg. can be used ~~by~~ in one m/c clock cycle. The CPU can access primary memory with a single load or store inst^n within a few clock cycles. Secondary memory is implemented in the storage devices, so

Scanned by TapScanner

access involves action by a driver & phy. device. This means that secondary accesses take at least 3 orders of magnitude more time than do primary mem. accesses.



Memory Hierarchy



Scanned by TapScanner

Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory acc. to the value of the pgm counter.

Instruction execution cycle, first fetches an inst^n from memory. The inst^n is then decoded and may cause operands to be fetched from memory. After the inst^n has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses

~~How~~ Sequence of memory addresses generated by the running pgm.

Address Binding: (Process Address Space Abstraction)

A program resides on a disk as a binary executable file. The program must be braught into memory and placed within a process for it to be executed. The process may be moved blw disk & memory during its execution. The collection of processes on the disk that is waiting to be brought into memory for ex^n forms the input queue.

The normal procedure is to select one of the processes in the input queue & load that process into memory. The process terminates, and its memory space is declared available.

Scanned by TapScanner

Binding of instructions & data to memory addresses can be done at any step along the way.

* Compile time: If you know at compile time where the process will reside in memory, then absolute code can be generated. For eg: if you know that a user process resides starting at location R, then the generated compiler code will start at the location & extend up from there.

* Load time: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code. In this case, final binding is delayed until load time. If the starting address changes, we need only to reload the user code to incorporate this changed value.

* Execution time: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Logical versus Physical Address Space

An address generated by the CPU is commonly referred to as a logical address. An address seen by the memory unit, i.e; loaded into mem. addr. register of the memory is referred to as a ~~memory address~~ physical address

Scanned by TapScanner

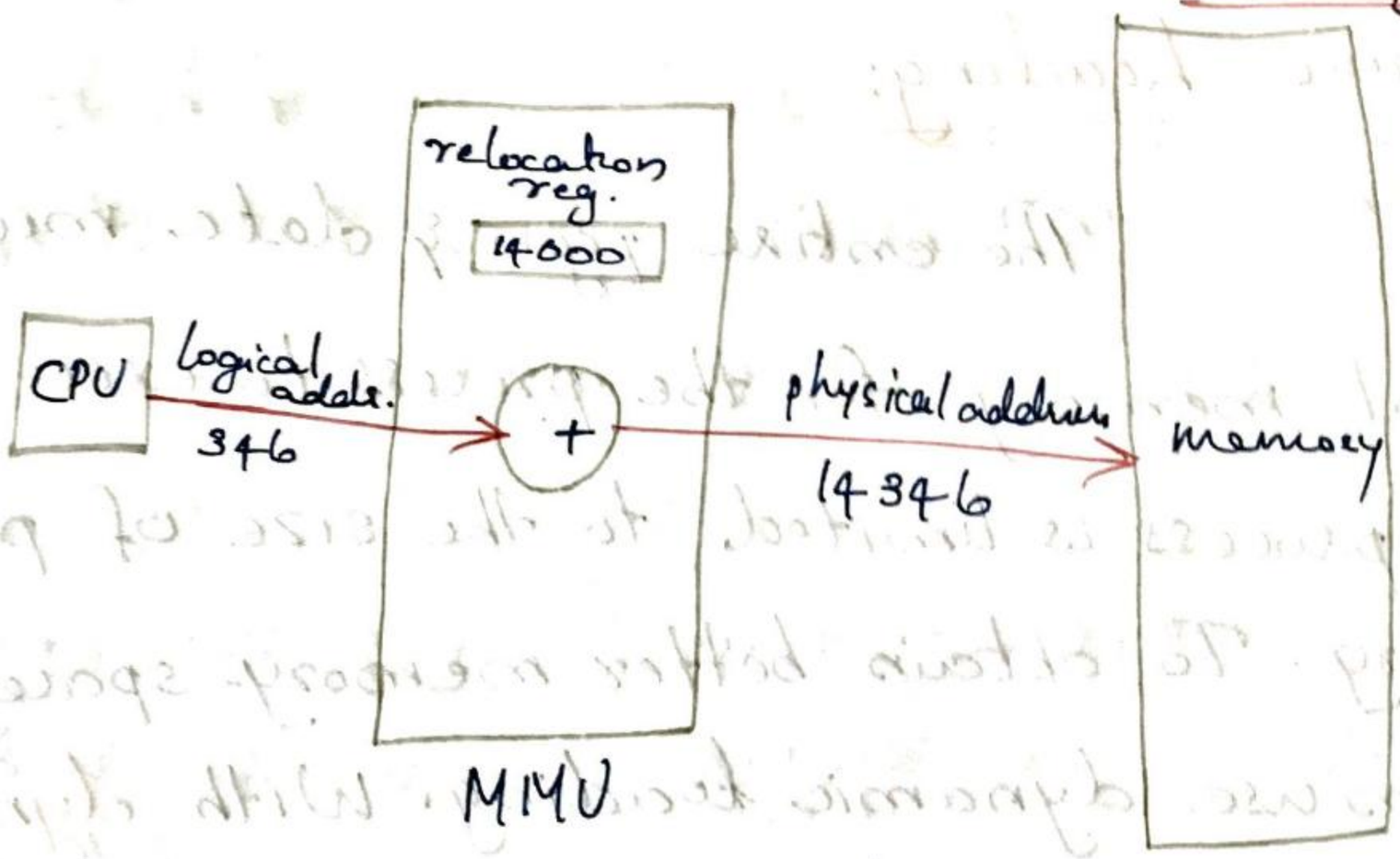
The compile time & load time address binding methods generate identical logical & physical addresses.

Execution time address binding scheme results in differing logical & physical addresses - refer to the logical address as a virtual address.

The set of all logical addresses generated by a programs is logical address space.

The set of all physical addresses corresponding to these logical addresses is a physical address space

The runtime mapping from virtual to physical addresses is done by a h/w device called the Memory Mgmt Unit(MMU)



Dynamic relocation using a relocation register.

The base register is called a relocation register. The value in the relocation reg. is added to every address generated by a user process at the time it is sent to memory.

Scanned by TapScanner

The user program deals with logical addresses. The memory mapping hlw converts logical addresses into physical addresses. The memory mapping hlw converts logical addresses into physical addresses.

Logical addresses (range 0 to max) – The user generates logical addresses & thinks the prouss runs in locations 0 to max.

Physical addresses (range R+0 to R+max).

The logical address space that is bound to a separate phy. addr. space.

Dynamic Loading:

The entire pgm & data must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. To obtain better memory-space utilizatn we can use dynamic loading. With dynamic loading a routine is not loaded until it is called. All routines are kept in ~~the~~ disk in a relocatable load format. The main pgm is loaded into memory & is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If not, the relocatable linking loader is called to load the desired routine into memory. Then the ctrl is passed to the newly loaded routine.

Scanned by TapScanner

Advantage: an unused routine is ~~not~~ never loaded
Although the total pgm size may be large, the portion that is used may be smaller.

Dynamic loading doesn't require special support from the operating system. It is the responsibility of the users to design their pgms to take advantage of such a method.

Dynamic Linking & Shared Libraries:

Static linking: ~~Some OS~~ in which s/m lang. libraries are treated like any other object module and ~~treated~~ are combined by the loader into the binary pgm image.

With dynamic linking, a stub is included in the image for each library routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory resident library routine, or how to load the library if the routine is not already present.

When this stub is executed, it checks to see whether the needed routine is already in memory. If not, the pgm loads the routine into memory. Either way stub replaces itself with the address of the routine, and executes the routine.

This feature can be extended to library updates. A library may be replaced by a new version

Scanned by TapScanner

and all pgms that reference the library will automatically use the new version.

Address binding contd.

Compile time: produce a relocatable object module.

The compiler translates a source pgm into relocatable code. The relocatable object module has 3 logical blks of addresses:

* Code segment: blk of m/c inst^{n}s.
* Data segment: blk of static variables.
* stack segment: stack that will be used when the pgm is executed.

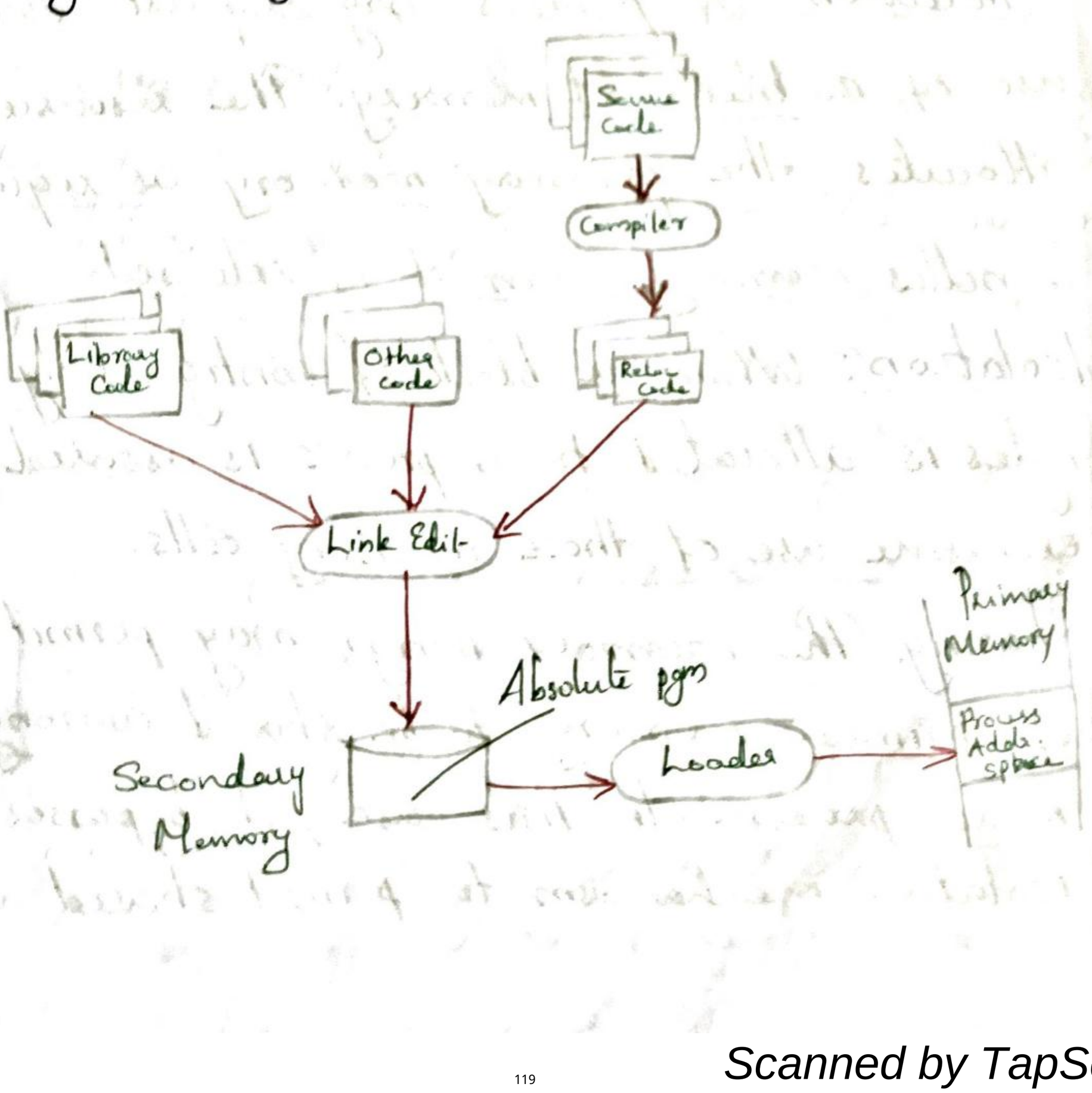
Link time: At link time, the code & data segments for each relocatable objt module are combined to form absolute pgm. Link editor combines all data segments into a single data segment. and all code segments into a single code segment. When the data segments are combined, the relative addresses of individual static variables will be changed.

The linkage editor then relocates the addresses in the inst^{n}s so that they reference the updated addresses in aggregated data segment.

Scanned by TapScanner

Load Time: Prior to loading an absolute pgm, the mem. mngr allocates a block of primary memory to the process. Then the loader copies the absolute pgm and data into this newly allocated memory.

All addresses in the absolute pgm were ~~sen~~ set by the link editor as if the module were to be loaded at primary memory location 0. The module is now to be loaded at a particular physical addr. in the primary memory: the first address in the block of memory assigned by the mem. mngr. The loader translates each logical address to the allocated primary memory addresses.



Scanned by TapScanner

The memory mngr addresses primary memory mgmt by implementing

Abstraction: The primary memory is abstracted so that s/w perceives the memory allocated to it as a large array of contiguously addressed bytes. The idea of a process address space is the main aspect of the abstraction, since it allows a process to use an abstract set of addresses to reference physical primary mem. locations.

Allocation: A process may request exclusive use of a block of memory. The resource mngr allocates the primary memory as requested, and handles memory when it is released.

Isolation: When a block of contiguously addressed bytes is allocated to a process is assured of exclusive use of those memory cells.

Sharing: The memory mngr may permit a block of primary memory to be shared among 2 or more processes. In this case, it bypasses the isolation mechanism to permit shared access.

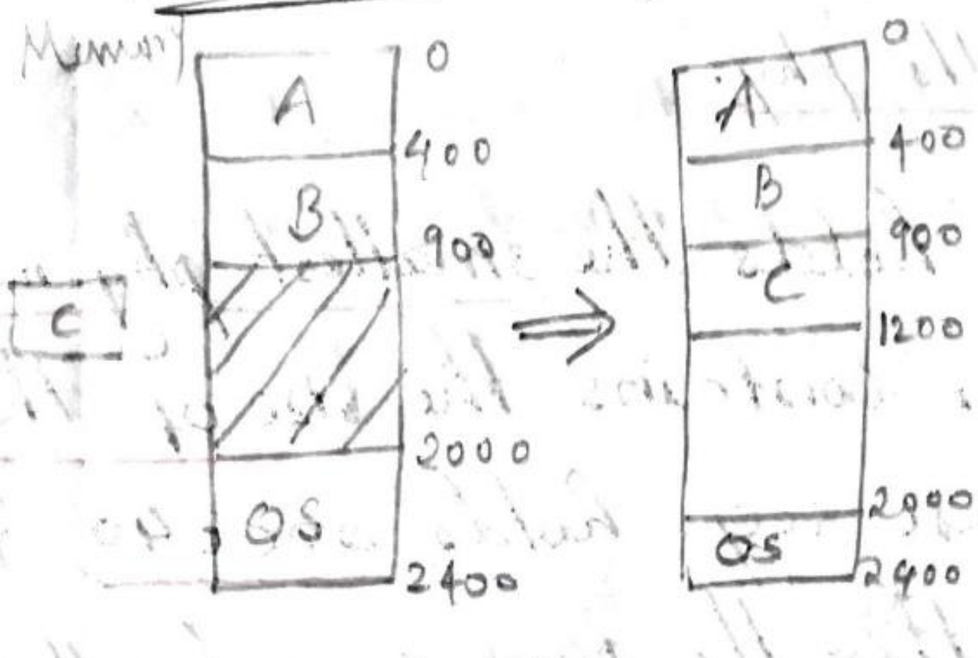
Scanned by TapScanner

Dynamic Address Space Binding: Dynamic Relocation

Relocation: is the process of assigning load addresses to various parts of a pgm and adjusting the code & data in the pgm to reflect the assigned addresses.

Each object file has various sections like code, data etc. To combine all the objects to a single executable, the linker merges all sections of similar type into a single section of that type. The linker then assigns run time addresses to each section & each symbol. Each section refers to one or more symbols which should be modified so that they point to the correct run time addresses based on info stored in a relocation table in the obj file. The relocation table is a list of pointers created by the compiler/assembler & stored in the obj or exe file.

Relocation



→ Put the OS in the highest memory

→ Compiler/Linker assumes that the process starts at 0.

→ When the OS loads the proces, it allocates a contiguous segment of mem. where process fits.

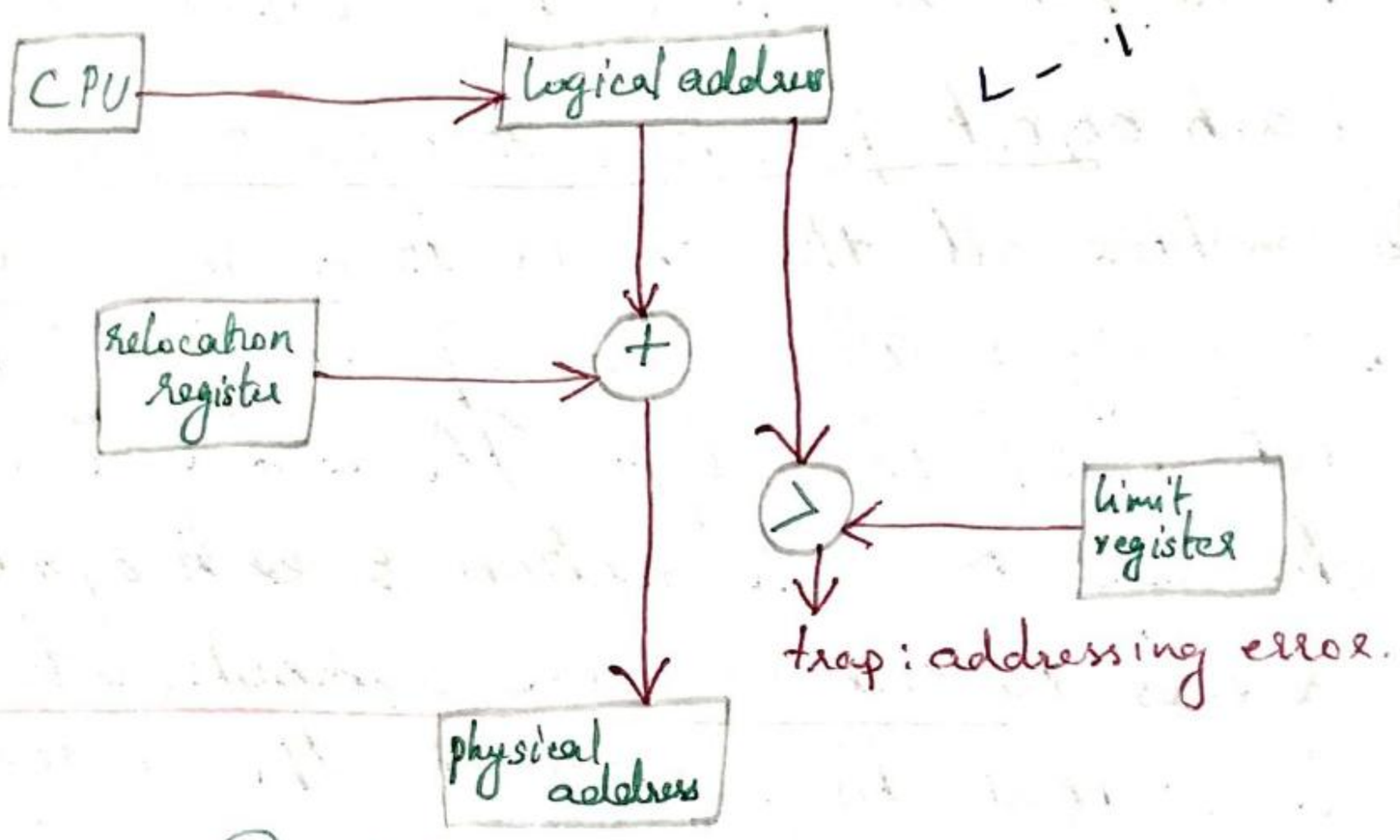
The first phy. addr. of the process is the base address and the largest phy. address the process can access is the limit address. The base address is also known as relocation address, & may be kept in a special reg.

Static Relocation: At load time, the OS adjusts the addresses in a process to reflect its posn in memory. Once a process is

Scanned by TapScanner

is assigned a place in memory & starts executing it, the OS cannot move it.

Dynamic Relocation



limit - 2000

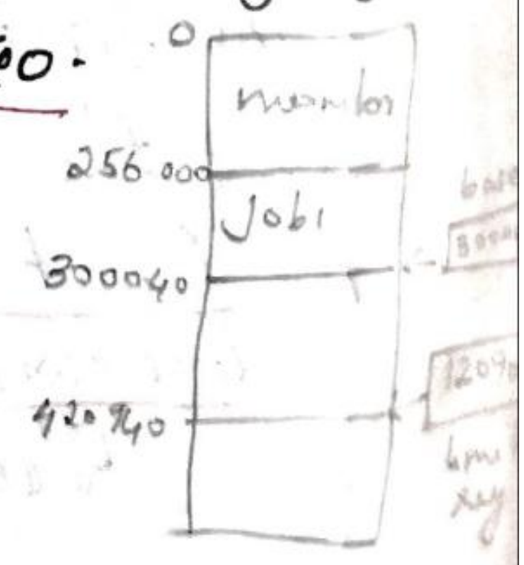
1000

L - 1

Dynamic Resolution

H/w adds relocation reg. (base) to virtual address to get a phy. address. H/w compares logical address with limit reg. address must be less than the ~~base~~ limit. If the test fails, the processor takes an address trap & ignores the phy.

The base register holds the smallest phy. mem. address; the limit register contains the size of the range. For eg: if the base reg. holds 300040 & limit register is 120900, then the pgm can legally access all addresses from 300040 through 420940.



Scanned by TapScanner

Memory Allocation Strategies

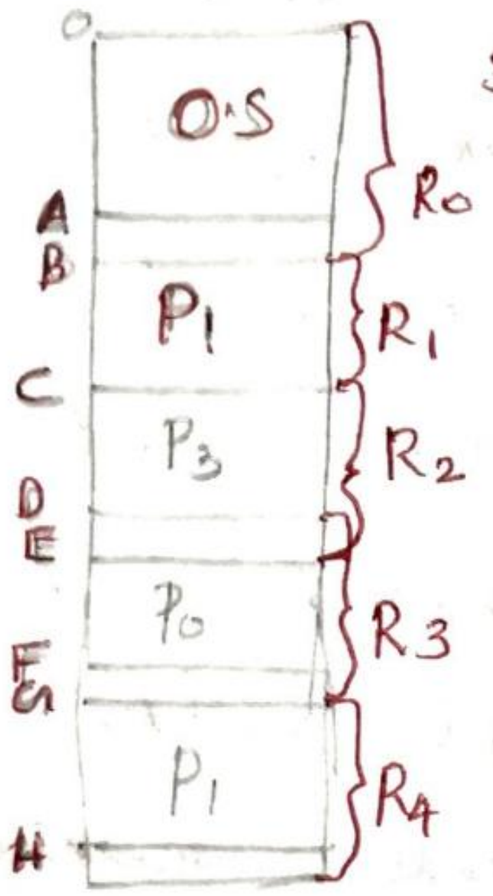
One of the simplest methods for memory allocation is to devide memory into several fixed sized partitions. Each partition may contain exactly one process. The degree of multiprogramming is bound by the no. of partitions. In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

The O.S keeps a table indicating which parts of memory are available (hole) and which are occupied

Input queue → free space → allocated space → CPU → release

In general, a set of holes, of various sizes, is scattered throughout memory at any given time. When a process arrives and needs memory, the s/m searches for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process, the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

Scanned by TapScanner



Snapshot of memory allocn.

$R_0 \rightarrow$ 0 to B
$R_1 \rightarrow$ B to C
$R_2 \rightarrow$ C to E
$R_3 \rightarrow$ E to G
$R_4 \rightarrow$ G to last addr.

Memory allocation strategies are separated into those devide primary memory into a fixed no. of fixed size blocks at the time the O·S is configured, and those that use dynamically determined, variable sized blocks.

Memory fragmentation: The memory mngr could allocate every single byte of memory to a process if any process needs memory. Parts of memory, (mem. fragmts) cannot be used at any given time b/c the mem. mngr is unable to allocate these parts in an efficient manner.

Fixed - Partition Memory Strategies:

Primary memory is statically devided into N fixed-size regions or partitions, where region R_i has N_i units of memory. $N_i \neq N_j$ (the regions are diff. sizes), so processes with small address spaces use small partitions and processes with large address spaces use large partitions.

Scanned by TapScanner

Memory allocation in this kind of s/m requires that a process's address space be less than or equal to the size of an allocated partition.

For eg:, if a process has an address space size of k, then it can be loaded into any R_i, where $N_i \geq k$. Upon allocation of region R_i to the process, $N_i - k$ units of the primary memory are unused during the time the the process is loaded, since the space is allocated to the process but not mapped into its address space. This phenomenon is called Internal fragmentation.

Fixed_partition memory mngrs were widely used in batch multipgmming s/ms, but they are generally not suitable its for use in any s/m in which k is not known ahead of time.

Variable_Partition Memory Strategies:

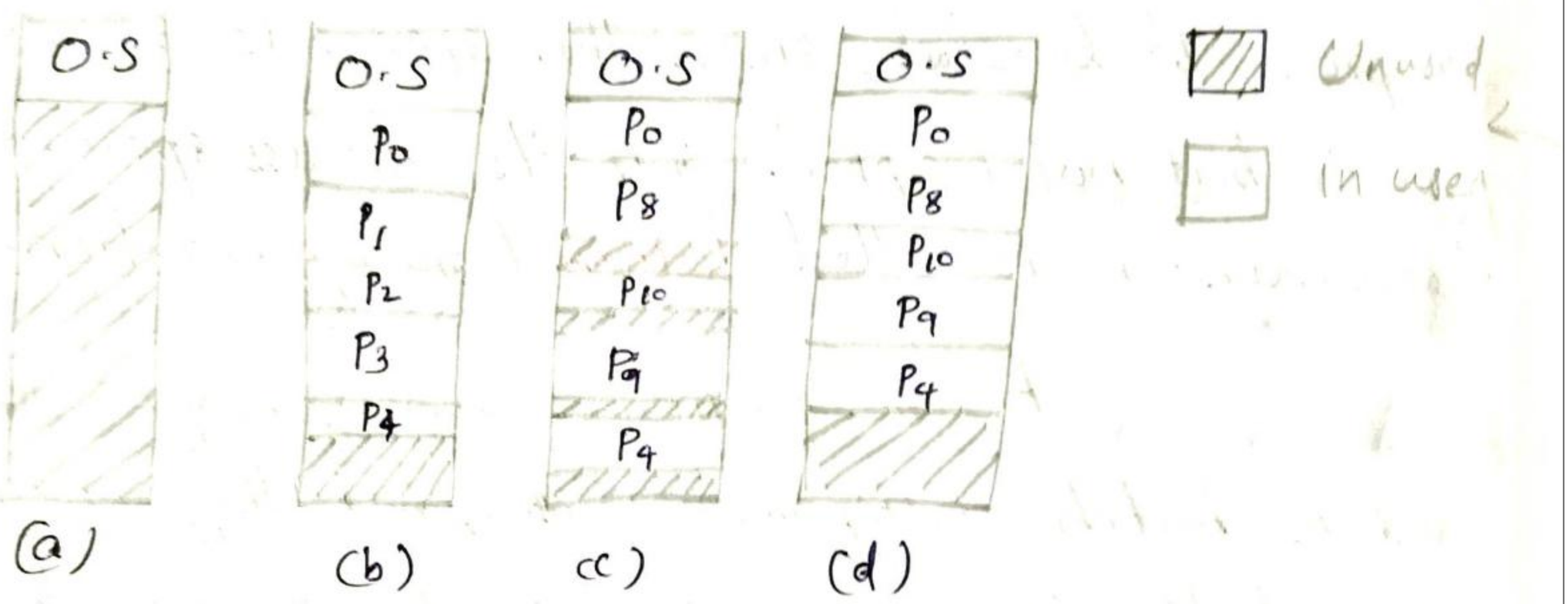
It dynamically defines regions according to the instantaneous space needs of a process. This approach removes the possibility of internal fragmentation.

The challenge for a variable_size block memory mngr is to keep track of the size of blocks of memory and to allocate them efficiently. When the s/m is initialized, the primary memory is configured as a single large block of N_0 units of memory. The scheduler allocates memory n_i units to process P_i as long as

$\sum n_i < N_0$.

Scanned by TapScanner

A small amnt of mem at the end of the memory space will be lost to external fragmentation - the part of the memory not marked as allocated to the O.S or to any P_i. When more processes can be allocated memory, the mem. mngr waits for one or more processes to release memory.



(a) Only the O·S is using memory

(b) The memory is filled by 5 processes that are adjacent in memory

(c) P_1, P_2, P_3 have released their memory and P_8, P_{10}, & P_9 have been assigned blks of memory. This causes external fragmentation for eg: blw the memory for P_8 & P_{10}

(d) The memory blks have been moved to eliminate the small memory fragments, producing a single large blk of unallocated memory. (Compaction)

Scanned by TapScanner

Dynamic Allocation

malloc
calloc

The s/m could allow a process to change the amount of memory allocated to it while it is executing, depending on its phase of computation. This scenario means that sometimes the process will request more memory than can fit in its current space + any adjacent holes.

In the above fig c. suppose Pq requests additional memory in an amount that exceeds its current space plus the space available in the holes above & below it. The memory mngr could block Pq until more adjacent space become available. But this strategy doesn't work in interactive s/ms, since the users might incur very long waits for service.

The memory mngr must keep a record of the usage of each allocatable block of memory. (linked list) The collection of unused memory can be kept in a conventional list. The mem. mngr maintains the list, adding blks when they are released, removing blocks when they are allocated & combining adjacent unused blocks.

A no. of diff. strategies can be used to allocate space to the processes that are competing for memory.

Best fit: The memory mngr places a process in the smallest block of unallocated memory in which it will fit. For eg: suppose a process requests 12KB of mem & the mem. mngr currently has a list of unallocated blocks

Scanned by TapScanner

of 6KB, 14KB, 19KB, 11KB & 13KB blocks. The best fit strategy will allocate 13KB block to the process.

Worst fit: The memory mngr places a process in the largest block of unallocated memory available. The idea is that this placement will create the largest hole after the allocations. If the list of unallocated blks consist of 6KB, 14KB, 19KB & 13KB blocks and a process requests a 12KB block, then worst fit will split 19KB block into 12KB block to allocate to the process, leaving a 7KB Block for future use.

First fit: If there are many holes in the memory, the mem. mngr begins transversing the list from the begining & allocates mem. from the first hole it encounters that is large enough to satisfy the request.

If the list of unallocated blocks consists of 6KB, 14KB & 19KB, 11KB and 13KB blocks & a process requests a 12KB, first fit will allocate the 14KB block to process.

Next fit: ~~It considers~~ It converts the list into a circular list, with the last block with the last block ptrng to the first free one.

Scanned by TapScanner

eg: 6KB, 14KB, 19KB, 11KB, & 13 KB.

When a process requests a block, it begins its search where the free pointer indicates. As soon as it finds a block to allocate, it assigns the space & then adjusts the free pointer to address the new fragment, or if there is no fragment, to the block that follows the allocated block.

Here the mem. mngr had last allocated a blk from the list b/w 19 KB & 11KB block, then when a process requests a 12KB block, next fit will allocate the 13KB blk to the process.

Swapping

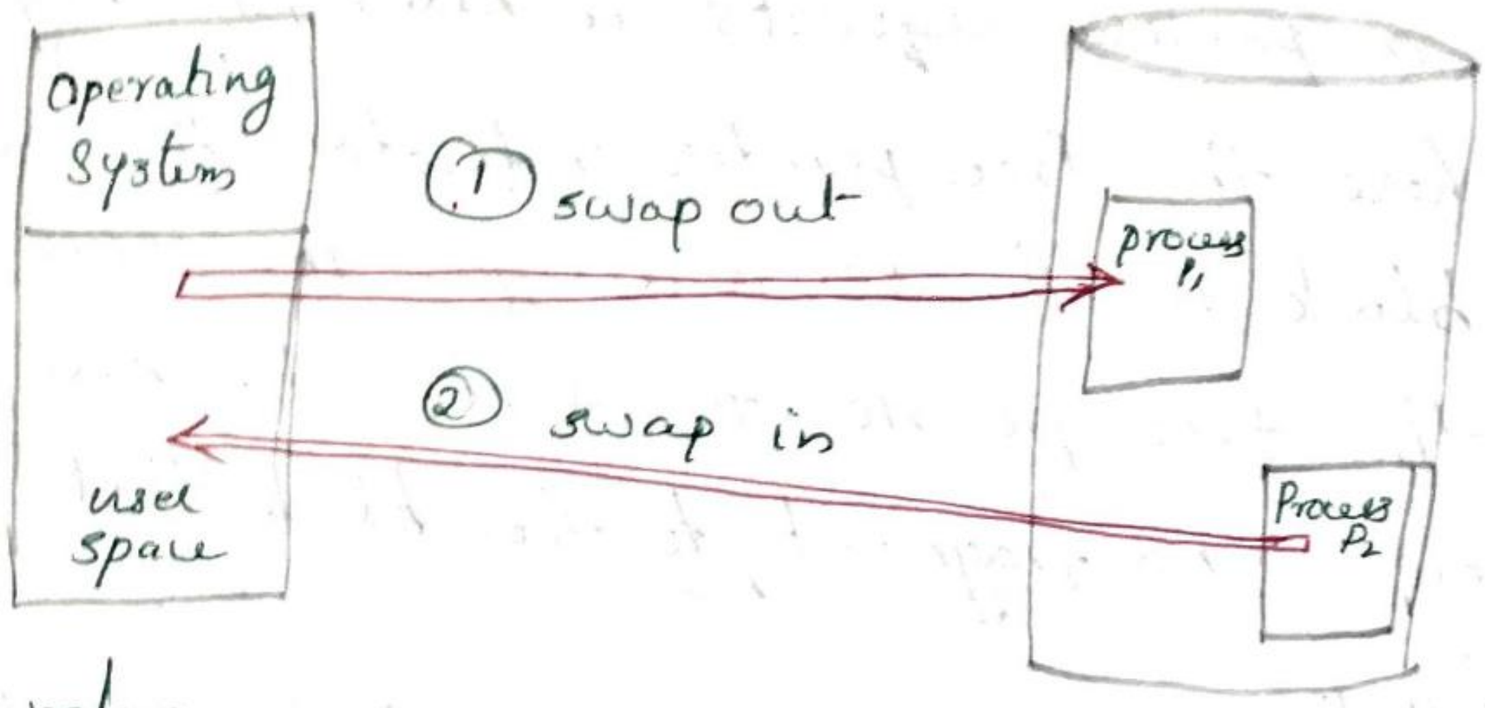
Swapping is mainly used in s/ms in which there is only a single thread process. They attempt to optimize s/m performance by removing a process from primary mem. when its thread is blocked, deallocating the memory, allocating to other proceses, & then reaquiring & reloading the swapped out process when its thread returns to the ready state.

A process, however, can be swapped temporarily out of memory to a backing store, & then brought back into memory for continued execution.

For example, assume a multiprogramming envt with a round-robin CPU scheduling algorithm. When a quantum expires, the mem. mngr will start to swap out the process just finished & to swap in another

Scanned by TapScanner

process to the memory space that has been freed.



m/ms.

When each process finishes its quantum, it will be swapped with another process.

A variant of this swapping policy is used for priority-based scheduling algms. If a higher-priority process arrives and wants service, the mem. mngr can swap out the lower-priority process so that it can load and execute the higher priority process. When the higher priority process finishes, the lower priority process can be swapped back in & continued. This variant of swapping is sometimes called rollout, roll in.

Normally a process that is swapped out will be swapped back into the same memory space that it occupied previously. If binding is done at assembly or load time, then the process cannot be moved to different locations. If ex^n time binding is being used, then a process can~~not~~ be swapped into a diff. mem-space, b/c the phy. addresses are computed dur

Scanned by TapScanner

Disadvantage:

→ Large backing store is required.

→ Context switch time ~~is~~ is high.

What is the cost of swapping out a process?

If the process holds S units of primary memory, then we can compute the overhead time required to copy the executable image to a storage device and the time to copy it back into the primary memory when it is swapped in. If a disk block is D units of primary memory, then the mem. mngr will need to write at least $R = \lceil S/D \rceil$ disk blocks in order to save executable image. It will need to perform the same no. of read op^{n}s in order to ~~save executable image~~ swap the addr. space back into primary memory. The cost to the process is the time the process spends competing to regain primary mem. after it and enters the ready state while being swapped out.

The delay in time — the swap time, 2R, and the time to reacquire memory is all overhead.

Memory Protection

→ protecting the operating s/m from user processes, and protecting user processes from one another. This protection is provided by using a relocation register.

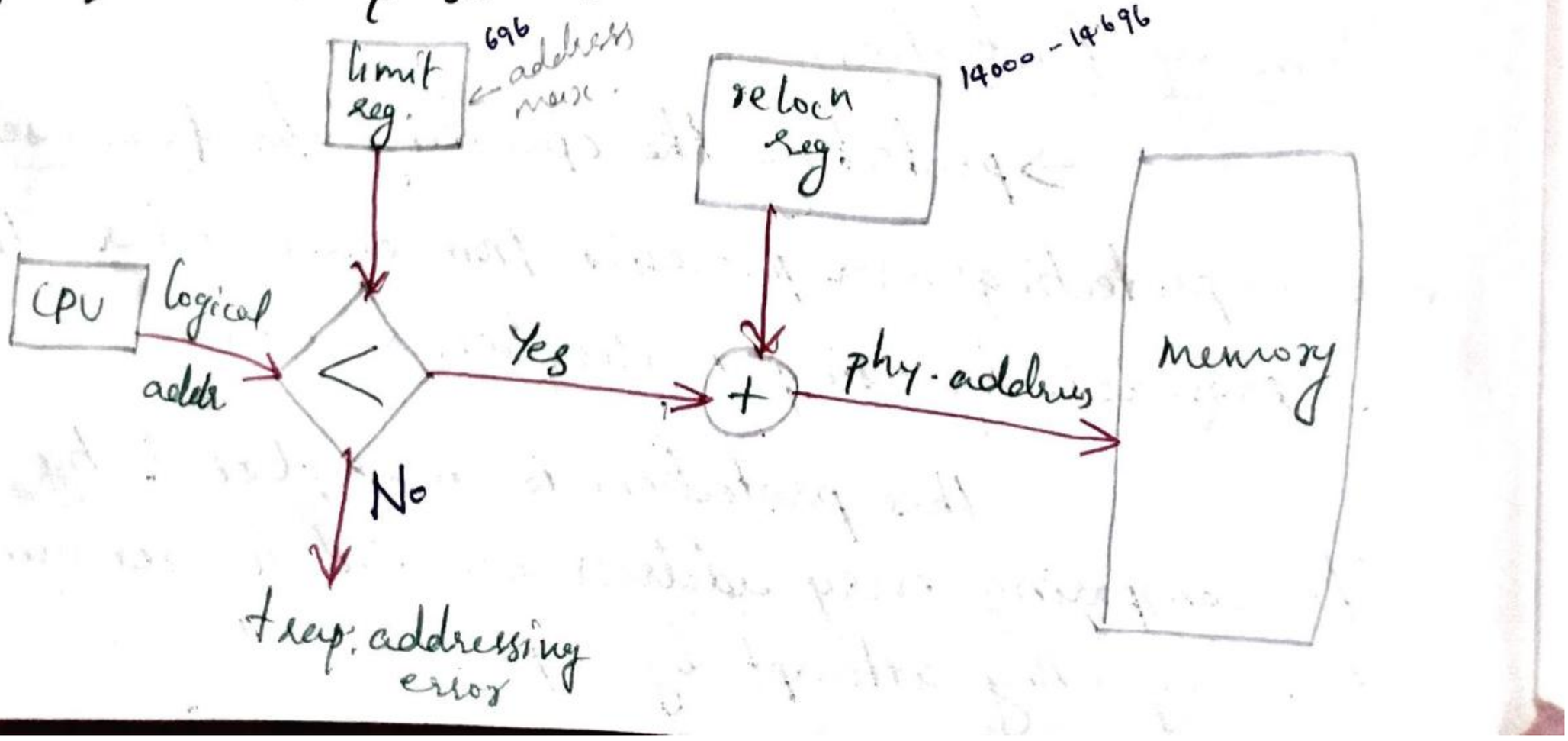
This protection is accomplished by the CPU h/w comparing every address generated in user mode with the regs. Any attempt by a pgm executing in user mode

Scanned by TapScanner

to access monitor memory or other users' memory results in a trap to the monitor, which treats the attempt as error. This scheme prevents the user pgm from modifying the code or data structures of either the O.S or other users.

The base & limit reg.s can be loaded by only the O-S which uses a special privileged inst^n. Since privileged inst^{n}s can be executed in only ~~mode~~ monitor mode, and since only the O.S can load the base & limit registers. This scheme allows the monitor to change the value of the regs. but prevents user pgms from changing the reg's contts.

The O-S, executing in monitor mode, is given unrestricted access to both monitor & user's memory. This provision allows the O-S to load users' pgms into users' memory, to dump out those pgms in case of errors, to access & modify parameters of s/m calls & so on.



Scanned by TapScanner

The relocation-reg. scheme provides an effective way to allow the O.S size to change dynamically

For eg: the O.S contains code & buffer space for device drivers. If a device driver is not commonly used, we donot want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called transient O.S code. Thus using this code changes the size of the O.S during pgm exn.

Virtual Memory

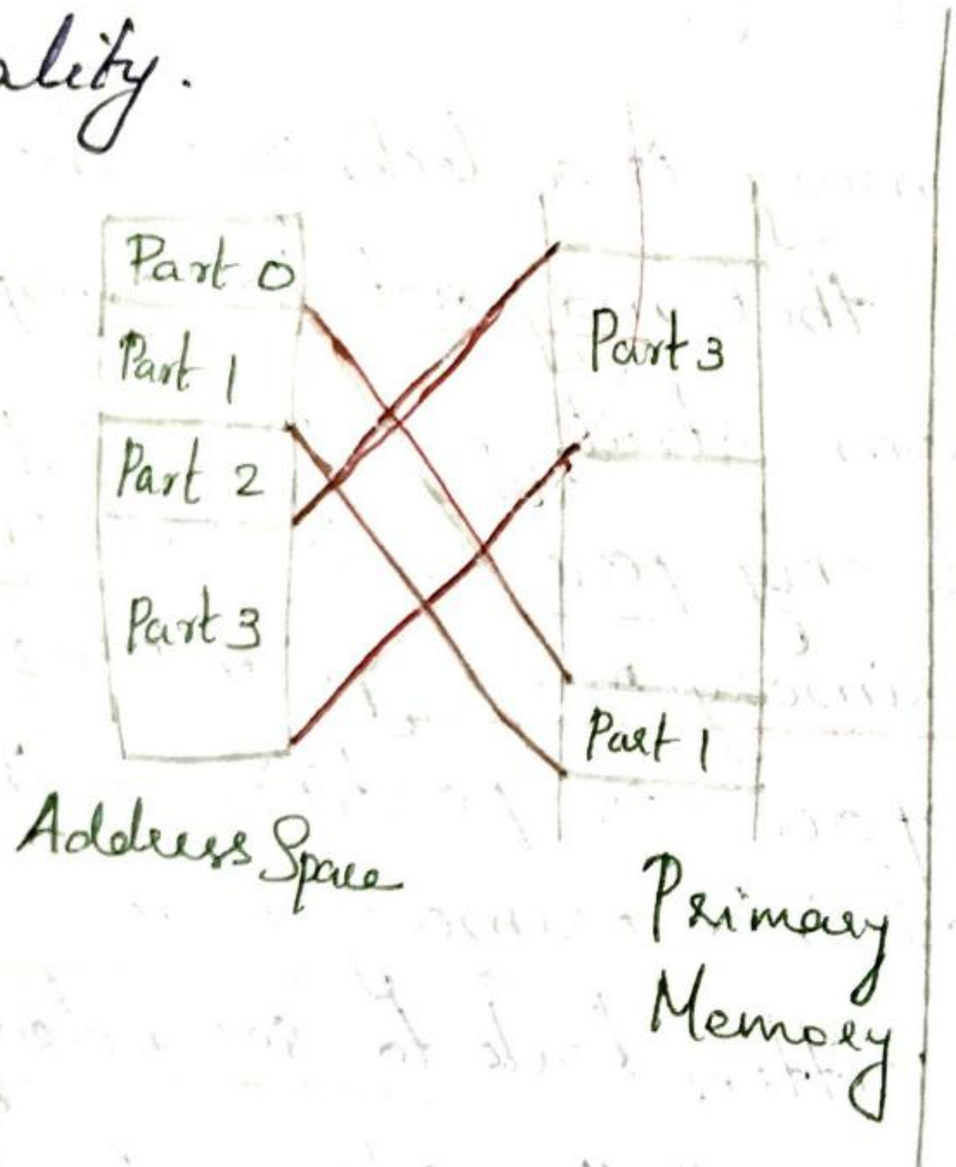
→ Paging & Segmentation

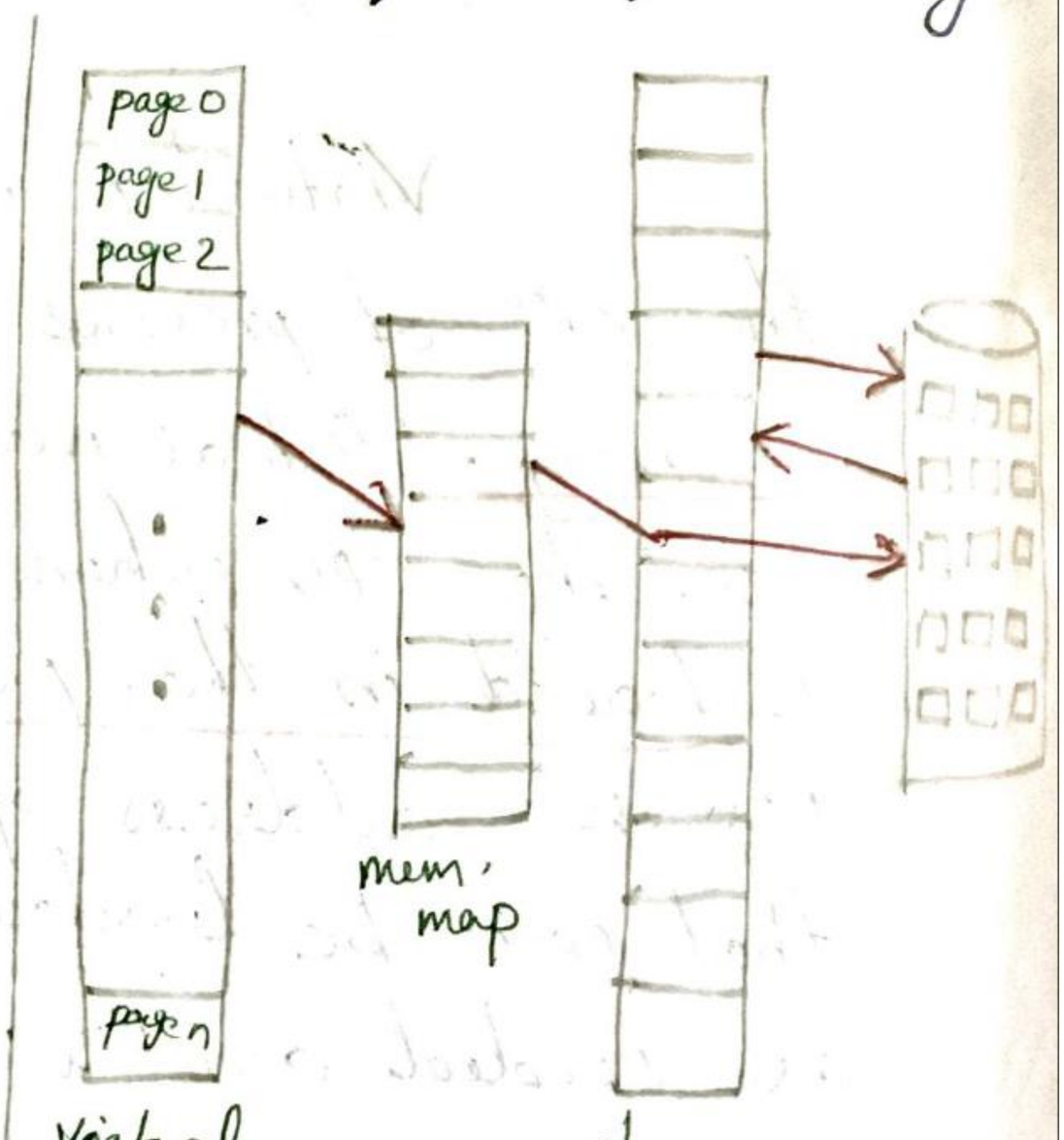
Virtual memory is a technique that allows the exn of praesses that may not be completely in memory. Virtual mem. strategy allows a proces to use the CPU when only part of its address space is loaded in the primary memory. In this approach process's address space is partitioned into parts that can be loaded into primary mem. when they are needed and written back to secondary memory. One major advantage of this scheme is that pgms can be larger than phy. memory. Virtual mem. abstracts m/m into an extremely large, uniform array of storage, separating logical mem as viewed by the user from phy. memory. This technique frees pgmmers from the concerns of mem. storage limitations.

Scanned by TapScanner

Spatial locality

For eg.: almost all pgms have a phase for initializing data structures; another for reading i/p data, one or more for the actual computation, others for recovery & reporting & one for output. Similar implicit ~~whers~~ partitions usually exist in the data segment. When a pgm is executing in one part of its addr. space, its spatial locality is the set of addresses used during that phase of the computation. As the computation moves to a diff. ~~st~~ phase, it changes locality.

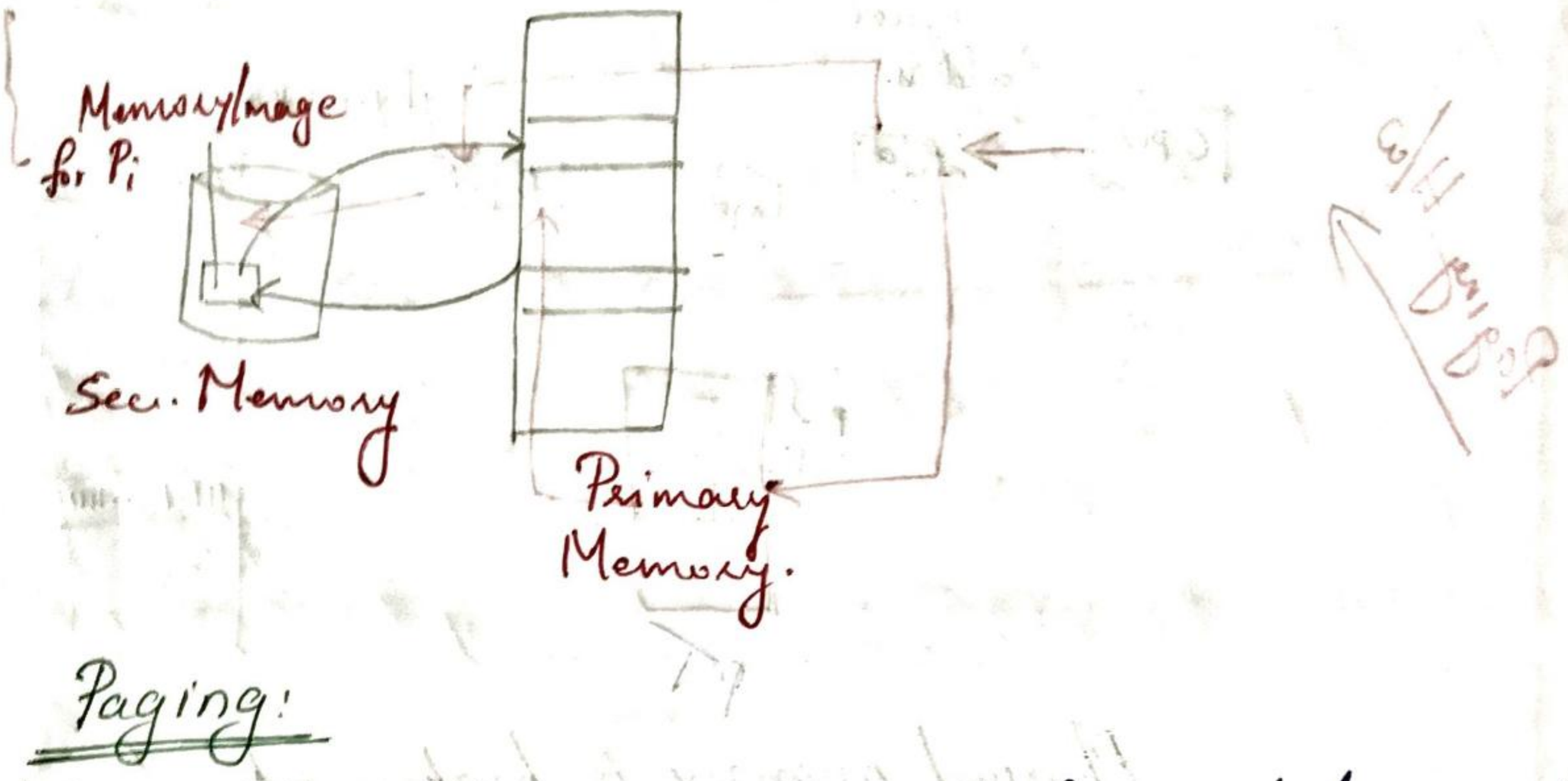




Virtual mem. that is larger than phy. memory.

Scanned by TapScanner

In a virtual mem. s/m, the mem. mngr copies portions of process P_i's address space into the primary memory whenever the P_i is referencing the instr^n. When P_i stops referencing the instr^n, it is updated in the secondary mem. & the copy is removed from the primary memory.



Paging:

A solution to the ext. fragmentation is to permit the logical address space of a process to be noncontiguous, thus allowing a process to be allocated phy. mem. wherever the latter is available.

Two complementary techniques ~~can also be~~ to achieve this solution:

* Paging ✓
* Segmentation ✓

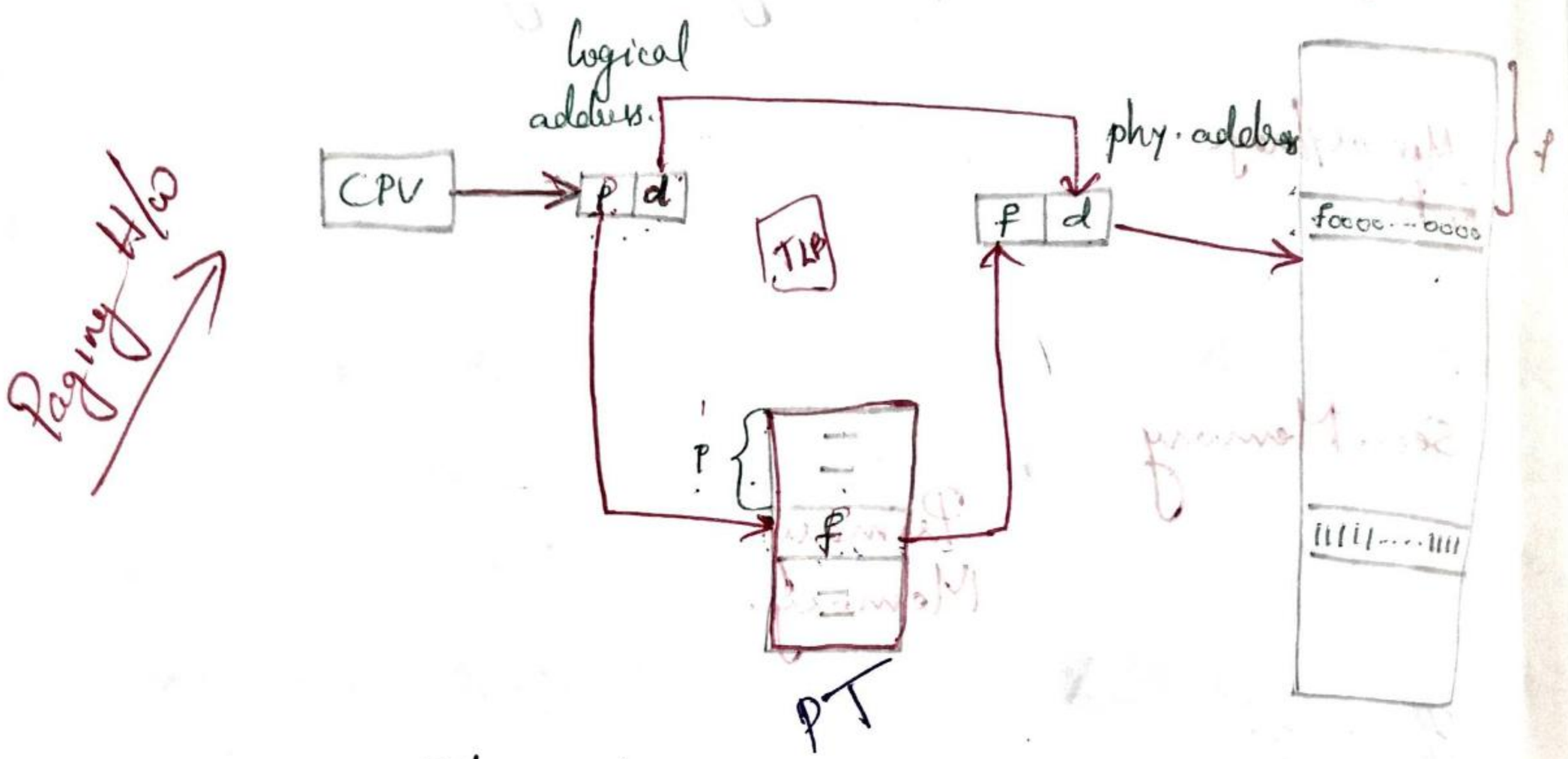
Paging:

Paging is a mem. mgmt scheme that permits the phy. addr. space of a process to be noncontiguous.

Paging s/ms devide the virtual address space into a set of fixed-size pages. These page boundaries are not visible to the s/w. The virtual mem. mngr transports pages back & forth b/w the 2 levels of mem.

Scanned by TapScanner

The physical address space is the portion of the primary mem. allocated to the process. The unit of allocn is a page frame, a blk of primary memory the same size as a page.



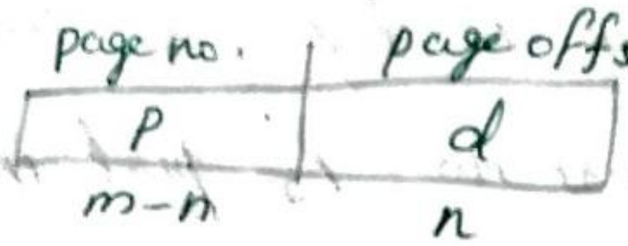
Physical memory is broken into fixed sized blks called frames. Logical mem. is also broken into blks of the same size called pages. When a process is to be executed, its pages are loaded into any available mem. frames from the backing store.

Every address generated by the CPU is devided into 2 parts: a page no. (p) & a page offset(d). The page no. is used as an index into a page table. The page table contains base address of each page in phy. memory. This base address is with offset to define the phy. the phy. mem. address that is sent to the mem. unit.

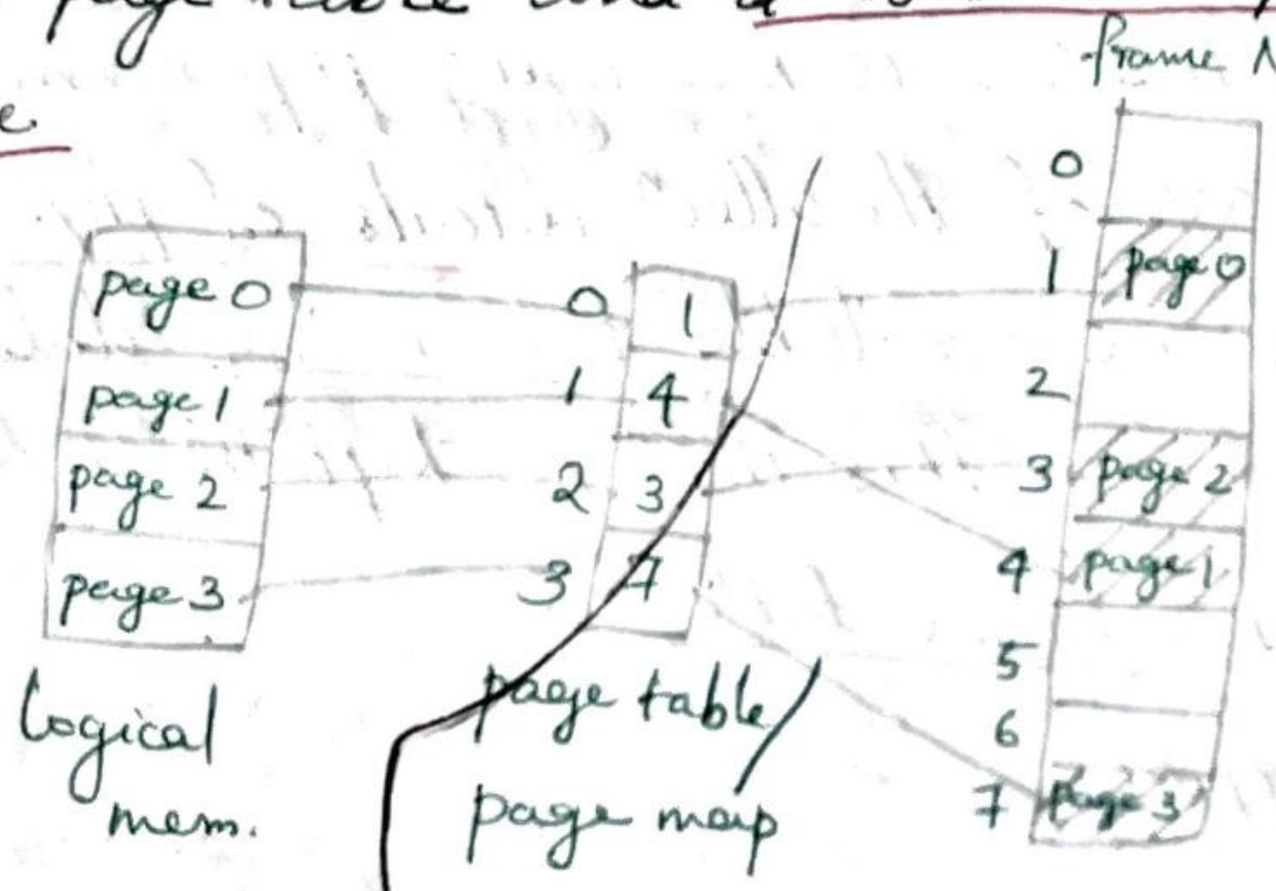
Scanned by TapScanner

If the size of logical addr. space is 2^m, and a page size is 2^n addressing units, then the higher order m-n bits of logical address designate the page number, & the n low order bits designate page offset.

The logical address is



where p is an index into the page table and 'd' is the displacement within the page.



Since the O.S is managing phy. memory, it must be aware of the alloc^n details of phy. mem. ie; which frames are allocated, which frames are available, how many total frames there are. This inf^n is kept in a datastructure called frame table. The frame table has one entry for each physical ~~memory~~ page frame.

When a process arrives in the s/m to be executed, its size, expressed in pages, is examined. Each page of the process needs one frame. Thus if the process requires

Scanned by TapScanner

n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame no. is put in the page table for this process. The next page is loaded into another frame, and its frame no. is put into the page table & so on.

Since the O.S is managing phy. memory, it must be aware of the alloc^{n} details of phy. memory which frames are available, how many total frames are there & so on. This inf^{n} is kept in a data structure called a frame table.

Hardware Support

Page table : A ptr to the page table is stored with other reg. values in the PCB. When the dispatcher is told to start a process, it must reload the user regs. & define the correct h/w page table values from the stored user page table.

Page table Implementation

Simplest way : The page table is implemented as a set of dedicated regs. Every access to mem. must go through the page table, so efficiency is a major consideration.

Scanned by TapScanner

The CPU dispatcher reloads these regs. just as it reloads other registers. Inst's to load or modify the page table regs. are, previleged, so that only the O.S can change the mem. map.

→ The page table to be very large.

$\frac{2^4}{2^7 \times 2^{10}} = \frac{1}{2}$

For these m/cs, the use of fast regs. to implement the page table is not feasible. The the page table is kept in m/m, & a page table base register (PTBR) pnt points to the page table.

The problem with this approach is the time required to access a user mem. location. If we want to access location i, we must first index into the page table, using the value in the PTBR offset by the pagenumber for i. This task requires a mem. access. It provides us with the frame no., which is combined with the page offset to produce the actual address. We can then access the desired place in memory. With this scheme, 2 mem. accesses are needed to access a byte (one for the pg table entry, one for byte).

The solution to this problem

TLB

Translation Look-aside Buffer (TLB)

Each entry in the TLB consists of two parts: a key & a value. The TLB contains only a few of the page table entries. When a logical address is generated by the CPU, its page no. is presented to the TLB. If the page no. is found, its frame no is immediately

Scanned by TapScanner

available and is used to access memory.

If the page number is not in the TLB (TLB miss), a memory reference to the page table must be meade. When the frame no. is obtained, we can use it to access memory. In addition, we add pgno & fr.no to the TLB, so that they will be found quickly on the next reference.

If the TLB is already full of entries, the O.S must select one for replacement. Se

Address space protection: Some TLBs store address-space identifiers (ASIDs) in each entry of the TLB. An ASID uniquely identifies each process & is used to provide address space protection for that process.

When the TLB attempts to resolve virtual page no.s, it ensures the ASID for the currently running process matches the ASID associated with the virtual page. If the ASIDs do not match, they are treated as a TLB miss.

Memory Protection

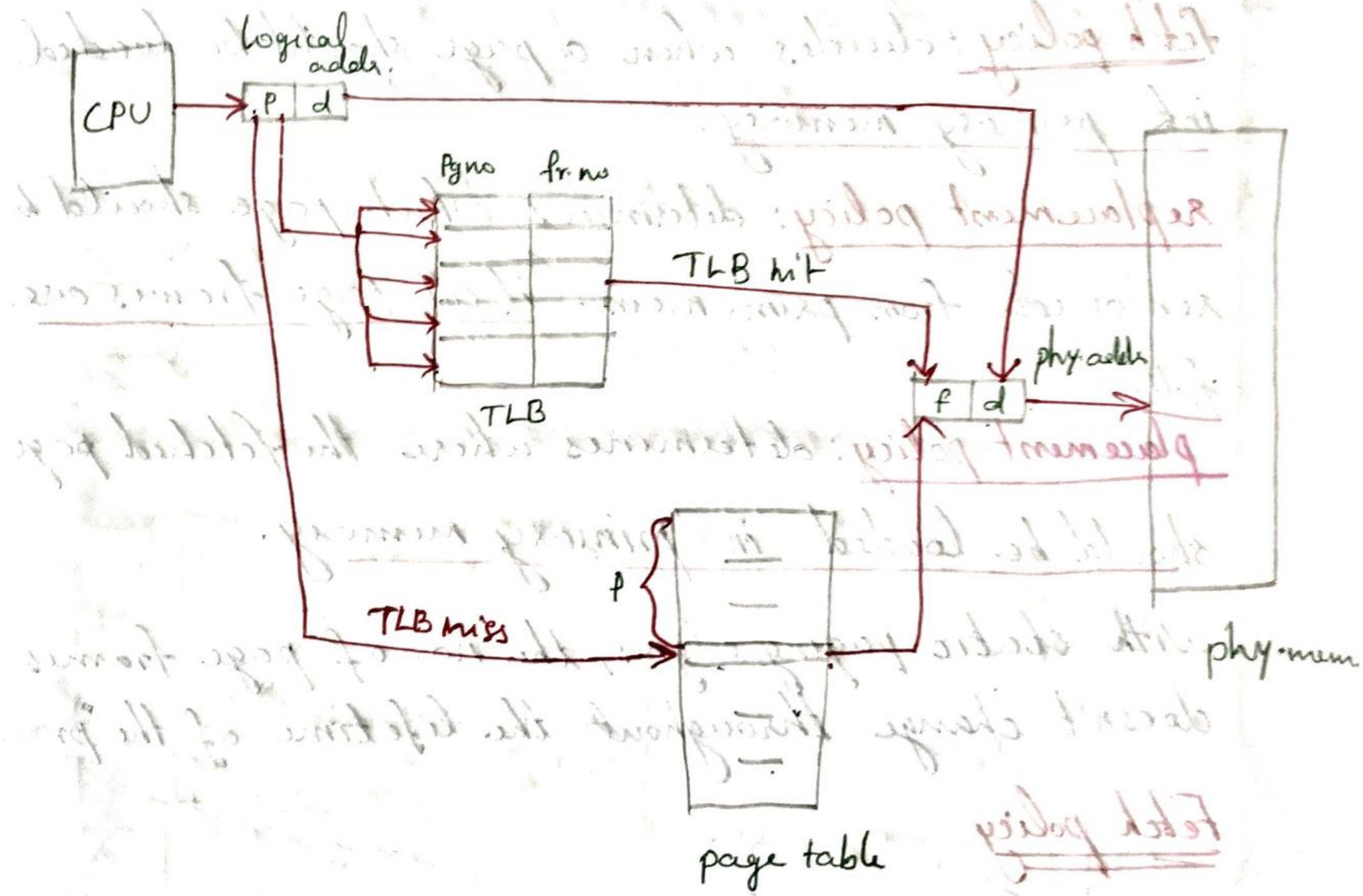
- protection bits that are associated with each frame. These bits are kept in page table. One bit can define a page to be read-write or read-only.

The protection bits can be checked to verify that no writes are being made to a read-only page.

One more bit is generally attached to each entry in the page table: a valid-invalid bit

Scanned by TapScanner

when this bit is set to "valid", this value indicates that the associated page is in the process' logical address space & is thus a valid page. If the bit is set to "invalid", this value indicates that the page is not in the process' logical address space.



Static Paging Algorithms

There are 2 basic families of paging algms.

- static allocation
- dynamic allocation.

With static paging algms, the process uses a fixed no. of page frames. In dynamic paging, the no. of allocated page frames changes as the process executes.

Scanned by TapScanner

In both forms of alloc^n algms, the paging policy defines how the page frames will be loaded & unloaded by the virtual mem. s/m. There are 3 basic policies in defining any paging algm.

fetch policy: decides when a page should be loaded into primary memory.

replacement policy: determines which page should be removed from prim. mem. if all page frames are full.

placement policy: determines where the fetched page should be loaded in primary memory.

With static paging algms, the no. of page frames doesn't change throughout the lifetime of the proces

Fetch policy

The fetch policy determines (when) a page should be brought into primary mem. The paging mechanism will not ordinarily have prior knowledge about the page reference stream for pgms executing on m/c.

prefetch policy: where pages are fetched into the mem prior to the time they are referenced.

demand paging: fetch policy where a page is loaded into primary mem. only when the process references.

Scanned by TapScanner

In demand paging, One of the follo. conditions applies at virtual time t when r_t appears in the ref. streams.

→ If r_t was loaded at virtual time t-1, then nothing is moved from primary memory & nothing is added to it

→ If r_t was not loaded at virtual time t-1, but there are empty pages frames allocated to the proces, then the missing page is placed into one of empty page frames

→ If r_t was not loaded at virtual time t-1 & all page frames contain a page, then the cont. of some page frame loaded with diff. page, y has its cont. replaced by page r_t.

Demanel Paging

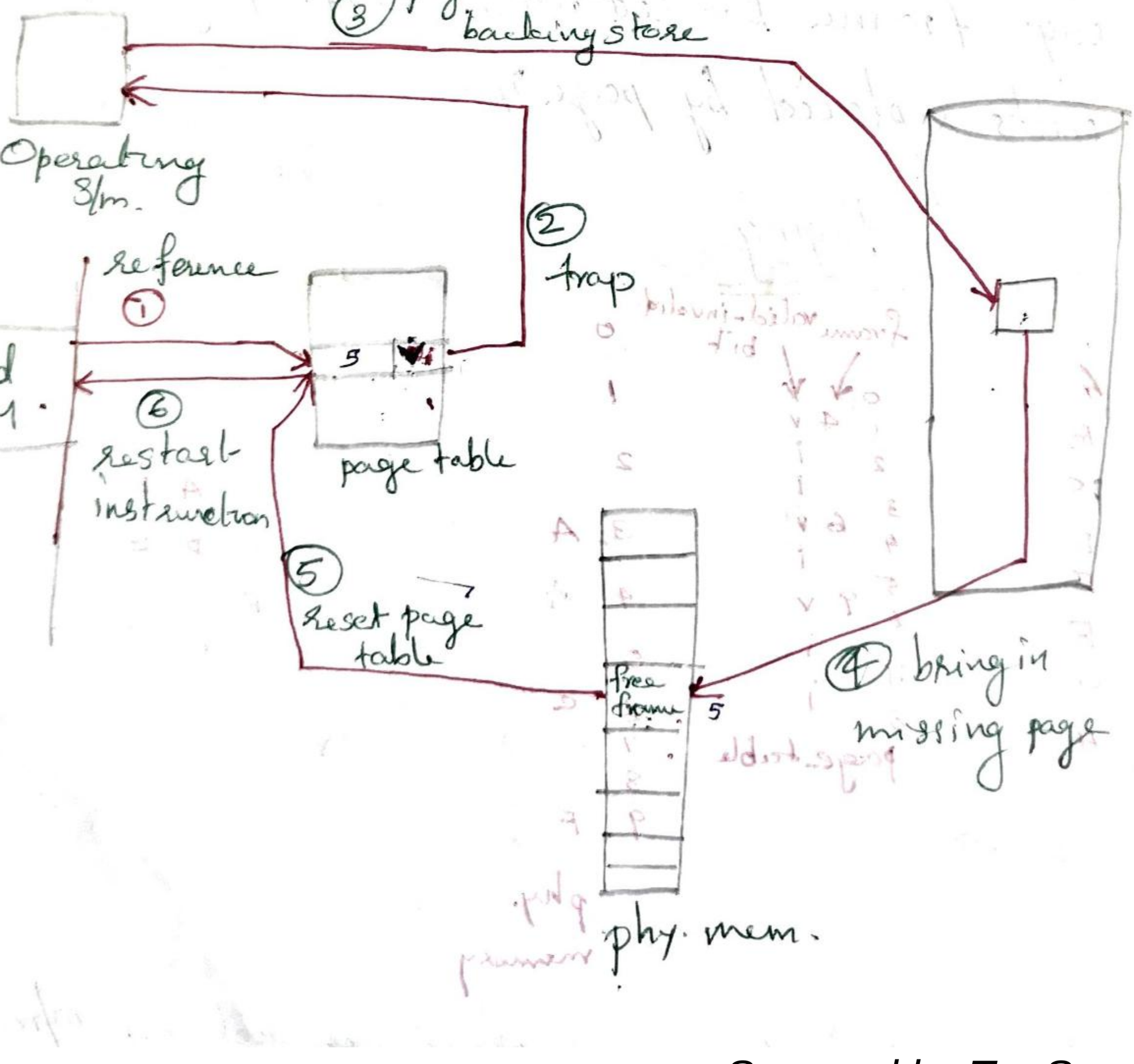
Page table when some pages are not in m/m.

Scanned by TapScanner

The valid-invalid bit can be used to distinguish b/w those pages that are in mem. & those pages that are on the disk.

When this bit is set to "valid", this value indicates that the associated page is in memory. If the bit is set to "invalid", indicates either the page is not valid or valid but is currently on disk. Access to a page marked invalid causes a page fault trap.

The procedure for handling this page fault.



Scanned by TapScanner

① We check an internal table for this process, to determine whether the ref. was a valid or invalid
② If the ref. was invalid, we terminate the process. If it was valid, but we have not yet brought th in that page, we now page it in.
③ We find a free frame from free frame list
④ We schedule a disk opⁿ to reseed the desired page into the newly allocated frame.
⑤ When the disk reseed is complete, we modify the internal table kept with the process & the page table to indicate the page is now in memory.
⑥ We start the instⁿ that was interrupted by the legal address ~~not~~ trap. The process can now access the page as though it had always been in mem.

Performance of Demand Paging

Effective access time for a demand paged memory.

effective access time = $(1-p) \times ma + p \times$ page fault time

p - probability of page fault

ma - memory access time

~~A page~~

Major components of the page-fault service time.

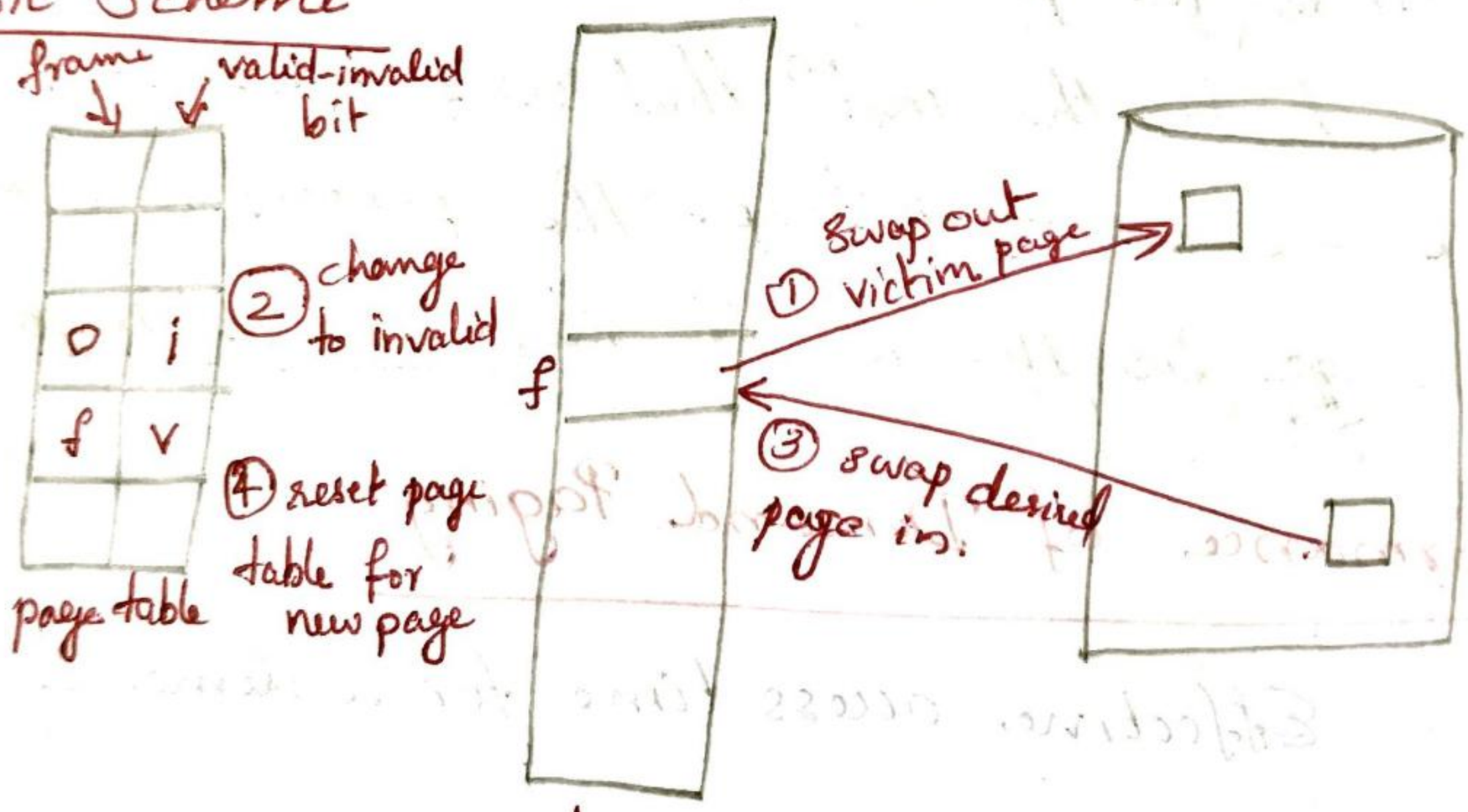
1. Service the page-fault interrupt
2. Read in the page
3. Restart the process.

Scanned by TapScanner

Page Replacement

▽ While a user process is executing, a page fault occurs. The h/w traps to the O.S, which checks its internal tables to see that this page fault is genuine one rather than an illegal memory access. The O.S determines where the desired page is residing on the disk, but then finds that there are no free frames on the free-frame list. All memory is in use.

Basic Scheme



Page Replacement Algorithms

We evaluate an algm by running it on a particular string of mem. references & computing the no. of page faults. The string of memory references is called a reference string.

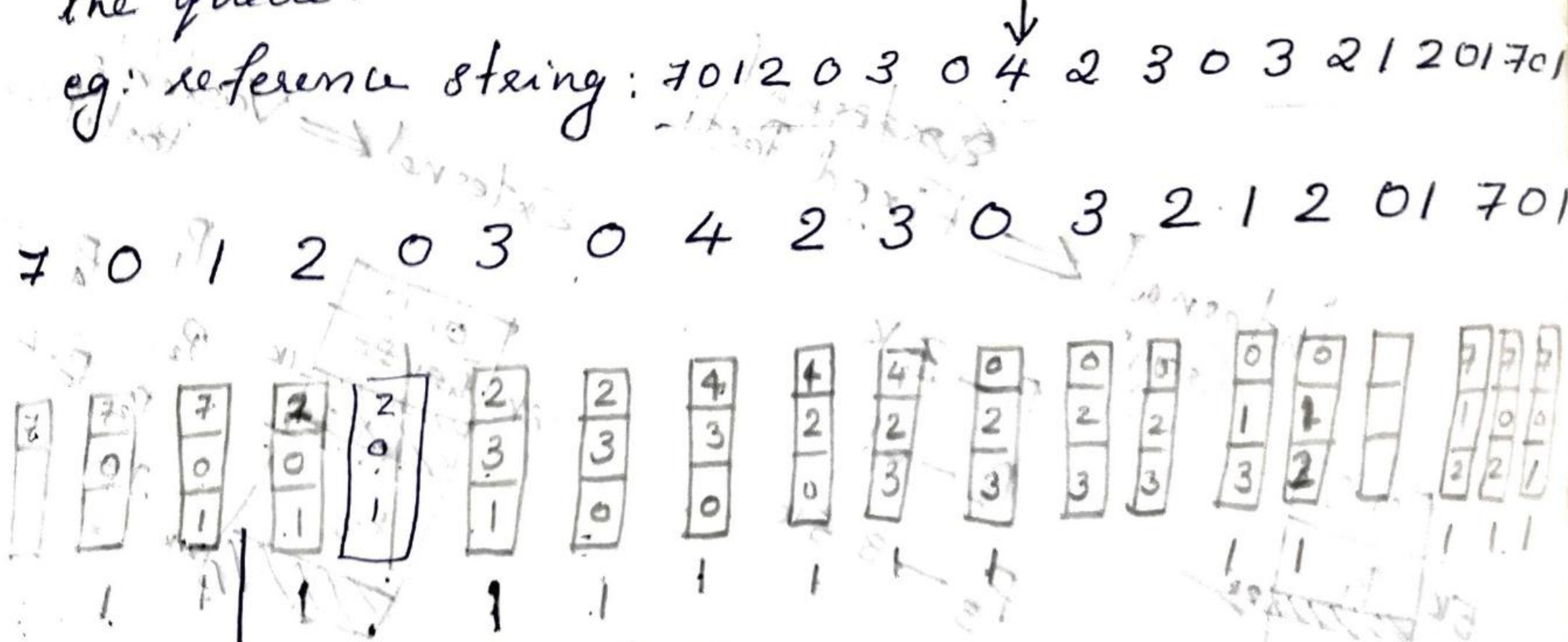
Scanned by TapScanner

We can generate reference strings artificially or we can trace a given s/m & record the address of each mem. reference.

FIFO Page Replacement

A FIFO replacement algm associates with each page, the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We replace the page at the head of the queue.

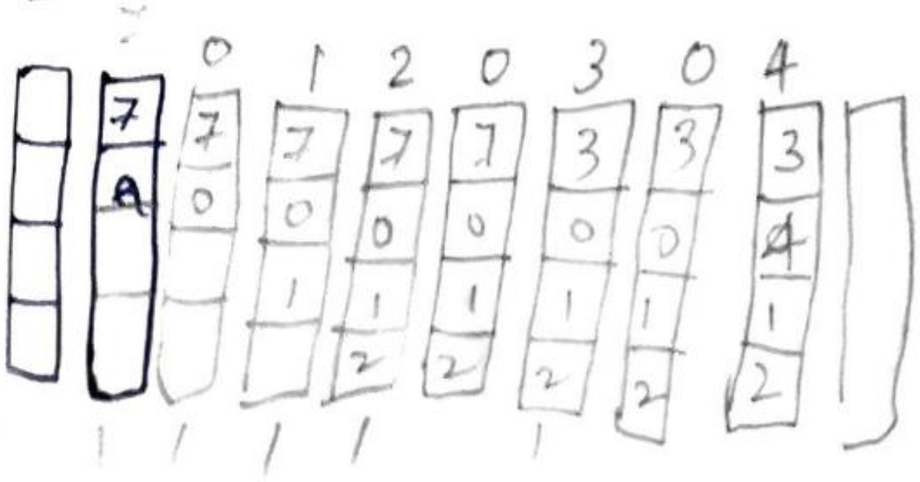
eg: reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



No. of page faults = 15

Belady's Anomaly:

Page fault rate may increase as the no. of allocated frames increases.



Scanned by TapScanner

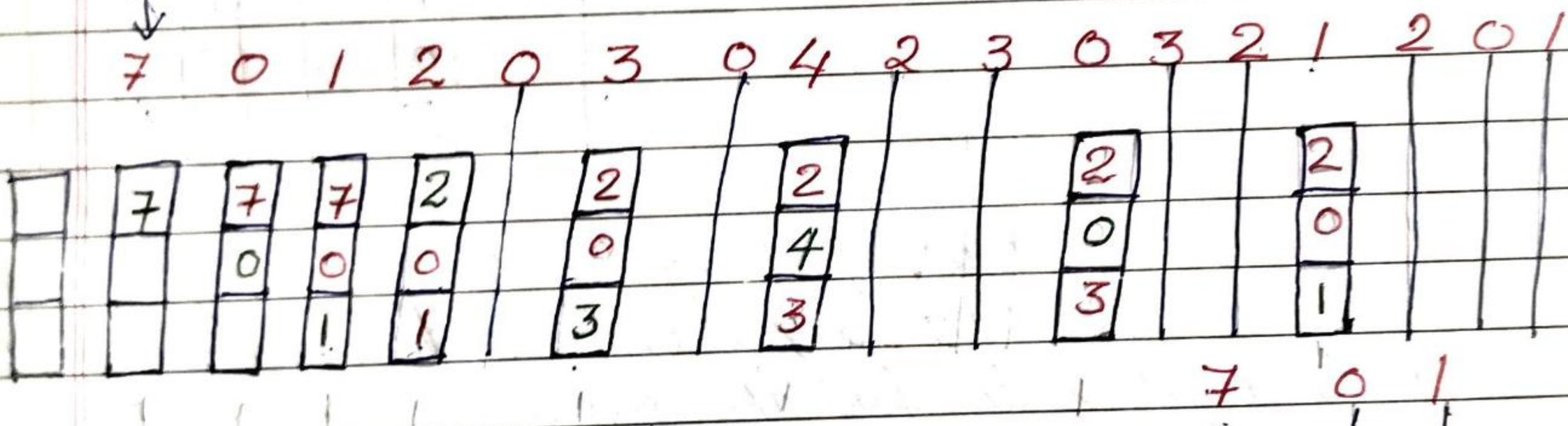
... (Replacement Algorithms:)

Optimal Page Replacement Algorithm:

An optimal page-replacement algm has the lowest page fault rate of all algorithms, & will never suffer from Belady's anomaly.

"Replace the page that will not be used for the longest period of 'K' time".

Eg: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 (reference string)



No. of page faults = 9

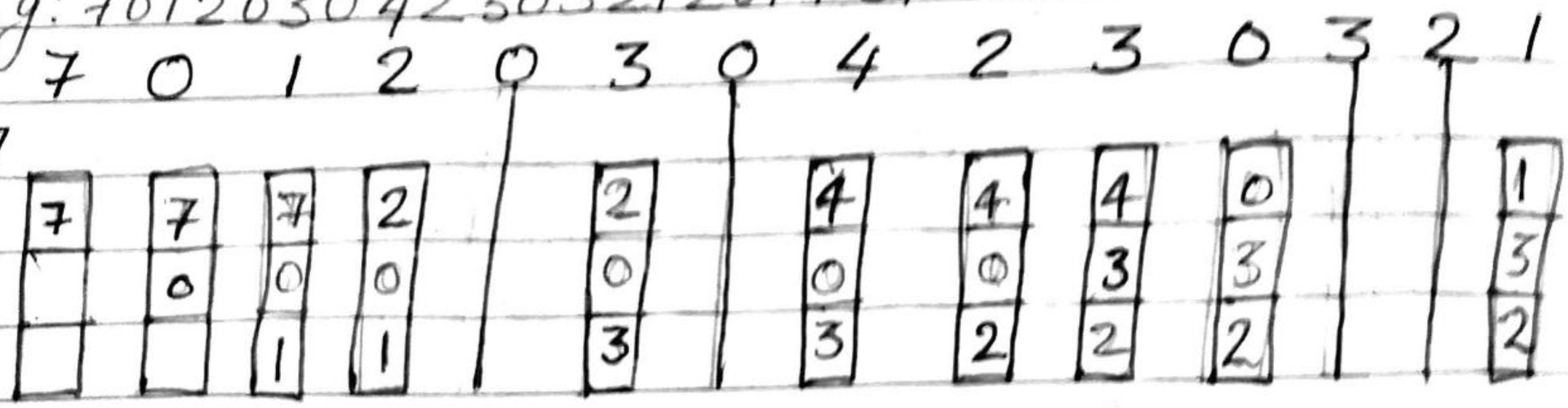
Optimal replacement algm is difficult to implement b/c it requires future knowledge of the reference string.

LRU Page Replacement Algorithm:

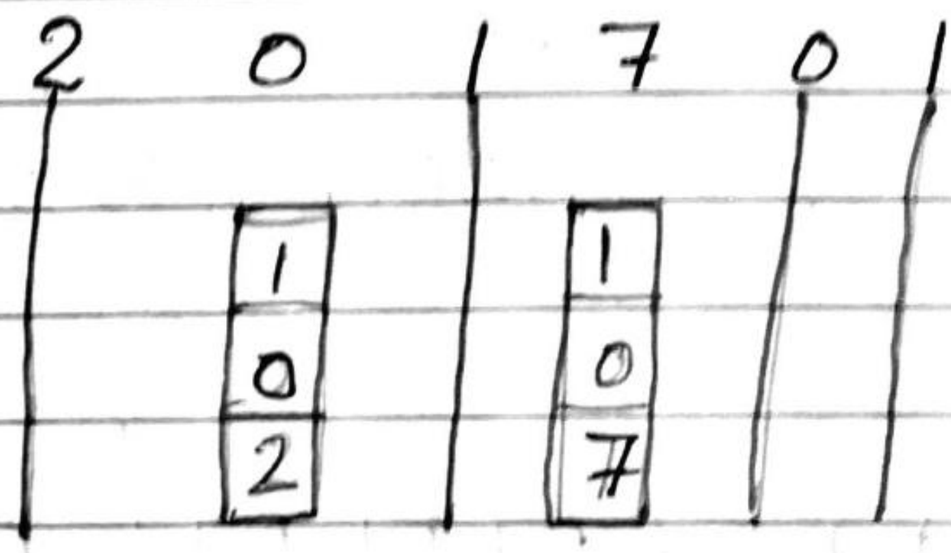
"Replace the page that has not been used for the longest period of time".
LRU replacement associates with each page the time of that page's last use.

Scanned by TapScanner

Eg: 7012030423032120 1701



No. of page faults = 12.



S^R - reverse of a ref. string S, then the page-fault rate for the OPT algm on S is the same as the page fault rate for the same on S^R.

Illy, the page fault rate for the LRU algm on S is the same as the page fault rate for LRU algm on S^R.

The major problem is how to implement LRU replacement. An LRU-replacement algm may require h/w assistance. The pblm is to determine an order for the frames defined by the time of last use.
Two Implementations are feasible:

Counters:

We associate with each page table entry a time of use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the conts. of the clock register are copied

Scanned by TapScanner

to the time of use field in the page table entry for that page. We replace the page with the smallest time value. This scheme requires a search the page table to find the LRU page and a write to memory for each memory access. The times must also be maintained when page tables are changed.

Stack:

Another approach to implement LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and on the top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page.

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

a b

Use of a stack to record the most recent page references.

we remove the ents in the stack.

stack beforea	stack before after b
2	7
1	2
0	1
7	0
4	4

Because entries must be removed from the middle of the stack, it is best implemented by a doubly linked list, with a head & tail pointer. There is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page.

Neither OPT nor LRU replacement suffers from Belady's anomaly.

Reference bit: The reference bit for a page is set, by the h/w, whenever that page is referenced. Reference bits are associated with each entry in the page table.

Scanned by TapScanner

Initially all bits are cleared by the O.S. As a user process executes, the bit associated with each page referenced is set by the h/w. After some time, we can determine which pages have been used & which have not been used by examining the reference bits. We do not know the order of use, but we know which pages were used and which were not used.

Approximation Page Replacement:
* Additional-Ref. Bits Algorithm
* Second Chance Algorithm
* Enhanced Second chance Algorithm

Additional Ref. Bits Algorithm:
- additional ordering information by recording the reference bits at regular intervals. We can keep 8 bit byte for each page in a table in memory. At regular intervals, a timer interrupt transfers ctrl. to the O.S. The O.S shifts the reference bit for each page into the higher-order bit of its 8 bit byte, shifting the ~~order~~ other bits right 1 bit, discarding the low order bit.

eg: A page with register value of 11000100 has been used more recently than has one with 01110111

Second Chance Algorithm:
When a page has been selected, we inspect its reference bit. If the value is 0, we proceed to replace this page. If the ref. bit is 1, we give that page a second chance, its ref. bit is cleared & its arrival time is reset to the current time. Thus a page that is given a second chance will not be replaced untill all other pages

Scanned by TapScanner

are replaced. If a page is used often enough to keep its ref. bit set will never be replaced.

Enhanced Second Chance Algorithm:

- both the (ref. bit) & the (modify bit) as an ordered pair.

1. (0,0) - neither recently used nor modified - best page
2. (0,1) - not " but modified-
3. (1,0) - recently used & not modified
4. (1,1) - recently used & modified

Counting - Based Page Replacement:

* Least frequently used (LFU): requires that the page with the smallest count be replaced. An actively used page should have a large ref. count.

* Most frequently used (MFU): is based on the argument that the page with the smallest count was probably just brought in & has yet to be used.

Page Buffering Algorithm:

S/ms commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

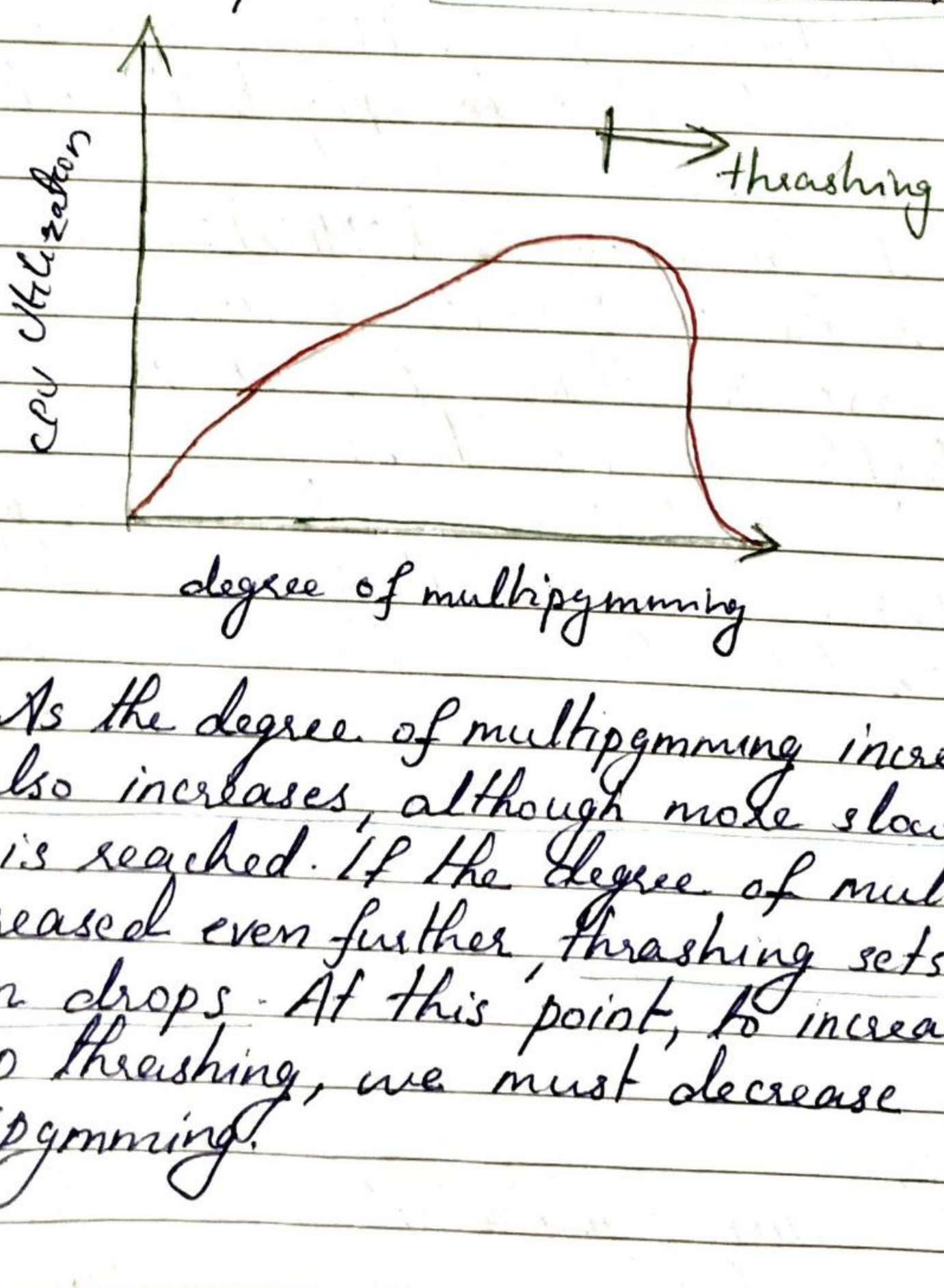
Thrashing:

High paging activity is called thrashing. A process is thrashing, if it is spending more time paging than executing.

Scanned by TapScanner

eg: If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the s/m. A process enters a new phase in its exn & need more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however and so they also fault taking frames from other processes. These faulting processes must use the paging device to swap pages in & out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

Then Repeats the same scenario



As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until the max. is reached. If the degree of multiprogramming is increased even further, thrashing sets in & CPU utilization drops. At this point, to increase CPU utilization & stop thrashing, we must decrease the degree of multiprogramming.

Scanned by TapScanner

Working Set Model

To prevent thrashing, we must provide as many frames as it needs. The working set strategy starts by looking at how many frames a process is actually using. This approach defines the locality model of process exn.

The locality model states that, as a process executes, it moves from locality to locality. A locality is a set of pages that are actively used together. A pgm is generally composed of several diff. localities, which may overlap.

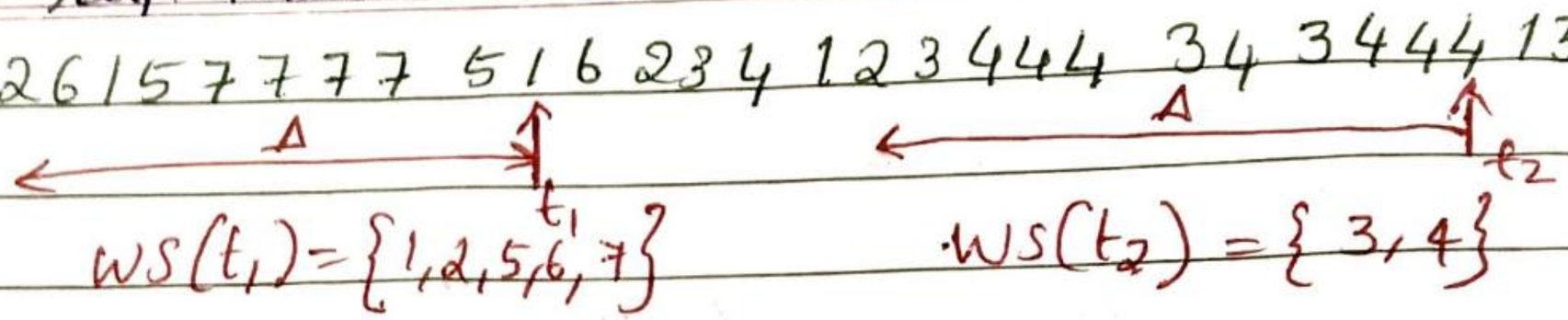
For eg: when a subroutine is called, it defines a new locality. In this locality, mem. references are made to the inst's of the subroutine, its local variables & a subset of the global variables. When the subroutine exited, the process leaves this locality, since the local variables and inst's of subroutine are no longer in active use.

Suppose that we allocate enough frames to a process to accomodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then it will not fault again until it changes localities.

The working-set model: is based on the assumption of locality. This model uses a parameter Δ, to define the working set window. The set of pages in the most recent Δ page references is the working set. If it is no longer being used it will drop from the working set Δ time units after its last reference.

Scanned by TapScanner

page ref. table



For eg: given the sequence of mem. refs, if Δ=10 mem. refs, then the working set at time t_1 is {1,2,5,6,7}. By the time t_2, the working set has changed to {3,4}

If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap several localities.

The most imp property of the working set is its size.

$D = \Sigma SS_i$ WSS_i → working set size for each process

D → total demand for frames.

The O.S monitors the working set of each process & allocates to that WS enough frames to provide it with its WS size. If there are enough frames another process can be initiated. If the sum of WSSizes increases, exceeding the total no. of available frames, the O.S selects a process to suspend.

This working set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. It optimizes CPU Utilization.

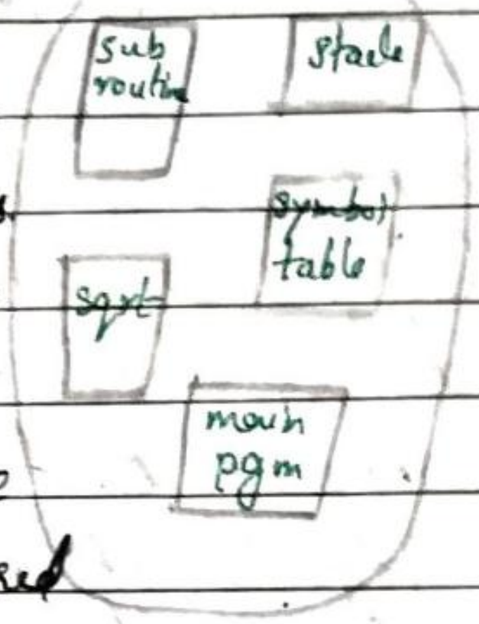
- difficulty: keeping track of working set.

The WS window is a moving window. At each mem. ref., a new ref. appears at one end and the oldest ref. drops off the other end.

Scanned by TapScanner

Segmentation

Users prefer to view memory as a collection of variable-sized segments, with no necessary ordering among segments.



main pgm ⟹ set of subroutines, procedures, functions, fns & modules. There may also be various data structures: tables, arrays, stacks variables & so on. Each of these modules or data elmts is referred to by name.

A logical address-space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name & the offset within the segment. The user therefore specifies each address by two quantities: a segment name & offset.

ie; logical address = <segment no, offset>

For eg: in Pascal separate segments:

1. Global variables
2. procedure call stack
3. code portion of each procedure
4. the local variables of each procedure & fn.

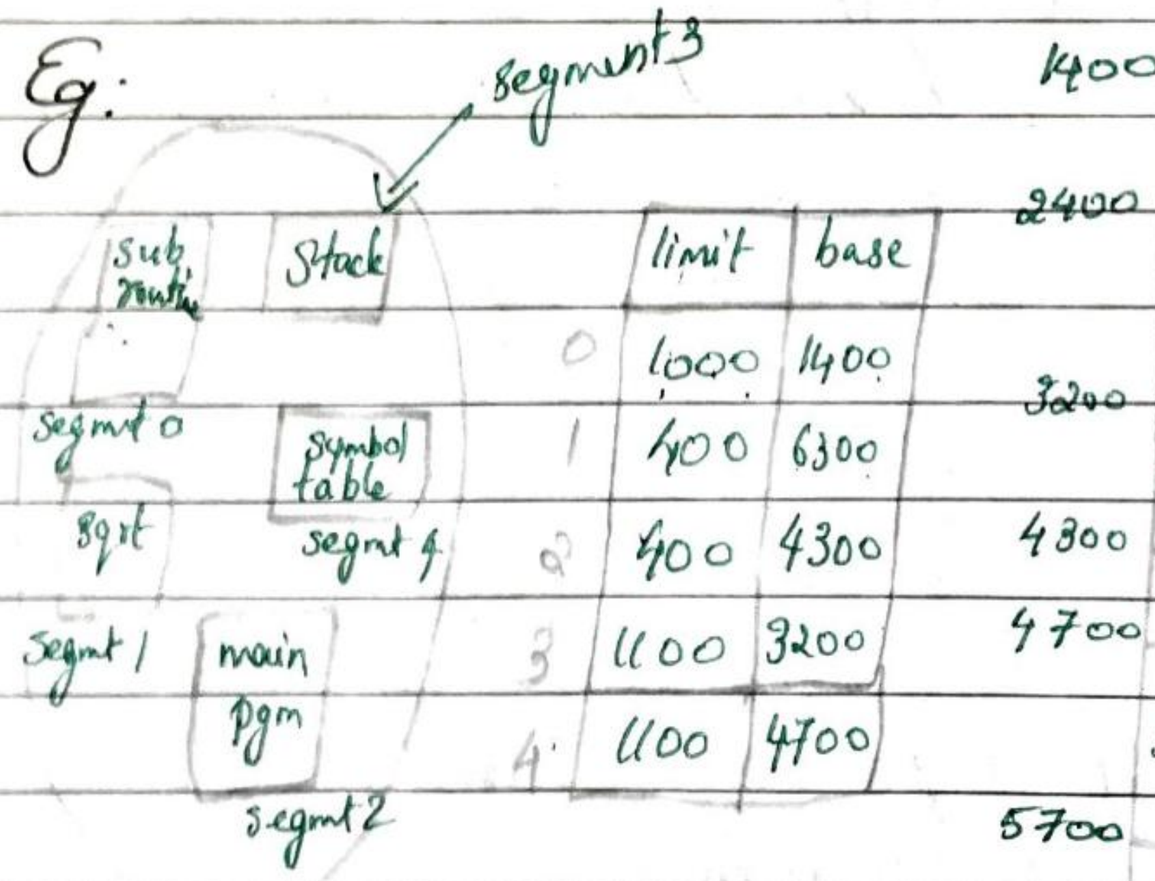
We must define an implementation to map 2 dimensional user defined addresses into one dimensional phy. addresses. This mapping is affected by a segment table.

Scanned by TapScanner

Each entry of the segment table has a segment base and a segment limit. These segment base contains the starting phy. address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

A logical address consists of two parts: a segment number, 's' & an offset into that segment, d. The offset 'd' of the logical address must be b/w 0 & the segment limit. If it is not, we trap to the O.S. If this offset is legal, it is added to the segment base to produce the address in phy. mem. of the desired byte. The segment table is an array of base-limit register pairs.

Eg:



logical addr. space

Registers:

Segment Table Register (STR): points to the location of the segment table itself.

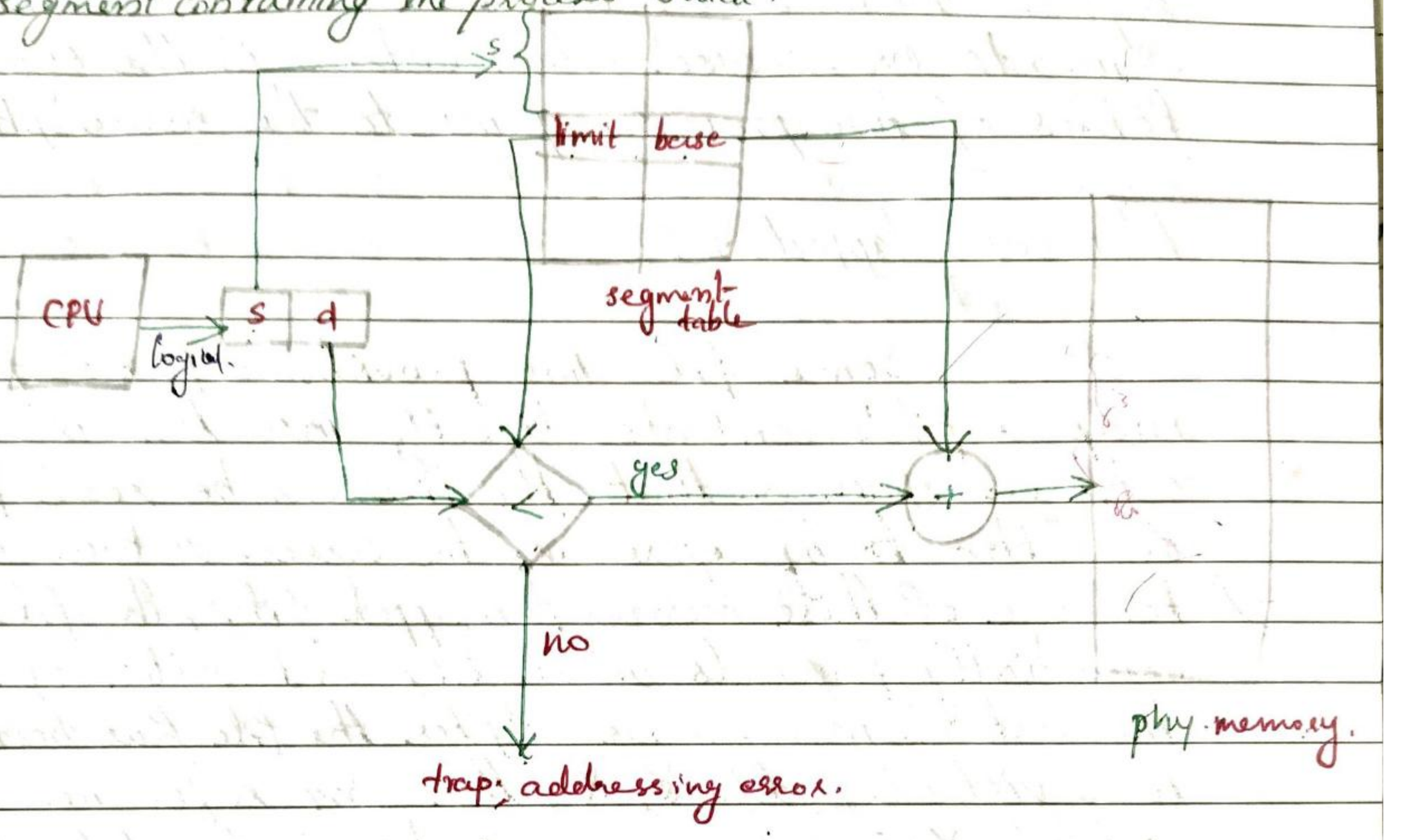
The code segment base value is kept in the Code Base Register also called Procedure Base Register (PBR) in some m/cs.

Scanned by TapScanner

P_1 – 1000 – 2000

P_2 – 2000 – 2500

A Data Base Register (DBR) is used to dynamically relocate static data references, and a Stack Base Register (SBR) points to a segment containing the process's stack.



Protection & Sharing:

Advantage:

→ ~~Association with the segments.~~

→ Association of protection with the segments. some segments ~~can be~~ are instructions, others are data. Inst^n's are non self modifying, so inst^n segments can be defined as read only or execute only. The memory mapping h/w will check the protection bits associated with each segment table entry to prevent illegal accesses to memory such as attempts to write read only segment or to use execute only segment as data.

→ Segmentation involves the sharing of code or data. Each process has a segment table associated

Scanned by TapScanner

with it, which the dispatcher uses to define the h/w segment table when this process is given the CPU. Segments are shared when entries in the segment tables of two processes point to the same phy. loc^n.

Memory Mapped Files:

Some file s/ms provide a function to map a file's contents directly into the virtual address space so that the file can be read from or written to by referencing the corres. virtual addresses. For one of these memory-mapped files, the file mngr essentially forwards all read() and write() op^{n}s to the virtual mem. mngr. After the file has been mapped to a virtual address, X, then any reference to virtual X+i references byte b_i in the file. When parts of the file are referenced, they are copied into memory on a page by page basis, just as other pages are handled by paging s/m.

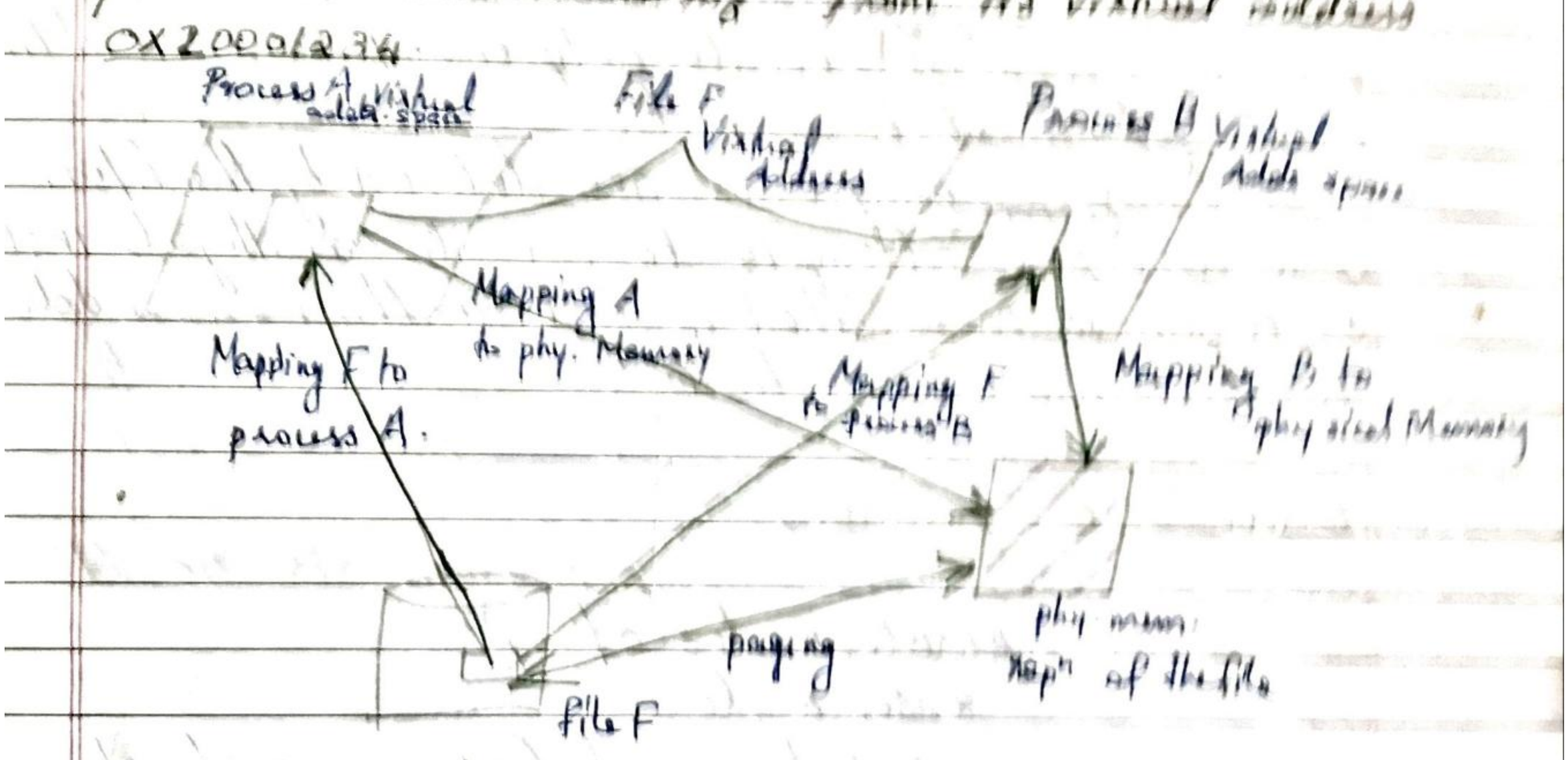
For eg:, if process a 64KB file and maps it into virtual address 0x20000000 to 0x2000FFFF, then it can read or write the first byte in the file by reading or writing mem. addr. 0x20000000 or it can reference the 16th byte by referencing 0x2000000F.

There is another aspect of memory mapped files that makes them even more useful. Since the inf^n is logically referenced as a filename, more than one process can map the same file into its own virtual address space at the same time.

Now suppose process A mapped a 64kB file into its locations 0x20000000 to 0x2000FFFF

Scanned by TapScanner

and process B opened the same file and mapped it to its virtual address 0x20000000 to the beginning of the same address space. Then process B could "communicate" info to process A by writing into at locations beginning, & process A could read info from its virtual address 0x20012345.



Writes to the file mapped in memory may not necessarily be immediate writes to the file on disk. Some O/Ss may choose to update the physical file when the operating O/s periodically checks if the page in memory mapping the file has been modified. Closing the file results in all the memory mapped data being written back to disk & removed from the virtual mem. of the process.

Hardware Protection

Single user; programmes operated O/Ss → complete control over the O/s.

Resident monitors: Early O.S.

Scanned by TapScanner

Sharing s/m resources: ↑ CPU utilization - Multiprogramming

S/m without sharing: an error in a pgm could cause problems for only the one pgm that was running. Without protection against these sorts of errors, either the computer must execute only one process at a time or all o/p must be suspect.

Many pgmming errors are detected by the h/w. If a user pgm attempts to execute an illegal inst^n, or to access mem. that is not in the user's addr. space. The h/w will trap to the O.S.

Dual Mode Operation:

Two separate modes of op^n:

* user mode
* monitor mode (supervisor mode, s/m mode & privileged mode).

A bit, called the mode bit, is added to the h/w of the computer to indicate the current mode:

monitor (0) or user (1).

With the mode bit, we are able to distinguish b/w a task that is executed on behalf of the O.S & one that is executed on behalf of the user.

At s/m boot time, the h/w starts in monitor mode. The O.S is then loaded, and starts in monitor mode. The O.S is then loaded, & starts user processes in user mode. Whenever a trap or interrupt occurs, the h/w switches from user mode to monitor mode.

The dual mode of op^n provides us with the means for protecting the O.S from errant users, & errant users

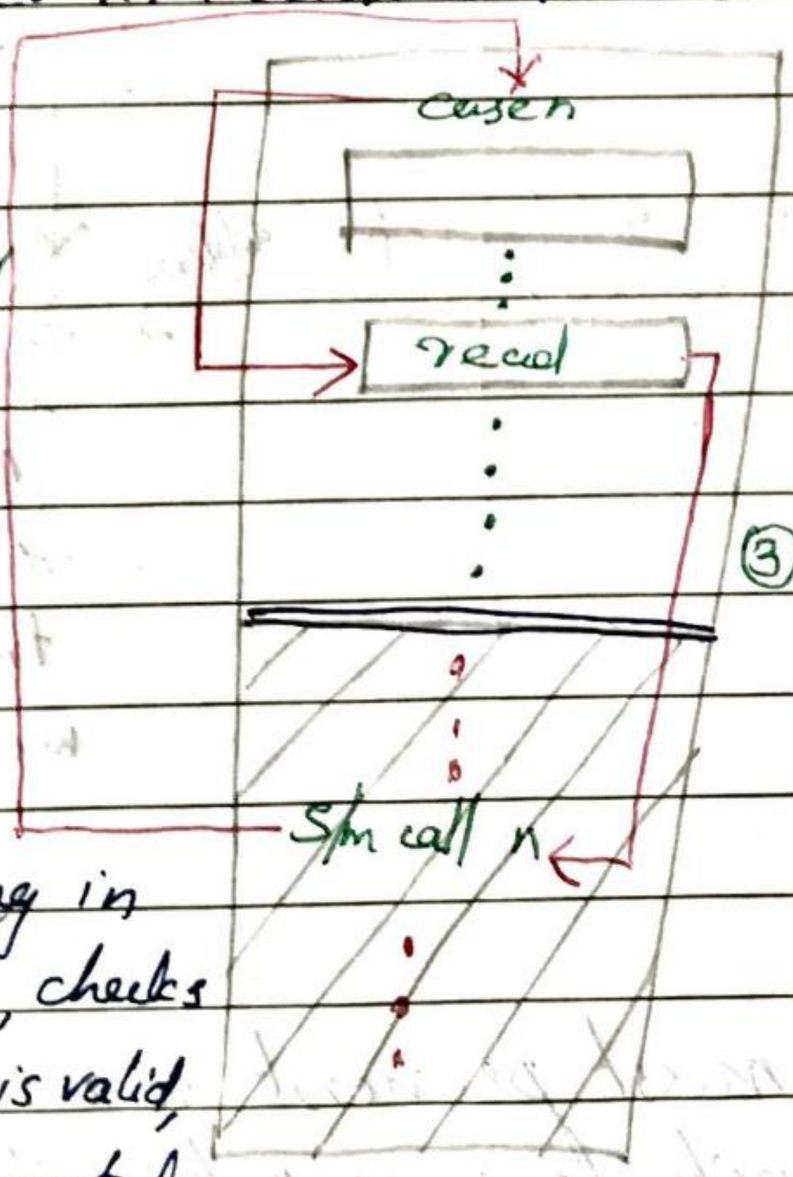
Scanned by TapScanner

from one another. The h/w allows privileged inst's to be executed only in monitor mode. If an attempt is made to execute a privileged inst^n in user mode, the h/w doesn't execute the inst^n, but rather treats the inst^n as illegal & traps it to the O.S.

I/O Protection

A user pgm may disrupt the normal op^n of the s/m by issuing illegal I/O instructions, by accessing mem. locations within O.S itself.

To prevent users from performing illegal I/O, we define all I/O inst's to be previleged inst's. Thus users cannot issue I/O instructions directly, they must do it through the O.S. For I/O protection to be complete, we must be sure that a user pgm can never gain ctrl of the computer in monitor mode.

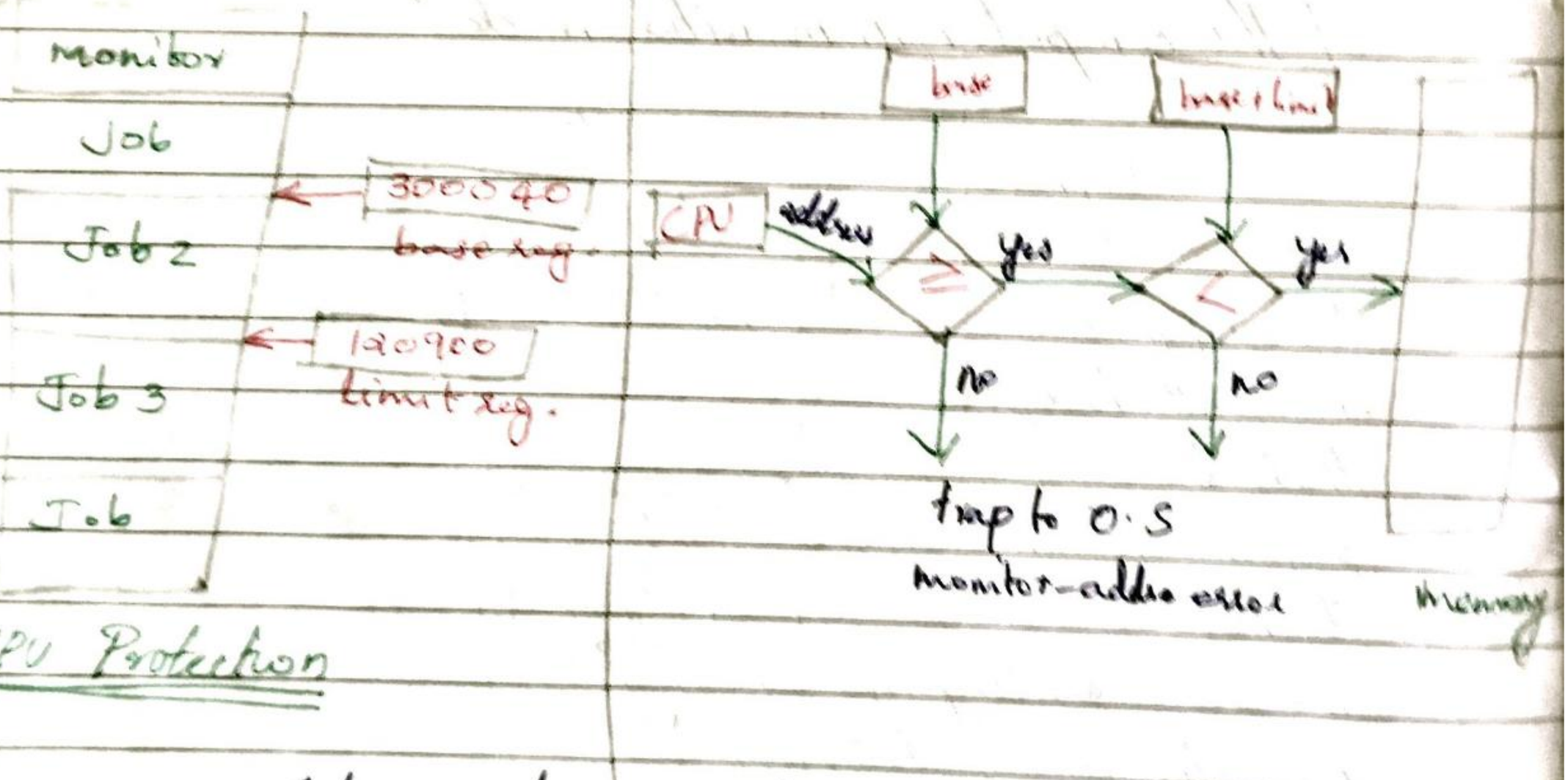


Scanned by TapScanner

Memory Protection

We want to protect the O.S from access by user pgms, & to protect user pgms from one another. To separate each pgm's memory space, we need the ability to determine the range of legal addresses that the pgm may access, & to protect the mem outside that space. We can provide this protection by using two regs., a base & limit.

The base reg. holds the smallest legal phy. memory address; The limit registes contains the size of the range. For eg: if the base reg holds 300040 and limit reg is 120900, then limit reg. is 120900, then the pgm can legally access all addresses from 300040 through 420940 inclusive



CPU Protection

We must prevent a user pgm from getting stuck in an infinite loop or not calling sys services, & never returning ctrl to the O.S. To accomplish this goal, we can use a timer. A timer can be set to interrupt the comp after a specified period. The period may be fixed or variable.

Scanned by TapScanner

A variable timer is generally implemented by a fixed-rate clock & a counter. The O.S sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches zero, an interrupt occurs.

Scanned by TapScanner

MODULE IV
FILE MANAGEMENT

Files:

General different storage media - magnetic disks, magnetic tapes & optical disks. The O.S provides a uniform logical view of inf^{n} storage. The O.S abstracts from the phy. properties of its storage devices to define a logical storage unit (the file). non volatile.

A file is a named collection of related inf^{n} that is recorded on secondary storage.

File Attributes:

Name: The symbolic file name is the only inf^{n} kept in humans readable form.

Identifier: The unique tag, usually a number, identifies the file within the file s/m; it is non-human-readable name for the file.

Type: This inf^{n} is needed for those s/ms that support diff types.

Location: This inf^{n} is a ptr to a device and to the loc^{n} of the file on that device.

Size: The current size of the file and possibly the max. allowed size are included in this attribute.

Protection: Access Control inf^{n} determines who can do reading, executing and so on.

Time, date & user identification: The inf^{n} may be kept for last creation, last modification & last use. These data can be useful for protection, security & usage monitoring.

The inf^{n} about all files is kept in the directory structure that also resides on secondary storage.

Scanned by TapScanner

File Operations

* Creating a file: Two steps are necessary to create files. First space in the file s/m must be found for the file. Second an entry for the new file must be made in the directory. The directory entry records the name of the file & the location in the file s/m, and possibly other inf^{n}.

* Writing a file: To write a file, we make a s/m call specifying both the name of the file and the inf^{n} to be written to the file. Given the name of the file, the s/m searches the directory to find the location of the file. The s/m must keep a write pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.

* Reading a file: To read from a file, we use a s/m call that specifies the name of the file and where the next block of the file should be put. Again, the directory is searched for the associated directory entry, and the s/m needs to keep a read ptr. to the location in the file where the next read is to take place. Once the read has taken place, the read ptr is updated.

* Repositioning within a file: The directory is searched for the appropriate entry and the current file position is set to a given value. Repositioning within a file doesn't need to involve any actual I/O. This file op^{n} is also known as file seek.

Scanned by TapScanner

classmate
Date
Page

*Deleting a file: To delete a file, we search the directory for the named file. Having found ~~be~~ the associated directory entry, we release all file space, so that it can be reused by other files & erase the directory entry.

*Truncating a file: The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged but lets the file be reset to length zero & its file space released.

Other common op^n's include appending new inf^n to the end of an existing file and renaming an existing file.

Open file table: The O.S keeps a small table containing inf^n about all open files. When a file op^n is reqsted, the file is specified via an index into this table, so no searching is required. When the file is no longer actively used, it is closed by the process and the OS removes its entry in the open-file table. The open file table has an open count associated with each file, indicating the no. of processes that have the file open. Each close decreases this count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open file table.

The pieces of inf^n are associated with an open file.

*File pointer: This pointer is unique to track the last read-write location as a current-file position ptr.

*File open count: Multiple processes may open a file, the s/m must wait for the last file to close before

Scanned by TapScanner

removing the open-file table entry. This counter tracks the no. of opens & closes and reaches zero on the last close. The s/m can then remove the entry.

* Disk location of the file: Most file op^{n}s require the s/m to modify data within the file. The info needed to locate the file on disk is kept in memory to avoid having to read it from disk for each op^n.

* Access rights: Each process opens a file in an access mode. This info is stored on the per-process table so the O.S can allow or deny subsequent I/O requests.

File Types:

A common technique for implementing file types is to include type as part of the filename. The name is split into two parts:— a name and an extension, usually separated by a period character. The s/m uses the extension to indicate the type of the file and the type of op^{n}s that can be done on that file. For instance, only a file with a .com, .exe or .bat extension can be executed.

Files: → Low level Files — byte stream
→ Structured files - structured sequential file
— Record Oriented
→ DBMS
→ Low level File Impl^n
→ Blk mgmt
→ High level file Abstractions
→ Directory

Scanned by TapScanner

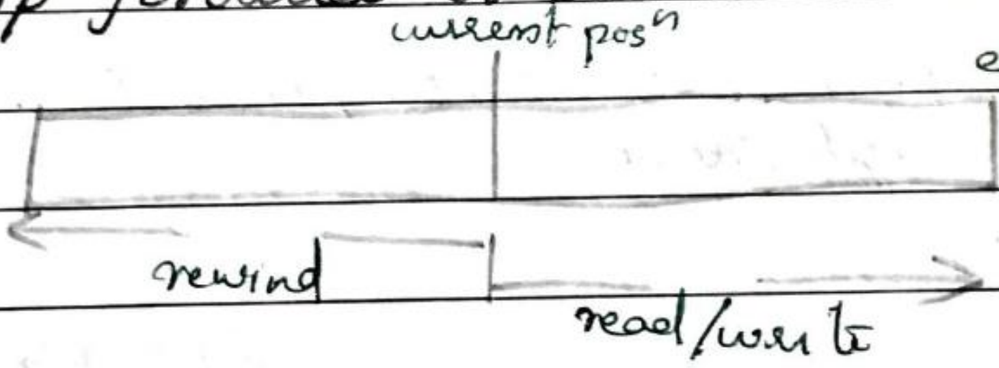
Date
Page

Access Methods

The inf^n in the file can be accessed in several ways. Some s/ms provide only one access method for files.

Sequential Access:

The simplest access method is sequential access. Inf^n in the file is processed in order, one record after the other. The bulk of the op^{n}s on a file is reads & writes. A read op^n reads the next portion of the file and automatically advances a file pointer, which tracks the I/o loc^n. Wr a write appends to the end of the file and advances to the end of the newly written material. Such a file can be reset to the begining and on some s/ms, a pgm may be able to skip forward or backward 'n' records for some integer n.



Direct Access:

A file is made up of fixed length logical records that allow pgms to read & write records rapidly in no particular order. The direct access method is based on a disk model of a file, since disks allow random access to any file block. We may read block 14, then read blk 53 & then write blk 7.

* Immediate access to large amounts of inf^n.

For the direct-access method, the file op^{n}s must be modified to include the blk no. as a parameter.

Ex; read n, where 'n' is the blk no. rather than read next & write n then write next.

The block no. provided by the user to the O.S is a relative blk no. A relative blk no. is an

Scanned by TapScanner

ans index relative to the begining of the file. The first relative block of the file is 0, the next is 1, and so on even though the actual absolute disk address of the block may be 14703 for the first blk & 3192 for the second.

Not all O.S support both seq. & direct access for files.

Seq. access	Impl^{n} for direct access
reset	cp = 0
read next	read cp; cp = cp+1;
write next	write cp; cp = cp+1

Indexed Sequential Access Method

To find a record in the file, we first search the index, and then use the ptr. to access the file directly and to find the desired record.



The i/p to an indexed seq. file s/m uses

Scanned by TapScanner

Low-Level Files:

A byte stream is a named sequence of bytes indexed by the non negative integers; a byte stream as a large array of bytes each with index. The first byte has index zero, the second has 1 & so on. In byte streams files, every process that opens a file uses a file positon to reference a particular byte in the file. When the file is opened the file positon references the first byte in the file.

The basic op's on a byte stream file:

open(filename): The filename is a character string that uniquely identifies a file. The op^n prepares the file for reading or writing. The open() op^n causes inf^n in the file descriptor to reflect that the file is being put into use.

close(fileID): This op^n deallocates the internal descriptors created when the file was opened, along with any other resources used by the file s/m to manage the byte stream I/O.

read(fileID, buffer, length): This op^n copies a block of length bytes begining at the current file pos^n into the buffer for the specified file identifier.

write(fileID, buffer, length): This op^n writes length bytes of the inf^n from the buffer to the current file pos^n & then increments the file positon by length.

seek(fileID, filePositon): This op^n changes the value of the file pos^n to the value of the parametes, filePositon. Subsequent read/write op^{n}s reference the data whose index corres. to the new value of the file pos^n.

Structured Files:

A structured file can be used to hold any kind of inf^n, including absolute pgm images, relocatable obj pgms, libraries, numeric data organized acc. to the needs of some set of appl^{n}s. Structured files are

Scanned by TapScanner

represented on the storage medium as a coll^n of blks.

Record Oriented Seq. Files:

A structured seq. file is a named seq. of logical records, indexed by nonnegative integers. As with byte stream files, access to the file is defined by a file pos^n, but in this case the pos^n indexes records in the file instead of bytes. Op^{n}s on the file are as follows:

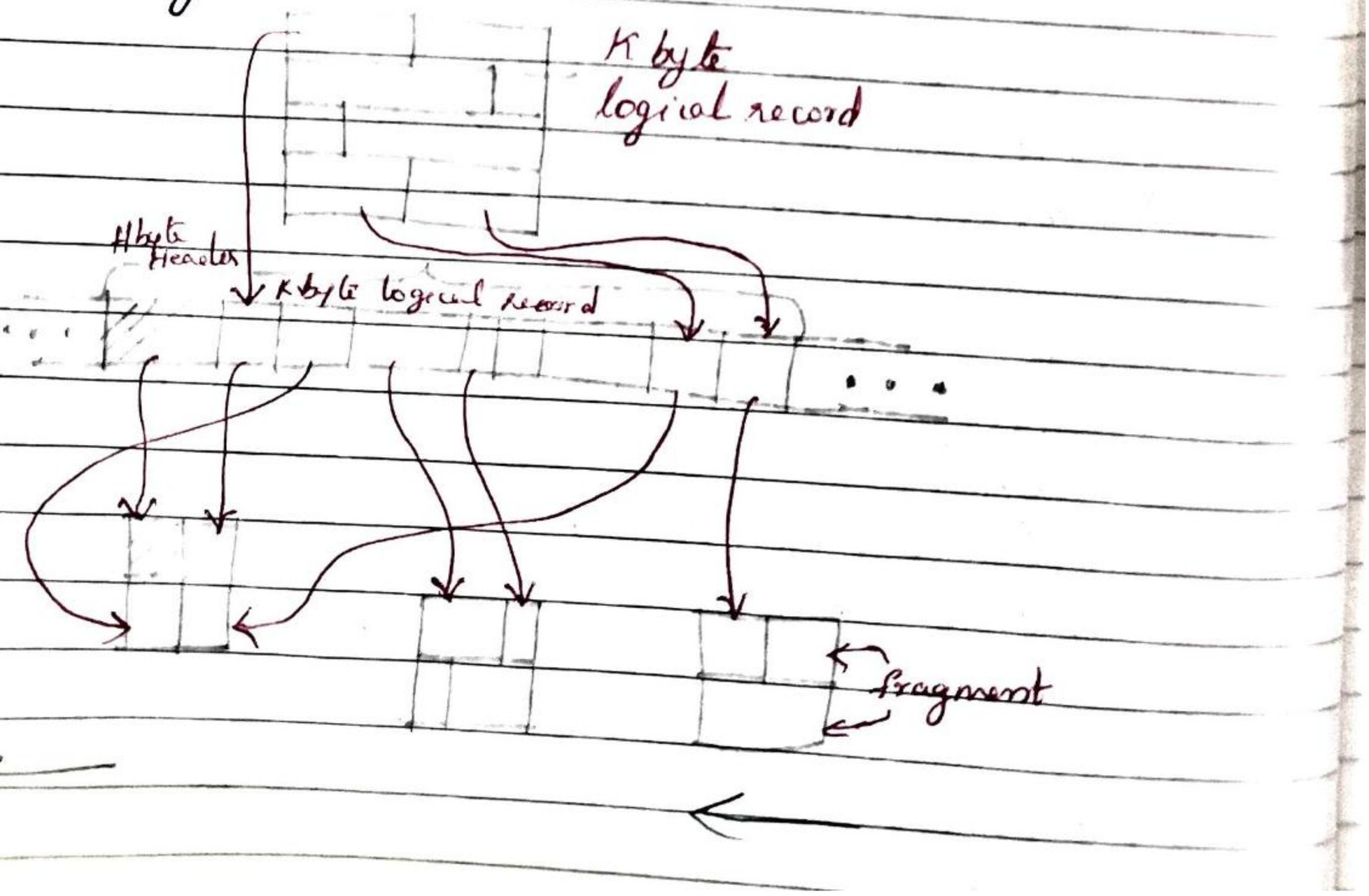
Open (Filename): Performs the same fn as open() & returns file identifier.

Close (File ID): Performs the same fn as close() on the given file identifier.

getRecord (FileID, record): Returns the record addressed by file pos^n.

putRecord (FileID, record): Writes the designated record at the current pos^n.

seek (File ID, position): Moves the file pos^n to point at the designated record.



Scanned by TapScanner

In the logical-physical record encoding, the fields of a data structure are marshaled into a list of bytes before they are stored on the device. When the record is read, the data is unmarshalled and the structure is reintroduced.

Suppose records are of different sizes - some holding an address and others holding a bitmap image. Then there must be a set of record block translation functions to be applied for the diff. types of records. This can be accomplished by prespecifying the set of record types or by using runtime decoding. In traditional record-oriented file structures, the record types ~~or~~ are built into the file s/m itself. In these cases, the access fns are op^{n}s designed & implemented at the time the file mngr is designed & implemented. For eg:, a file mngr may support strings of chars. where a record is a string. Now a putRecord() op^n might write a specific no. of bytes onto a stream to the storage device.

Indexed seq. file s/ms uses a more general read/write i/f than do pure seq. files:

getRecord(fileID, index): Returns the record with an index field set to the index value.

putRecord(fileID, record): As an access op^n, writes the designated record into the file at a pos^n chosen by the file s/m and then returns the value of the index field as a result.

deleteRecord(fileID, index): Removes the record with the index field set to the index value.

Indexed seq. file

Database Management Systems:

Multimedia Storage:- highly structured files, to contain nos, chars, formatted text, exe pgms, graphics, images,

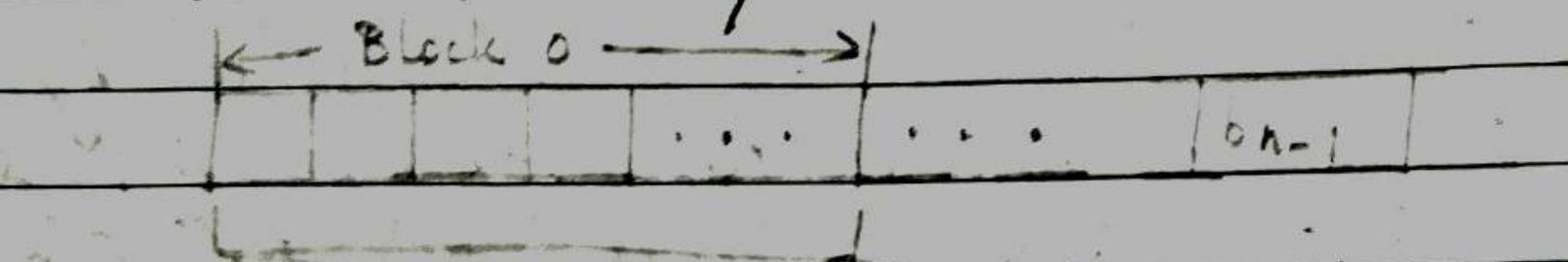
Scanned by TapScanner

audio/video streams & so on.

→ There must be considerable translation fnality in appl^{n}s that use the multimedia document.

→ The file mngr must provide a means for the appl^{n} to export very comprehensive access routines to be used by the file mngr when mnging the multimedia file.

Low Level File Implementations:



Mag. Disk Realization

Mag. Tape realization.

The file mngr provides a minimal mechanism to enable a process to read from and write to inf^{n} on storage devices. The file mngr implements the file i/f on a variety of storage devices; some of these devices only allow seq. access to their data, & others allow random access to the blocks on the device. The file mngr implements stream block translation so that the appl^{n} can read/write the file as a sequence of bytes. A tape realization of the low-level file abstraction requires the logical byte streams to be mapped to logical blks, which are then mapped onto the phy. records on tape. The contiguous bytes are grouped into logical records & then are stored in phy. records.

Bytes 0 through k-1 are stored in logical records, which is stored in phy. record 0, where k is the no. of bytes in a tape phy. record. The order of logical blocks

Scanned by TapScanner

on the byte stream maps to contiguous blks in a mag. tape realization of the low level file. So for k-byte physical blocks on the tape, bytes b_0 to b_{k-1} are stored in phy. block 0 of the tape; bytes b_k to b_{2k-1} are stored in phy. block 1 on the tape.

A disk realization also maps logical blks of contiguous bytes from the byte stream but the mapping of logical to phy. blks will not normally be to contiguous blks on the disk. Instead, the disk realization provides a mechanism for mnging a much more general coll^n of blks to store the bytes of a particular file so that they can be accessed as if they were stored as a contiguous byte stream.

The OPEN() & CLOSE() Operations:

When a file is created, the file mngr creates a data structure instance called a file descriptor, in which it stores detailed inf^n about the file. This file descriptor is kept on the storage device with the conts. of the file. Different file mngrs store diff. inf^n in the file descriptor, but most maintain the foll. inf^n:

* External name: A char. string name for the file used at the cmd line or from an appl^n pgm. This is the symbolic name that users associate with the file.

* Sharable: A field that specifies whether multiple processes can have the file open at the same time. This field identifies the file as private, read sharable, ex^n sharable, write sharable & so on.

* Owner: The identifier associated with the user whose process created the file. In some cases, ownership may be

Scanned by TapScanner

assigned by some other policy. The file s/m may allow a process to pass ownership to another users.
* Protection Settings: The min. modes of protection are for reading & writing. The protection settings indicate whether the owner is able to read from or write to the file without overriding the protection.
* Length: The no. of bytes contained in the file.
* Time of creation: The s/m time at which the file was created.
* Time of last modification: The s/m time at which the file was last written to.
* Time of last access: The s/m time at which the file was last read from or executed or had any other opn performed on it.
* Reference count: The count of the no. of directories in which the file appears if the directory s/m allows a file to appear in more than one.
* Storage device details: The field that contains the details of how the blks in the file can be accessed.

The file manager checks to be sure that the process is authorized to access the file. This authorization involves comparing the protection flags from the ext. file descriptor for the file with the protection keys held by the user/process. If the process attempts to open a file for which it doesn't have appropriate rights, the authorization procedure will disallow the open() opn. Once the file mngr has determined that the process is authorised to access the file, it creates its own internal version of the file descriptor, called open file descriptor. The open file descriptor contains all the inf^{n} that is in the ext. file descriptor along with any additional file specific inf^{n} that the

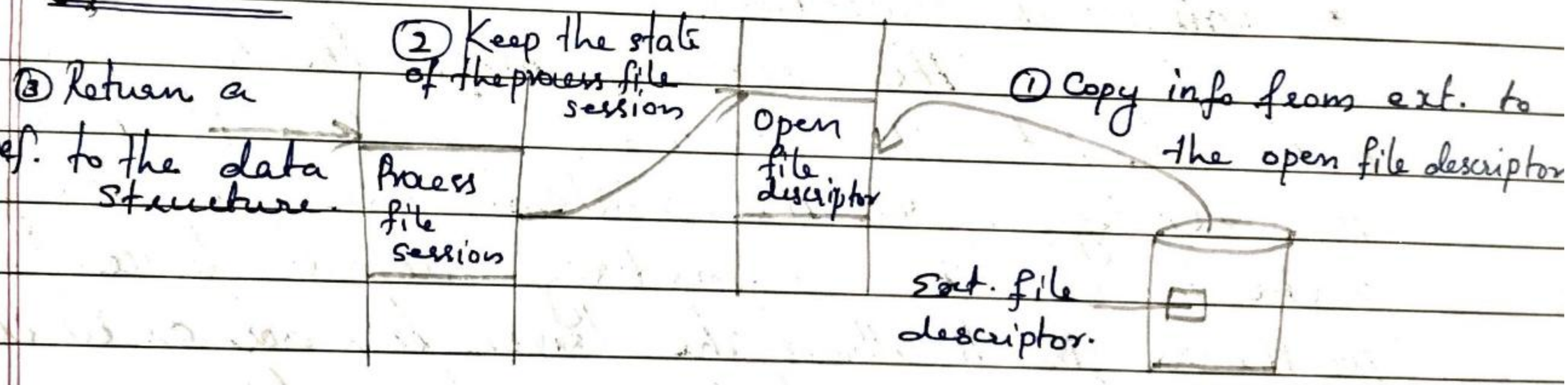
Scanned by TapScanner

file manager may choose to keep.

*Locks: Locks can be read locks, meaning the file is currently opened for reading. If the file is shareable other processes can open it only for reading. A write lock indicates a process has opened the file for writing.

* Current state: The state of the file, this can be archieved, meaning that it has been written to a very low level of the storage hierarchy & cannot be opened for use without a significant delay. If the state is closed, the file resides in an online storage device & can be opened for use within ms. The file can be open for reading, open for writing, open ~~of~~ for ex^n, open for appending & so on.

Open():



1) Locates the file and its ext. file descriptor on the storage device.

2) Extracts inf^n regarding the file from the ext. file descriptor & inf^n regarding the process from the process descriptor.

3) Authenticates that the process is permitted access to the file.

4) Creates an entry in an open file descriptor table to maintain basic inf^n about any process's use of the file.

5) Creates an entry in a process file session table to keep track of the state of each specific process's interaction with the file.

Scanned by TapScanner

The close() opⁿ causes the file mangr to complete all pending opⁿs, to release I/O buffers, to release locks the process may hold on the file, to update the ext. file descriptor, and to deallocate the file status table entry.

Block Management

→ refers to the task of keeping track of which storage device blocks are associated with which file. There are 3 well known techniques for organizing the phy. storage blocks for a file.

* Contiguous Allocation:

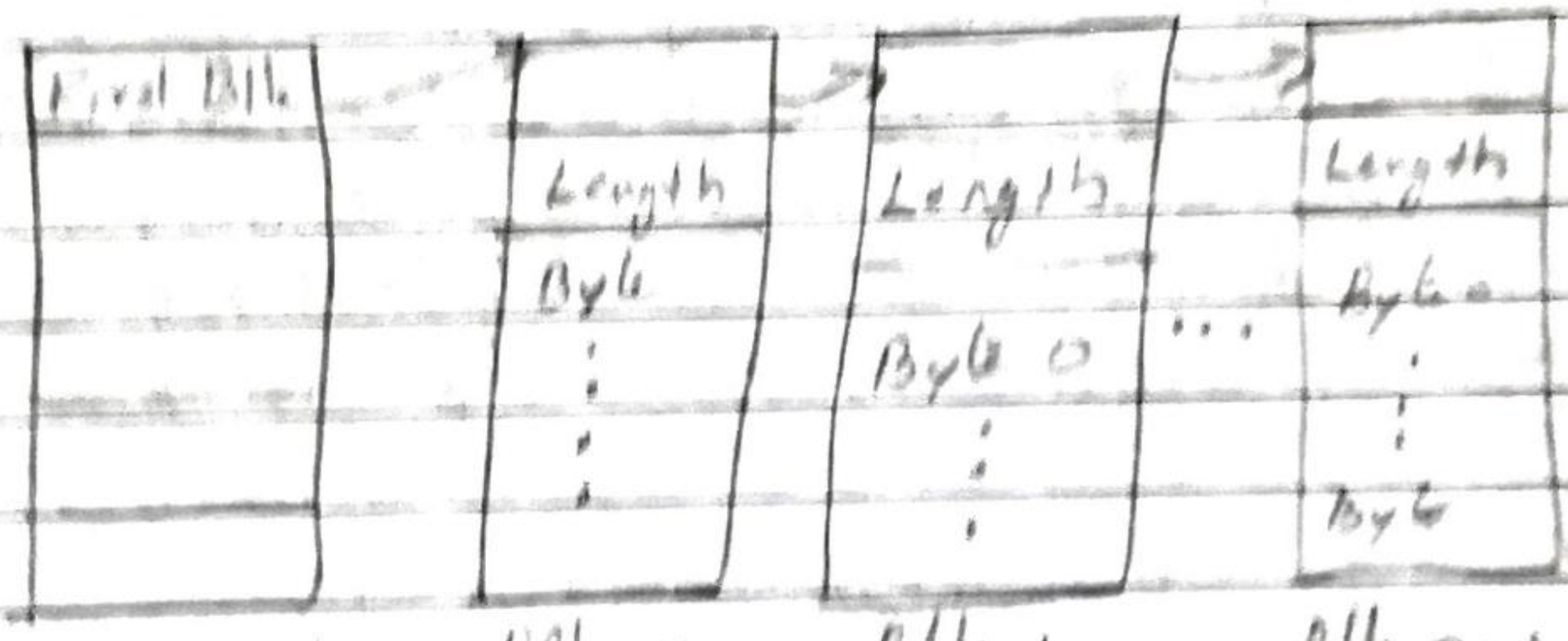
This strategy maps the N logical blks in the file into N contiguously addressed physical blocks. This strategy makes the randomly accessed storage device behave like a sequentially accessed device for that particular file. It allows the driver to read or write to an entire file in a short amount of time.

~~Cons~~ Contiguous allocation doesn't accomodate dynamic file sizes, since it maps logical structure to the physical structure. If a file is stored in some contiguous set of blks & data is added to the end of the file then either the next contiguous phy. blk on the storage device must be made available or the entire file must be copied to a larger group of unallocated contiguous blks.

Scanned by TapScanner

LINKED LISTS

Linked lists of blks use explicit ptrs among an arbitrary set of phy. blks making up the file. Logical blk i+1 need not be physically located near logical blk i, since i will contain a link that addresses the phy blk that contains logical blk i+1. The file status table entry for a file will include a copy of the file position & a ptr to the first device blk used in the file.

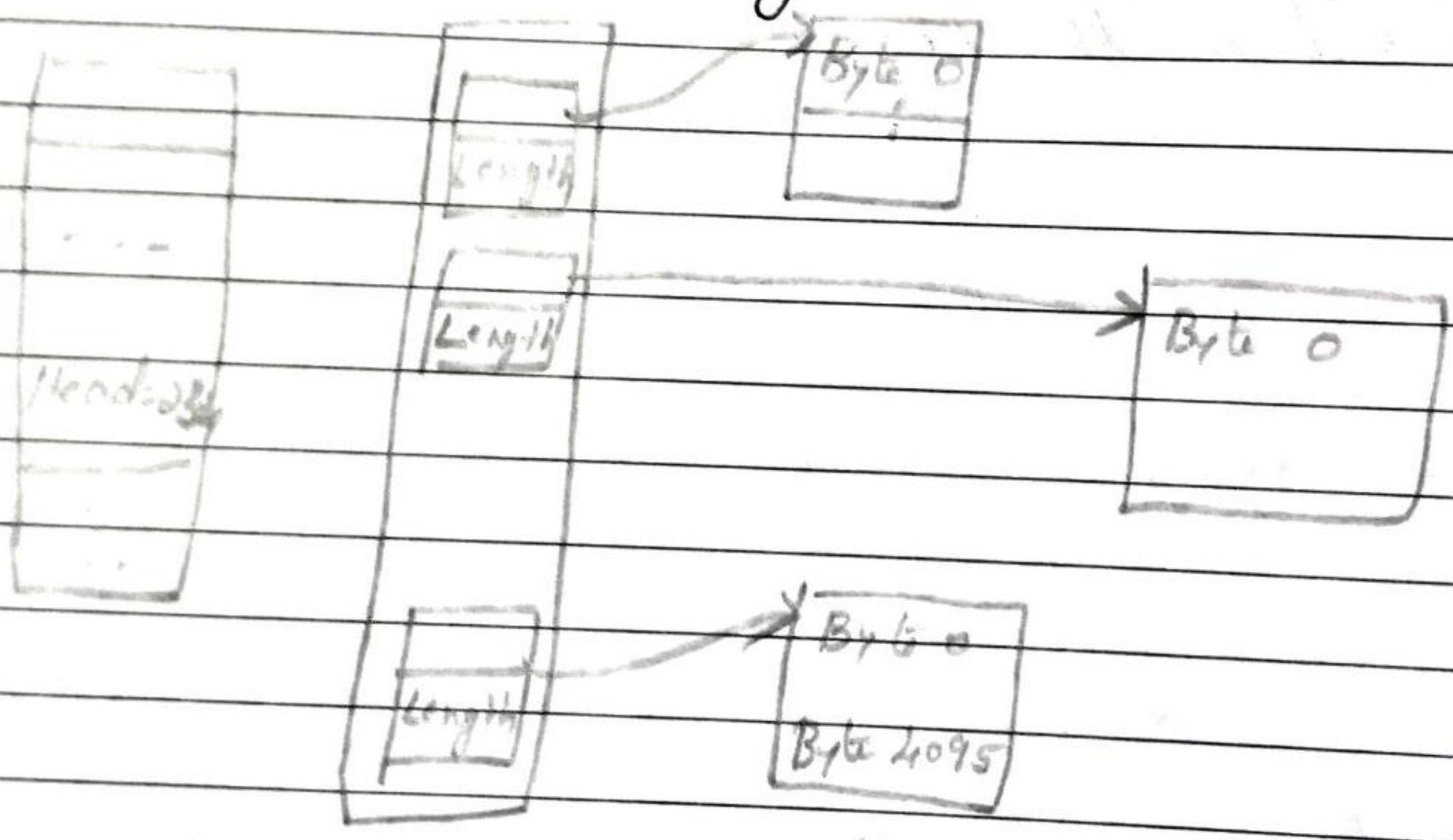


Each blk in the file contains overhead inf" used by the file manager. In the eg: two fields in a blk are used to hold the next-blk ptr & a count of the no. of bytes actually stored in the block. The length field enables the file mngr to store variable no. of bytes in each block, thus potentially enabling the file mngr to read reduce the no. of blk allocations & deallocations in a dynamic file.

Scanned by TapScanner

Indexed Allocation

— uses a unique block in the file as an index for all the other blks used to store data. The block length field is shown in the index blk along with the blk ptr, which simplifies file ptr placement bc this value can be used when the file manager uses the ptrs in the index blk for seeking.



Blk N-1

In indexed allocation, the file mngr has an in-memory table that provides the disk addr of each blk in the file. This requires

Scanned by TapScanner

Storage Abstractions

The approaches for supporting storage abstractions

Structured Sequential Files:

- contain collections of logical records. Records are referenced using an index as in byte stream files. The two types of files differ in that structured sequential files cause a full record to be read from or written to.

File mngrs that handle structured files must contain the same kinds of inf^{n} that are kept for low-level file as well as additional fields to support data structure.

* Type: A tag describing the type of the file. For eg: the type field might distinguish relocatable objt files from absolute objt from absolute objt files.

* Access methods: A composite of functions to be used to read from, write to, append to, update or otherwise access file.

* Other: Specific structured file types may had have other fields to represent the relationship of this file mngs handling this file type.

Indexed Sequential Files:

Each record contains an index field used to select it from the file. An application pgm provides an index value with each read or write opⁿ. The file mngs implements a mechanism

Scanned by TapScanner

to search the storage device to find the phy. blk that contains the record. The impl^{n} may use the same file structure as structured seq. files. The record index ordinarily determines the order in which the records are stored in the file: Record 0 is stored as the first record in the first blk, record 1 is the second record in the first blk & so on, until the first blk is full. The next record is stored in the second blocke & so on.

Database Management Systems (DBMS)

Multimedia Documents

Scanned by TapScanner

Mod 4

Protection & Security

Protection

to increase reliability of any complex s/m that makes use of shared resources.

Reason behind protection:

* need to prevent mischievous, intentional violation of an access restriction by a user. An unprotected resource cannot defend against use by an unauthorised or incompetent user. A protection-oriented s/m provides means to distinguish b/w authorized & unauthorized usage.

The role of protection in a computer s/m is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways.

* Some are fixed in the design of the s/m
* Others are formulated by the mgmt of a s/m
* others are defined by the users to protect their own files & pgms.

Protection mechanism & policies.

Mechanisms determine how something will be done; policies decide what will be done.

Domain of Protection

A computer s/m is a collection of processes & objects. By objects, we mean both H/w objts & S/w objts. H/w objts - CPU, memory segments, printers, disks & tape drives) S/w objts - files, pgms & semaphores).

Scanned by TapScanner

Each objt has a unique name that differentiates it from all other objts in the sm, & each can be accessed only through well-defined and meaningful operations

A process should be allowed to access only those resources for which it has authorization.

Protection Domain

- specifies the resources that the process may access. Each domain defines a set of objts & the types of op^n's that may be invoked on each object.

The ability to execute an op^n on an objt is an access right. A domain is a collection of access rights each of which is an ordered pair <objt-name, rights-set>. For eg, if domain D has the access right <file F, {read, write}> then a process executing in domain D can both read & write file F; it cannot, however perform any other op^n on that objt.

Domains do not need to be disjoint; they may share access rights.

Security Mechanisms

Protection mechanisms that can be used to ctrl access to a secure entity. A security policy is a specification that determines how the protection mechanisms should be used. For eg: a protection mechanisms might be able to selectively allow certain subjects to access secure entities. Then security policy will need to specify which of the subjects should have access to secure entities & which subjts should not. ←

Scanned by TapScanner

Access Matrix

The rows of the access matrix represent domains and the columns represent objects. Each entry in the matrix consists of a set of access rights. B/c the column defines objts explicitly. The entry access (i,j) defines the set of operations that a process, executing in domain D_i can invoke on objt O_j.

objt / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

4 domains & 4 objts, 3 files (F_1, F_2, F_3) & one printer.
When a process executes in domain D_4, it can read files F_1 & F_3. In D_4, read & write F_1 & F_3.

We must ensure that a process executing in domain D_i can access only those objts specified in row i, and then only as allowed by the access matrix entries.

The users normally decide the conts. of the access-matrix entries. When a user creates a new object O_j, the column O_j is added to the access matrix with the appropriate initialization entries. The user may decide to enter some rights in some entries in column j & other rights in other entries as needed.

When we switch a process from one domain to another, we are executing an op^n (switch) on an objt (domain). We can ctrl domain switching by including domains among the objts of

Scanned by TapScanner

objts of access matrix.

Processes should be able to switch from one domain to another. Domain switching from domain D_i to domain D_j is allowed to occur iff the access right switch $\in$ access(i, j)

object / domain	F_1	F_2	F_3	printer	D_1	D_2	D_3	D_4	Domain as objts.
D_1	read		read			switch			
D_2				print			switch	switch	
D_3		read	execute						
D_4	read write		read write		switch				

From D_1 → D_2

A process executing in domain D_2 can switch to domain D_3 or to domain D_4. A process in domain D_4 can switch to D_1 & in one in domain D_1 can switch to D_2.

Allowing controlled change to the conts. of the matrix entries requires 3 additional op^ns: copy, owner & control.

Copy

The ability to copy an access right from one domain (or row) of access matrix to another is denoted by an asterik (*) appended to the access right. The copy right allows the copying of the access right only within the columns for which the right is defined.

→ A right is copied from access(i, j) to access (k, j); it is then removed from access(i, j), this action is a transfer of a right rather than a copy

→ Propagation of the copy may be limited - ie; when the right R^* is copied from access(i, j)

Scanned by TapScanner

to access (k, i) only the right R (not R*) is created.
ie; 3 copy rights - copy, transfer, limited copy.

Owner

A mechanism to allow addition of new rights & removal of some rights. The owner ctrl these opns. If access(i, j) includes the owner right, then a process executing in Domain D_i can add & remove any right in any entry in column j.

Copy fig

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

Owner fig

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write*
D_3	execute		

object / domain	F_1	F_2	F_3
D_1	owner execute		
D_2		owner read* write*	read* owner write*
D_3		write	write

Domain D_1 is the owner of F_1 & thus can add & delete any valid right from F_1. Here D_2 D_3 removed execute access right of D_3 on F_1.

Scanned by TapScanner

Contest for Protection & Security.

A protection mechanism may rely on physical isolation of the computer or on logical isolation that prevents remote users from accessing the computer with telecommⁿ n/ws.

An intruder is a subject that gains access to a computer by spoofing the authentication mechanism, or by finding a loophole in the authentication policy. An intruder might be a person or a s/w probe. Here are some well-known types of attacks.

* **Masquerading**: The intruder is able to fool the authentication mechanism by pretending to be a diff. user. The intruder approaches the authentication mechanism & provides login identification. The authentication requires additional infⁿ to authenticate the identificⁿ for eg: a password. The intruder supplies the additional infⁿ & gains access to the m/c masquereading as an authenticated user.

* **Guessing a login & password**: Naive users tend to use naive passwords. If the intruder knows the person for whom they are attempting to masquerade, then they may be able to make good guesses about the password.

* **Snooping for login & password**: Notice that "About Me" and personal web page are a rich source from which an intruder can gather infⁿ about a user. As another eg: suppose that a human user keeps a

Scanned by TapScanner

m/c readable file with paswds to computers. If the intruder can ~~gather~~ find this file and manage to copy it, then he or she will be able to masquerade as that user.

*Alternate entrances: Since computer s/ms are connected to n/ws there is another avenue of attack that the intruder can use. A classic security bug was that a popular OS was installed with a default password used by the community of mail pgms. If the s/m admin didn't change this pswd, then any intruders could masquerade as another mail pgm to gain access to ~~g~~ the s/m.

Software Intruder:

S/w intruder silently moves into the comp. and begins to violate the security policy.

S/w Intruder attacks:

Confinement & allocating rights: The idea behind confinement is that any interaction b/w two threads in two processes results in inf^n implicitly being learned about the communicating partner. In s/w, the goal of confinement is to prevent any intruder process from learning or inferring any unauthorized inf^n from its interactions with a process that has a secure entity.

Trojan horse: An intruder provides some entity to the s/m, which the s/m accepts & incorporates into its trusted s/w. Viruses are eg: A virus is embedded in free s/w, a device driver update, or other unit of s/w that is likely to be installed without proper authentication. Once the Trojan horse has been installed, the s/w intruder has

Scanned by TapScanner

potentially gained entrance into the trusted domain of the computer.

Denial of Service: The purpose of a denial of service attack is not to access a secure entity, but rather is to provide so much bogus external work for a system that it is unable to perform its real work. A denial of service attack can be launched by having a program that makes execute slow that rapidly transmits numerous reqts to the server.

Authentication

Authentication refers to the task of ensuring that a subject who attempts to access the secure entity is in fact actually the subject that it claims to be. For eg: if a subject claims to be the Sheriff of Nohern then sys's authentication mechanism is responsible for checking to be sure that the entity is the Sheriff of Nohern.

External authentication is concerned with determining whether a user is the person he or she claims to be. For eg:, if a person logs into a sys with a particular acc name, then the basic ext. authentication mechanism will check to ensure that the person logging in is actually the person who is supposed to be using the account.

Internal authentication is intended to ensure that an executing thread cannot appear to be owned by a user other than its true owner. This means that the thread must be authenticated to be associated with a given process, & the process must be authenticated to be associated with a particular user.

Scanned by TapScanner

classmate
Date
Page

External User Authentication:

The user's initial interaction with the comp. s/m is with the login portion of the OS. In the login dialogue, the OS attempts to verify that users are who they claim to be. — External authentication

Scanned by TapScanner

User Authentication:

Authentication is based on one or more of three items: user possession (a key or card), user knowledge (user identifier and password) and a user attribute (fingerprint, retina pattern or signature).

* Password

→ most common approach where the user identifies herself by user ID or account name, she is asked for a password. If the user-supplied password matches the password stored in the s/m, the s/m assumes that the user is legitimate.

Different pswds may be associated with diff. access rights. For eg: diff. pswds may be used for each of the following file op's: reading, appending & updating.

* Password Vulnerabilities:

Passwords can often be guessed, accidently exposed, sniffed or illegaly transferred from an authorised user to unauthorised one, as we show next.

Two common ways to guess a password.

→ One is for the intruder to know the user or to have inf^n about the user. People use obvious inf^n (such as the names of their cats & spouses) as their pswds.

→ to use brute force, trying enumeration, or all possible combinations of letters, numbers & punctuation until the pswd is found.

The failure of pswd security due to exposure can result from visual or electronic monitoring. An intruder can look over the shoulder of a user (shoulder surfing) when the user is logging in, & can learn the pswd easily by watching the keyboard.

Scanned by TapScanner

Anyone with access to the n/w on which a computer resides could seamlessly add a n/w monitor allowing her to watch all data being transferred on the n/w (Snuffing), including users IDs & pswds.

Exposure is particularly severe pblm if the pswd is written down where it can be read or lost. Some s/ms force the user to select hard to remember or long pswds. This requirement can cause a user to record the pswd, providing much less security than s/ms that allow easy pswds.

Illegal transfer is the result of human nature. Sometimes users break account-sharing rules to help friends

Passwords can be either generated by the s/m or selected by a user. S/m generated pswds may be difficult to remember and thus users may write them down. User-selected pswds, are often easy to guess.

Encrypted Passwords:

One pblm with all these approaches is the difficulty of keeping the pswd secret within the computer. The UNIX s/m uses encryption to avoid the necessity of keeping its pswd list secret. The s/m contains a fn that is extremely difficult to invert, but it is simple to compute. i.e; given a value $f(x)$, however it is impossible to compute x. This fn is used to encode all pswds. Only encoded pswds are stored. When a user presents a pswd, it is encoded and compared against the stored encoded pswd. Even if the stored encoded pswd is seen, it cannot be decoded, so the pswd cannot be determined.

Scanned by TapScanner

The function f(x) is typically an encryption algm.

One-time Passwords:

When a session begins the s/m randomly selects and presents one part of a pswd pair & the user must supply the other part. This approach can be generalised to the use of an algm as a password. The algm might be an integer fn, for eg. The s/m selects random integers & present it to the user. The user applies the fn & replies with the correct result. The s/m also applies the function. If the two results match, access is allowed.

Here the s/m and user share a secret. The secret is never transmitted of overs a medium that allows exposure. Rather, the secret is used as input to the fn, along with shared seed. A seed is a random no./alphanumeric sequence. The secret & seed are used as i/p to the fn f(secret, seed). The result of this fn is transmitted as the pswd to the computer. B/c the comp. also knows the secret & the seed, it can perform the same computation. The next time that the user needs to be authenticated, another seed is generated & the same steps ensue. This time pswd is diff.

Biometrics

Palm or hand-readers are common to secure phy. access. The parameters can include a temperature map, finger length, finger width, & line patterns. These devices are too expensive.

Scanned by TapScanner

Secure Web Traffic

SSL - Secure Socket Layer & its successor transport layer security (TSL) are critical n/w components to support web-based authentication. SSL & ~~TSL~~ TSL protocol use cryptography so that when inf^{n} is transported over the internet, it is an encrypted form rather than ordinary clear text.

Kerberos

is a set of n/w protocols that can be used to authenticate access to one comp by a user at a diff. comp. over an unsecure n/w.

Kerberos assumes that inf^{n} flowing over the n/w could be tampered with during ~~transf^{n}~~ transm^{n}. & it doesn't assume that the O.S on 2 m/cs are necessarily secure.

Kerberos provides an authentication server & protocol to allow the client & server to transmit authenticated msgs to the companion process in a specific session.

Steps:

1. Client asks the authentication server for credentials of server process.
2. The Auth. Server returns the credential ticket & a session key with the latter encrypted using the client's key. The ticket has fields containing the client's identification & copy of session key encrypted using server's key.

Scanned by TapScanner

3. After the client obtains the credentials, it ~~dep~~ decrypts the ticket & session key keeping a copy of the session key so that it can authenticate inf^{n} from the server.

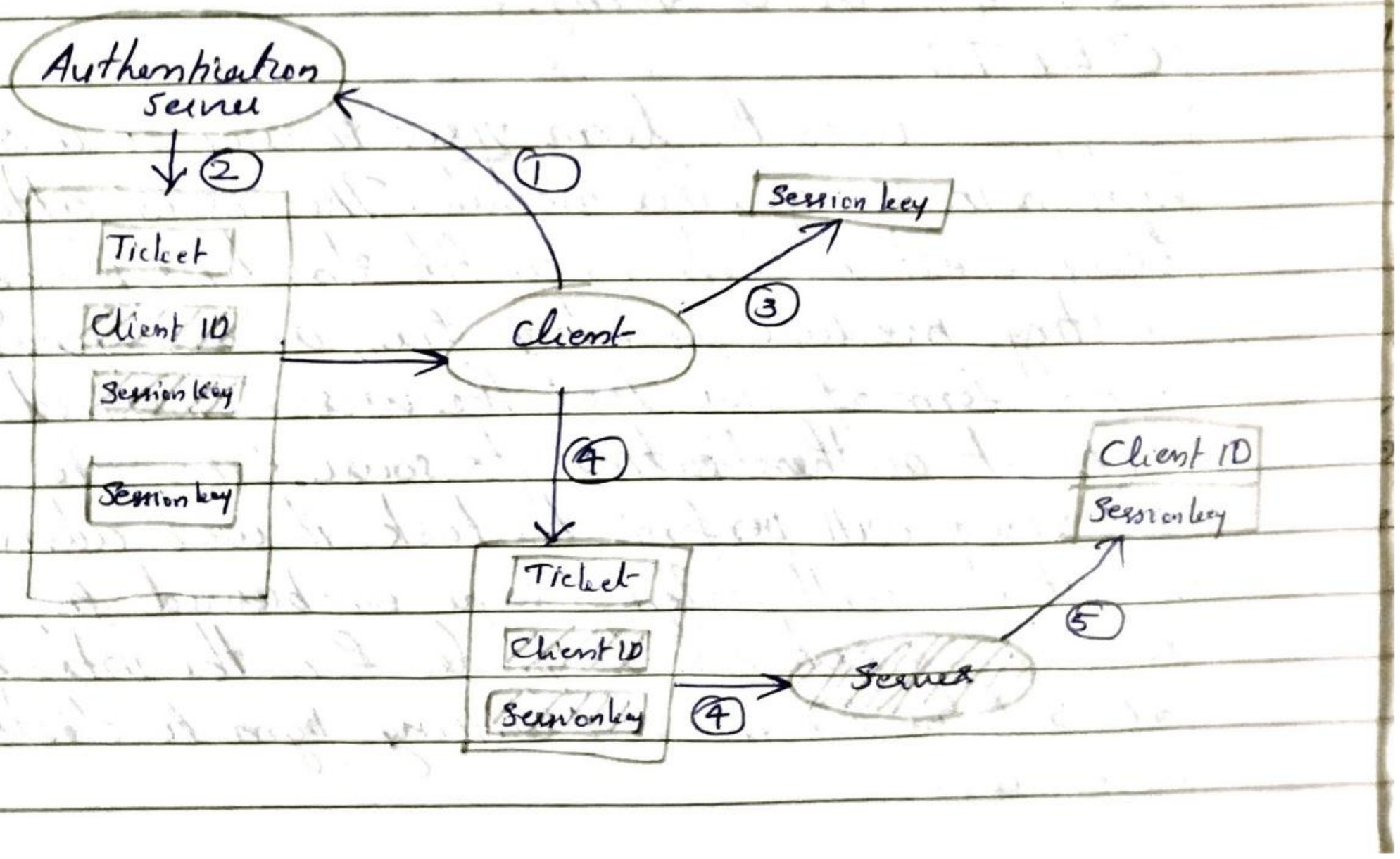
4. The client then sends a copy of the ticket with the encrypted fields intact to the server.

5. The server decrypts the copy of the ticket ~~with the encrypted~~ so that it can obtain a secure copy of both the client's identification & the session key.

Ticket Granting Service

AS-Db.

Since the authentication server can encrypt inf^{n} for the client & the server, it can give the client a "container" - the ticket - that contains inf^{n} that the client cannot read but that it can pass on to the server.



Scanned by TapScanner

Software Authentication

It is the task of verifying that a s/w module was created by a trusted source & that the s/w can be expected to perform only the actions that are publicly attributed to it. S/W authentication addresses the problm of preventing code that may contain a virus from being loaded into a protected computer domain.

CERT Internet security center uses the term "worm" to refer to s/w that actively attempts to penetrate a computer s/m by somehow fooling the authentication mechanism.

"Code Red virus" — that was designed to find websites that used a particular version of Microsoft s/w & then to exploit a vulnerability in that s/w causing it to overflow a buffer & penetrate the target m/c. CERT refers to the Code Red s/w as a worm.

CERT:

uses the term virus to refer to a s/w module hidden inside another module. The virus attempts to become established in a file s/m by replacing some existing module as a bug fix, as an upgrade, or as some form of s/w that the user is inspired to install without authenticating its source. The file containing the virus will perform the task it was advertised to do but it also will perform some unobserved fn, such as leaving undetected loopholes for the intruder to use at a later time or planting pgm to destroy s/m resources.

Scanned by TapScanner

There is a third class of s/w that has become popular with the wide use of web browsers: mobile code

Mobile Code: refers to s/w that is dynamically downloaded from a n/w source when it is needed, and then presumably destroyed after it has served its purpose. The most popular eg: of mobile code is probably Java applets that execute in a web browser. When a user requests conf. from a n/w site, the n/w site may download the ~~mod~~ mobile code along with inf^n conf. The idea is that by downloading the mobile code, subsequent interactions b/w user's web & the n/w nlc can be far more efficient

Certificates: are a basic mechanism for s/w authentication. A s/w is a digital signature that is associated with mobile code. The n/w host that supplies the mobile code & the user's computer exchange encrypted authentication inf^n regarding the mobile code. The mechanism will tell the target computer's user that mobile code from a particular source is to be downloaded. The user can then either explicitly accept or reject the download based on knowing the identity of the source of the mobile code.

Authorization ~~Authorization~~ Authorization

Authorization mechanisms are intended to ensure that users, processes or threads are allowed to use secure entities in the computer only if the protection policy permits such use.

After the login is complete, the user is free to direct the command line interpreter process to attempt access to arbitrary inf^n & resources.

Scanned by TapScanner

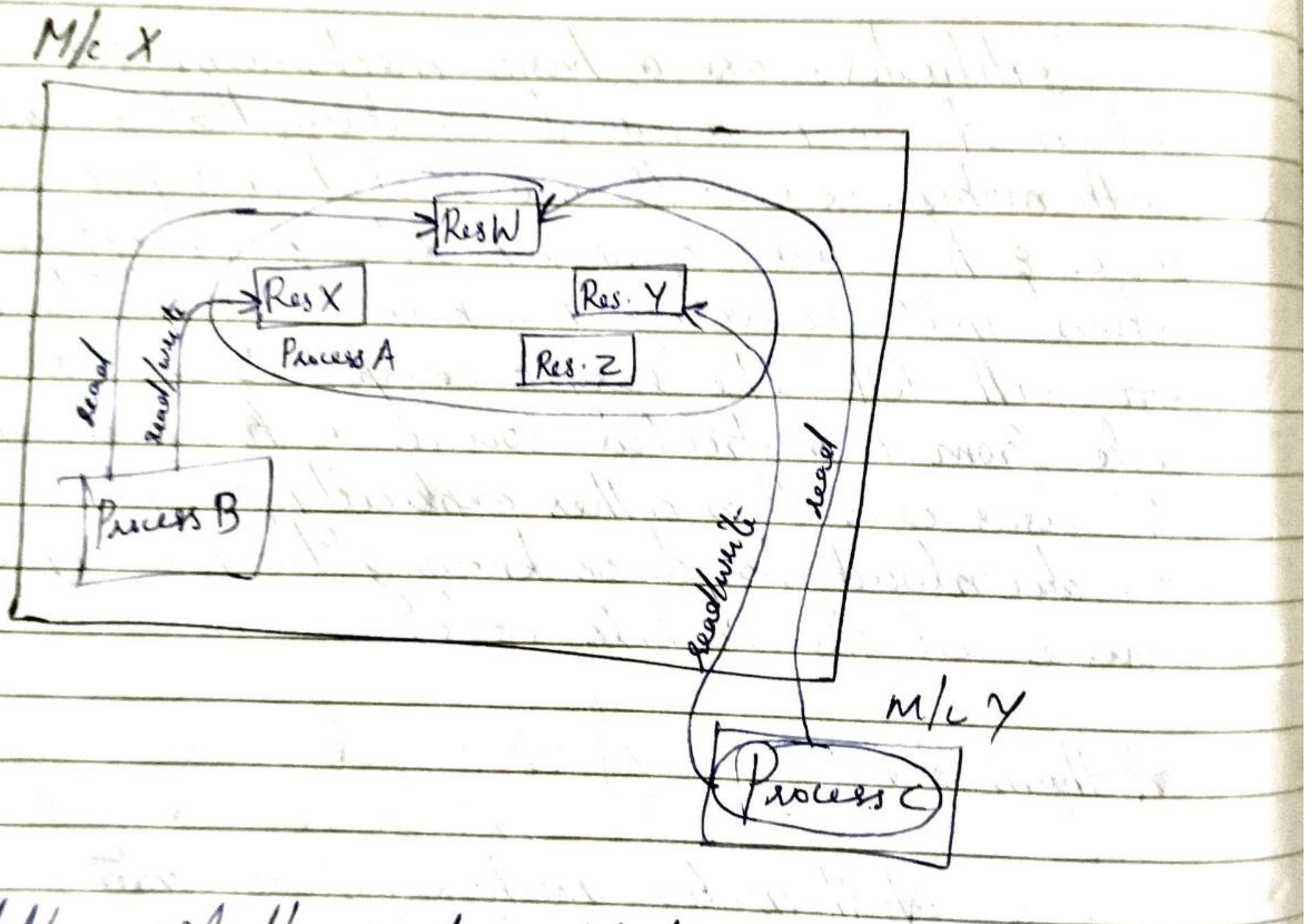
Authorization is part of the task of managing resources, particularly resource sharing. Suppose process A has resources W, X, Y & Z. Some of these resources are to be shared with other processes.

→ Process B has read access rights to W, while process C has read/write access rights to W.

→ Process B has read/write access to X, while C has no access to X.

→ Process C has read/write access to Y, while B has no access to Y.

→ Process A has private access to Z.



Adhoc Authorization Mechanisms

OS designers have always recognised the threat related to protection & security & have traditionally added some form of authorization mechanism

Scanned by TapScanner

Two Examples: File protection & room locks

File Protection: UNIX authorization mechanism.

Every UNIX user is identified by a user identification, called a UID. Each user also can belong to various groups of users, noted by group identification. GID.

ls -lg

user login → owner; owner group ID

Fig: -r--r-------- gjn rtsg 2335 Apr 11 1996 Makefile.
→drwxr-xr-x gjn ctrg 512 Feb 5 10:27 Tools
directories
---x--x--- gjn faculty 37541 Feb 4 12:42 bsmfix
→drwxr-xr-x gjn rtsg 512 Feb 5 11:34 bin
-rw-rw-r-- gjn faculty 853 Jan 6 1996 cover.txt

UNIX file protection marks specify the kinds of access that the owner, members of an explicit group, and all other process have to the file. In the 10-character mask on the left side of the display, the most significant character is 'd' if the file is directory & otherwise.

The next 3 characters specify read(r), write(w) & execute(x) ~~perf~~ permission for the file owner.

The next 3 characters are the same permissions for group members

The last 3 characters for all other processes.

If a triplet has an 'r' in the first position the corresponding user community (owner, group or world) has read permission on the file/directory; a '-' means the user doesn't have read permission. The second position permits represents write permission 'w' & third position represents execute permission x.

Scanned by TapScanner

For eg: File Make file is set so that gjn & any member of rtsg have read permission to the file. Any user process can read the cover-tex file, but only faculty & gjn can write to the file.

The file manager implements the authorization mechanism. Whenever a process attempts to open a file (a to reading, writing or executing), the file mngr's open() function first determines the identity of the process that is attempting the open(). From this, the file mngr will know if this process is the owner, a member of any group, or a process operating in behalf of any other user.

Suppose a process P_i attempts to open the file named cover.tex in fig.

If the process executing the open() is owned by the same user that owns the file, then the file mngr authorizes the access.

If, if the process is owned by some user that is a member of the faculty group, the file mangr will authorize the access & complete the open() op?

If the process is owned by a user that is neither the owner nor a member of the faculty group, the file mngr will deny access.

allow only read

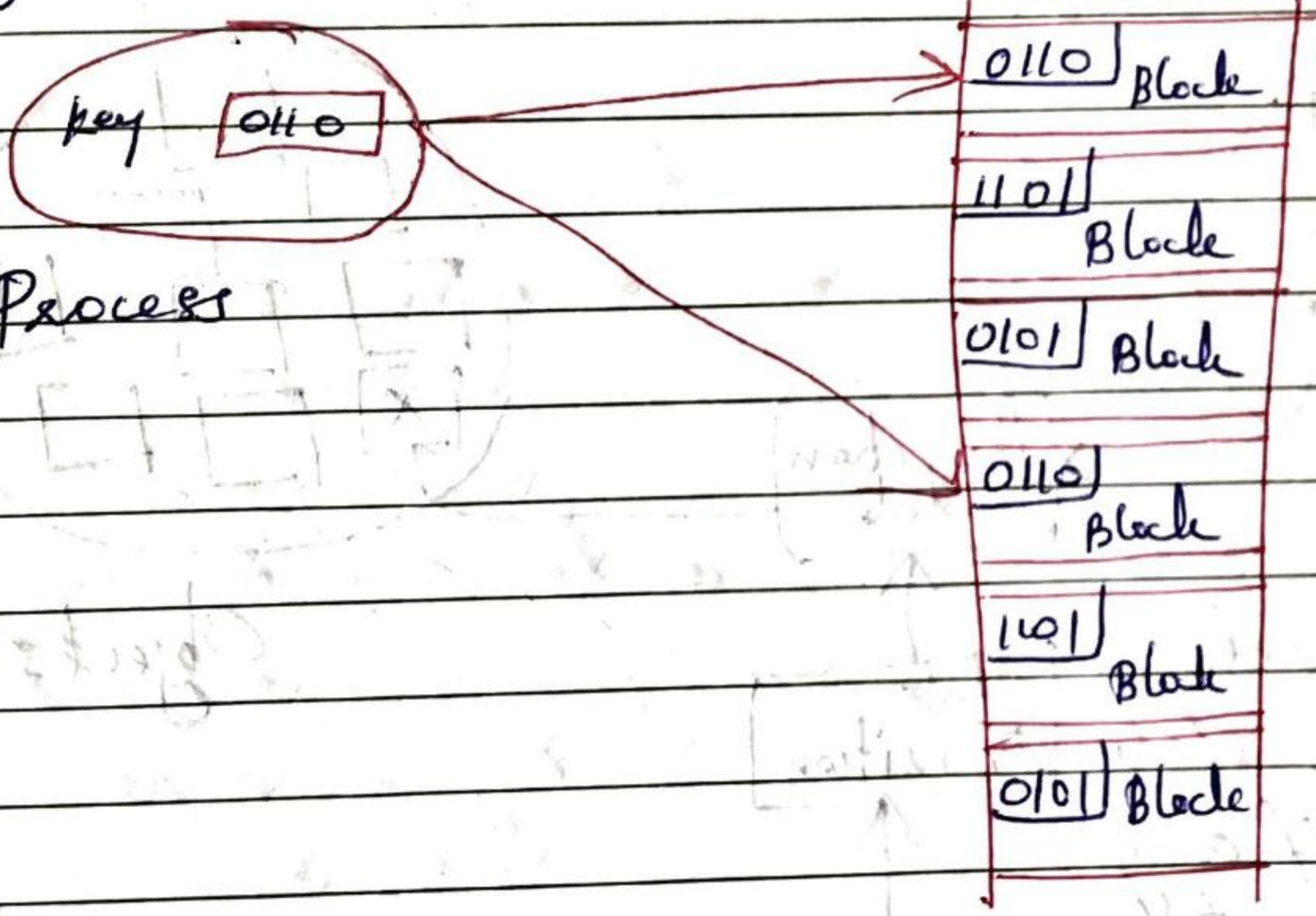
Scanned by TapScanner

Memory Locks & Keys

Authorization mechanism for memory objts. An allocatable unit of memory may be a word, a partition, a page or a segment.

Mem. locks on each allocatable blk of 'k' bytes - Each sub blk can be assigned a 'k' bit lock value & each process descriptor includes a k-bit key setting. When a block of memory is allocated to a process, the lock is set to be the same as the process's key.

Process key & mem locks can only be set with previleged instr's. The access is checked by hw on each mem. access by keeping the key value in a CPU reg. & by incorporating lock reg.



Memory locks are binary bit patterns assigned to allocatable memory blocks. A process provides a key with the same no. of bit pos's as a mem. lock. If the key bit pattern is the same as the mem. lock pattern, the process has access to the block.

Scanned by TapScanner

General Model for Authorizations

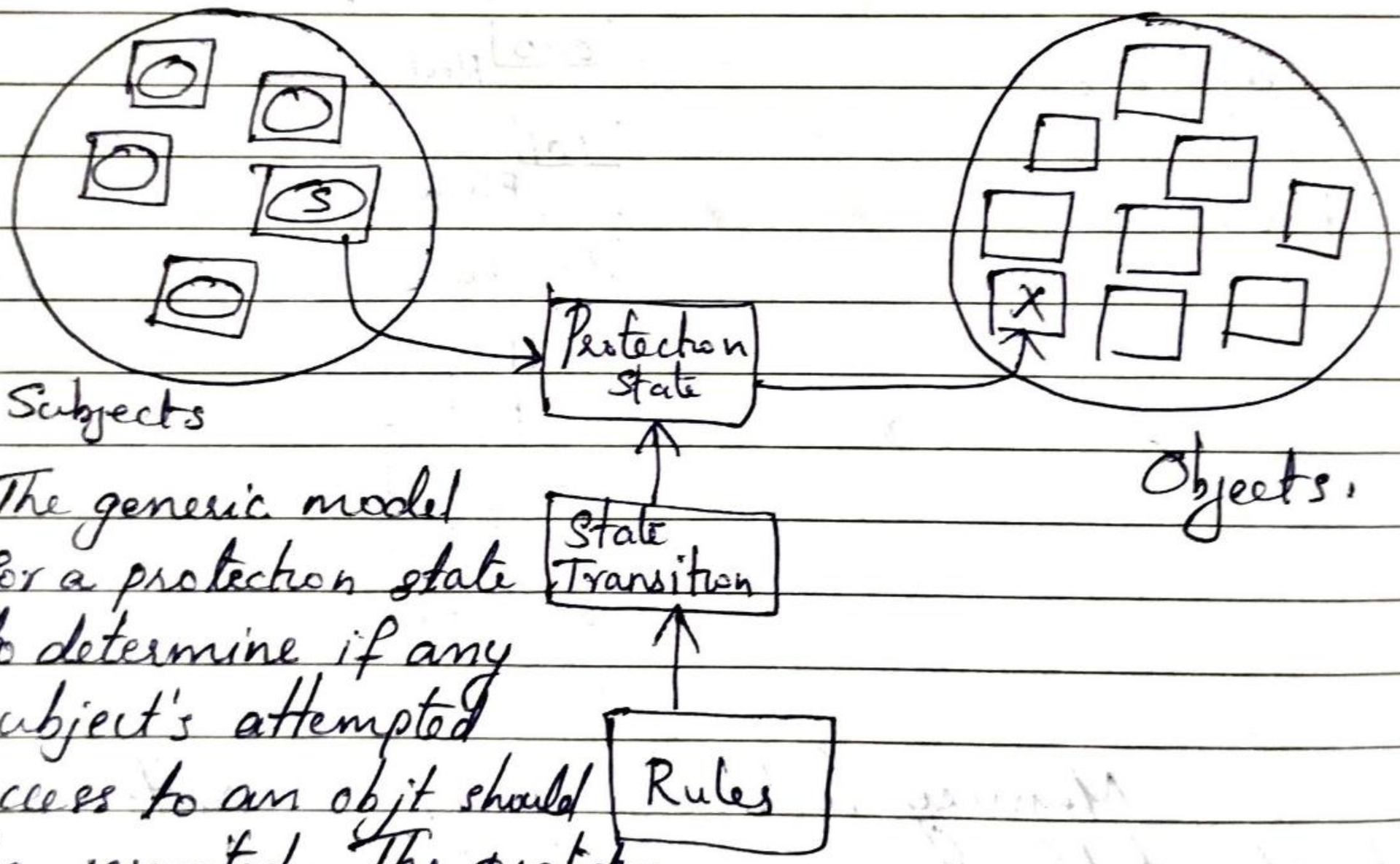
A s/m has active parts & passive parts. Its active parts, such as processes or threads, act on behalf of users. Its passive parts, corresponding to resources & other system entities, are called objects.

- Protection domain: The particular set of rights a process has at any given moment.

Access right

- A protection system is composed of a set of objects, a set of rules specifying the protection policy & a set of subjects. It represents the accessibility of objects by subjects as defined by the s/m's protection state.

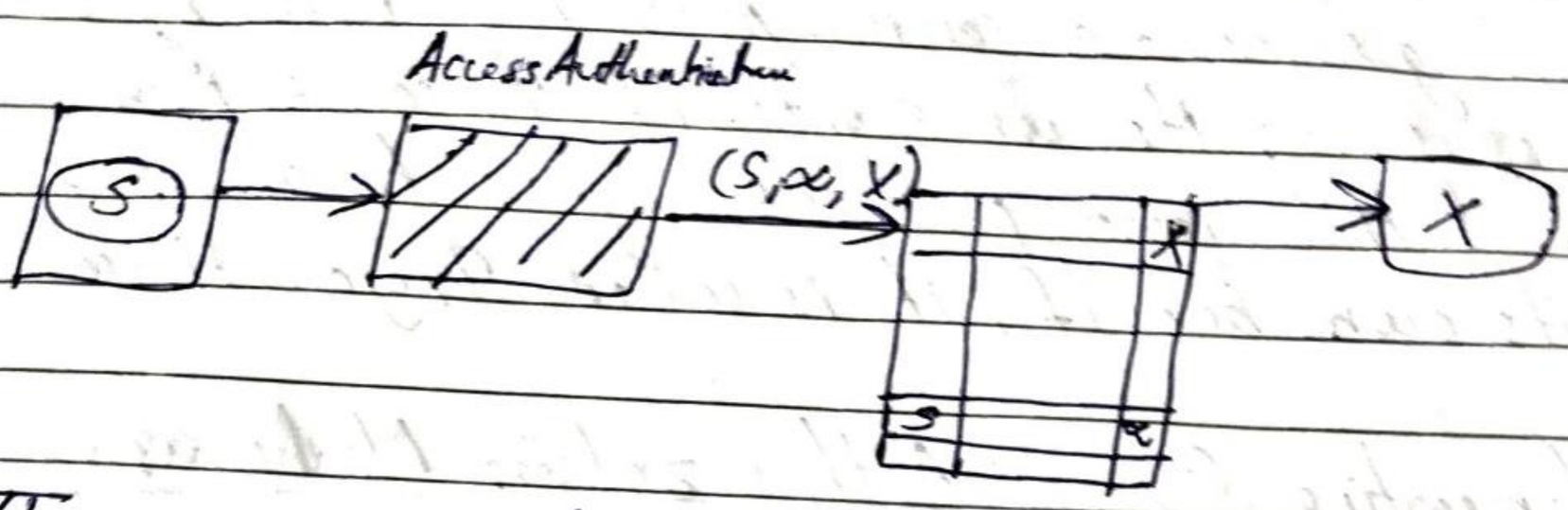
Protection S/m:



The generic model for a protection state to determine if any subject's attempted access to an obj it should be permitted. The protection state changes acc. to a fixed set of state transitions and the allowable state transitions are defined by the rules. The desired security policy as a set of rules

Scanned by TapScanner

Representing the Protection State with an Access Matrix



These are several protection pblms:

* Sharing parameters: A process's resource policy may be violated if other processes can indiscriminately change parameter values within its address space. For eg: suppose a process calls a procedure in some other process's address space & the callee's procedure modifies parameters passed into the procedure so that when the caller regains ctrl variables in its addr. space have been changed by the called procedure.

* Confinement: is a generalization of the pblm of sharing parameters. Suppose a process wishes to limit the dispersion of infn to some particular envt. The challenge is to contain all rights to resources so that they don't propagate outside some chosen set of parameters.

* Allocating rights: A protection s/m may allow a process to provide another process with specific rights to use its resources. In some cases, the first process needs to be able to revoke those rights at any time. Rights should be only temperarily allocated by one process to another. A pblm may occur if a process provides rights to another process & then the receiving process passes the rights on to other processes without the knowledge or permission of the resource owner. Some protection s/ms disallow this rights propagation without the explicit permission of the resource owner.

Scanned by TapScanner

Trojan Horse: It is a special case of the pblm of allocating rights - a service pgm being used by a client process using its own rights. If the server pgm takes adv. of the client process's rights to access resources on its own behalf, it is called Trojan Horse.

Implementing General Authorization Mechanisms:

The generalized protection mechanism is based on a means to save the protection state, to query the state to validate ongoing access & to change the state. There are several issues to consider in implementing the mechanisms:

→ The access matrix is not the only possible rep^{n} of the protection state, but it is the basis of most implem

→ The access matrix must be represented in some secure storage medium, only to be read & written by highly trusted mechanisms.

→ The protection s/m should be able to authenticate the source of each request by a subject rather than being passed the subject's identity as a parameter through procedure call.

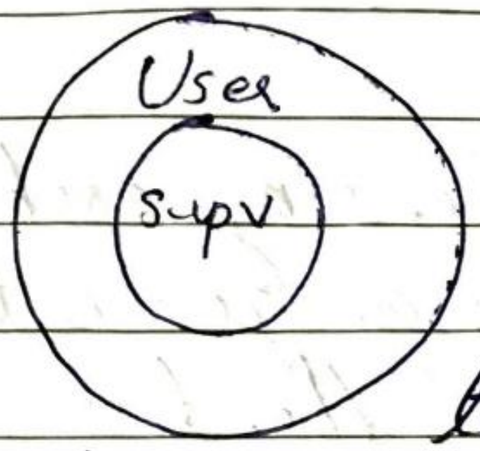
→ The goal of the design is to route all accesses through the protection monitor. Such routing will ensure that the current protection state will be used to validate each access.

→ The protection monitor must be a protected mechanism to implement the rules. It must not be

Scanned by TapScanner

possible for other subjects to compromise the monitor & state transition mechanisms.

Protection Domains



Two level Domain Architecture:

The inner domain represents pgms executed in the supervisor mode in the content of protection, the process is said to execute in the supervisor mode domain. Pgms operating in the users domain, for eg: rights to memory as well as right to execute the extended inst^n set. If p is a process, then subject S_1 = (P, user mode) has fewer rights than does S_2 = (P, supervisor mode).

A two level protection domain s/m corresponds to the familiar notion of a CPU with user & supervisor modes. In this view the domains are represented as concentric circles with the innermost circle representing the domain where processes have most rights.

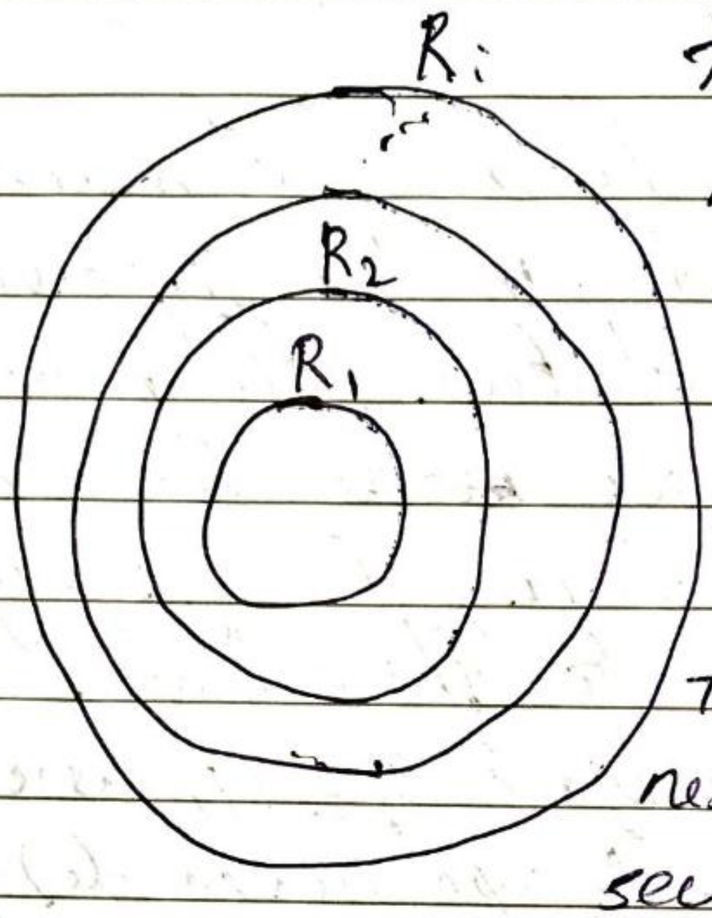
This org^n of domains — as concentric rings — doesn't represent arbitrary domain relationships such as could be created using the UNIX set UID bit on executable files.

The generalization of the two-level domain is a set of N concentric rings called a domain ring architecture for protection. The ~~ring archi~~

Suppose the protection s/m is organized as N rings of protection in which Rings R_0 through R_s support the O.S domain & rings R_{s+1} through R_{N-1} are

Scanned by TapScanner

used by applicatons. In this model, the h/w supervisor mode would ordinarily be used when s/w is executing in the lowest-numbered rings, perhaps only in R_0.



The ring architecture is a generalizeation of the two level ring. The inner most ring represents a domain where processes have the most rights. The next outer ring has the second most rights, and so on. The outer ring represents a domain where the processes have fewest rights.

Implementation of Access Matrix

Global Table:

A global table consisting of a set of ordered triples <domain, object, rights-set>. Whenever an op^n M is executed on an object Oj within domain D_i, the Global table is searched for a triple $< D_i, O_j, R_k >$, $M \in R_k$. If this triple is found this op^n is allowed to continue. otherwise, an exception cond^n raised. This imp^n suffers from several drawbacks.

The table is usually large & thus

Scanned by TapScanner

cannot be kept in m/m, so additional I/O is needed
Virtual memory techniques are often used for
managing this table.
It is difficult to take advantage of special
groupings of objts or domains. For eg:, if every-
one can read a particular objt, it must have
a separate entry in every domain.

Access Lists for Objects:

Each column in the access matrix can be implemented as an access list for one objt
The empty entries can be discarded. The resulting list for each object consists of ordered pairs
<domain, rights-set> which define all domains with a non empty set of access rights for that object.

This approach can be extended easily to define a list plus a default set of access rights. When an opⁿ M on an objt O_j is attempted in domain D_i we search access list for object O_j, looking for an entry $\langle D_i, R_k \rangle$ with $M \in R_k$. If the entry is found, we allow the opⁿ. if it is not, we check the default set. If M is in default set, we allow the access. Otherwise access is denied.

Capability Lists for Domains: token of authority.

Rather than associating the columns of access matrix with the objts as access lists, we can associate each row with its domain.

Scanned by TapScanner

A capability list for a domain is a list of objects together with the op^n's allowed on those objts. An object is often represented by its physical name or address is called a capability. To execute op^n M on object Oj, the process executes the op^n M, specifying the capability for objt Oj as a parameter. Simple possession of the capability means that access is allowed.

The capability list is associated with a domain, but it is never directly accessible to a process executing in that domain. Rather, the capability list itself a protected object, maintained by the operating s/m & accessed by the user indirectly.

Locks-Key Mechanism.

→ Each object has a tag to denote its type as either a capability or as accessible data.

→ The addr. space associated with a pgm consists into 2 parts. One part is accessible to the pgm & contains the pgm's normal data & inst^n's
the other part, containing the capability list is ~~acces~~ accessible only by the O.S.

A capability is defined as a protected object reference which, by virtue of its possession by a user process, grants that process the capability to interact ob with an objt in certain ways.

Scanned by TapScanner

Open !

File S/m Implementation

Structures & Operations used to implement file s/m operations:

On disk & in-memory structures are used to implement a file s/m.

On-disk the file s/m may contain inf^{n} about how to boot an O.S stored there, the total no. of blks, the no. & location of free blocks, directory structure & individual files.

* Boot Control Block: can contain inf^{n} needed by the s/m to boot an O.S from that partition. If the disk doesn't contain an O.S this blk can be empty. It is typically the first block of partition. In UFS - boot block & in NTFS, it is partition boot sector.

* Partition Ctrl Block: contains partition details such as the no. of blocks in the partition, size of blks, free block count & free blk ptrs & free FCB count & FCB pointers. In UFS, - superblk & NTFS - Master File Table.

* Directory Structure: is used to organise files

* FCB: contains many of the file's details, including permissions ownership, size & loc^{n} of the data blks. In UFS, - inode. In NTFS, this inf^{n} is stored within the Master File Table.

Scanned by TapScanner

In memory information is used for both file sys mgmt & performance improvement via caching. The structures can include:

→ An in-memory partition table containing info about each mounted partition.

→ An in-memory directory structure that holds the directory info of recently accessed directories.

→ The s/m-wide-open file table: contains copy of FCB of each open file, as well as other info.

→ The per-process open-file table: contains a ptr to the appropriate entry in the s/m wide open file table as well as other info.

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blks

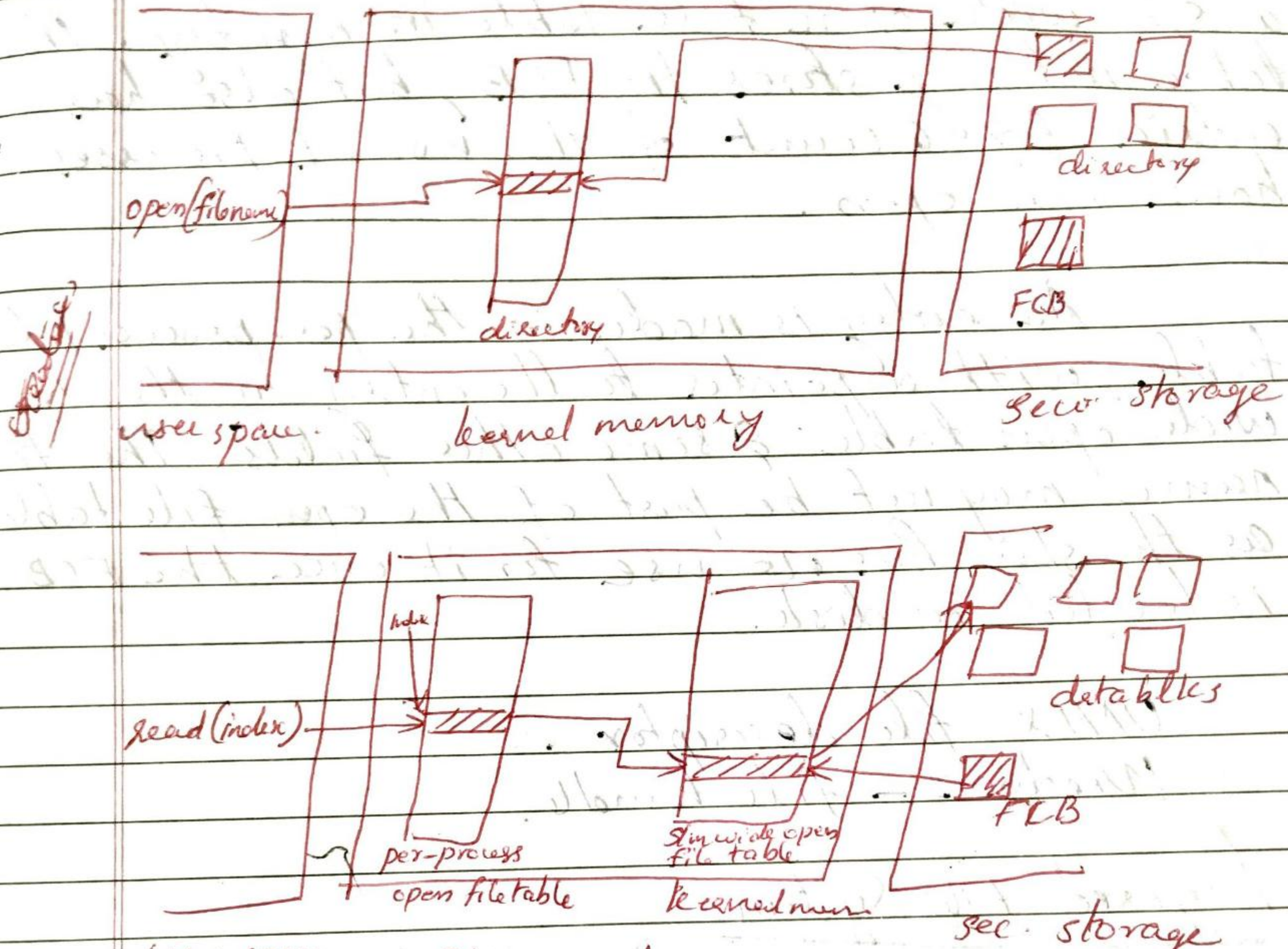
FCB

Create

To create a new file, an appl^{n} pgm calls the logical file s/m. The logical files s/m knows the format of the directory structures.

To create a new file, it allocates a new FCB; ~~reads~~ update directory with new file name & FCB & write it back to the disk.

Scanned by TapScanner



The logical file s/m can call the file org^n module to map the directory I/o into disk-blk no.s. The file org^n module also allocates blks for storage of the file's data.

Open

The Open call passes a file name to the file s/m. When a file s/m is opened, the directory structure is searched for the given filename. Parts of directory structure are usually cached in memory to speed directory op^{n}s.

Scanned by TapScanner

Once the file is found, the FCB is copied into a s/m wide open file table in memory. This table not only stores the FCB, but also has entries for a count of the no. of processes that have file open.

An entry is made in the per-process file table, with a pointer to the entry in the s/m wide open table & some other fields. The file name may not be part of the open file table, as the s/m has no use for it once the FCB is located on disk.

UNIX file descriptor.
Windows — file handle.

Layered File System

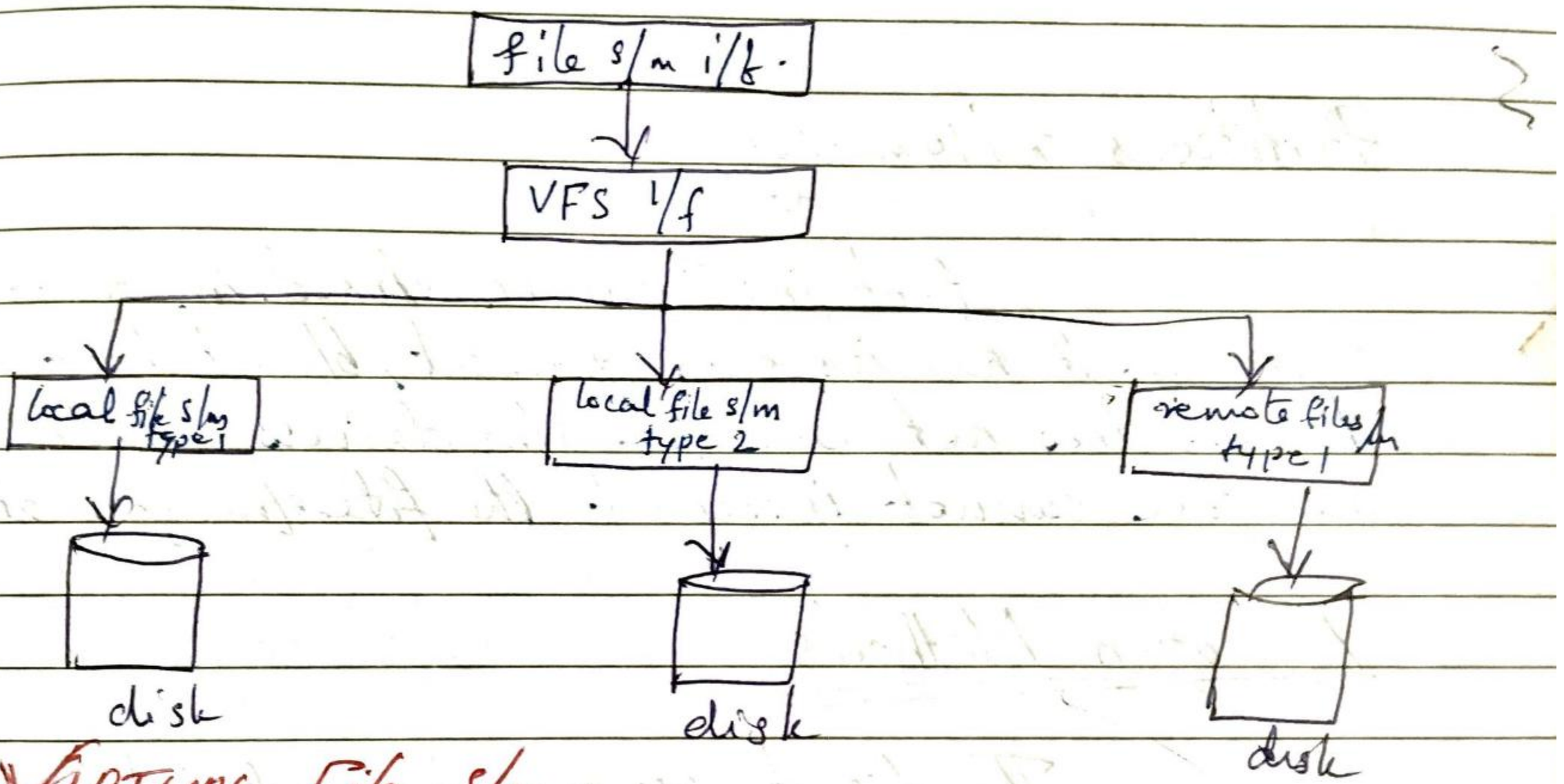
Appl^n pgms
↓
logical file s/m
↓
file-org^n module
↓
basic file s/m
↓
I/O ctrl
↓
devices

Basic file s/m needs only to issue generic cmds to the appropriate device driver to read & write phy. blks on the disk. Each phy. blk is identified by its numeric disk address.

File Organization module: knows about files & their logical blks as well as phy. blks. By knowing the type file allocation used & the location of the file, the file org^n module can translate logical blk addresses to phy. blk addresses for the basic file s/m to transfer-

Scanned by TapScanner

Logical file System: manages metadata information. Metadata includes all of the file-s/m structure, excluding the actual data. It maintains file structure via file ctl blocks. A file ctl block (FCB) contains inf^n about the file including ownership, permissions & loc^n of the file conts. It is also responsible for protection & security.



VIRTUAL File S/m

The file s/m implementation consists of 3 major layers.

First layer is called Virtual File S/m i/f, based on open, read, write & close calls & file descriptors.

Second layer is called the Virtual File System Layer it serves 2 important functions:

* It separates file s/m generic op^ns from their implementation by defining a clean VFS i/f. Allows transparent access to diff. types of file s/ms mounted locally.

* The VFS is based on a file rep^n structure called vnode that contains a numerical descriptor for a

Scanned by TapScanner

n/w wide unique file. This n/w wide uniqueness is required for support of n/w file s/ms.

VFS distingenishes local files from remote ones, & local files are further distingenished acc. to their file s/m types.

Partitions & Mounting

Boot inf^n can be stored in a separate partition. It has its own format, b/c at boot time the s/m does not have file s/m drivers loaded & therefore cannot intepret the file s/m format.

Allocation Methods

The direct access nature of disks allows us flexibility in the impl^n of files. The main pblm is how to allocate space to these files so that disk space is utilized effectively & files can be accessed quickly.

> Contiguous Allocation

requires each file to occupy a set of contiguous blks on the disk. Disk addresses define a linear ordering on the disk.

Pblms :→ External fragmentation

→ determining how much space is needed for a file. When the file is created, the total amt of space it will need must be found & allocated.

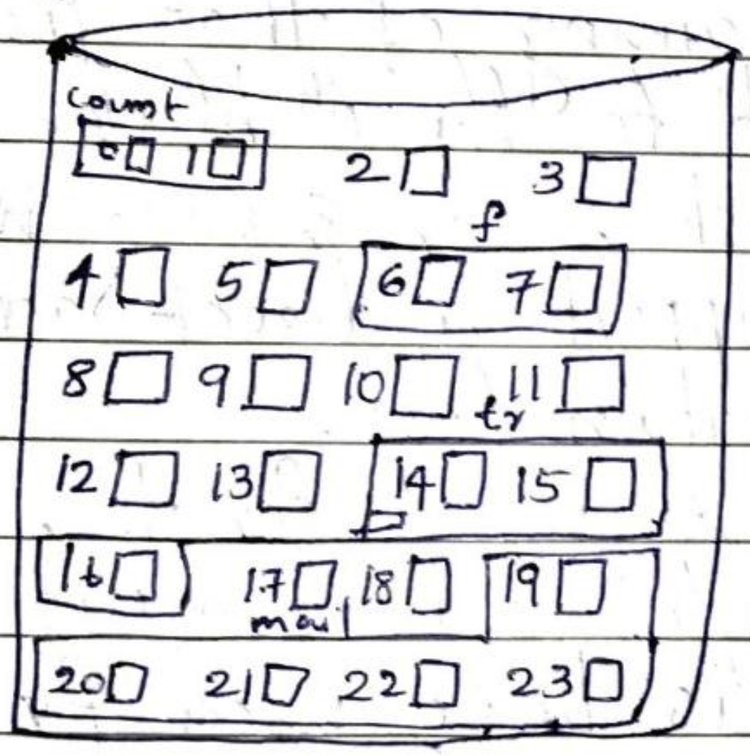
Scanned by TapScanner

→ If we allocate too little space to a file, we may find that the file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence we cannot make the file larger in place. Two possibilities

* first the user pgm can be terminated, with an appropriate error msg. The user must then allocate more space & run the pgm again.

* The other possibility is to find a larger hole, copy the contents of the file to the new space & releases the previous space. This series of actions may be repeated as long as space exists, although it can be time-consuming.

directory



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

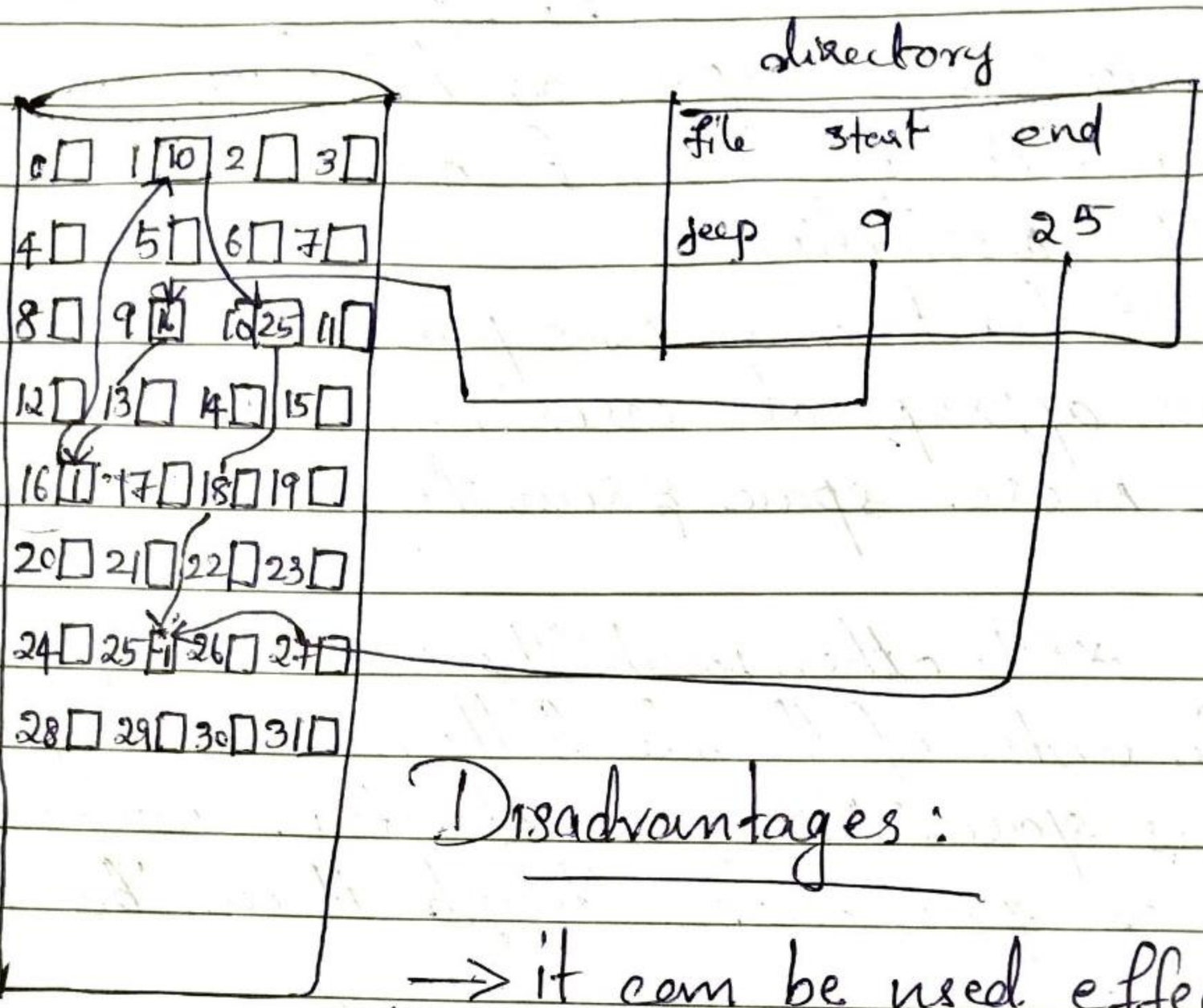
Linked Allocation

→ each file is a linked list of disk blocks. the disk blks may be scattered anywhere on the disk. The directory contains a pointer to the first & last blocks of the file.

To create a new file, we simply create a new entry in the directory. With linked allocation, each

Scanned by TapScanner

directory entry hers a ptr to the first blk of the file.



Disadvantages:

→ it can be used effectively only for seq. access files. To find the i^{th} block of a file we must start at the beginning of that file & follow the pointers until we get to the i^{th} block. Each access to a pointer requires a disk read, & sometimes a disk seek. It is inefficient to support a direct access capability for linked allocation files.

→ The space required for the pointers. If a pointer requires 4 bytes out of a 512 byte blk then .78% of the disk is being used for ptrs rather than info.

→ Reliability: Since files are linked together by ptrs scattered all over the disk, consider what would happen if a ptr were lost or damaged

File Allocation Table. FAT

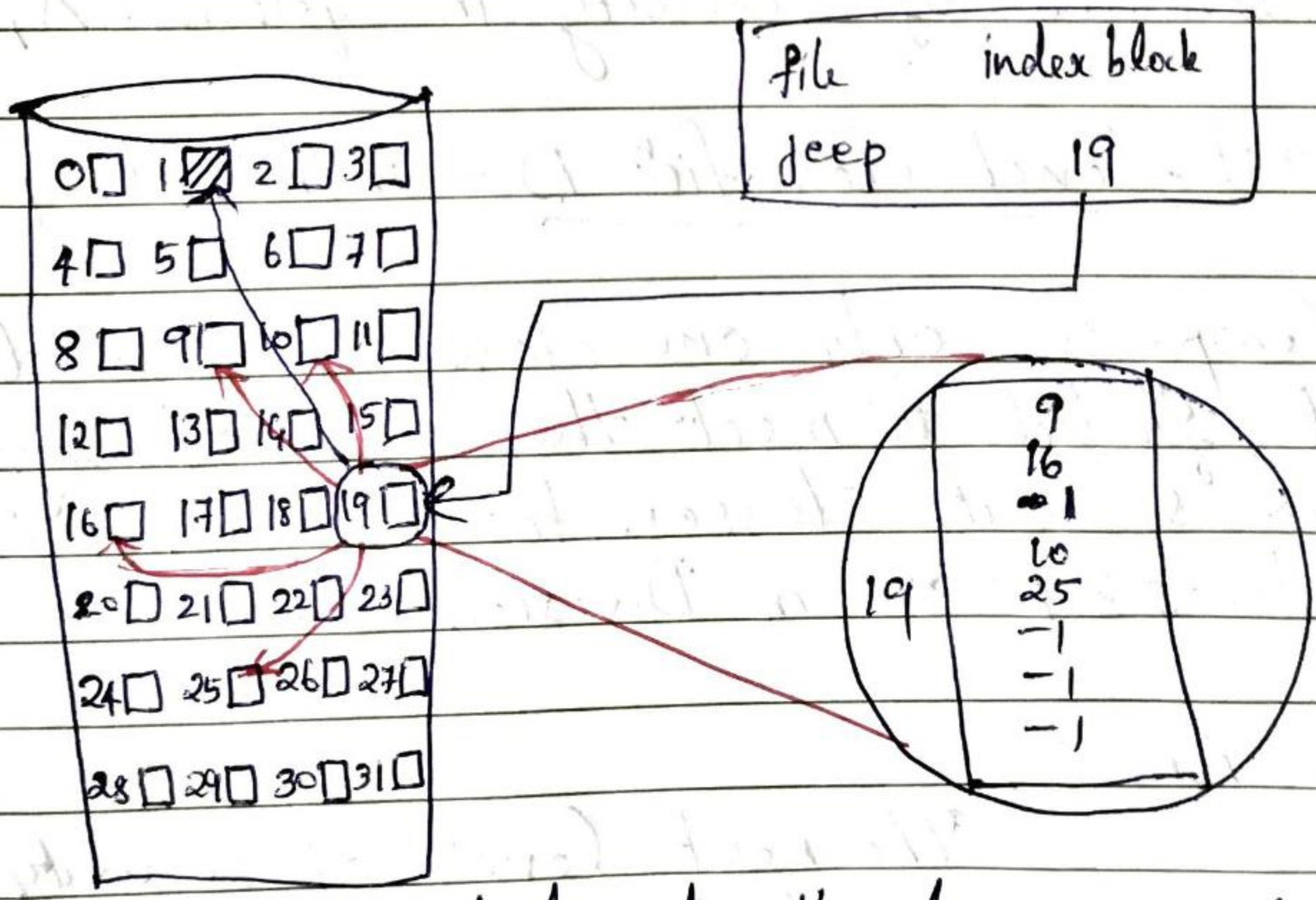
Scanned by TapScanner

Indexed Allocation:

Linked allocation solves the external-fragmentation & size-declaration pblm of contiguous allocation. But linked allocation cannot support efficient direct access, ~~since the~~ Indexed allocation solves this pblm by bringing all the pointers together into one location: the index block.

Each file has its own index block, which is an array of disk-blk addresses.

The i^{th} entry in the index block to the i^{th} block of the file. The directory contains the address of the index block. To read the i^{th} block, we use the ptr in the i^{th} index-block entry to find & read the desired block.



Indexed allocation supports direct access, without suffering from external fragmentation because any free block on the disk may satisfy a request for more space.

Scanned by TapScanner

Indexed allocation suffer from wasted space. The ptr overhead ~~is~~ of the index blk is generally greater than the ptr overhead of linked allocation. With linked allocation, we lose the space of only one ptr/blk. With indexed allocation, an entire index block must be allocated, even if only one or two ptrs non null.

File Sharing
Protection — Authorization
File s/m Implementation
Allocation Methods.
Directory & Disk Structure.
Directory Implementation.

Computer - Security Classifications

4 divisions of security in Systems: A, B, C, & D.

Lowest-level classific^{n} D - minimal protection

→ Comprises only one class, & is used for s/ms that failed to meet the reqmts of any one of the other security classes. For instance, MS-DOS & Windows 3.1 are in Division D.

Division C,

The next level of security, provides discretionary protection & accountability of users & their actions through the use of audit capabilities.
Two levels: → C_1 & C_2.

Scanned by TapScanner

A C_1 class s/m incorporates some form of ctrls that allow users to protect private inf^n & to keep other users from accidently reading or destroying their's data.

The sum total of all protection s/ms within a comp. s/m (hlw, slw, firmware) that correctly enforce a security policy is known as Trusted Computer Base. (TCB)

A C_2 - class s/m adds to the reqmts of a C_1 s/m via an individual-level access ctrl. For eg: access right of a file can be specified to the level of a single individual. In addition, the s/m administrator is capable of selectively auditing the actions of any one or more users based on individual identity.

Division - B

Mandatory protection s/ms have all the properties of a class C_2 s/m plus they attach sensitivity labels to each object. The B1 class TCB maintains the security label of each obj't in the s/m the label is used for decisions pertaining to mandatory access-control. For eg: a user at confidential level could not access a file at the more sensitive secret level. The TCB also denotes the sensitivity level at the top & bottom of each page of any human readable o/p.

A B_2 class s/m extends the sensitivity labels to each resource such as storage objects. Phy. devices are assigned min & max. security levels that the s/m uses to enforce constraints imposed by the phy. envt's in which devices are located.

Scanned by TapScanner

B3 class s/m allows the creation of a ACL that denote users or groups not granted access to a given named object. The TCB also contains a mechanism to monitor events that may indicate a violation of security policy. The mechanism notifies the security administrator & if necessary terminates the event in least disruptive manner.

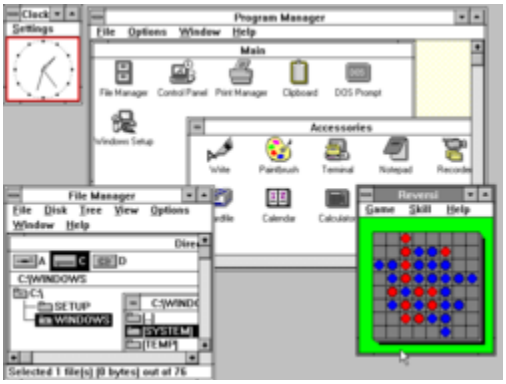
Division A – Highest level classification.

A class A, is functionally eqt. to a B3 s/m, but uses formal design specifications & verification techniques, granting a high degree of assurance that the TCB has been implemented correctly.

Scanned by TapScanner

Windows 3.x

Main articles: Windows 3.0 and Windows 3.1x



Windows 3.0, released in 1990

Windows 3.0, released in 1990, improved the design, mostly because of virtual memory and loadable virtual device drivers (VxDs) that allow Windows to share arbitrary devices between multi-tasked DOS applications Windows 3.0 applications can run in protected mode, which gives them access to several megabytes of memory without the obligation to participate in the software virtual memory scheme. They run inside the same address space, where the segmented memory provides a degree of protection. Windows 3.0 also featured improvements to the user interface. Microsoft rewrote critical operations from C into assembly. Windows 3.0 is the first Microsoft Windows version to achieve broad commercial success, selling 2 million copies in the first six months.

Windows 3.1, made generally available on March 1, 1992, featured a facelift. In August 1993, Windows for Workgroups, a special version with integrated peer-to-peer networking features and a version number of 3.11, was released. It was sold along with Windows 3.1. Support for Windows 3.1 ended on December 31, 2001.

Windows 3.2, released 1994, is an updated version of the Chinese version of Windows 3.1. The update was limited to this language version, as it fixed only issues related to the complex writing system of the Chinese language. Windows 3.2 was generally sold by computer manufacturers with a ten-disk version of MS-DOS that also had Simplified Chinese characters in basic output and some translated utilities.

Windows 9x

Main article: Windows 9x

The next major consumer-oriented release of Windows, Windows 95, was released on August 24, 1995. While still remaining MS-DOS-based, Windows 95 introduced support for native 32-bit applications, plug and play hardware, preemptive multitasking, long file names of up to 255 characters, and provided increased stability over its predecessors. Windows 95 also introduced a redesigned, object oriented user interface, replacing the previous Program Manager with the Start menu, taskbar, and Windows Explorer shell. Windows 95 was a major commercial success for

Microsoft; Ina Fried of CNET remarked that "by the time Windows 95 was finally ushered off the market in 2001, it had become a fixture on computer desktops around the world."[24] Microsoft published four OEM Service Releases (OSR) of Windows 95, each of which was roughly equivalent to a service pack. The first OSR of Windows 95 was also the first version of Windows to be bundled with Microsoft's web browser, Internet Explorer. Mainstream support for Windows 95 ended on December 31, 2000, and extended support for Windows 95 ended on December 31, 2001.

Windows 95 was followed up with the release of Windows 98 on June 25, 1998, which introduced the Windows Driver Model, support for USB composite devices, support for ACPI, hibernation, and support for multi-monitor configurations. Windows 98 also included integration with Internet Explorer 4 through Active Desktop and other aspects of the Windows Desktop Update (a series of enhancements to the Explorer shell which were also made available for Windows 95). In May 1999, Microsoft released Windows 98 Second Edition, an updated version of Windows 98. Windows 98 SE added Internet Explorer 5.0 and Windows Media Player 6.2 amongst other upgrades. Mainstream support for Windows 98 ended on June 30, 2002, and extended support for Windows 98 ended on July 11, 2006.

On September 14, 2000, Microsoft released Windows Me (Millennium Edition), the last DOS-based version of Windows. Windows Me incorporated visual interface enhancements from its Windows NT-based counterpart Windows 2000, had faster boot times than previous versions (which however, required the removal of the ability to access a real mode DOS environment, removing compatibility with some older programs), expanded multimedia functionality (including Windows Media Player 7, Windows Movie Maker, and the Windows Image Acquisition framework for retrieving images from scanners and digital cameras), additional system utilities such as System File Protection and System Restore, and updated home networking tools. However, Windows Me was faced with criticism for its speed and instability, along with hardware compatibility issues and its removal of real mode DOS support. *PC World* considered Windows Me to be one of the worst operating systems Microsoft had ever released, and the 4th worst tech product of all time.

Windows NT

Main article: Windows NT

Early versions (Windows NT 3.1/3.5/3.51/4.0/2000)

Main articles: Windows NT 3.1, Windows NT 3.5, Windows NT 3.51, Windows NT 4.0, and Windows 2000

In November 1988, a new development team within Microsoft (which included former Digital Equipment Corporation developers Dave Cutler and Mark Lucovsky) began work on a revamped version of IBM and Microsoft's OS/2 operating system known as "NT OS/2". NT OS/2 was intended to be a secure, multi-user operating system with POSIX compatibility and a modular, portable kernel with preemptive multitasking and support for multiple processor architectures. However, following the successful release of Windows 3.0, the NT development team decided to

rework the project to use an extended 32-bit port of the Windows API known as Win32 instead of those of OS/2. Win32 maintained a similar structure to the Windows APIs (allowing existing Windows applications to easily be ported to the platform), but also supported the capabilities of the existing NT kernel. Following its approval by Microsoft's staff, development continued on what was now Windows NT, the first 32-bit version of Windows. However, IBM objected to the changes, and ultimately continued OS/2 development on its own.

Windows NT was the first Windows operating system based on a hybrid kernel. The hybrid kernel was designed as a modified microkernel, influenced by the Mach microkernel developed by Richard Rashid at Carnegie Mellon University, but without meeting all of the criteria of a pure microkernel.

The first release of the resulting operating system, Windows NT 3.1 (named to associate it with Windows 3.1) was released in July 1993, with versions for desktop workstations and servers. Windows NT 3.5 was released in September 1994, focusing on performance improvements and support for Novell's NetWare, and was followed up by Windows NT 3.51 in May 1995, which included additional improvements and support for the PowerPC architecture. Windows NT 4.0 was released in June 1996, introducing the redesigned interface of Windows 95 to the NT series. On February 17, 2000, Microsoft released Windows 2000, a successor to NT 4.0. The Windows NT name was dropped at this point in order to put a greater focus on the Windows brand.

Windows XP

Main article: Windows XP

The next major version of Windows NT, Windows XP, was released on October 25, 2001. The introduction of Windows XP aimed to unify the consumer-oriented Windows 9x series with the architecture introduced by Windows NT, a change which Microsoft promised would provide better performance over its DOS-based predecessors. Windows XP would also introduce a redesigned user interface (including an updated Start menu and a "task-oriented" Windows Explorer), streamlined multimedia and networking features, Internet Explorer 6, integration with Microsoft's .NET Passport services, modes to help provide compatibility with software designed for previous versions of Windows, and Remote Assistance functionality.

At retail, Windows XP was now marketed in two main editions: the "Home" edition was targeted towards consumers, while the "Professional" edition was targeted towards business environments and power users, and included additional security and networking features. Home and Professional were later accompanied by the "Media Center" edition (designed for home theater PCs, with an emphasis on support for DVD playback, TV tuner cards, DVR functionality, and remote controls), and the "Tablet PC" edition (designed for mobile devices meeting its specifications for a tablet computer, with support for stylus pen input and additional pen-enabled applications). Mainstream support for Windows XP ended on April 14, 2009. Extended support ended on April 8, 2014.

After Windows 2000, Microsoft also changed its release schedules for server operating systems; the server counterpart of Windows XP, Windows Server 2003, was released in April 2003.[31] It was followed in December 2005, by Windows Server 2003 R2.

Windows Vista

Main article: Windows Vista

After a lengthy development process, Windows Vista was released on November 30, 2006, for volume licensing and January 30, 2007, for consumers. It contained a number of new features, from a redesigned shell and user interface to significant technical changes, with a particular focus on security features. It was available in a number of different editions, and has been subject to some criticism, such as drop of performance, longer boot time, criticism of new UAC, and stricter license agreement. Vista's server counterpart, Windows Server 2008 was released in early 2008.

Windows 7

Main article: Windows 7

On July 22, 2009, Windows 7 and Windows Server 2008 R2 were released as RTM (release to manufacturing) while the former was released to the public 3 months later on October 22, 2009. Unlike its predecessor, Windows Vista, which introduced a large number of new features, Windows 7 was intended to be a more focused, incremental upgrade to the Windows line, with the goal of being compatible with applications and hardware with which Windows Vista was already compatible. Windows 7 has multi-touch support, a redesigned Windows shell with an updated taskbar, a home networking system called HomeGroup, and performance improvements.

Windows 8 and 8.1

Main articles: Windows 8 and Windows 8.1

Windows 8, the successor to Windows 7, was released generally on October 26, 2012. A number of significant changes were made on Windows 8, including the introduction of a user interface based around Microsoft's Metro design language with optimizations for touch-based devices such as tablets and all-in-one PCs. These changes include the Start screen, which uses large tiles that are more convenient for touch interactions and allow for the display of continually updated information, and a new class of apps which are designed primarily for use on touch-based devices. The new Windows version required a minimum resolution of 1024×768 pixels, effectively making it unfit for netbooks with 800×600-pixel screens.

Other changes include increased integration with cloud services and other online platforms (such as social networks and Microsoft's own OneDrive (formerly SkyDrive) and Xbox Live services), the Windows Store service for software distribution, and a new variant known as Windows RT for use on devices that utilize the ARM architecture. An update to Windows 8, called Windows 8.1, was released on October 17, 2013, and includes features such as new live tile sizes, deeper

OneDrive integration, and many other revisions. Windows 8 and Windows 8.1 have been subject to some criticism, such as removal of the Start menu.

Windows 10

Main article: Windows 10

On September 30, 2014, Microsoft announced Windows 10 as the successor to Windows 8.1. It was released on July 29, 2015, and addresses shortcomings in the user interface first introduced with Windows 8. Changes on PC include the return of the Start Menu, a virtual desktop system, and the ability to run Windows Store apps within windows on the desktop rather than in full-screen mode. Windows 10 is said to be available to update from qualified Windows 7 with SP1, Windows 8.1 and Windows Phone 8.1 devices from the Get Windows 10 Application (for Windows 7, Windows 8.1) or Windows Update (Windows 7).

In February 2017, Microsoft announced the migration of its Windows source code repository from Perforce to Git. This migration involved 3.5 million separate files in a 300 gigabyte repository. By May 2017, 90 percent of its engineering team was using Git, in about 8500 commits and 1760 Windows builds per day.

Multilingual support

Multilingual support has been built into Windows since Windows 3.0. The language for both the keyboard and the interface can be changed through the Region and Language Control Panel. Components for all supported input languages, such as Input Method Editors, are automatically installed during Windows installation (in Windows XP and earlier, files for East Asian languages, such as Chinese, and right-to-left scripts, such as Arabic, may need to be installed separately, also from the said Control Panel). Third-party IMEs may also be installed if a user feels that the provided one is insufficient for their needs.

Interface languages for the operating system are free for download, but some languages are limited to certain editions of Windows. Language Interface Packs (LIPs) are redistributable and may be downloaded from Microsoft's Download Center and installed for any edition of Windows (XP or later) – they translate most, but not all, of the Windows interface, and require a certain base language (the language which Windows originally shipped with). This is used for most languages in emerging markets. Full Language Packs, which translates the complete operating system, are only available for specific editions of Windows (Ultimate and Enterprise editions of Windows Vista and 7, and all editions of Windows 8, 8.1 and RT except Single Language). They do not require a specific base language, and are commonly used for more popular languages such as French or Chinese. These languages cannot be downloaded through the Download Center, but available as optional updates through the Windows Update service (except Windows 8).

The interface language of installed applications are not affected by changes in the Windows interface language. Availability of languages depends on the application developers themselves.

Windows 8 and Windows Server 2012 introduces a new Language Control Panel where both the interface and input languages can be simultaneously changed, and language packs, regardless of type, can be downloaded from a central location. The PC Settings app in Windows 8.1 and Windows Server 2012 R2 also includes a counterpart settings page for this. Changing the interface language also changes the language of preinstalled Windows Store apps (such as Mail, Maps and News) and certain other Microsoft-developed apps (such as Remote Desktop). The above limitations for language packs are however still in effect, except that full language packs can be installed for any edition except Single Language, which caters to emerging markets.