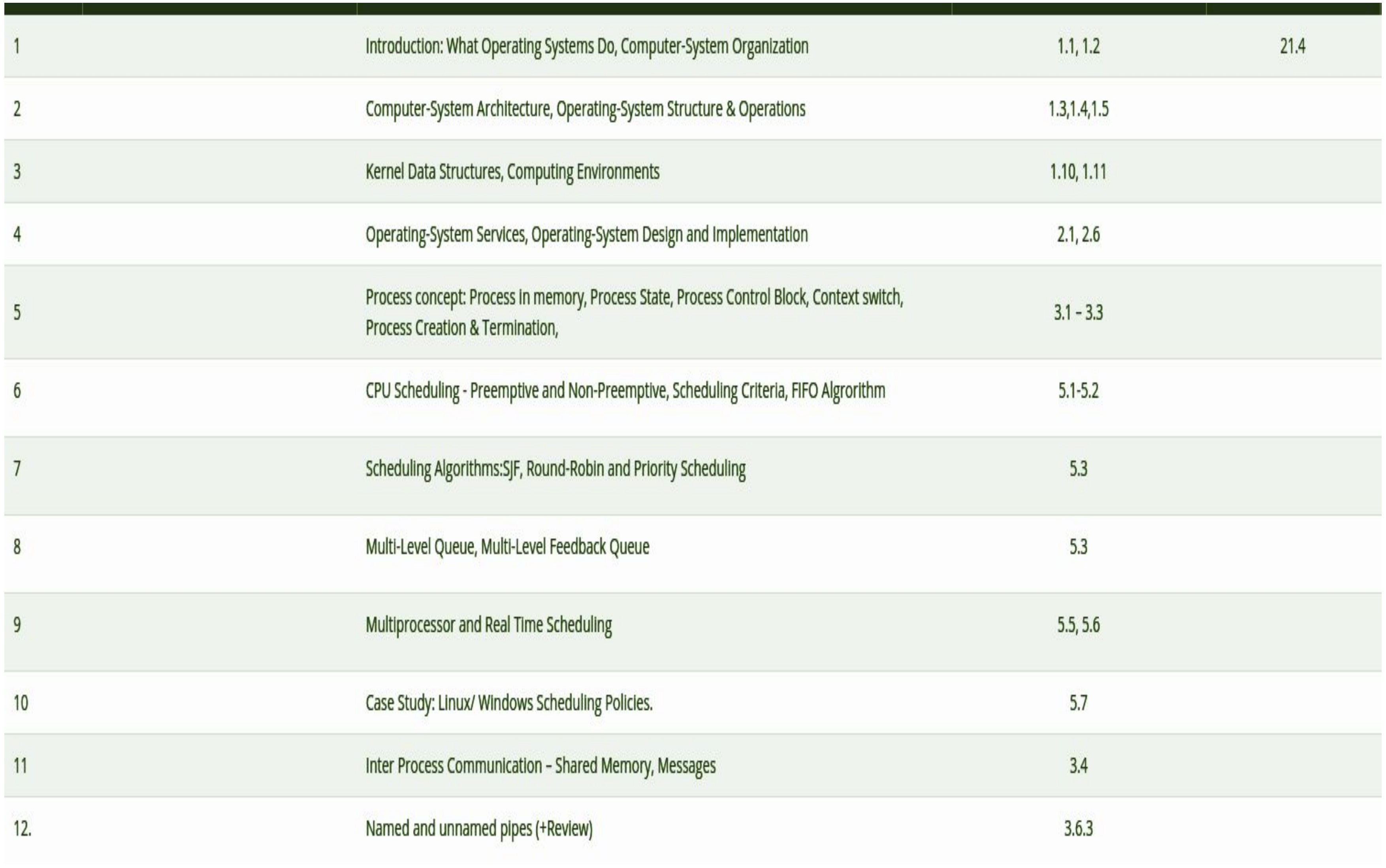
OPERATING SYSTEMS

Unit1_Unit2_Unit3: Revision Class #3

Nitin V Pujari
Faculty, Computer Science
Dean , IQAC, PES University

OPERATING SYSTEMS

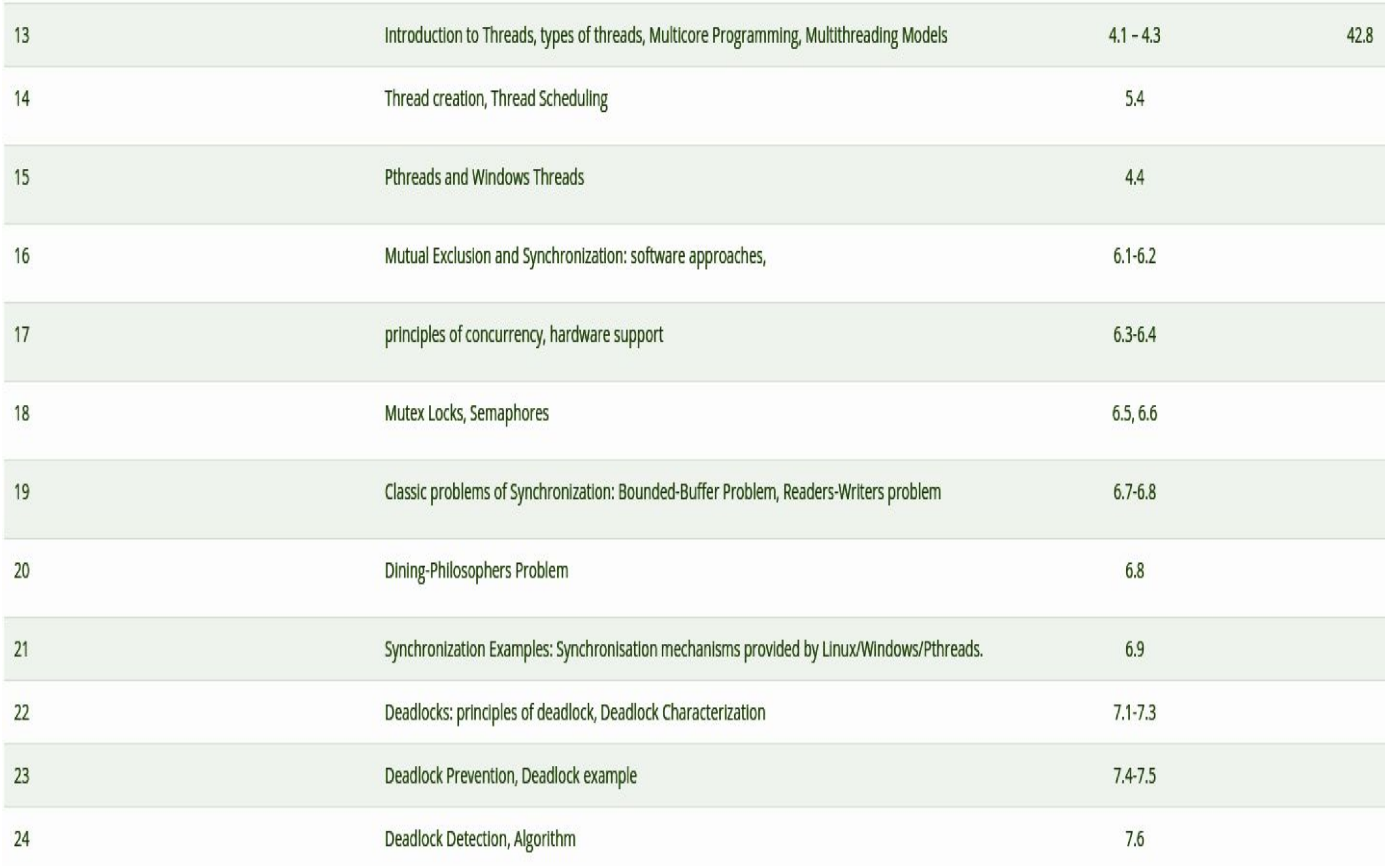
Course Syllabus => Unit 1



1		Introduction: What Operating Systems Do, Computer-System Organization	1.1, 1.2	21.4
2		Computer-System Architecture, Operating-System Structure & Operations	1.3,1.4,1.5	
3		Kernel Data Structures, Computing Environments	1.10, 1.11	
4		Operating-System Services, Operating-System Design and Implementation	2.1, 2.6	
5		Process concept: Process in memory, Process State, Process Control Block, Context switch, Process Creation & Termination,	3.1 - 3.3	
6		CPU Scheduling - Preemptive and Non-Preemptive, Scheduling Criteria, FIFO Algrorithm	5.1-5.2	
7		Scheduling Algorithms:SJF, Round-Robin and Priority Scheduling	5.3	
8		Multi-Level Queue, Multi-Level Feedback Queue	5.3	
9		Multiprocessor and Real Time Scheduling	5.5, 5.6	
10		Case Study: Linux/ Windows Scheduling Policies.	5.7	
11		Inter Process Communication - Shared Memory, Messages	3.4	
12.		Named and unnamed pipes (+Review)	3.6.3	

OPERATING SYSTEMS

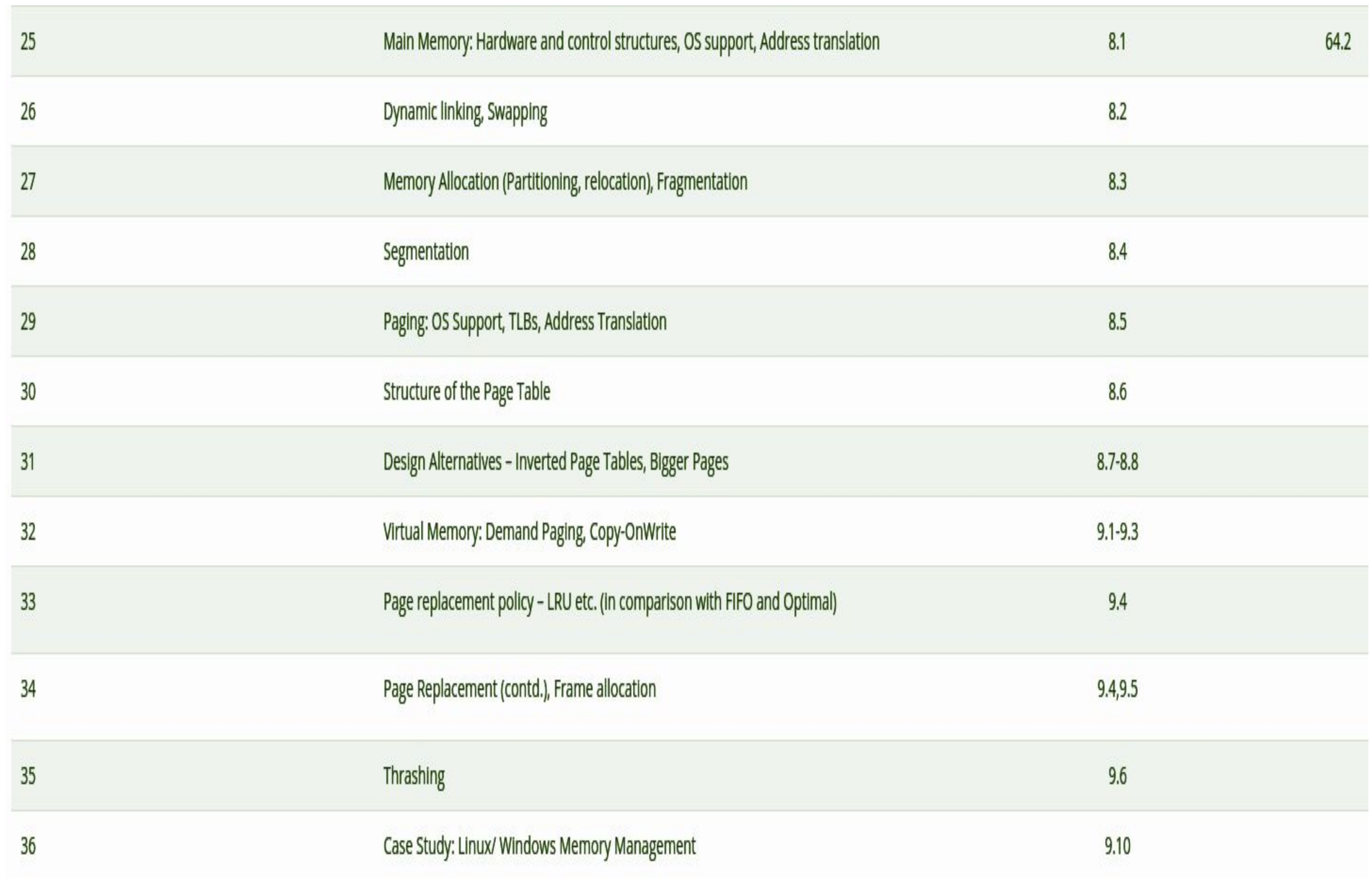
Course Syllabus => Unit 2



13	Introduction to Threads, types of threads, Multicore Programming, Multithreading Models	4.1 - 4.3	42.8
14	Thread creation, Thread Scheduling	5.4	
15	Pthreads and Windows Threads	4.4	
16	Mutual Exclusion and Synchronization: software approaches,	6.1-6.2	
17	principles of concurrency, hardware support	6.3-6.4	
18	Mutex Locks, Semaphores	6.5, 6.6	
19	Classic problems of Synchronization: Bounded-Buffer Problem, Readers-Writers problem	6.7-6.8	
20	Dining-Philosophers Problem	6.8	
21	Synchronization Examples: Synchronisation mechanisms provided by Linux/Windows/Pthreads.	6.9	
22	Deadlocks: principles of deadlock, Deadlock Characterization	7.1-7.3	
23	Deadlock Prevention, Deadlock example	7.4-7.5	
24	Deadlock Detection, Algorithm	7.6	

OPERATING SYSTEMS

Course Syllabus => Unit 3



25	Main Memory: Hardware and control structures, OS support, Address translation	8.1	64.2
26	Dynamic linking, Swapping	8.2	
27	Memory Allocation (Partitioning, relocation), Fragmentation	8.3	
28	Segmentation	8.4	
29	Paging: OS Support, TLBs, Address Translation	8.5	
30	Structure of the Page Table	8.6	
31	Design Alternatives - Inverted Page Tables, Bigger Pages	8.7-8.8	
32	Virtual Memory: Demand Paging, Copy-OnWrite	9.1-9.3	
33	Page replacement policy - LRU etc. (In comparison with FIFO and Optimal)	9.4	
34	Page Replacement (contd.), Frame allocation	9.4,9.5	
35	Thrashing	9.6	
36	Case Study: Linux/ Windows Memory Management	9.10	

Revision Class Outline => Student Queries

- How valid is the diagram which shows the stack and heap expanding towards each other given that they can be on different segments ?

- Sir, in the textbook for shortest job first/next, they have spoken about a formula to find the CPU burst time of the next process (exponential average), could you explain that ?

- Sir, can you explain how the CPU burst prediction is done in detail ?

- Comparative study on various CPU scheduling strategies => like advantages and disadvantages

Revision Class Outline => Student Queries

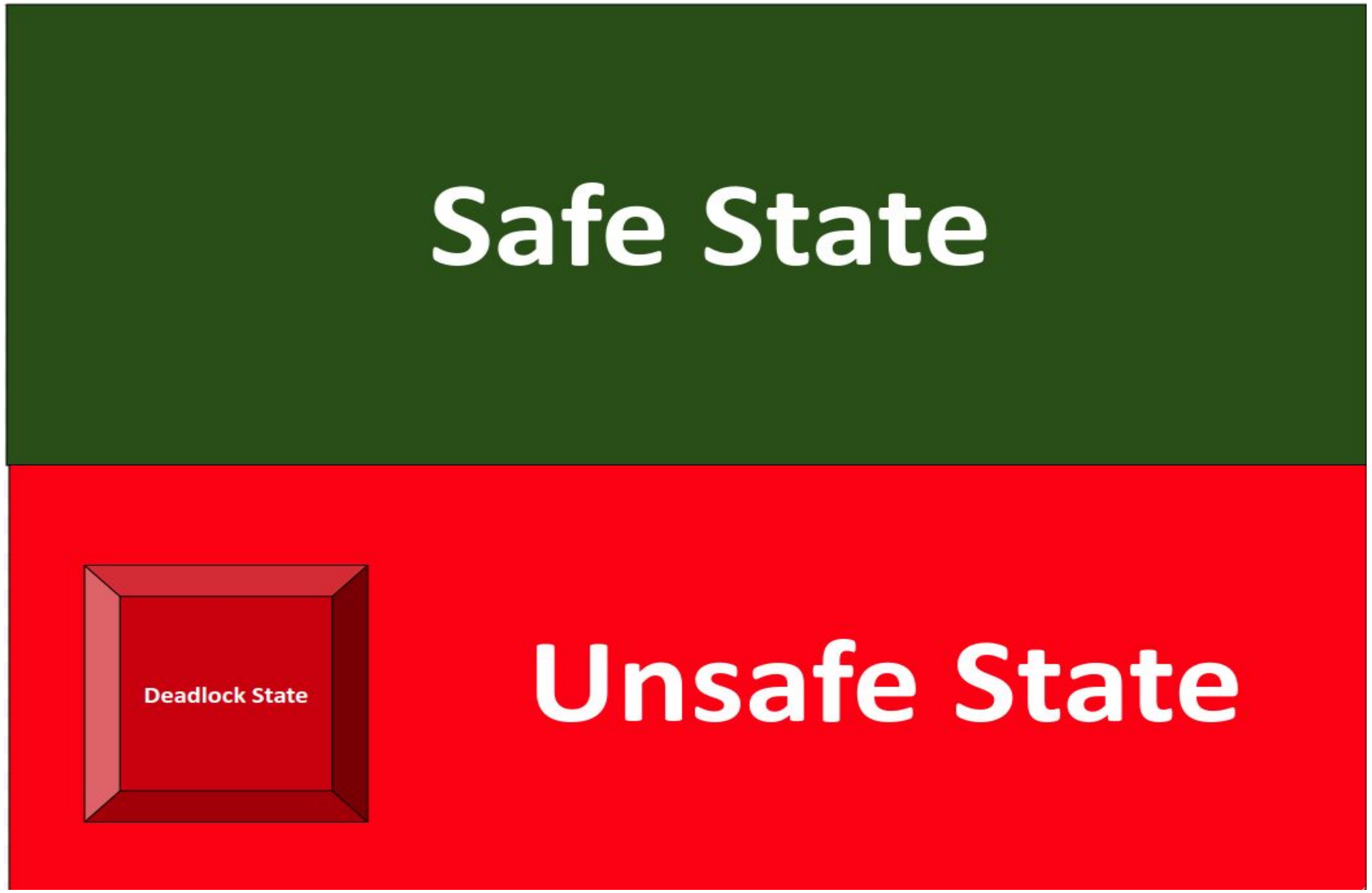
- Revision on Real time scheduling sir
- Revision for Round Robin Scheduling
- Recursive Mutexes, Semaphores, Spinlocks
- Revision of Banker's safety algorithm for detecting deadlocks

Revision Class Outline => Student Queries

- Page replacement algorithm in general
- Demand paging performance calculation
- Inverted page table

Revision Class Outline => Student Queries

- Safe and Unsafe State



Revision of Banker's safety algorithm for detecting deadlocks

Safe State

- **Safe state:** A safe state is a state in which all the processes can be executed in some arbitrary order with the available resources such that no deadlock occurs.

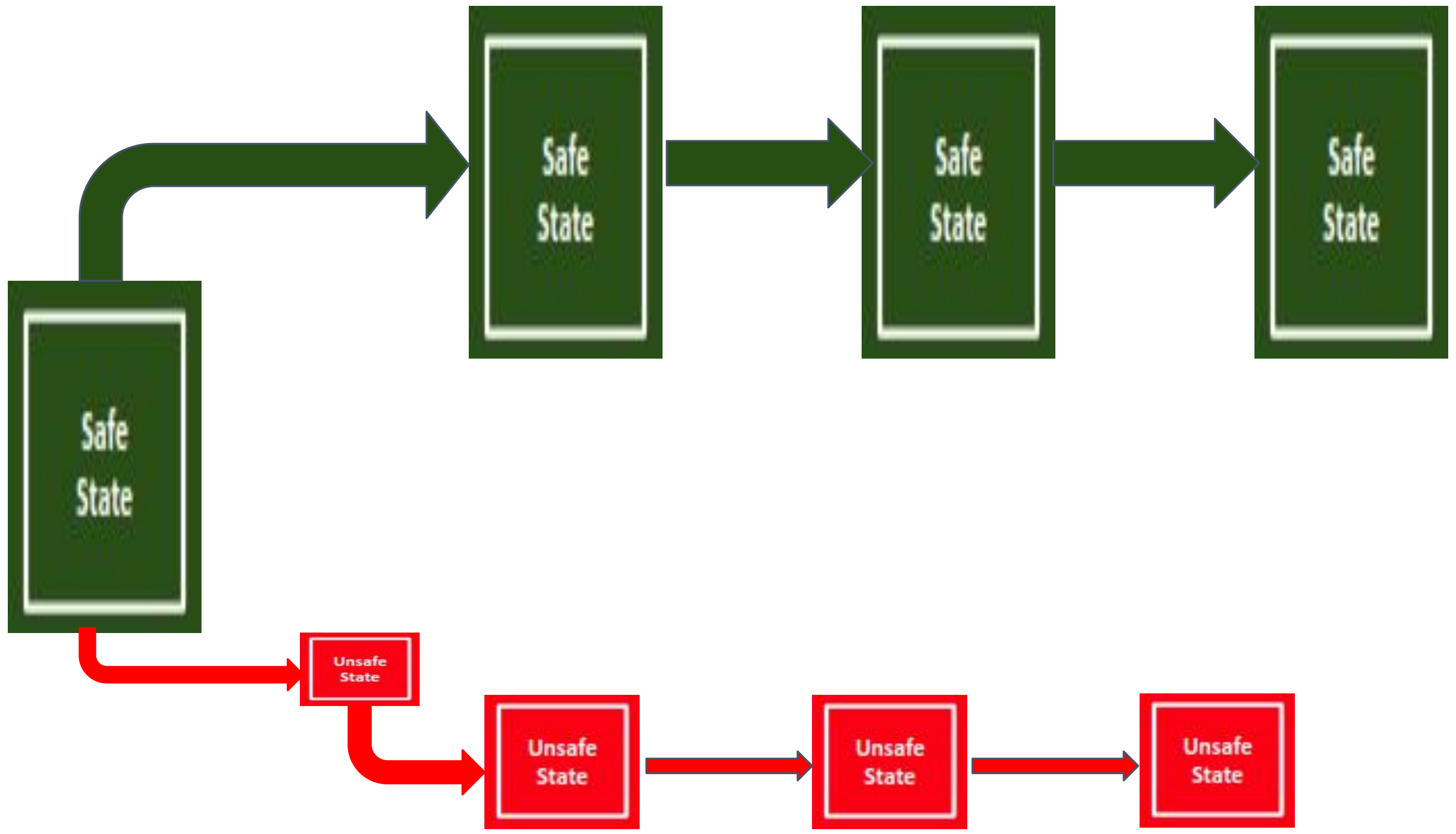
Revision of Banker's safety algorithm for detecting deadlocks

1. If it is a safe state, then the requested resources are allocated to the process in actual.

2. If the resulting state is an unsafe state then it rollbacks to the previous state and the process is asked to wait longer.

Revision Class Outline => Student Queries

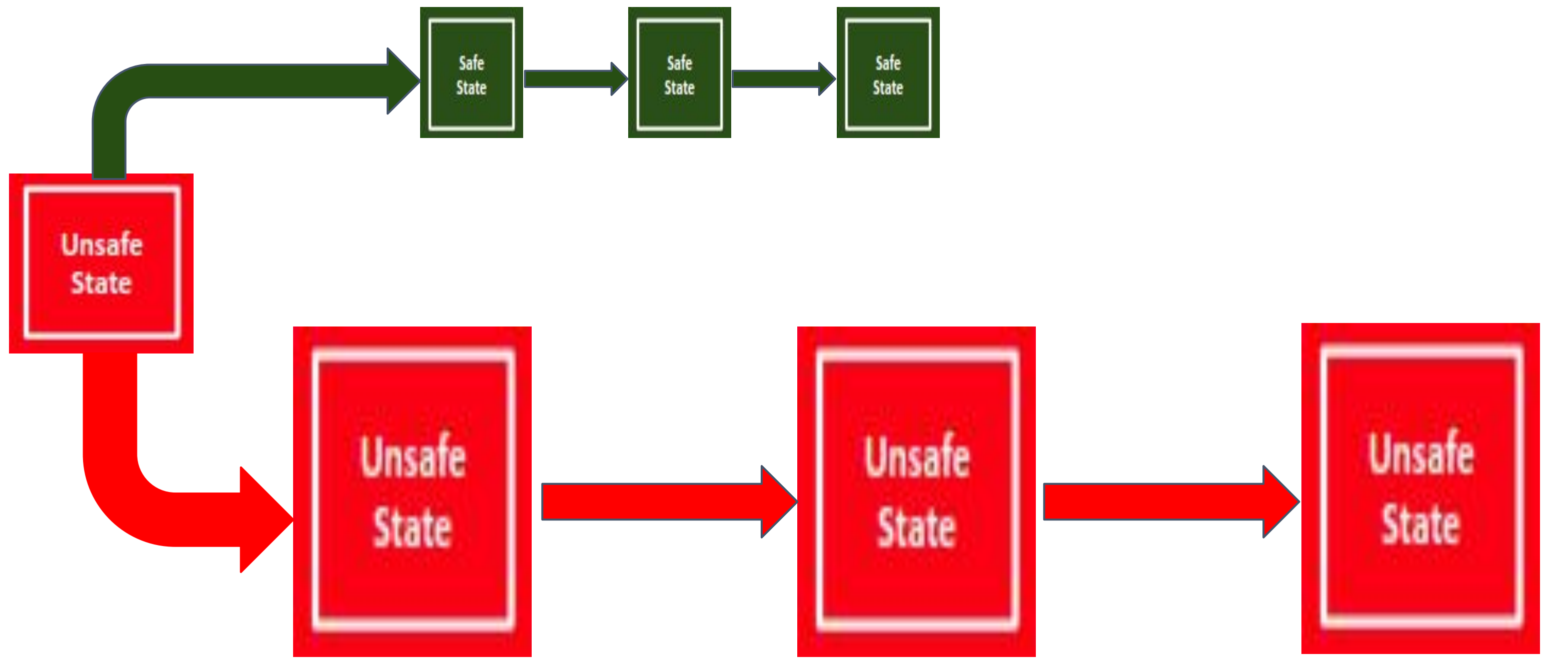
Safe to Unsafe State



Revision of Banker's safety algorithm for detecting deadlocks

Revision Class Outline => Student Queries

Unsafe State to Safe State



Revision of Banker's safety algorithm for detecting deadlocks

Banker's Algorithm

- Banker’s Algorithm is a **Deadlock Avoidance algorithm**.
- It is also used for **Deadlock Detection**.
- This algorithm tells that if any system can go into a deadlock or not by analyzing the currently allocated resources and the resources required by it in the future.
- This algorithm can be used when several instance of resources are present, which is typically cannot be handled by RAG
- There are various data structures which are used to implement this algorithm

Revision of Banker's safety algorithm for detecting deadlocks

Banker’s Algorithm

The Bankers Algorithm consists of the following two algorithms

1. Safety Algorithm

2. Request-Resource Algorithm

Revision of Banker's safety algorithm for detecting deadlocks

Revision Class Outline => Student Queries

- **Available**: A vector of length ***m*** indicates the number of available resources of each type
- **Allocation**: An ***n* x *m*** matrix defines the number of resources of each type currently allocated to each process
- **Request**: An ***n* x *m*** matrix indicates the current request of each process. If ***Request* [*i*][*j*] = *k***, then process P_i is requesting ***k*** more instances of resource type R_j.

Revision of Banker's safety algorithm for detecting deadlocks

Banker’s Algorithm: Safety Algorithm

Step1: Let Work and Finish be vectors of length m and n, respectively.

Initialize: Work = Available

Finish [i] =false for i= 0,1,2,..n-1.

Step2: Find an i such that both:

(a) Finish[i] = false

(b) $Need_i \leq Work$

If no such i exists, go to step 4.

Step 3: Work= Work + $Allocation_i$

Finish[i] =true

go to step 2.

Step 4: If Finish[i] == true for all i, then the system is in a safe state.

Revision of Banker's safety algorithm for detecting deadlocks

Banker's Algorithm: Resource Request Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
   (a) ***Work = Available***
   (b) For ***i* = 1,2, …, *n***, if $\textbf{\textit{Allocation}}_i \neq \mathbf{0}$, then ***Finish***[i] = ***false***; otherwise, ***Finish***[i] = ***true***

2. Find an index ***i*** such that both:
   (a) ***Finish***[***i***] == ***false***
   (b) $\textbf{\textit{Request}}_i \leq \textbf{\textit{Work}}$

   If no such ***i*** exists, go to step 4

Revision of Banker's safety algorithm for detecting deadlocks

Banker's Algorithm: Resource Request Algorithm

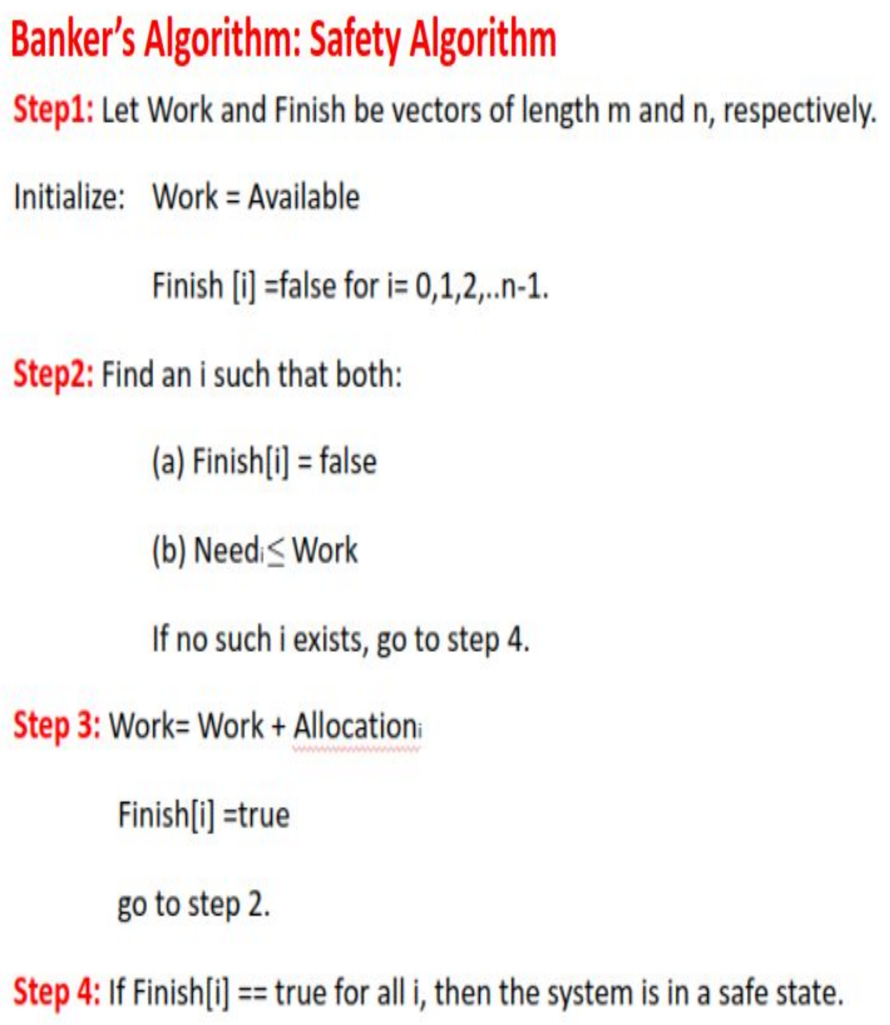
3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2

4. If ***Finish[i] == false***, for some ***i***, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if ***Finish***[*i*] == ***false***, then ***P***$_i$ is deadlocked

Algorithm requires an order of O(*m* x *n*^{2}) operations to detect whether the system is in deadlocked state

Revision of Banker's safety algorithm for detecting deadlocks

Revision Class Outline => Student Queries



Banker's Algorithm: Safety Algorithm

Step1: Let Work and Finish be vectors of length m and n, respectively.

Initialize: Work = Available

Finish [i] =false for i= 0,1,2,..n-1.

Step2: Find an i such that both:

(a) Finish[i] = false

(b) $Need_i \leq Work$

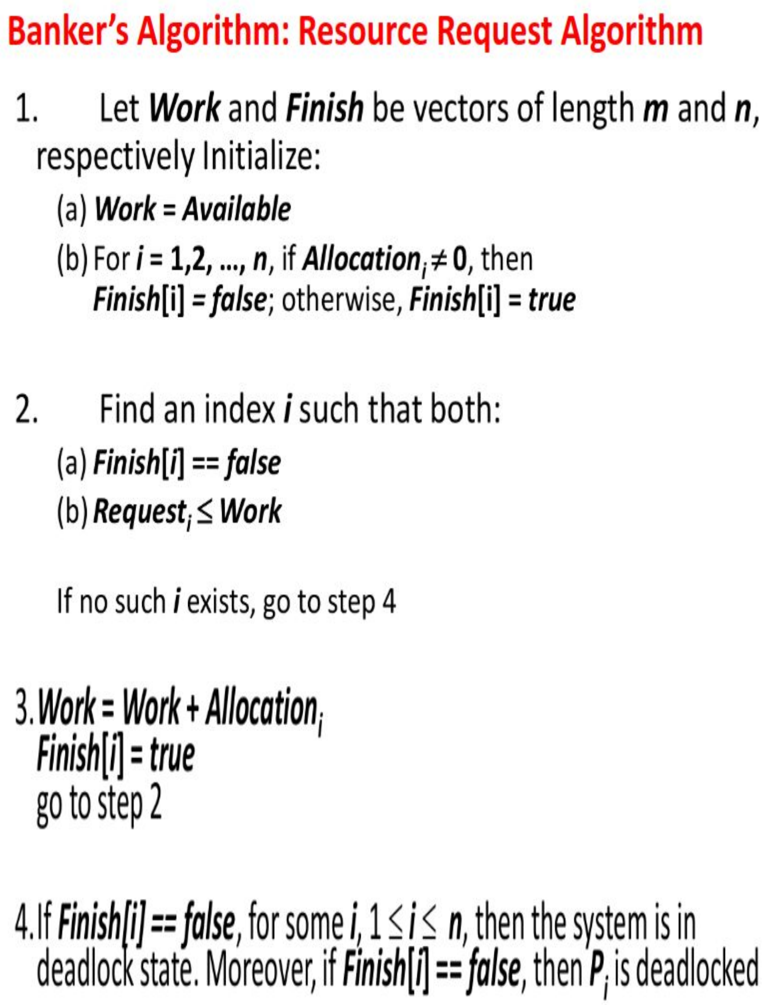
If no such i exists, go to step 4.

Step 3: Work= Work + $Allocation_i$

Finish[i] =true

go to step 2.

Step 4: If Finish[i] == true for all i, then the system is in a safe state.



Banker's Algorithm: Resource Request Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
   (a) ***Work = Available***
   (b) For ***i* = 1,2, …, *n***, if $Allocation_i \neq 0$, then ***Finish***[i] = ***false***; otherwise, ***Finish***[i] = ***true***

2. Find an index ***i*** such that both:
   (a) ***Finish[i] == false***
   (b) $Request_i \leq Work$

   If no such ***i*** exists, go to step 4

3. $Work = Work + Allocation_i$
   Finish[i] = true
   go to step 2

4. If ***Finish[i] == false***, for some ***i***, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if ***Finish[i] == false***, then P_i is deadlocked

Revision of Banker's safety algorithm for detecting deadlocks

Example - Banker's Safety Algorithm

Suppose we have 4 processes(P0, P1, P2, P3) and 3 resource types(A, B, C) each having (5,2,4)) instances. Suppose at time t1 if the snapshot of the system taken is as follows then find the system is in a safe state or not, after finding the need

RMax		
A	B	C
5	2	4

Available=>Rmax-Allocated		
A	B	C
1	0	2

RMax as an example		
A (Tape Drives)	B(Printers)	C(Scanner)
5	2	4

Process	Allocation			Max		
	A	B	C	A	B	C
P0	2	0	1	3	1	1
P1	0	1	0	0	1	1
P2	1	0	1	3	1	1
P3	1	1	0	1	1	3
Total	4	2	2	4	4	6

Revision of Banker's safety algorithm for detecting deadlocks

Example - Banker's Safety Algorithm

Suppose we have 4 processes(P0, P1, P2, P3) and 3 resource types(A, B, C) each having (5,2,4)) instances. Suppose at time t1 if the snapshot of the system taken is as follows then find the system is in a safe state or not, after finding the need

RMax		
A	B	C
5	2	4

Available=>Rmax-Allocated		
A	B	C
1	0	2

i	0 to n-1

Process	Allocation			Max		
	A	B	C	A	B	C
P0	2	0	1	3	1	1
P1	0	1	0	0	1	1
P2	1	0	1	3	1	1
P3	1	1	0	1	1	3
Total	4	2	2	4	4	6

Process	Need = Max(i) - Allocated(i)		
	A	B	C
P0	1	1	0
P1	0	0	1
P2	2	1	0
P3	0	0	3
Total	3	2	4

Revision of Banker's safety algorithm for detecting deadlocks

Example - Banker's Safety Algorithm

Suppose we have 4 processes(P0, P1, P2, P3) and 3 resource types(A, B, C) each having (5,2,4)) instances. Suppose at time t1 if the snapshot of the system taken is as follows then find the system is in a safe state or not, after finding the need

RMax		
A	B	C
5	2	4

Available=>Rmax-Allocated		
A	B	C
1	0	2

i	Not Initialised

Work = Available		
A	B	C
1	0	2

Process	Allocation			Max		
	A	B	C	A	B	C
P0	2	0	1	3	1	1
P1	0	1	0	0	1	1
P2	1	0	1	3	1	1
P3	1	1	0	1	1	3
Total	4	2	2	4	4	6

Process	Need = Max(i) - Allocated(i)		
	A	B	C
P0	1	1	0
P1	0	0	1
P2	2	1	0
P3	0	0	3
Total	3	2	4

Process	Flag
P0	False
P1	False
P2	False
P3	False

Initialize: Work = Available

Finish [i] =false for i= 0,1,2,..n-1.

Safe Sequence			

Revision of Banker's safety algorithm for detecting deadlocks

PES UNIVERSITY ONLINE

Example - Banker's Safety Algorithm

Suppose we have 4 processes(P0, P1, P2, P3) and 3 resource types(A, B, C) each having (5,2,4)) instances. Suppose at time t1 if the snapshot of the system taken is as follows then find the system is in a safe state or not, after finding the need

RMax		
A	B	C
5	2	4

Available=>Rmax-Allocated		
A	B	C
1	0	2

i	1

Work = Available		
A	B	C
1	0	2
1	1	2

Process	Allocation			Max		
	A	B	C	A	B	C
P0	2	0	1	3	1	1
P1	0	1	0	0	1	1
P2	1	0	1	3	1	1
P3	1	1	0	1	1	3
Total	4	2	2	4	4	6

Process	Need = Max(i) - Allocated(i)		
	A	B	C
P0	1	1	0
P1	0	0	1
P2	2	1	0
P3	0	0	3
Total	3	2	4

Process	Flag
P0	False
P1	True
P2	False
P3	False

Safe Sequence			
P1			

Revision of Banker's safety algorithm for detecting deadlocks

Example - Banker’s Safety Algorithm

Suppose we have 4 processes(P0, P1, P2, P3) and 3 resource types(A, B, C) each having (5,2,4)) instances. Suppose at time t1 if the snapshot of the system taken is as follows then find the system is in a safe state or not, after finding the need

RMax		
A	B	C
5	2	4

Available=>Rmax-Allocated		
A	B	C
1	0	2

i	0

Process	Allocation			Max		
	A	B	C	A	B	C
P0	2	0	1	3	1	1
P1	0	1	0	0	1	1
P2	1	0	1	3	1	1
P3	1	1	0	1	1	3
Total	4	2	2	4	4	6

Process	Need = Max(i) - Allocated(i)		
	A	B	C
P0	1	1	0
P1	0	0	1
P2	2	1	0
P3	0	0	3
Total	3	2	4

Process	Flag
P0	True
P1	True
P2	False
P3	False

Work = Available		
A	B	C
1	0	2
1	1	2
3	1	3

Safe Sequence			
P1	P0		

Revision of Banker's safety algorithm for detecting deadlocks

Example - Banker's Safety Algorithm

Suppose we have 4 processes(P0, P1, P2, P3) and 3 resource types(A, B, C) each having (5,2,4)) instances. Suppose at time t1 if the snapshot of the system taken is as follows then find the system is in a safe state or not, after finding the need

RMax		
A	B	C
5	2	4

Available=>Rmax-Allocated		
A	B	C
1	0	2

i	2

Process	Allocation			Max		
	A	B	C	A	B	C
P0	2	0	1	3	1	1
P1	0	1	0	0	1	1
P2	1	0	1	3	1	1
P3	1	1	0	1	1	3
Total	4	2	2	4	4	6

Proces s	Need = Max(i) - Allocated(i)		
	A	B	C
P0	1	1	0
P1	0	0	1
P2	2	1	0
P3	0	0	3
Total	3	2	4

Process	Flag
P0	True
P1	True
P2	True
P3	False

Work = Available		
A	B	C
1	0	2
1	1	2
3	1	3
4	1	4

Safe Sequence			
P1	P0	P2	

Example - Banker's Safety Algorithm

Suppose we have 4 processes(P0, P1, P2, P3) and 3 resource types(A, B, C) each having (5,2,4)) instances. Suppose at time t1 if the snapshot of the system taken is as follows then find the system is in a safe state or not, after finding the need

RMax		
A	B	C
5	2	4

Available=>Rmax-Allocated		
A	B	C
1	0	2

i	3

Process	Allocation			Max		
	A	B	C	A	B	C
P0	2	0	1	3	1	1
P1	0	1	0	0	1	1
P2	1	0	1	3	1	1
P3	1	1	0	1	1	3
Total	4	2	2	4	4	6

Process	Need = Max(i) - Allocated(i)		
	A	B	C
P0	1	1	0
P1	0	0	1
P2	2	1	0
P3	0	0	3
Total	3	2	4

Process	Flag
P0	True
P1	True
P2	True
P3	True

Work = Available		
A	B	C
1	0	2
1	1	2
3	1	3
4	1	4
5	2	4

Safe Sequence			
P1	P0	P2	P3

Example - Banker's Resource Request Algorithm - Next Class

Suppose we have 4 processes(P0, P1, P2, P3) and 3 resource types(A, B, C) each having (5,2,4)) instances. Suppose at time t1 if the snapshot of the system taken is as follows then find the system is in a safe state or not, after finding the need

RMax		
A	B	C
5	2	4

Available=>Rmax-Allocated		
A	B	C
1	0	2

i	0

P2	(2,1,0)

Work = Available		
A	B	C
1	0	2

Process	Allocation			Max		
	A	B	C	A	B	C
P0	2	0	1	3	1	1
P1	0	1	0	0	1	1
P2	1	0	1	3	1	1
P3	1	1	0	1	1	3
Total	4	2	2	4	4	6

Proces s	Need = Max(i) - Allocated(i)		
	A	B	C
P0	1	1	0
P1	0	0	1
P2	2	1	0
P3	0	0	3
Total	3	2	4

Process	Flag
P0	False
P1	False
P2	False
P3	False

Safe Sequence			
P1	P0	P2	P3

Revision Class Outline => Student Queries => Inverted Page Table

- Inverted Page Table is the global page table which is maintained by the Operating System for all the processes.
- In inverted page table, the number of entries is equal to the number of frames in the main memory.
- It can be used to overcome the drawbacks of page table.
- There is always a space reserved for the page regardless of the fact that whether it is present in the main memory or not. However, this is simply the wastage of the memory if the page is not present.
- We can save this wastage by just inverting the page table.
- We can save the details only for the pages which are present in the main memory.
- Frames are the indices and the information saved inside the block will be Process ID and page number.

0				
1				
2	P2	10	P5	11
3	P3	15		
4	P15	4		
5				
6				
7				
8				
9				
10				

Inverted Page Table

Revision Class Outline => Student Queries

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- One entry for each real page of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries
- TLB can accelerate access
- But how to implement shared memory ?
- One mapping of a virtual address to the shared physical address

Revision Class Outline => Student Queries

- By contrast, the size of a page table is determined by the amount of virtual memory used.

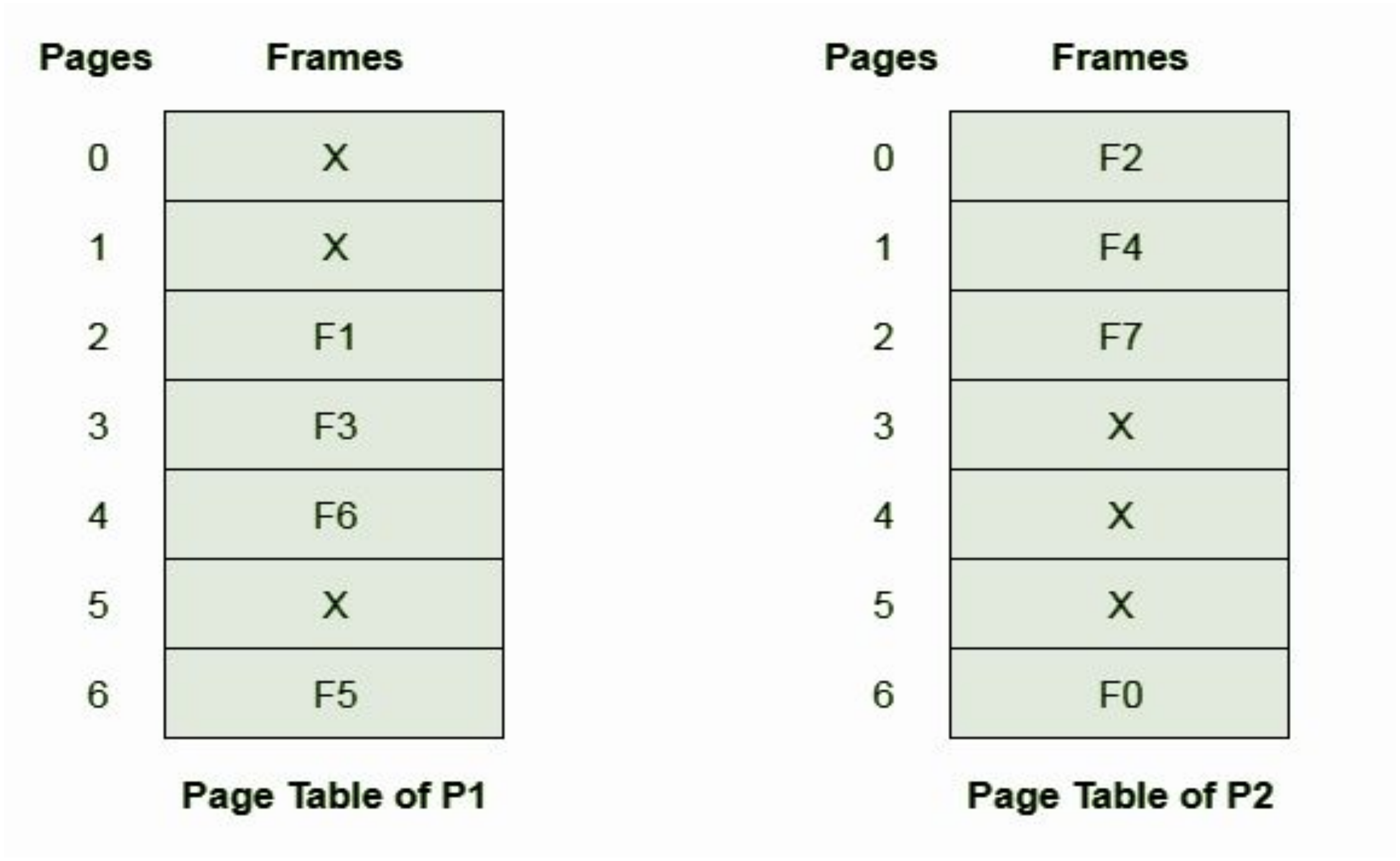
- If only small programs are run, the page table sizes will be small; if large programs (in terms of their virtual memory size) are run, the page tables will be large (by about the same factor: a page table entry is probably about 4 or 8 bytes, and each 4K page requires one page table entry).

- Since each process requires its own page tables (the page tables are swapped as part of a context switch), if several large virtual memory programs are run, the total amount of page table space needed is the sum of the page table space for each process.

- An inverted page table has one major disadvantage: it is more complex and expensive to use (in terms of time) for mapping the virtual address to a physical address.

OPERATING SYSTEMS

Revision Class Outline => Student Queries



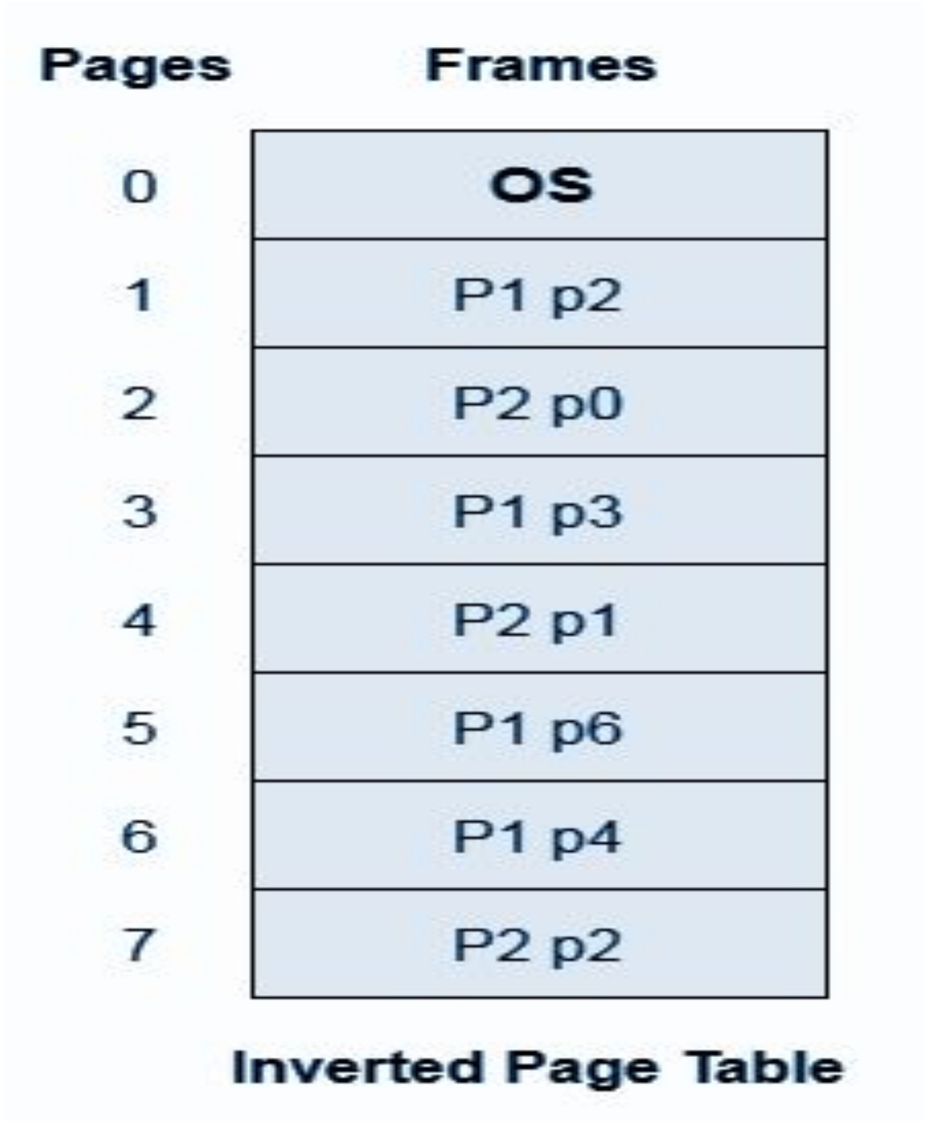
Pages	Frames
0	X
1	X
2	F1
3	F3
4	F6
5	X
6	F5

Page Table of P1

Pages	Frames
0	F2
1	F4
2	F7
3	X
4	X
5	X
6	F0

Page Table of P2

Revision Class Outline => Student Queries



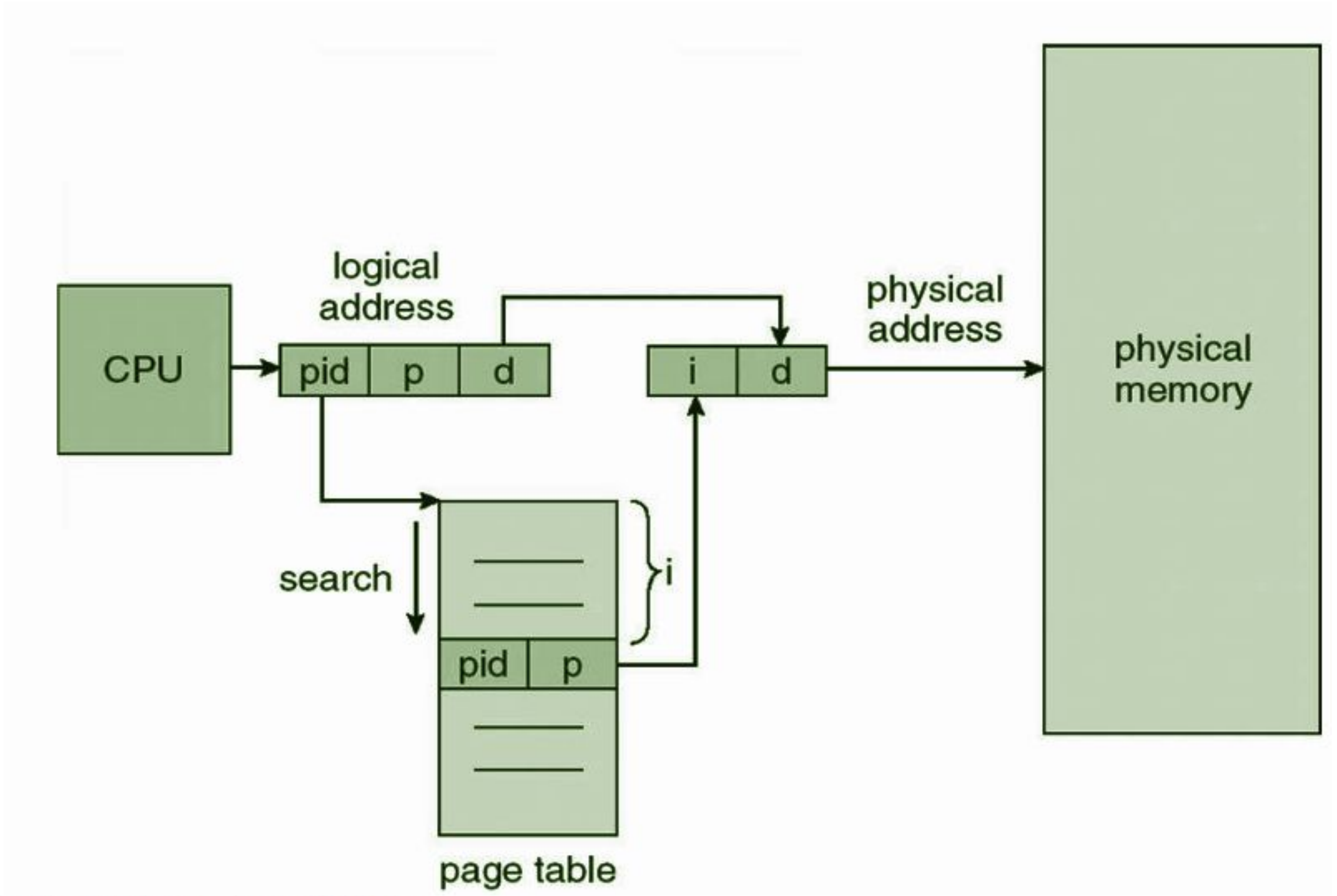
Pages	Frames
0	OS
1	P1 p2
2	P2 p0
3	P1 p3
4	P2 p1
5	P1 p6
6	P1 p4
7	P2 p2

Inverted Page Table

Inverted Page Table

OPERATING SYSTEMS

Revision Class Outline => Student Queries



Inverted Page Table

OPERATING SYSTEMS

Revision Class Outline => Student Queries

- The simplest form of an inverted page table contains one entry per physical page in a linear array.
- Since the table is shared, each entry must contain the process ID of the page owner.
- And since physical pages are now mapped to virtual, each entry contains a virtual page number instead of a physical.
- The physical page number is not stored, since the index in the table corresponds to it.
- As usual, information bits are kept around for protection and accounting purposes.
- Assuming 16 bits for a process ID, 52 bits for a virtual page number, and 12 bits of information, each entry takes 80 bits or 10 bytes, so the total page table size is 10 * 128KB = 1.3MB

OPERATING SYSTEMS

Revision Class Outline => Student Queries

- In order to translate a virtual address, the virtual page number and current process ID are compared against each entry, traversing the array sequentially.

- When a match is found, the index of the match replaces the virtual page number in the address to obtain a physical address.

- If no match is found, a page fault occurs.

- While the table size is small, the lookup time for a simple inverted page table can be very large.

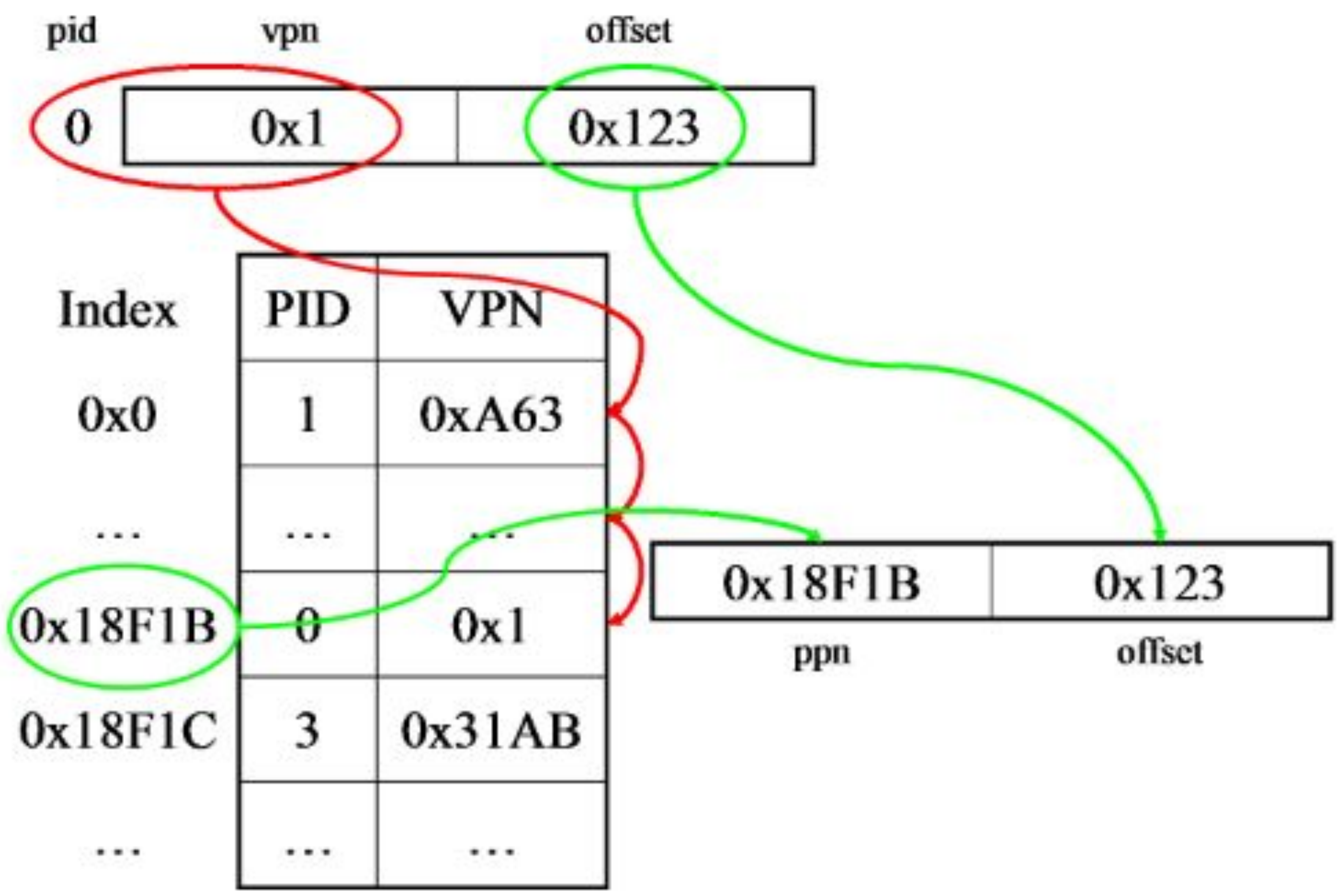
- Finding a match may require searching the entire table, or 128K memory accesses!

- On average, we expect to take 64K accesses in order to translate an address. This is far too inefficient, so in order to reduce the amount of required searching, hashing is used

- Linear inverted page table
  - Page table entry contains
    - Process ID (table shared between all processes)
    - Virtual page number
  - Table indexed by physical page number

OPERATING SYSTEMS

Revision Class Outline => Student Queries



Inverted Page Table

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean, IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com