★★ Module-3 ★★

Page No.
Date 27 | 3 | 23

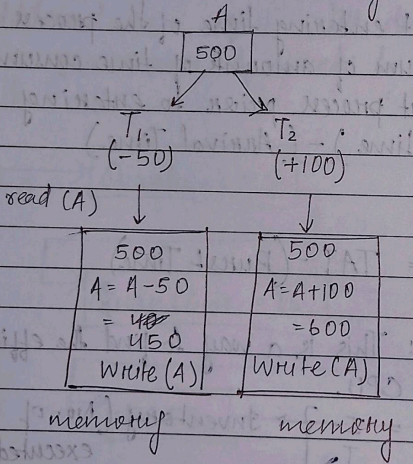
- Process Synchronization -

→ Background
→ The Critical-Section Problem
→ Semaphores
→ Classical Problems of Synchronization
→ Monitors

★ Processes here are co-operative.

★ Background - Concurrent access (2 or more processes access the data simultaneously) to shared data may result in data inconsistency.

Concurrent access anomaly



Ans. should be 550
but here, the race situation occurs & o/p will either be 600 or 450. (both incorrect)
∴ Synchronization is required.

→ Maintaining data inconsistency requires mechanisms to ensure the orderly execution of cooperating processes.

→ Race Condition : The situation where several processes access & manipulate shared data concurrently. The final value of the shared data depends on which process finishes last.

→ To prevent must be sy

Q. Let T1 tran 10% of the.

T1
r(A)
A = A-5
W(A)
r(B)
B = B+5
W(B)

r → read
W → write

↑ These are
This is S
Let A = 50
O/p T1 :

O/p T2 :

Before
450

Date

→ To prevent race conditions, concurrent processes must be synchronized.

Q. Let T_1 transfers ₹50 from A to B, & T_2 transfer 10% of the balance from A to B.

r → read
w → write

T_1	T_2
r(A)	r(A)
A = A - 50	temp = 0.1 * A
w(A)	A = A - temp w(A)
r(B)	r(B)
B = B + 50	B = B + temp
w(B)	w(B)

These are the steps of transaction T_1 & T_2.
This is Scenario -1 (Serial Execution)

Let A = 500, B = 1000

O/p T_1 : r(A) 500
A = 500 - 50 = 450
w(A) 450
r(B) 1000
B = 1000 + 50 = 1050
w(B) 1050

O/p T_2 : r(A) ~~500~~ 450
temp = 0.1 * ~~500~~ 450 = ~~50~~ 45
A = ~~500 - 50~~ 450 - 45 = 405
w(A) ~~450~~ 405
r(B) 1050
B = 1000 + ~~50~~ 45 = ~~1050~~ 1095
w(B) 1095

Before (A+B) = After (A+B)
450 + 1050 = 405 + 1095
1500 = 1500
∵ Equal. ∴ Database is in consistent state.

Scenario - 2

A = 500, B = 1000

T_1	T_2
read(A)	read(A)
A = A - 50	temp = A * 0.1
	A = A - temp
	W(A)
	rR(B)
W(A)	
r(B)	
B = B + 50	
W(B)	
	B = B + temp
	W(B)

$T_1(A+B) = T_2(A+B)$

450 + 1050 = 405 + 1095

1500 = 1500

∴ DB is consistent

A = 200, B = 700

T_1	T_2
A = 150	A = 180
B = 750	temp = 20
	B = 700 + 20 = 720

$T_1(A+B) \neq T_2(A+B)$

∴ Inconsistent

Scenario - 2 allows both T_1 & T_2 to update the values simultaneously

Page No.
Date 29 03 23

The Critical-Section Problem:-

→ 'n' processes, all competing to use some shared data.
→ Each process has a code segment, called critical section in which the shared data is accessed, and modified.
→ The problem: ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Critical Section-

→ general structure of process P_i :-

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

* entry section: the process makes a request to enter its critical section.
* critical section: updates the shared value.
* exit section: comes out of critical section.
* remainder section: executing the remaining portion of the program.

Algorithm for Process P_i :-

```
do { while (turn == j);
        critical section
     turn = j;
        remainder section
} while (true)
```

Algorithm for Pi: (Critical)

```
while (turn == i);
        C.S.
turn = i
        R.S.
} while (true)
```

↖ when false, then execute the 3 steps.

Solution to a Critical Section problem.

→ The soln to designed to solve the CSP should satisfy the following 3 properties:-

① Mutual Exclusion.
Mutual Exclusion implies that only one process can be inside the critical section at any time. If any of the other processes require the critical section, they must wait until it is free.

② Process Requirement -
→ Currently no process is executing in its critical section.
→ But there exist some processes that wish to enter their critical section.
→ Then, the selection of the processes that wish to enter their critical will enter the critical section next cannot be postponed indefinitely.

③ Bounded Waiting Requirement -
→ A bound must exist on -
→ the no. of times other processes are allowed to enter their critical sections.
→ After a process has made a request to enter its critical section & before that request is granted.

Page No.
Date 3 4 23

- Peterson's Solution -

→ It is a 2 process solution. (won't work for more than 2 processes)

→ It was the first sol" which solved the previous 3 requirements.

→ These 2 processes share 2 variables:

→ int turn;

→ Boolean flag[2];

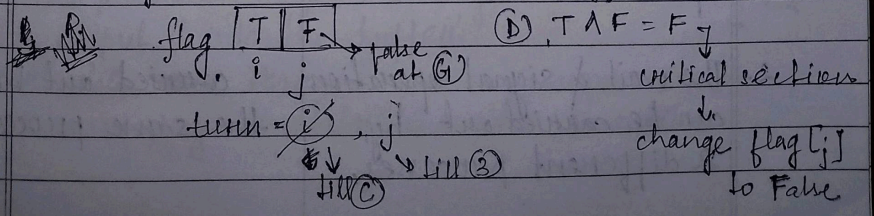
→ The variable "turn" indicates whose turn it is to enter the critical section.

→ The flag array is used to indicate if a process is ready to enter the critical section.

→ flag[i] = true implies that process P_i is ready.

Algorithm for process P_i		Algo. for process P_j
1. do{	A.	do{
2. flag[i] = true;	B.	flag[j] = true;
3. turn = j;	C.	turn = i;
4. while (flag[j] &&	D.	while (flag[i] &&
5. turn == j);	E.	turn == i);
6. critical section	F.	critical section
7. flag[i] = false;	G.	flag[j] = false;
8. remainder section	H.	remainder section
9. } while (true);	I.	} while (true);

(we may start either with P_i or P_j).



- Semaphores -

→ It is a synchronization tool.
→ Semaphore S - integer variable.
→ can only be accessed via 2 indivisible (atomic) operations.

```
wait (S):
    while S ≤ 0 do no-op;
    S--;

signal (S):
    S++;
```

→ 2 Types of Semaphores -

① Counting Semaphore : integer value can range over an unrestricted domain.

② Binary Semaphore : integer value can range only between 0 & 1.

→ Semaphore (Mutex)

5/4/23

Q. What's the difference b/w a Binary Semaphore & a Mutex ?

Binary Semaphore	Mutex
1. Semaphore is a signaling operation.	Locking operation
2. The wait & signal operation can be carried out by 2 different processes.	carried out by the same process.

3. Semaphore is used as a synchronization tool.	used when we want exclusive access to a resource.

Deadlock & Starvation -

Deadlock - 2 or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

→ Let 'S' & 'Q' be 2 semaphores initialized to 1

S=1

Q=1

P_0		P_1	
wait(S);	① S=0	wait(Q);	② Q=0
wait(Q);	③ waiting	wait(S);	④ waiting
...		...	
signal(S);		signal(Q);	
signal(Q);		signal(S);	

Both ③ & ④ waiting ∞.

Classical Problems of Synchronization -

1. Bounded Buffer Producer - Consumer Problem.
2. Readers & Writer Problem.
3. Dining - Philosophers ~~Pb~~ ".

① - B-B ~~&~~ P-C Problem -

→ The problem describes 2 processes, the producer & the consumer, who share a common, fixed-size buffer.

→ The producer's job is to generate data & ~~it~~ put it into the buffer.

→ At the same time, the consumer is consuming the data one piece at a time.

→ The problem is to make sure that the producer won't try to add data into the buffer if it's full & that the consumer won't try to remove data from an empty buffer.

→ The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer it notifies the producer, who starts to fill the buffer again.

→ Similarly, the consumer can go to sleep if it finds the buffer is empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

→ The sleeping can be reached by means of inter process communication typically using semaphores

Shared data

semaphore: full = 0, empty = n → counting semaphores
mutex = 1; → binary semaphore

→ Full indicates the no. of buffer places that contain the data item, whereas the empty represents the no. of empty spaces in buffer.

→ At the doesn't

Produce
do{
wait
wait
add
sig
sig
}while

Consum
do{
wait
wait
item
sign
sign
con
}while

7/4/23

full = 5
consum
wait

produce

→ At the initial state, we'll assume that the buffer doesn't contain any data item.

Producer Process

```
do{
    produce an item
    wait(empty);        empty decreases by 1
    wait(mutex);        producer gains control over buffer
    add item to buffer
    signal(mutex);      control is revoked
    signal(full);       full = 1 (1 item in buffer)
}while(1);
```

Consumer Process

```
do{
    wait(full);         full decreases by 1
    wait(mutex);        consumer gains control of buffer
    remove an item from buffer
    signal(mutex);      control revoked
    signal(empty);      empty increases by 1
    consume the item
}while(1);
```

7/4/23 full = 5, empty = n-5, mutex = 1

consumer

wait(full) decreases by 1. full = 5, 4, 3, 2, 1, 0

control accessed

control revoked

empty = (n-4), (n-3), (n-2), (n-1), n = n

producer empty = (n-4), (n-3), (n-2), (n-1), n

control access → n, (n-1), (n-2), (n-3), (n-4)

control revoked

full = 0, 1, 2, 3, 4, 5

n-5

Page No.
Date 17 | 4 | 23

Readers-Writers Problem

A data set is shared among a no. of concurrent processes.

→ Readers: only read the data set; they don't perform any updates.

→ Writers: can both read & write.

* Problem: allow multiple readers to read at the same time.
→ Only one single writer can access the shared data at the same time.

→ Several variations of how readers & writers are considered - all involve some form of priorities.

* Readers-Writers Problem is of 2 types.
① First RW Problem (Reader has priority)
② Second RW Problem (Writer has priority)

→ Shared Data:
→ Data set
→ Semaphore rw_mutex or(wrt) initialized to 1
→ Semaphore mutex initialized to 1
→ Integer read_count " " 0

Semaphore soln to the RW problem

Writer Process:
wait(wrt);
// writing is performed.
signal(wrt);

Date

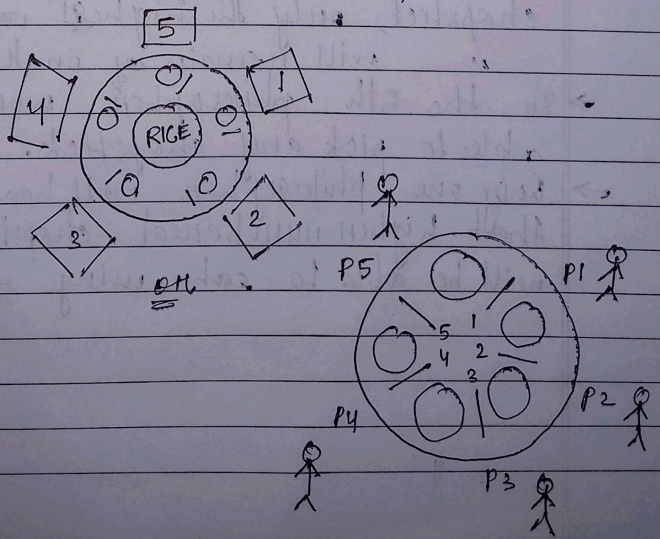
Reader Process

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

19.4.23 Dining - Philosophers Problem

→ The dining philosophers problem is a classic synchronizing problem involving the allocation of limited resources among a group of processes in a deadlock-free & starvation-free manner.

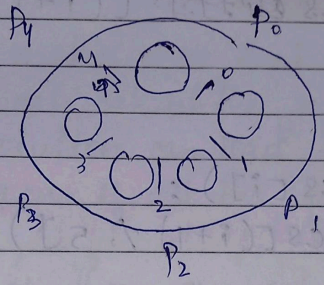
→ Consider 5 philosophers sitting around a table, in which there are 5 chopsticks evenly distributed & an endless bowl of rice in the center, as shown in the diagram:



→ Chopsticks are placed b/w each pair of adjacent philosophers.
→ When it is time for a philosopher to eat, it must first acquire 2 chopsticks - one from their left & one from their right.
→ A philosopher can use the chopstick only if it is not being used by another philosopher.
→ After an individual philosopher finishes eating, they need to put down both chopsticks so that the chopsticks become available to others.
→ The problem is to design a solution such that no philosopher will starve.
→ The solution to the problem was originally proposed by Dijkstra.
→ It assigns a partial order to the resources (the chopsticks) & establishes the convention that all resources will be requested in order.
→ Each of the philosopher will always pickup the lowers numbered chopstick first, & then the higher no. chopstick.
→ In this case, if four of the five philosophers simultaneously pick up their lowers numbered chopstick, only the highest numbered will remain on the table.
→ So, the 5th philosopher won't be able to pick any chopstick.
→ Only one philosopher will have access to that higher numbered chopstick, so he will be able to eat using 2 chopsticks.

Struct
Proble
24/4/23

From implementation point -



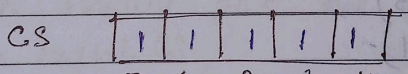
Structure of Dining ~~Philosopher~~ Philosophers Problem.

⇒ ~~Shared~~ Shared data:

- semaphore chopstick[5]; (initially all values are 1)

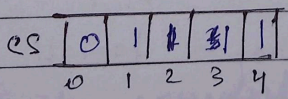
24/4/23

Chopstick : CS



→ all CS on table ✓

wait (CS[0]) → picked up by either P_0 or P_4.



→ Philosophers i :: (For ith philosopher)

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

$P_0 \rightarrow (0, 1)$ → P_0 picks up CS → 0 & 1
$P_1 \rightarrow (1, 2)$
$P_2 \rightarrow (2, 3)$
$P_3 \rightarrow (3, 4)$
$P_4 \rightarrow (4, 0)$

→ Philosophers : 1

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

Page No.
Date

→ When all processes are holding all the resources, none can proceed due to lack of further resources & such scenario is called a Deadlock.

→ The solution makes sure that no 2 neighboring philosophers can eat at the same time.
→ But this solution can lead to a deadlock.
→ This will happen if all the philosophers pick their right / left chopstick simultaneously. Then none of them can eat & deadlock occurs.

→ Some of the ways to avoid deadlock are:
→ There should be at most 4 philosophers on the table.
→ An even philosopher should pick the right CS & then the left CS while an odd philosopher should pick the left CS & then the right CS.

- Limitation of Semaphore -

→ Their use is not enforced, but is by convention only.

→ The programmer has to keep track of all calls to wait & to signal the semaphore.

→ With improper use, a process may block indefinitely, such situation is called a deadlock.

→ The alternative to semaphore is monitor

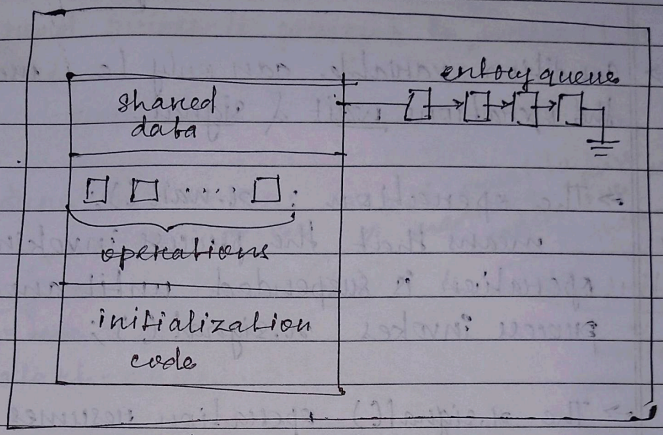
Date

Semaphore	Monitor
① Semaphore is an integer variable.	① Monitor is an abstract data type.
② The value of semaphore S indicates the no. of shared resources available in the system.	② The monitor type contains shared variables & the set of procedures that operate on the shared variable.
③ When any process access the shared resources it performs wait() operation & when it releases the resources it performs signal() operation.	③ When any process wants to access the shared variables in the monitor, it needs to access it through the procedures.
④ Semaphores do not have condition variables.	④ Monitors has condition variable

Monitors

Page No.
Date 26 | 4 | 23

→ High level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.



Schematic view of a monitor

Programming structure:

```
monitor monitor-name
{
  shared variable declarations
  procedure body P1 (...) {...
    ...
  }
  procedure body P2 (...) {
    ...
  }
  procedure body Pn (...) {
    ...
  }
  {
    initialization code
  }
}
```

Date

- Monitor with

→ To allow a process to wait within the monitor, a condition variable must be declared, as condition

condition x, y;

⇒ Condition variable can only be used with the operations wait & signal.

⇒ The operation : x.wait();
means that the process invoking this operation is suspended until another process invokes x.signal();

⇒ The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

- Monitor solution to the R-W Problem -

```
monitor readers-writers
begin
readcount : integer;
busy : boolean;
oktoread, oktowrite : conditions;

Procedure start-read;        → this procedure is to start reading
begin
if busy then oktoread.wait;  → oktoread suspended
readcount = readcount + 1;
oktoread.signal;             → oktoread allowed to execute

end start-read;
```

means a write, read procedure is ongoing on

28/4/23

Assignment 9.

Page No.
Date

```
Procedure end-read;
begin
readcount = readcount - 1;
if readcount = 0 then oktowrite.signal;
// no read process is present so write() is allowed.
end end-read;

Procedure start-write;
begin
if (busy or readcount <> 0) then oktowrite.wait;
busy = true;
end start-write;

Procedure end-write;
begin
busy = false;
if oktoread.queue then oktoread.signal
else oktowrite.signal
end end-write;

begin (* Initialization code *)
readcount = 0; busy = false;
end;

end readers-writers;
```

writer is active

not equal to meaning some readers are active

then

write process is suspended

readers are waiting in queue so

reader is allowed to execute.

28/4/23

Assignment

8. Write the monitor solution to the producer-consumer problem & the dining-philosophers problem.

- Deadlock -

→ In concurrent computing, a deadlock is a state in which each member of a group is waiting for some other member to take action.

→ In OS, a deadlock occurs when a process enters a waiting state because the requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process.

- System Model -

→ Resource types $R_1, R_2, \ldots, R_m$

→ Each resource type R_i has W_i instances.

→ Each process utilizes a resource as follows:
  → request
  → use
  → release

- Deadlock Characterization -

imp

→ Deadlock can arise if four conditions hold simultaneously.

① Mutual Exclusion : only one process at a time can use a resource.

② Hold & wait : a process is holding at least one resource & is waiting to acquire additional resources held by other processes.

③ No preemption : a resource can be released

voluntarily by the process holding it, after the process has completed its task.

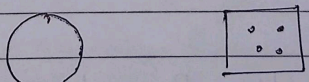
(iv) Circular wait : there exists a set $\{P_1, P_2, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by P_n, & P_n is waiting for a resource that is held by P_0.

- Resource Allocation Graph -

→ A set of vertices V and a set of edges E
→ V is
→ V is partitioned into 2 types :
→ $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
→ $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

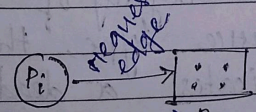
→ request edge : directed edge from $P_i \rightarrow R_j$
→ assignment edge : " " " $R_j \rightarrow P_i$

Representation :

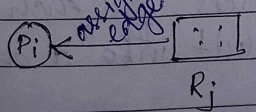


Process

Resource Type with 4 instances

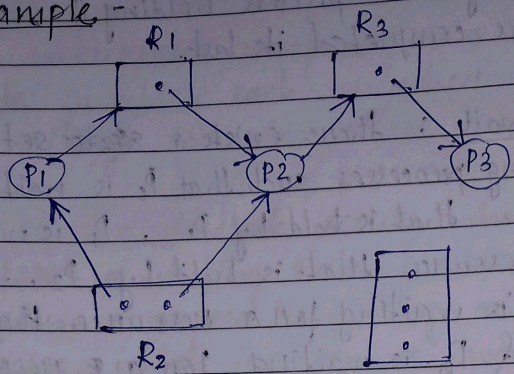


P_i requests instance of R_j



P_i is holding an instance of R_j

example -



(resource type R_2 has 2 instances, out of which one is assigned to P1 & the other is assigned to P2)

→ If in the resource allocation graph, we find at least one resource having more than one instances, then that graph belongs to the category of multiple instances of each resource type. otherwise the graph belongs to the category of single instance of each resource type.

1/5/23

Concluding Remarks -

→ If the graph is for the category of single instance for each resource type, then, a cycle in the graph indicates a deadlock.

→ But for multiple instances of each resource type, a cycle may result in a deadlock.

→ For multiple instances, we'll use Banker's Algorithm, to conclude if the system is in a deadlock state or not.

Basic Fact -

1. If the graph has no cycle ⇒ No deadlock
2. If the graph contains a ~~deadlock~~ cycle ⇒
   → if only one instance per resource type, then deadlock
   → if several instances per resource type, then possibility of deadlock.

* Methods to handle Deadlock -
(1) Deadlock Prevention
(2) Deadlock Avoidance
(3) Deadlock Detection & Recovery

(1) Deadlock Prevention
→ This strategy states that deadlock can be prevented if any one of the "necessary conditions for deadlock" is made to false.

(i) → Mutual Exclusion : not required for sharable resources ; must to hold for non-sharable resources. ∴ It can't be made false.

(ii) → Hold & wait : must guarantee that whenever a process requests a resource, it doesn't hold any other resources.

(iii) → No Preemption :
→ If a process that is holding some ~~requests~~ resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
→ Preempted resources are added to the list of resources for which the process is waiting.

→ Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

(iv) → Circular wait :

To make the circular wait condition to be false, let us impose a partial ordering among the resources & the process should request the resources in the increasing order of enumeration.

$$f : R \rightarrow N$$

every resource is given a number.

e.g.

f(CPU) = 3

f(Printer) = (4) → Let a process is using resource (4), then this process can request resources (5) > (4)

f(Scanner) = 5

↘ these are resource.

→ let us define a func^n : $f : R \rightarrow N$.

→ This func^n will assign a unique +ve integer values to the resources.

→ To prove, that the circular wait condition doesn't condition, we will use the method of contradiction.

Proof

→ Suppose, a circular wait condition exists between a set of processes $\{P_0, P_1, \ldots, P_n\}$.

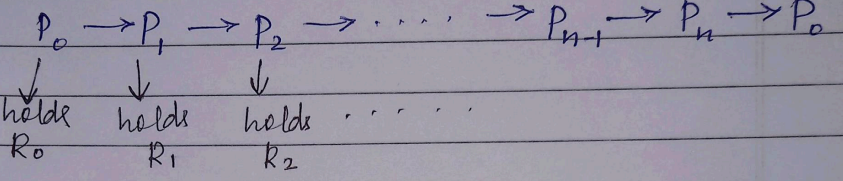
→ let the process P_i is holding the resource R_i & since, a circular wait condition exists,

P_0 is waiting for P_1 or

P_1 " " " P_2

:

or



From here, we can say: $f(R_0) < f(R_1)$

$< f(R_2) < \cdots < f(R_n)$

$< f(R_0)$

From above: $f(R_0) < f(R_0)$

which is not possible.

By transitivity, we can write $f(R_0) < f(R_0)$ which is false, hence our assumption is wrong & hence a circular wait condition doesn't exist.

(2) Deadlock Avoidance -

false, hence our assumption is wrong & hence a circular wait condition doesn't exist.

5/5/23

(2) Deadlock Avoidance -

→ The simplest model states that each process should declare the maximum no. of resources of each type that it may need.

→ The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

→ Resource-allocation state is defined by the no. of available & allocated resources, & the maximum demands of the processes.

Safe-state : When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe-state.

→ System is in safe-state if there exists a

safe-sequence of all processes.

Basic Facts.

→ If a system is in safe state ⇒ no deadlocks.
→ " " " " unsafe " ⇒ possibility of deadlock.
→ Avoidance ⇒ ensure that a system will never enter into an unsafe state.

- Data Structures for the Banker's Algorithm -

Let n = no. of processes
m = no. of resource types.

Processes	Resource Types			
	R_1	R_2	....	R_m
P_1	x	y	...	z
P_2	a	b	...	c
:	:	:		:
P_n				

→ Available : Vector of length 'm'.
If available[j] = k,
there are k instances of resource type R_j available

AVL:	0	1	0	7	3	1
	R_1	R_2	...	R_j		m

→ Max : (n x m) matrix.
If Max[i, j] = k, then process P_i may request at most 'k' instances of

resource type R_j.

→ Allocation : (n x m) matrix.
If Allocation[i,j] = k, then
P_i ~~may~~ is currently allocated 'k' instances of R_j.

→ Need : (n x m) matrix.
If Need[i,j] = k, then
P_i may need k more instances of R_j to complete its task.

Need[i,j] = Max[i,j] - Allocation[i,j]

Safety Algorithm (safety-state algo):

① Let Work & Finish be vectors of length 'm' & 'n', respectively.

m → work → resource ~~type~~
n → Finish → process

Initialize:
Work = Available // resources that are available are put into 'work'.
Finish[i] = false for i = 1, 2, 3, ..., n
↳ none of the processes have completed their execution.

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$_i$
Finish[i] = true
go to step ②.

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

- Example of Banker's Algorithm -

Given Q. Check if the system is in the safe state.

5 processes P_0 through P_4

3 resource types A (10 instances)
B (5 ")
C (7 ")

Given, snapshot at time T_0 :

$10-(2+3+2)$
$5-(1+1)$
$7-(2+1+2)$

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2	7	4	3
P_1	2	0	0	3	2	2				1	2	2
P_2	3	0	2	9	0	2				6	0	0
P_3	2	1	1	2	2	2				0	1	1
P_4	0	0	2	4	3	3				4	3	1

Step (1) ∴ None of the P_i has 000 in 'Need'
∴ No P_i has finished its execution.

Step (2) we have AVL : 3 3 2

Step (3) we can fulfill P_1 which has 1 2 2 in need

∴ work = 3 3 2
+ 2 0 0
= 5 3 2

Now P_1 is done.

Similarly,

work for P_3 : 5 3 2
+ 2 1 1
7 4 3 (P_3 done)

for P_0 : 7 4 3
+ 0 1 0
7 5 3 (P_0 done)

for P_4 : 7 5 3
+ 0 0 2
7 5 5 (P_4 done)

for P_2 : 7 5 5
+ 3 0 2
10 5 7 (P_2 done)

instance of A B C completed.

∴ The safe sequence is : $\langle P_1, P_3, P_0, P_4, P_2 \rangle$

∵ There exists a safe-sequence, the system is in the safe-state.
Sometimes, we may have more than one safe sequence & so,

⇒ Order of execution is irrelevant.
(We can start with any process)

- Resource-Request Algorithm for Process P_i -

Request$_i$: request vector for process P_i.
If Request$_i[j] = k$, then process P_i wants 'k' instances of resource type R_j.

① If Request$_i \leq$ Need$_i$, go to step ②.
Otherwise raise error.

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

(3) Pretend to allocate requested resources to P_i by modifying the state as follows:

$Available = Available - Request_i$;
$Allocation_i = Allocation_i + Request_i$;
$Need_i = Need_i - Request_i$;

→ If safe ⇒ the resources are allocated to P_i.
→ If unsafe ⇒ P_i must wait, & the old resource-allocation state is restored.

→ If safe ⇒ the resources are allocated to P_i
→ If unsafe ⇒ P_i must wait, & the old resource-allocation state is restored.

8/5/23 (referring to previous question)

Q. Suppose the process P_1 makes a request (1, 0, 2)

	Need	Avl
P_1:	1 2 2	3 3 2

(1 0 2)

Step ① ✓ → Request ≤ Need, i.e. (1 0 2) ≤ (1 2 2)

Step ② ✓ → Request ≤ Available, i.e. (1 0 2) ≤ (3 3 2)

Step ③ → Both True

→ After virtual allocation, the new state is obtained as -

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

~~10 - (1+3+2)~~ → 5 - (2) → 7 - (7)

We have avl: (2 3 0)

→ We can fulfill P_1

∴ work: 2 3 0 + 0 2 0 = 2 5 0

2 3 0 + 3 0 2 = 5 3 2

(P_1 done)

P_3

2 5 0 + 0 1 1 = 2 6 1

5 3 2 + 2 1 1 = (7 4 3)

(P_3 done)

P_4

2 6 0 + 6 0 0 = 8 6 0

7 4 3 + 0 0 2 = (7 4 5)

(P_4 done)

P_2

8 6 0 + 4 3 1 = 12 9 1

7 4 5 + 3 0 2 = 10 4 7

(P_4 done) (P_2 done)

P_0

12 9 1 + 7 4 3 = 19 13 4

(P_0 done)

P_0

10 4 7 + 0 1 0 = 10 5 7

(P_0 done)

∴ Safe sequence: $\langle P_1, P_3, P_4, P_2, P_0 \rangle$

∵ There exists a safe sequence, the system is in the safe state & hence, the virtual allocation is converted into actual allocation & ~~&~~.

Q: Can request for (0, 2, 0) by P_0 be granted.

	Allocation	Need	Avl
P_0	0 3 0	7 2 3	2 1 0
✓ P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
✓ P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

P_1

$$\begin{array}{r} 2\ 1\ 0 \\ +\ 3\ 0\ 2 \\ \hline 5\ 1\ 2 \end{array}$$

P_3

$$\begin{array}{r} 5\ 1\ 2 \\ +\ 2\ 1\ 1 \\ \hline 7\ 2\ 3 \end{array}$$

P_4

$$\begin{array}{r} 7\ 2\ 3 \\ +\ 0\ 0\ 2 \\ \hline 7\ 2\ 5 \end{array}$$

P_0

$$\begin{array}{r} 10\ 2\ 7 \\ 0\ 3\ 0 \\ \hline 10\ 5\ 7 \end{array}$$

P_2

$$\begin{array}{r} 9\ 5\ 4 \\ 6\ 0\ 0 \\ \hline 15\ 5\ 4 \end{array} \qquad \begin{array}{r} 7\ 2\ 5 \\ 3\ 0\ 2 \\ \hline 10\ 2\ 7 \end{array}$$

Limit was 10 : unsafe state

∴ Roll back to previous safe state

P_0

Q. Can request (3, 3, 0) by P_4 be granted.

- Deadlock Detection -

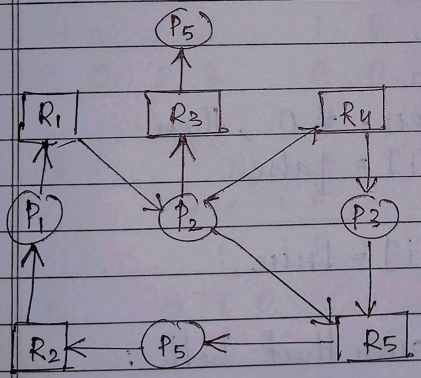
→ Allow system to enter deadlock state.
→ Detection algorithm.
→ Recovery scheme.

Single Instance of each resource type
→ Maintain wait-for graph.
→ Nodes are processing
→ $P_i \rightarrow P_j$ if P_i is waiting for P_j

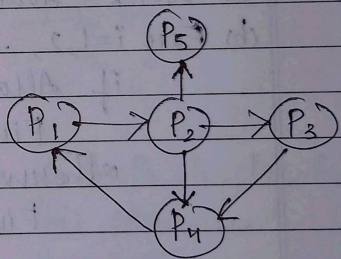
→ Periodically invoke an algorithm that searches for a cycle in the graph.

→ An algorithm to detect a cycle in a graph requires an order of n^2 operations where 'n' is the no. of vertices in the graph.

7/7



Resource Allocation Graph (RAG)



Corresponding wait-for graph (WFG)

10/5/23 - Several instances of a Resource Types -
(Variant of Bankers Algorithm)

→ Available: A vector of length 'm' indicates the no. of available resources of each type.

→ Allocation: An (n x m) matrix defines the no 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.

- Detection Algorithm =

① Let 'Work' & 'Finish' be vectors of length 'm' & 'n', respectively.

Initialize:
(a) Work = Available
(b) For i=1,2, ..., n
if Allocation$_i$ ≠ 0, then
Finish[i] = false
otherwise
Finish[i] = true.

② For an index 'i' such that both:
(a) Finish[i] == false
(b) Request$_i$ ≤ Work

If no such 'i' exists, go to step ④.

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

④

(iii) If Finish[i] == false, for some i, $(1 \le i < n)$ then the system is in deadlock state.

- Example of detection algorithm:

Q. Five processes P_0 through P_4;
3 resource types: A (7 instances)
B (2 ")
C (6 ")

Snapshot at time T_0:

	Allocation			Request			Available (work)		
	A	B	C	A	B	C	A	B	C
✓ P_0	0	1	0	0	0	0	~~0~~	~~0~~	~~0~~
✓ P_1	2	0	0	2	0	2	~~0~~	~~1~~	~~0~~
✓ P_2	3	0	3	0	0	0	~~3~~	~~1~~	~~3~~
✓ P_3	2	1	1	1	0	0	~~5~~	~~1~~	~~3~~
✓ P_4	0	0	2	0	0	2	~~7~~	~~2~~	~~4~~
							7	2	6

P_0 : Avl
0 0 0
+ 0 1 0
0 1 0

P_2 0 1 0
+ 3 0 3
3 1 3

P_1 : 3 1 3
2 0 0
5 1 3

P_3 : 5 1 3
2 1 1
7 2 4

P_4 7 2 4
0 0 2
7 2 6

∴ $\langle P_0, P_2, P_1, P_3, P_4 \rangle$
This sequence will result in Finish[i] = true for all i.
Hence, the system is not in a deadlock state.

Q. Suppose P_2 requests an additional instance of type 'C'. (0 0 1)

∴ ~~Allocation~~

P_2: 0 0 0
0 1 0
0 1 0 → for P_2: Request 0 0 1

Deadlock

∴ 0 ≤ 0 ✓
0 ≤ 1 ✓
1 ≤ 0 ✗
∴ Deadlock

~~Prev~~

- Recovery from deadlock -

① Process Termination -
→ Abort all deadlocked processes.
→ Abort one process at a time until the deadlock cycle is eliminated.
→ In which order should ~~be~~ we choose to abort?
(1) Priority of the process.
(2) How long the process has computed, & how much longer to completion.
(3) Resources the process has used.
(4) Resources that the process needs to complete.
(5) How many processes will need to be terminated
(6) Is process interactive or ~~bat~~ batch.

② Resource Preemption -
→ Selecting a victim - minimize cost.
→ Rollback - return to some safe state, restart process for that state.
→ Starvation - same process may always be picked as victim, include no. of rollback in cost factor.

→ Memor
memor
allocat
→ It check
to pr
→ It decid
time.
→ It trac
free, o
updat
- Prev
→ The
proc
→ The
addre