CS & IT ENGINEERING

Operating System

Deadlocks

Revision

Lecture No.- 08

By- Dr. Khaleel Khan Sir

Recap of Previous Lecture

Topic Process Synchronization

Topics to be Covered

Topic	System Model
Topic	Basic Facts
Topic	Avoidance Algorithms
Topic	Deadlock Detection

Topic : System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  - CPU cycles, memory space, I/O devices
- Each resource type R_i has $\underline{\underline{W_i}}$ instances.
- Each process utilizes a resource as follows:
  - request ✓
  - use ✓
  - release ✓

Deadlock (Lockup):

Two/more Processes (Threads) are said to be in deadlock iff, they wait for the happening of an event which would never happen;

<Infinite Waiting>

→ Processes gets blocked

→ Resource utiliz & thruput declines

Topic : Deadlock with Semaphores

- Data:
  - A semaphore S_1 initialized to 1
  - A semaphore S_2 initialized to 1
- Two threads T_1 and T_2
- T_1:

  wait(s_1)
  wait(s_2)

- T_2:

  wait(s_2)
  wait(s_1)

T_1 & T_2 Should get PreEmpted b/w the Two wait opns ;

Topic : Deadlock Characterization

Necessary Condition

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one thread (Process) at a time can use a resource Shared
- **Hold and wait:** a thread holding at least one resource is waiting to acquire additional resources held by other threads
- **No preemption:** a resource can be released only voluntarily by the thread holding it, after that thread has completed its task (of resource)
- **Circular wait:** there exists a set $\{T_0, T_1, ..., T_n\}$ of waiting threads such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by T_2, ..., T_{n-1} is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

Topic : Resource-Allocation Graph

(R.A.G)

V E

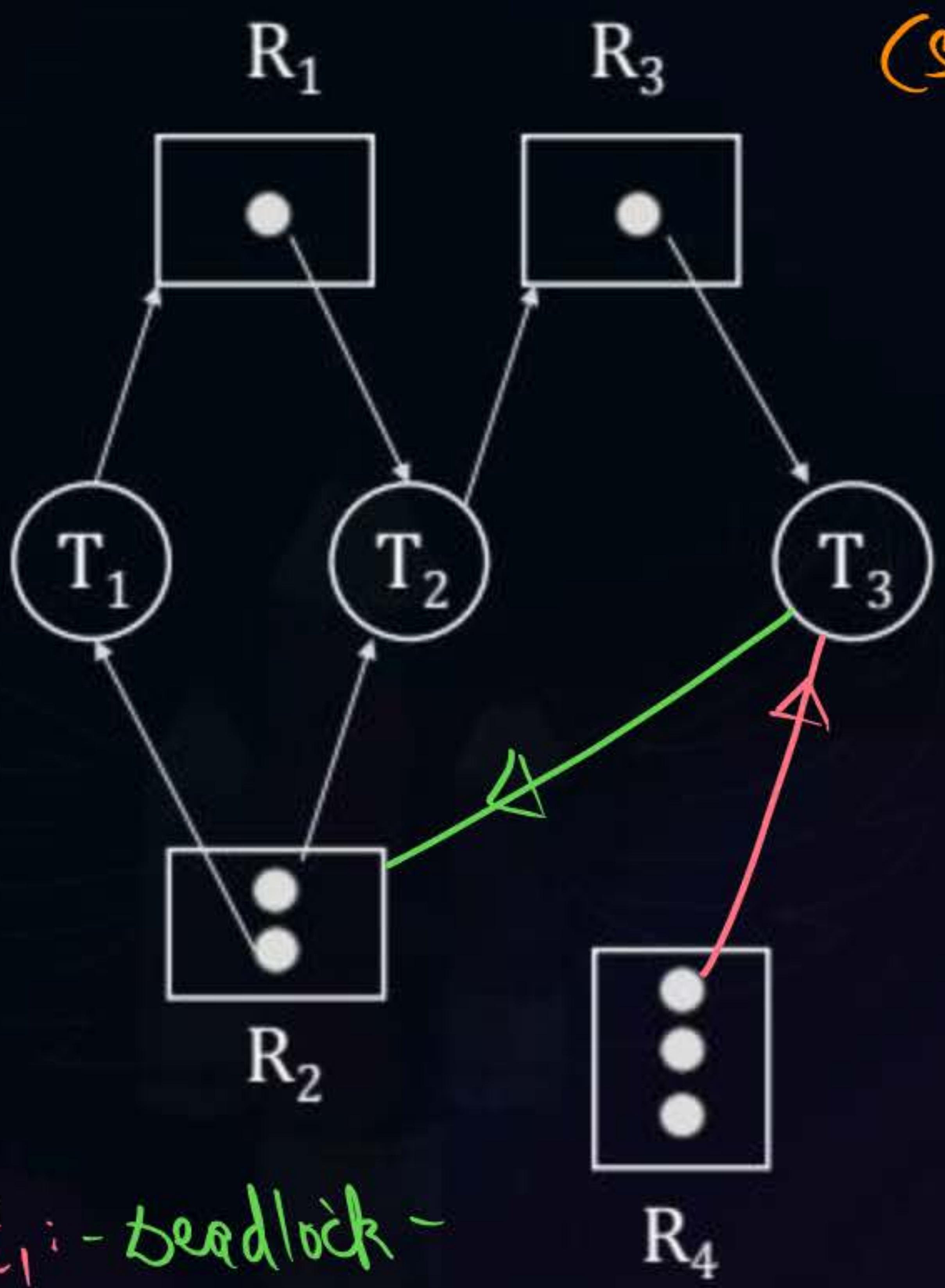
Process Resource Req Assigned

- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $T = \{T_1, T_2, ..., T_n\}$, the set consisting of all the threads in the system.
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system
- **request edge –** directed edge $T_i \rightarrow R_j$
- **assignment edge –** directed edge $R_j \rightarrow T_i$

Topic : Resource Allocation Graph Example

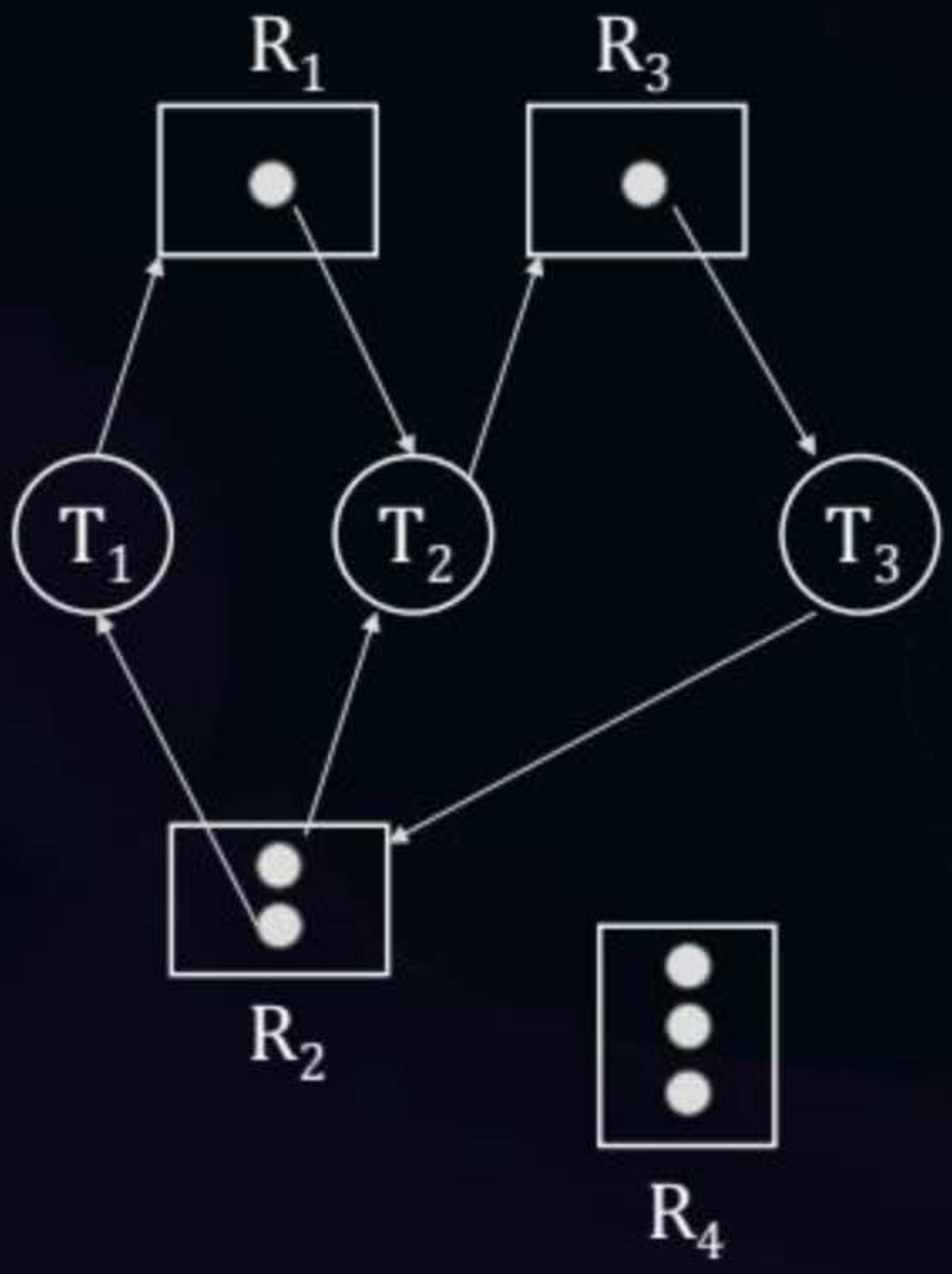
- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waiting for an instance of R_1
- T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- T_3 is holds one instance of R_3

t_0: NOT-IN-deadlock (safe)



t_1: - deadlock -

Topic : Resource Allocation Graph with a Deadlock

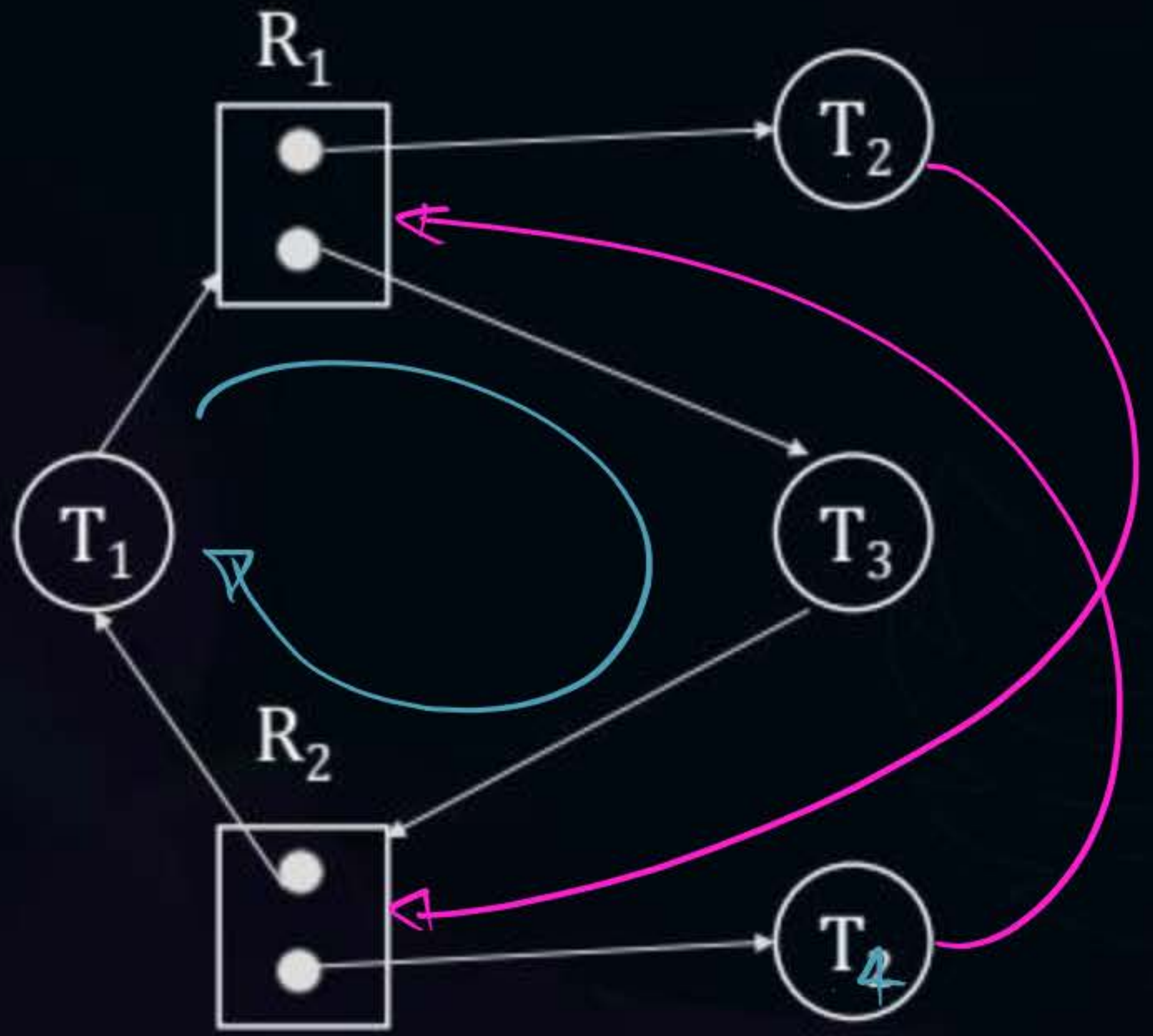


Topic : Graph with a Cycle But no Deadlock

G_2 :

t_0: ⟨No-Deadlock⟩

t_1:



Topic : Basic Facts

- If graph contains no cycles ⇒ no deadlock <Safe>
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock (N & S)
  - if several instances per resource type, possibility of deadlock (May | May Not)

Topic : Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance

  : by Dissatisfying/Negating one/more of Nec. Cond's
- Allow the system to enter a deadlock state and then recover (Detec. & Recovery)
- Ignore the problem and pretend that deadlocks never occur in the system. (Ostrich Algo)

Strategies

(i) Prevention

(ii) Avoidance ⟨Banker's Algo⟩
- S.I. Resource: Res-Alloc. Graph Algo
- Multi-Instance (General)

(iii) Detection & Recovery

(iv) Ignorance ⟨Ostrich Algo.⟩ (NO STRATEGY)

Topic : Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

- **Mutual Exclusion –** not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait –** must guarantee that whenever a thread requests a resource, it (Hold or Wait) does not hold any other resources
  - Require threads to request and be allocated all its resources before it begins execution or allow thread to request resources only when the thread has none allocated to it.
  - Low resource utilization; starvation possible

Topic : Deadlock Prevention (Cont.)

- **No Preemption:**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released – Self PreEmption
  - Preempted resources are added to the list of resources for which the thread is waiting
  - Thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait:**
  - Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration

Topic : Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:

  first_mutex = 1

  second_mutex = 5

Topic : Circular Wait

code for thread_two could not be written as follows:

```
/* thread one runs in this function */
void *do work one (void *param)
{
    pthread mutex_lock (&first mutex); pthread_mutex_lock (&second_mutex);
    /**
    * Do some work
    */
    pthread_mutex_unlock (&second_mutex); pthread_mutex_unlock (&first mutex);

    pthread_exit(0);
}
```

Topic : Circular Wait

```
/* thread two runs in this function */
void *do_work_two (void *param)
{
    pthread_mutex_lock (&second mutex); pthread_mutex lock (&first_mutex);
    /**
    *Do some work
    */
    pthread_mutex_unlock (&first_mutex); pthread_mutex_unlock (&second_mutex);

    pthread_exit(0);
}
```

Topic : Deadlock Avoidance

Requires that the system has some additional a priori information available

- Simplest and most useful model requires that each thread declare the maximum number 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 number of available and allocated resources, and the maximum demands of the processes

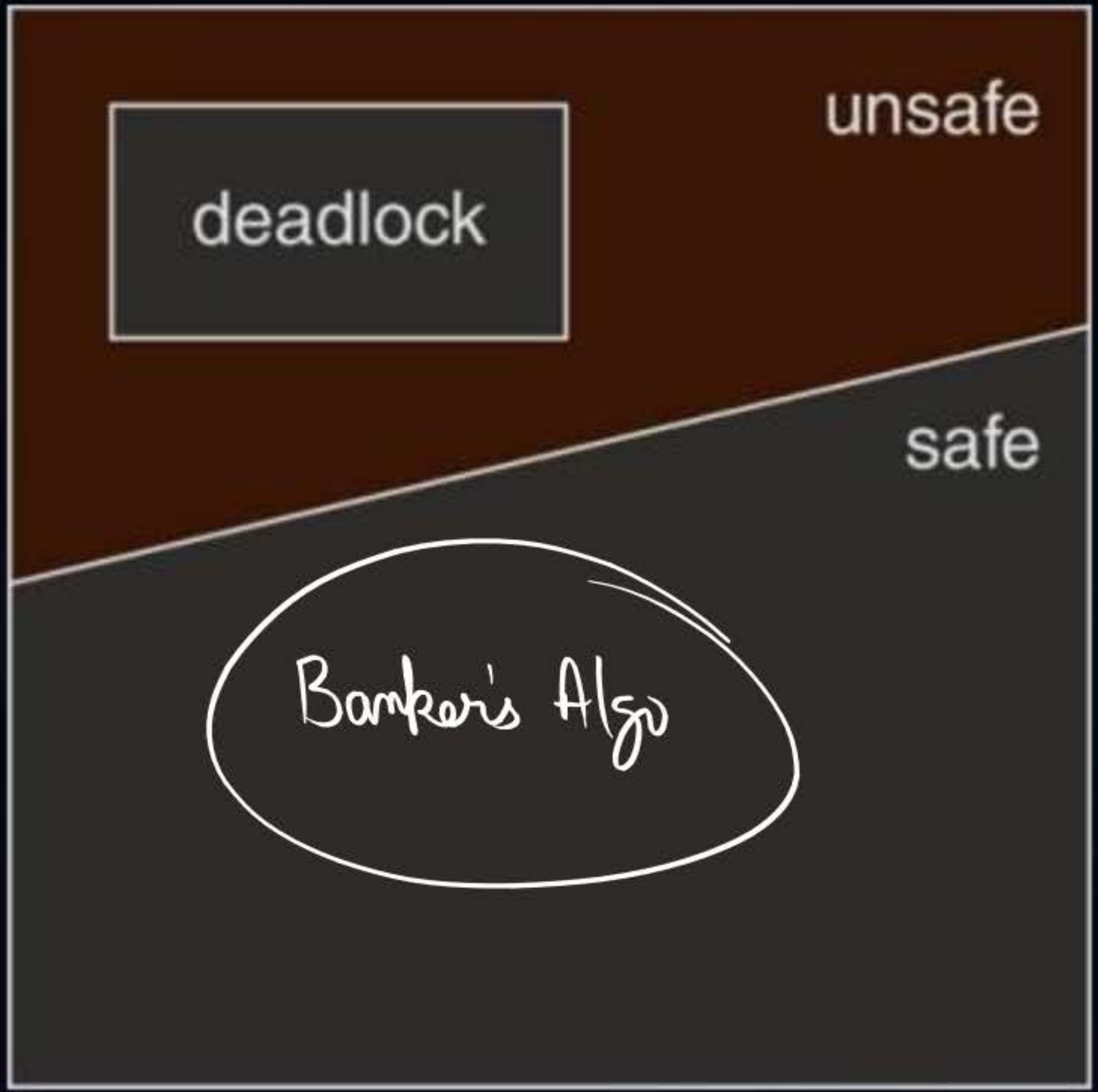
Topic : Safe State

- When a thread 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 sequence $<T_1, T_2, ..., T_n>$ of ALL the threads in the systems such that for each T_i, the resources that T_i can still request can be satisfied by currently available resources + resources held by all the T_j, with $j < I$
- That is:
  - If T_i resource needs are not immediately available, then T_i can wait until all T_j have finished
  - When T_j is finished, T_i can obtain needed resources, execute, return allocated resources, and terminate
  - When T_i terminates, T_{i+1} can obtain its needed resources, and so on

Topic : Basic Facts

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

Topic : Safe, Unsafe, Deadlock State



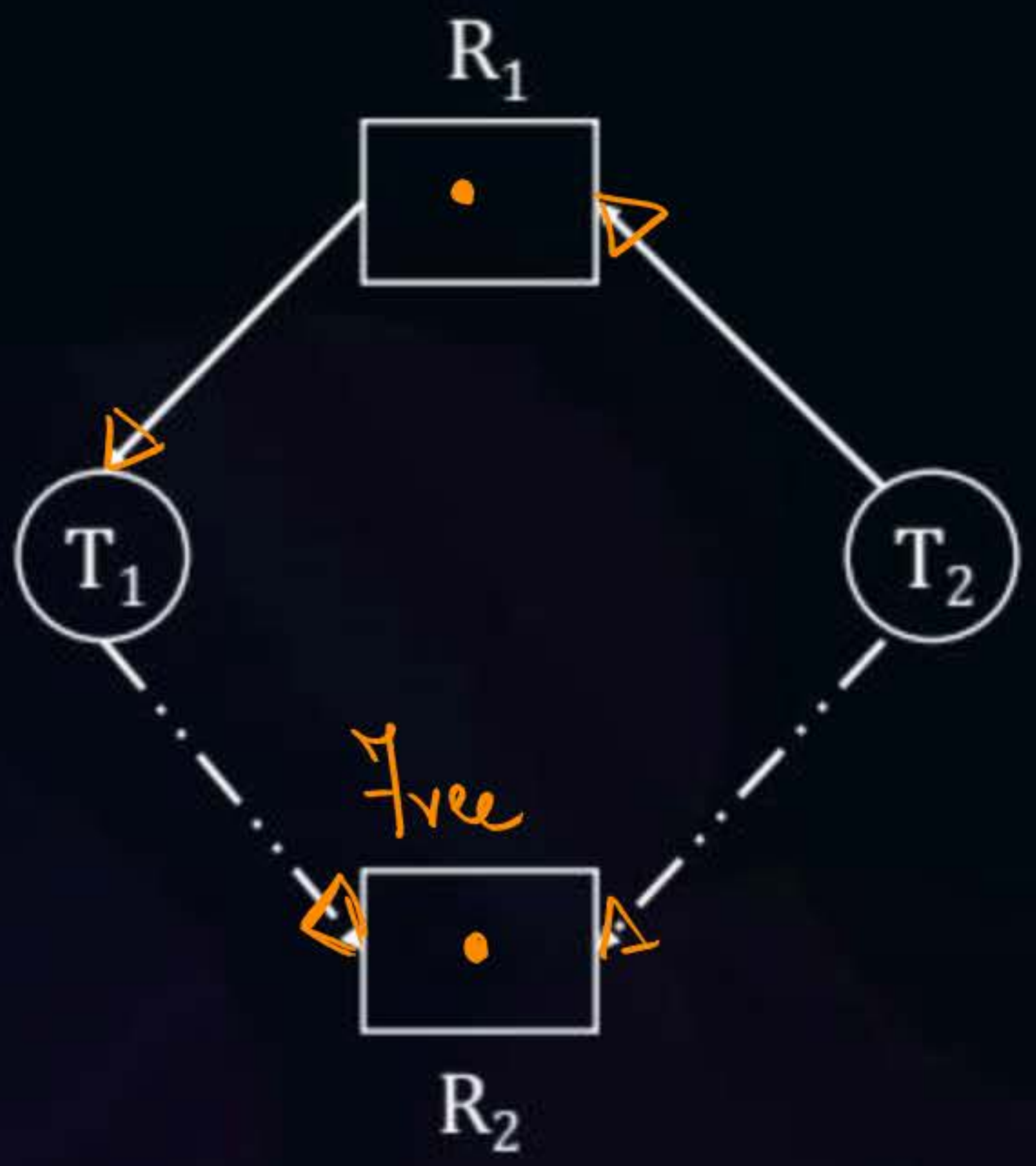
Topic : Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's Algorithm

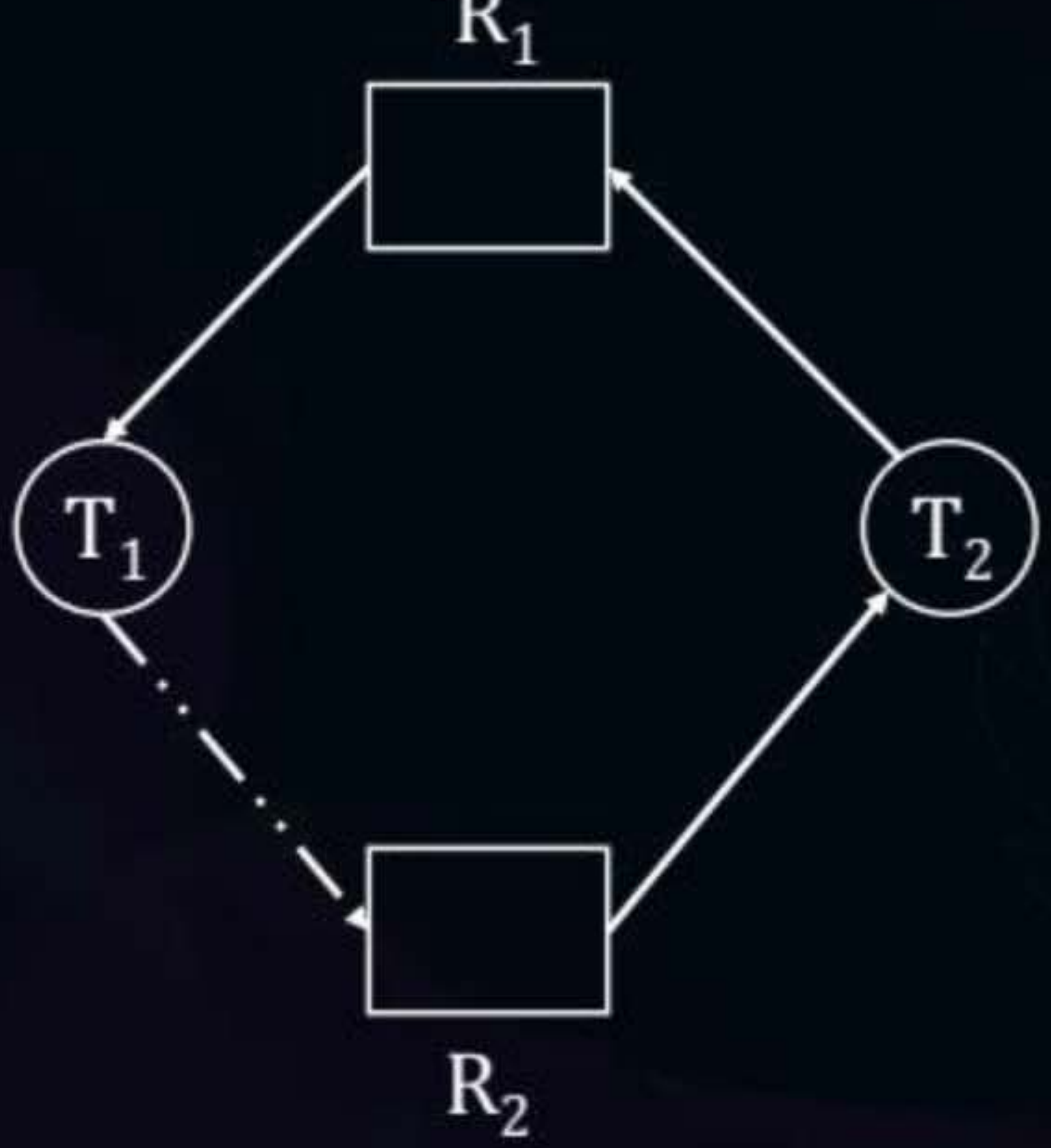
Topic : Resource-Allocation Graph Scheme

- Claim edge $T_i \rightarrow R_j$ indicated that process T_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a thread requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the thread
- When a resource is released by a thread, assignment edge reconverts to a claim edge
- Resources must be claimed a priori in the system

Topic : Resource-Allocation Graph



Topic : Unsafe State In Resource-Allocation Graph



Topic : Resource-Allocation Graph Algorithm

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

Topic : Banker's Algorithm

(Multi-Instance Resource)

- Multiple instances of resources
- Each thread must a priori claim maximum use
- When a thread requests a resource, it may have to wait
- When a thread gets all its resources it must return them in a finite amount of time

(i) n : no. of Processes
(ii) m : no. of Resources
(iii) Max[1..n, 1..m]
(iv) Alloc[1..n, 1..m]
(v) Need[1..n, 1..m]
↳ Max - Alloc

(vi) Available[1..m]
(vii) Total[1..m]
(viii) Request[1..n, 1..m]

$$\text{Total}[j] = \text{Avail}[j] + \sum_{i=1}^{n} \text{Alloc}(i,j)$$

Topic : Data Structures for the Banker's Algorithm

- Let n = number of processes, and m = number of resources types.
- **Available:** Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max:** n x m matrix. If Max [i,j] = k, then process T_i may request at most k instances of resource type R_j
- **Allocation:** n x m matrix. If Allocation[i,j] = k then T_i is currently allocated k instances of R_j
- **Need:** n x m matrix. If Need[i,j] = k, then T_i may need k more instances of R_j to complete its task

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

Topic : Safety Algorithm

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

  Initialize:

  Work = Available

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

- Find an i such that both:

  (a) Finish [i] = false

  (b) $Need_i \leq Work$

  If no such i exists, go to step 4

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

- If Finish [i] == true for all i, then the system is in a safe state

Topic : Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process T_i. If $Request_i[j] = k$ then process T_i wants k instances of resource type R_j

1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \le Available$, go to step 3. Otherwise T_i must wait, since resources are not available
3. Pretend to allocate requested resources to T_i by modifying the state as follows:

$$\begin{cases} Available = Available - Request_i; \\ Allocation_i = Allocation_i + Request_i; \\ Need_i = Need_i - Request_i; \end{cases}$$

- If safe $\Rightarrow$ the resources are allocated to T_i
- If unsafe $\Rightarrow$ T_i must wait, and the old resource-allocation state is restored

Topic : Example of Banker's Algorithm

- 5 threads T_0 through T_4;

  3 resource types:

  A (10 instances), B (5instances), and C (7 instances)

t_0: ⟨T_1; T_3; T_4; T_0; T_2⟩

- Snapshot at time T_0:

	Allocation	Max	Available	Need	
	A B C	A B C	A B C	A B C	
T_0	0 1 0	7 5 3	3 3 2	7 4 3	
T_1	2 0 0	3 2 2		1 2 2	
T_2	3 0 2	9 0 2		6 0 0	
T_3	2 1 1	2 2 2		0 1 1	
T_4	0 0 2	4 3 3		4 3 1	

Topic : Example (Cont.)

- The content of the matrix Need is defined to be Max – Allocation

	Need
	A B C
T_0	7 4 3
T_1	1 2 2
T_2	6 0 0
T_3	0 1 1
T_4	4 3 1

- The system is in a safe state since the sequence $< T_1, T_3, T_4, T_2, T_0 >$ satisfies safety criteria

Topic : Example: P_1 Request (1,0,2)

Granted

- Check that Request ≤ Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

	Allocation	Need	Available
	A B C	A B C	A B C
T_0	0 1 0	7 4 3	2 3 0
T_1	3 0 2	0 2 0	
T_2	3 0 2	6 0 0	
T_3	2 1 1	0 1 1	
T_4	0 0 2	4 3 3	

- Executing safety algorithm shows that sequence < T_1, T_3, T_4, T_0, T_2> satisfies safety requirement
- Can request for (3,3,0) by T_4 be granted? NOT - Granted
- Can request for (0,2,0) by T_0 be granted? NOT - Granted

Topic : Deadlock Detection

: System can go into deadlock

→ Detection

→ Recovery

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

S.I.R : RAG → W.f.G → Cycle

M.I.R → (Safety Algo)

Topic : Single Instance of Each Resource Type

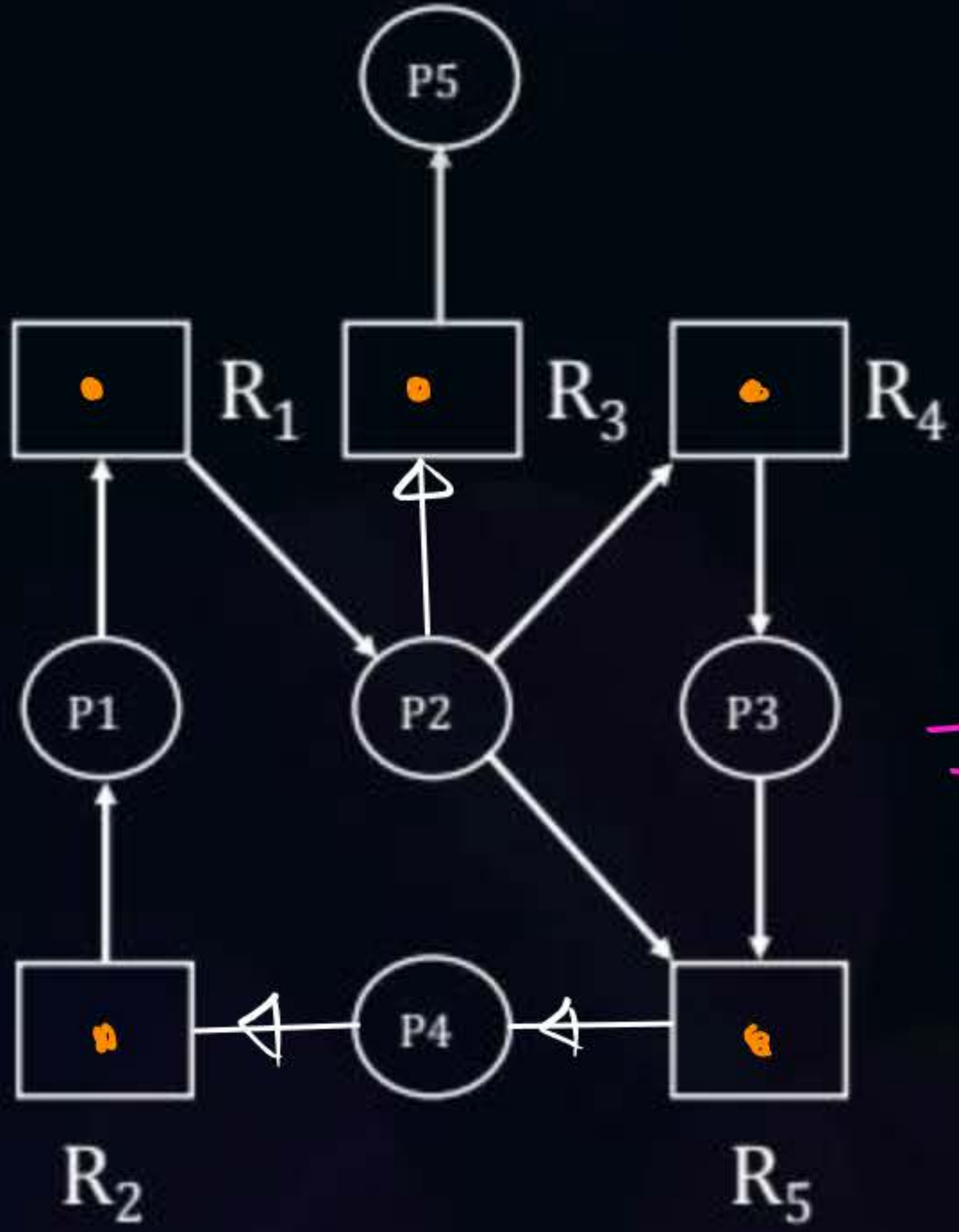
- Maintain wait-for graph
  - Nodes are threads
  - $T_i \rightarrow T_j$ if T_i is waiting for T_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Topic : Resource-Allocation Graph and Wait-for Graph

$P_i = T_i$

$T_1 - T_2 - T_3 - T_4 - T_1$

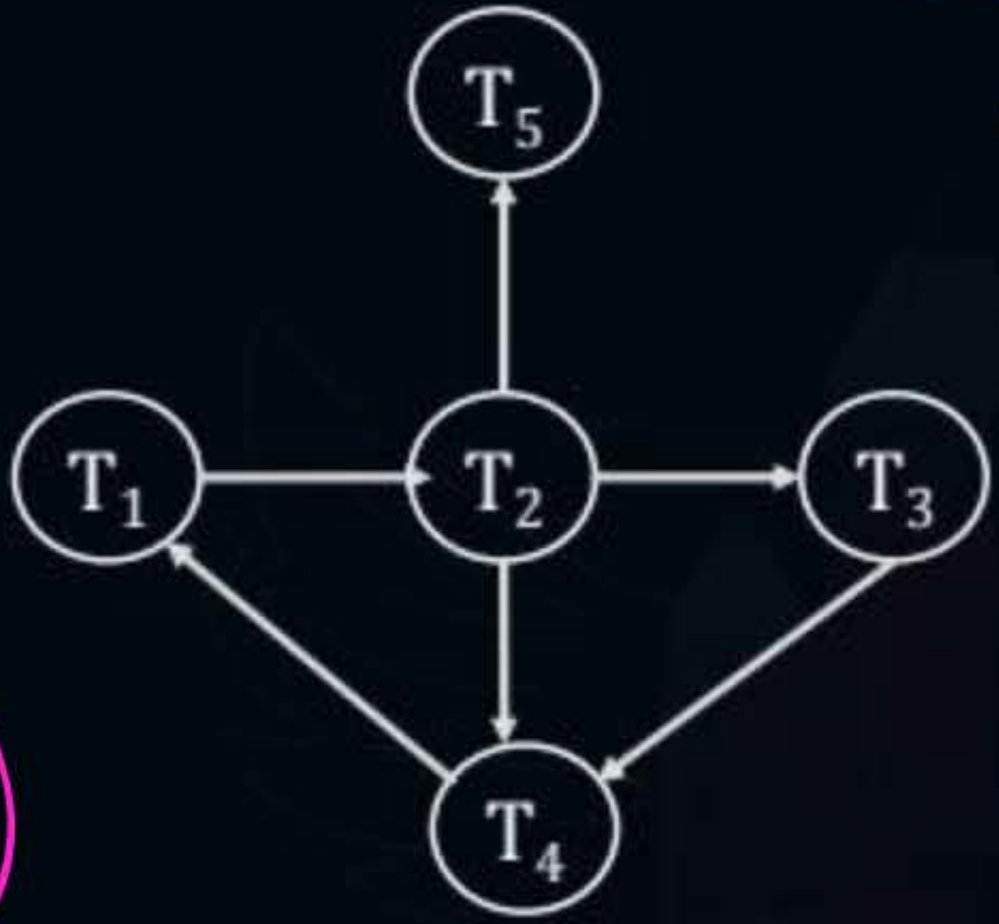
Deadlock



(a)

Resource-Allocation Graph

(wait-for-Graph)



(b)

Corresponding wait-for graph

Topic : Several Instances of a Resource Type

- **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 thread.
- **Request:** An n x m matrix indicates the current request of each thread. If Request [i][j] = k, then thread T_i is requesting k more instances of resource type R_j.

Topic : Detection Algorithm

1. Let Work and Finish be vectors of length m and n, respectively Initialize:
   - Work = Available
   - 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:
   - Finish[i] == false
   - $Request_i \leq Work$
   - If no such i exists, go to step 4

Topic : Detection Algorithm (Cont.)

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 T_i is deadlocked

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

Topic : Example of Detection Algorithm

Five threads T_0 through T_4; three resource types

A (7 instances), B (2 instances), and C (6 instances)

Snapshot at time T_0:

	Allocation	Requested	Available
	A B C	A B C	A B C
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

Sequence $<T_0, T_2, T_3, T_1, T_4>$ will result in Finish[i] = true for all i

O/P

Deadlocked Processes/Threads

Recovery

Topic : Example (Cont.)

- T_2 requests an additional instance of type C

	Requested
	A B C
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

- State of system?
- Can reclaim resources held by thread T_0, but insufficient resources to fulfill other processes; requests
- Deadlock exists, consisting of processes T_1, T_2, T_3, and T_4

Topic : Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked threads "caused" the deadlock.

Topic : Recovery from Deadlock: Process Termination

- Abort all deadlocked threads
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?

1. Priority of the thread
2. How long has the thread computed, and how much longer to completion
3. Resources that the thread has used
4. Resources that the thread needs to complete
5. How many threads will need to be terminated
6. Is the thread interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim –** minimize cost
- **Rollback –** return to some safe state, restart the thread for that state
- **Starvation –** same thread may always be picked as victim, include number of rollback in cost factor

→ Consider a System with n-processes & a Single Resource 'R' having '8' Instances, Each Process needs 3 (2) instances of 'R' to Complete.

a) what is the min(n) to cause deadlock?

"8"

$P_1 - 2$
$P_2 - 2$
$P_3 - 2$
$P_4 - 2$

"4"

$P_i \rightarrow 3(R)$

b) what is the Max(n) for deadlock freedom?

→ "7"

"3"

[NAT]

Y/N (1/0)

#Q. The following is a question about Dining Computer Scientists. There are 6 computer scientists seated at a circular table. There are 3 knives at the table and 3 forks. The knives and forks are placed alternately between the computer scientists. A large bowl of food is placed at the center of the table. The computer scientists are quite hungry, but require both a fork and knife to eat.

Consider the following policies for eating and indicate, if it can result in Deadlock.

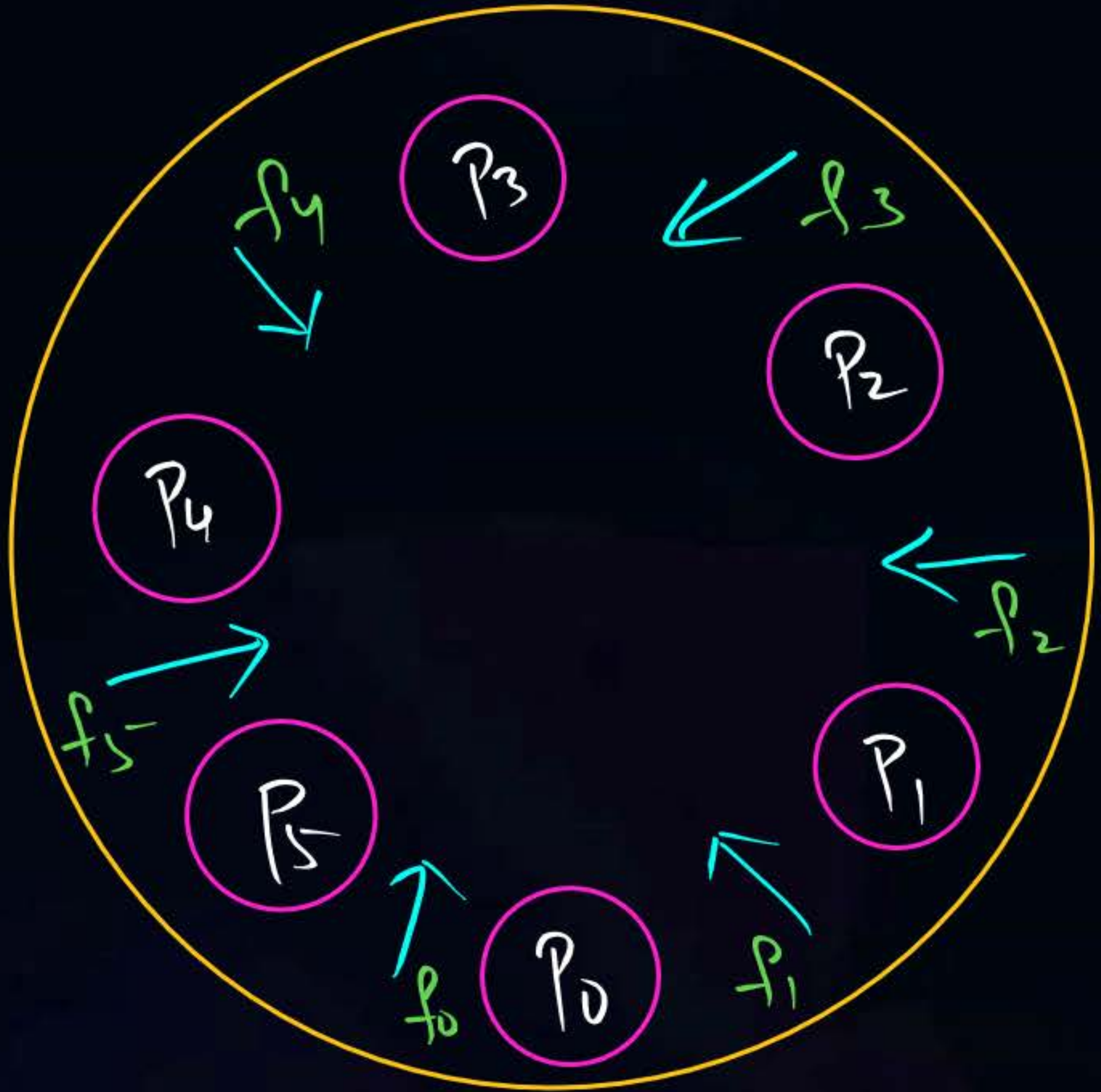
Algorithm

- Attempt to grab the fork that sits between you and your neighbor until you are successful.
- Attempt to grab the knife that sits between you and your neighbor until you are successful.
- Eat
- Return the fork
- Return the knife

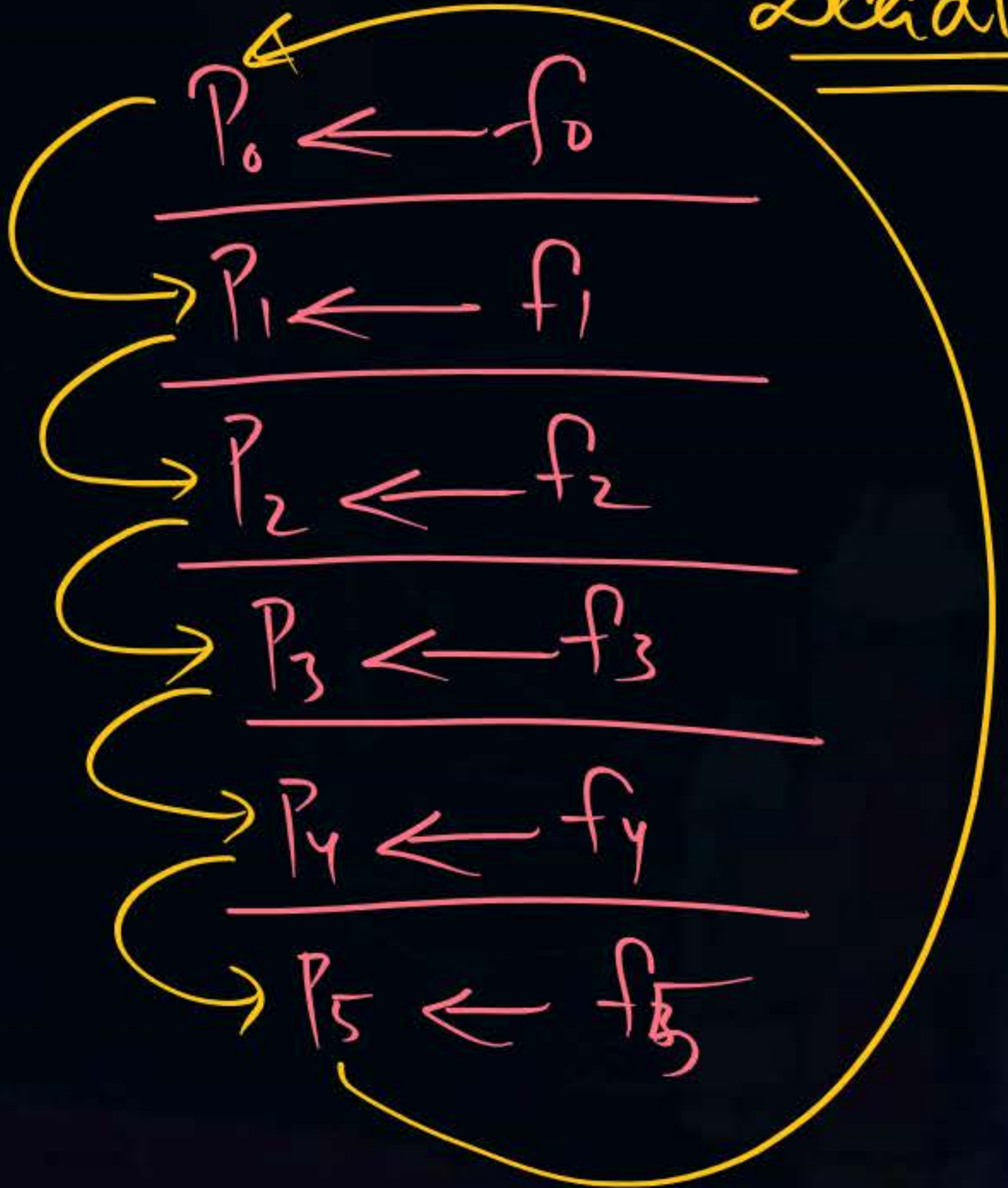
"Deadlock is NEVER Possible"

Dining Philosopher

$n = 6$; $m = 6$

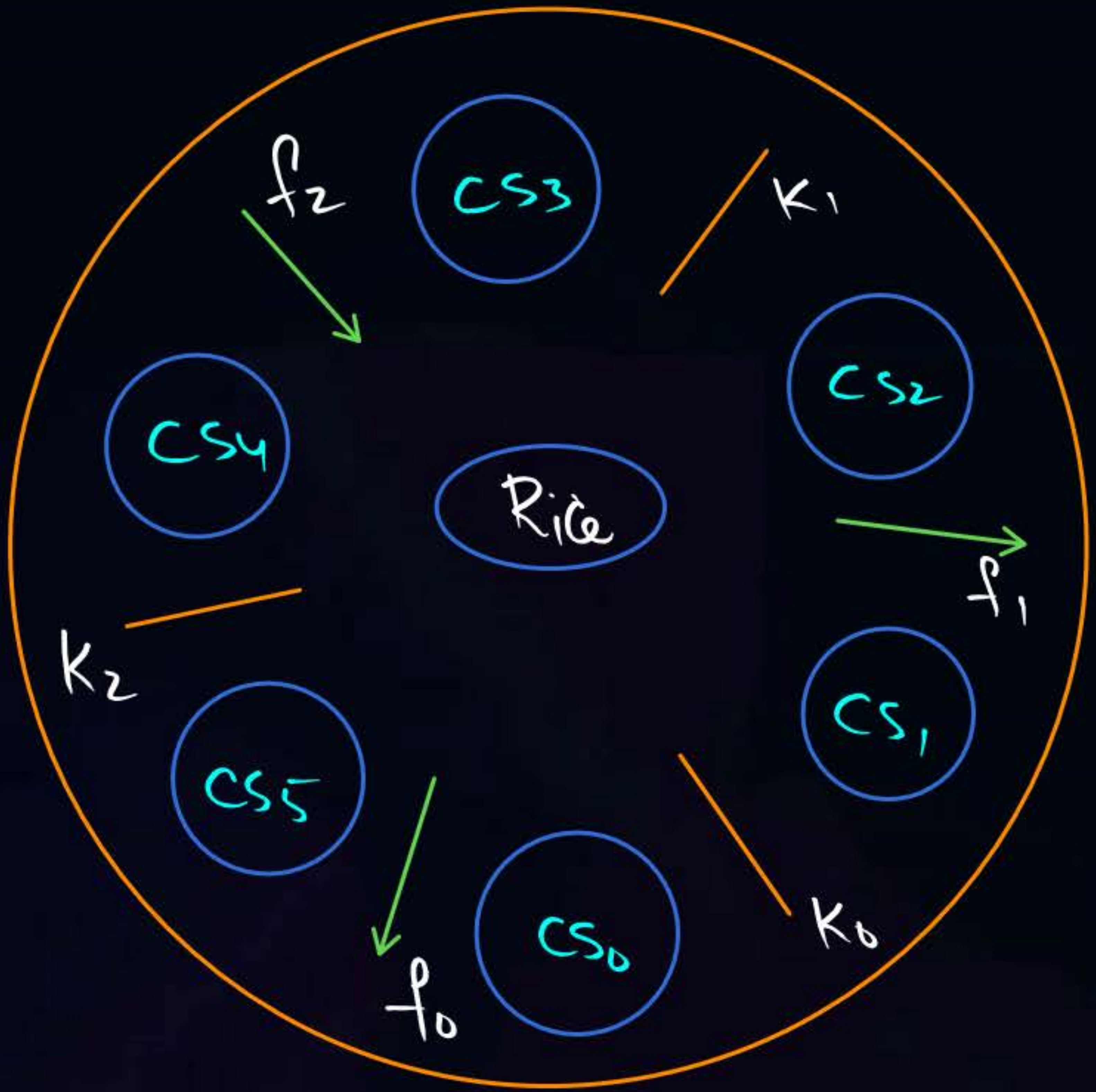


deadlock



$w(f(i))$;

$w(f((i+1)\%N))$



Algorithm – CS(i)

1) Grab-fork
2) Grab-Knife
3) <eat>
4) Rel_fork
5) Rel-Kime

$CS_0 \leftarrow f_0, K_0$

$CS_1 \leftarrow f_1$, Blked

$CS_2 \leftarrow$ Blocked

$CS_3 \leftarrow f_2, K_1$

$CS_4 \leftarrow$ Blocked

$CS_5 \leftarrow$ Blocked

2 mins Summary

Topic One → Deadlock concept

Topic Two → Characterization

Topic Three → Strategies

Topic Four → Prevention, Avoidance, Detection & Recovery

Topic Five → Problem Solving

THANK - YOU