Deadlock

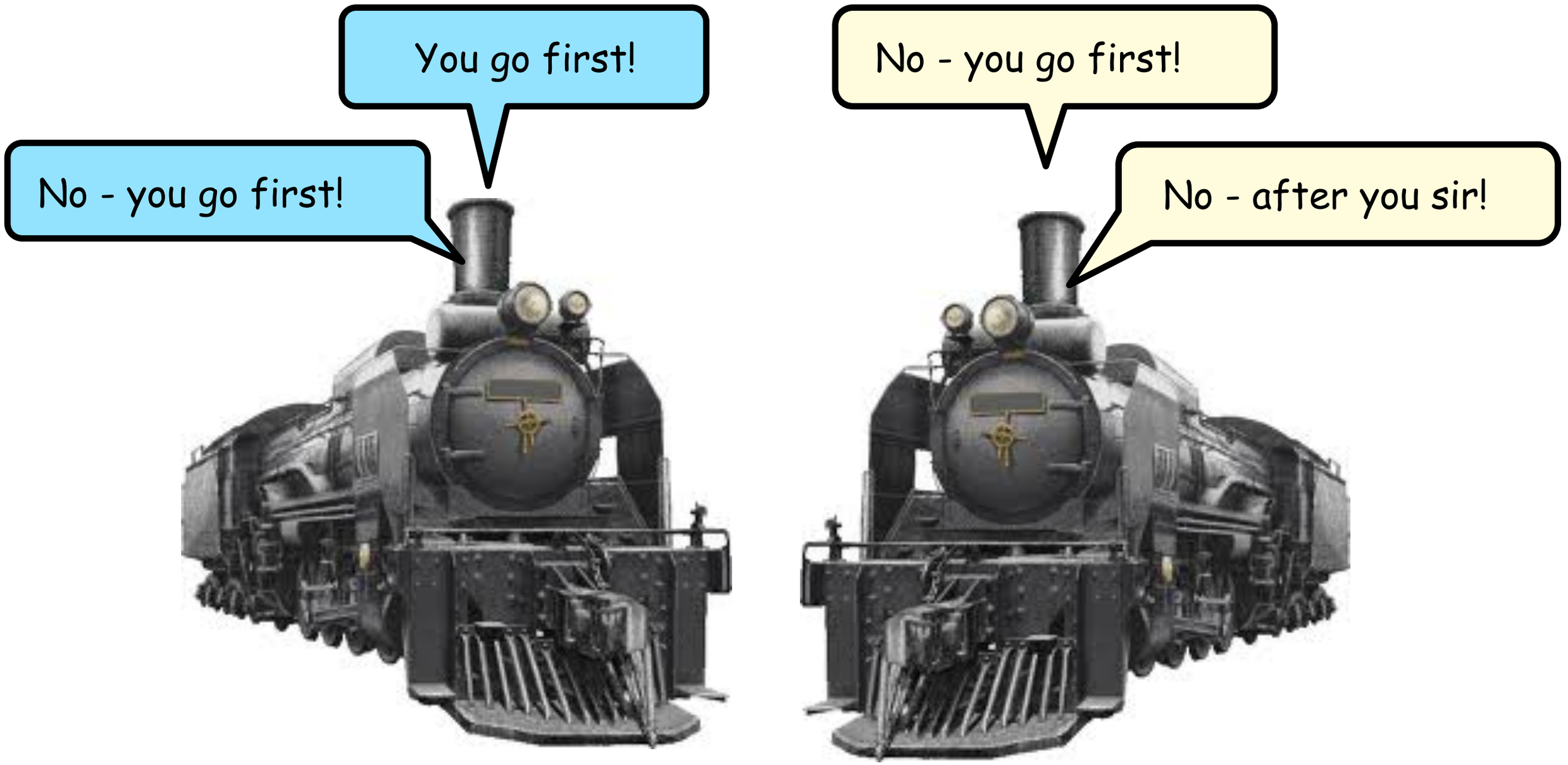
Module 4

Lecture

Operating systems 2019

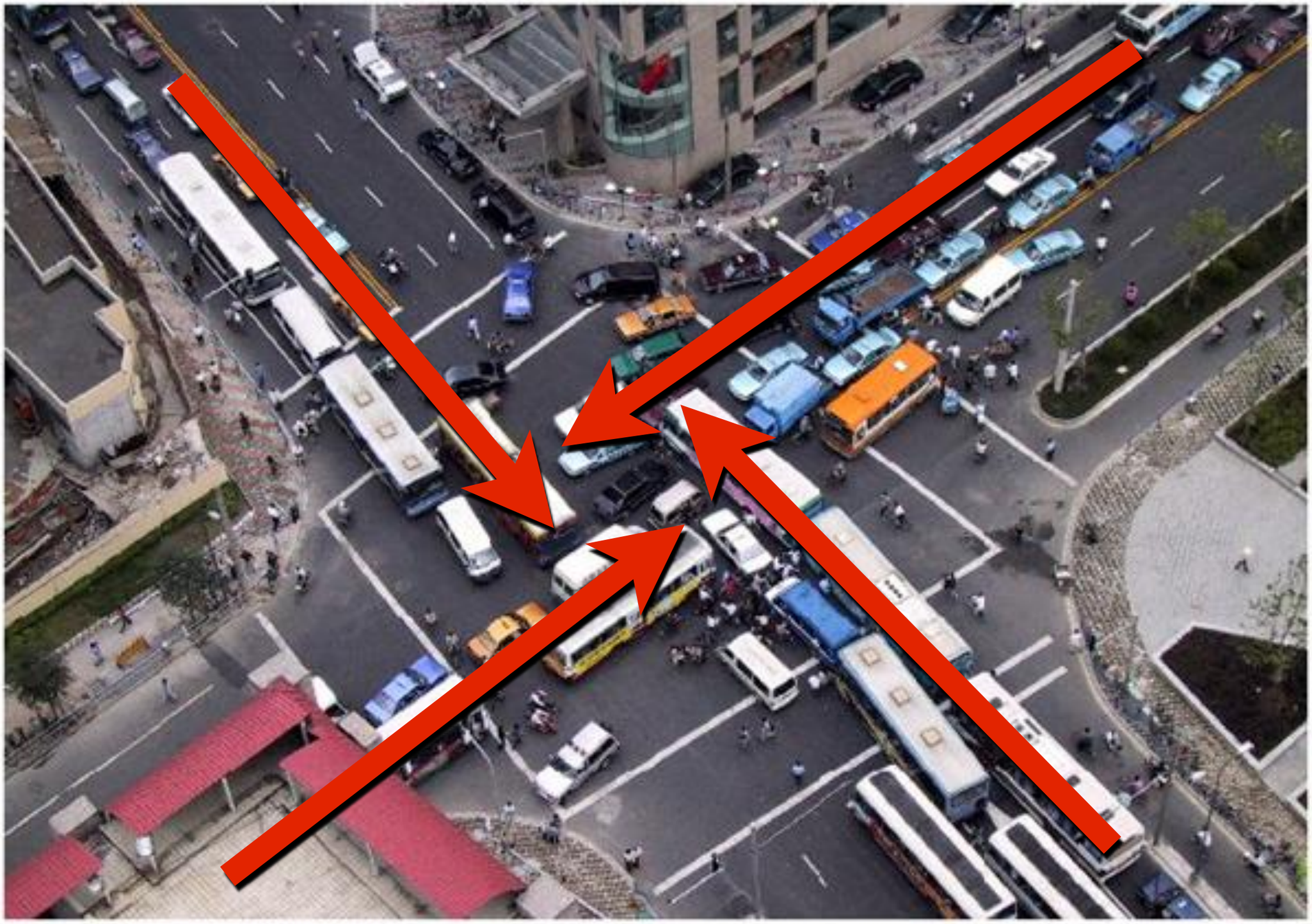
1DT044, 1DT096 and 1DT003

February 2019 karl.marklund@it.uu.se Uppsala university

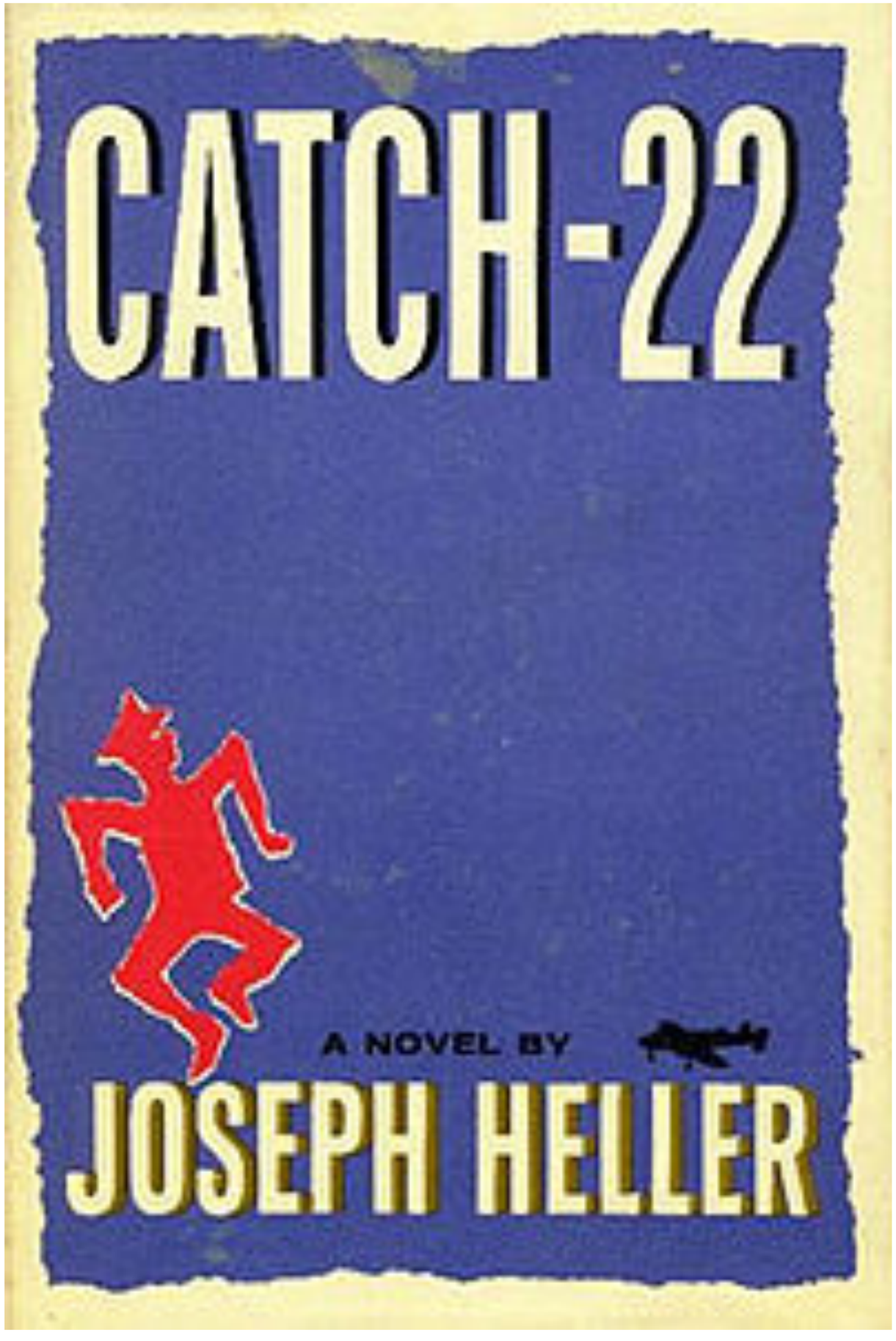


When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

An illogical statute passed by the Kansas legislature in early 20th century.



Mutual cooperation among drivers would give the maximum benefit (prevention of **gridlock**), but this may not happen because of the desire to maximize one's own benefit (shortest travel time) given the uncertainty about the other drivers' commitment to cooperation.



Catch-22 is a satirical, historical novel by the American author Joseph Heller, first published in 1961. The novel, set during the later stages of World War II from 1944 onwards, is frequently cited as one of the great literary works of the twentieth century.

There was only one catch and that was Catch-22, which specified that a concern for one's safety in the face of dangers that were real and immediate was the process of a rational mind. Orr was crazy and could be grounded. All he had to do was ask; and as soon as he did, he would no longer be crazy and would have to fly more missions. Orr would be crazy to fly more missions and sane if he didn't, but if he were sane he had to fly them. If he flew them he was crazy and didn't have to; but if he didn't want to he was sane and had to. Yossarian was moved very deeply by the absolute simplicity of this clause of Catch-22 and let out a respectful whistle.

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, such as sending a message or more commonly releasing a lock.

Source: Coulouris, George (2012). Distributed Systems Concepts and Design. Pearson. p. 716.

Process 0

- Holds Disk 0
- Needs Disk 1

User Space

Process 1

- Holds Disk 1
- Needs Disk 0

Kernel Space

The OS Controls the hardware and coordinates its use among the various application programs for the various user.

Computer Hardware

Disk 0

Disk 1

Process 0

- Holds Disk 0
- Needs Disk 1

User Space

Process 1

- Holds Disk 1
- Needs Disk 0

holds

needs

needs

holds

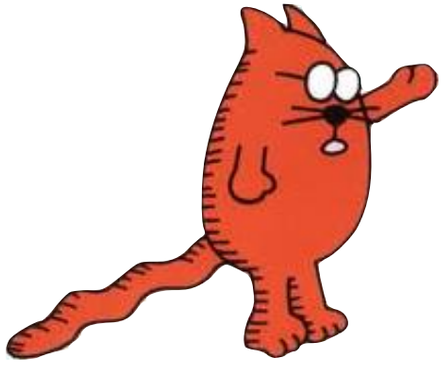
Kernel Space

The OS Controls th hardware and coordinates its u e various application progra for the various user.

Computer Hardware

Disk 0

Disk 1



Deadlock

P1 and P2 each hold one disk drive and both needs the one held by the other.

Two tasks **`T0`** and **`T1`**, semaphores A and B, both initialized to **`1`**.

`T0` waits for **`T1`** to release **`B`**.

`A = Semaphore(1)` `B = Semaphore(1)`		
`T0`	**`Time`**	**`T1`**
`wait(A)`	↓	`wait(B)`
`wait(B)`		`wait(A)`

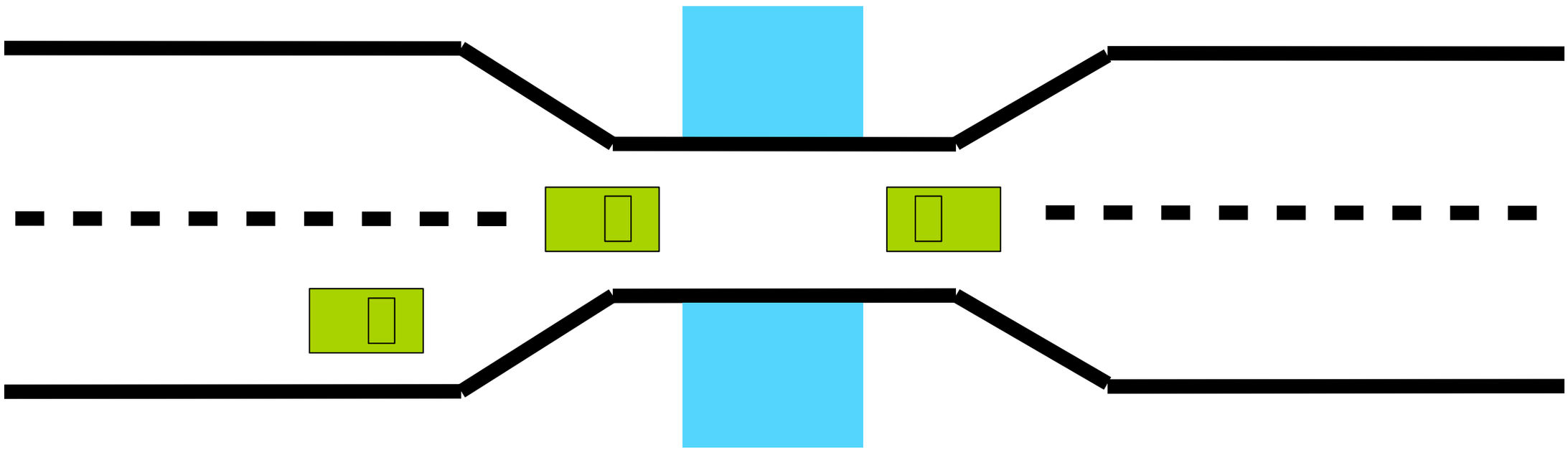
At the same time as **`T1`** waits for **`T0`** to release **`A`**.

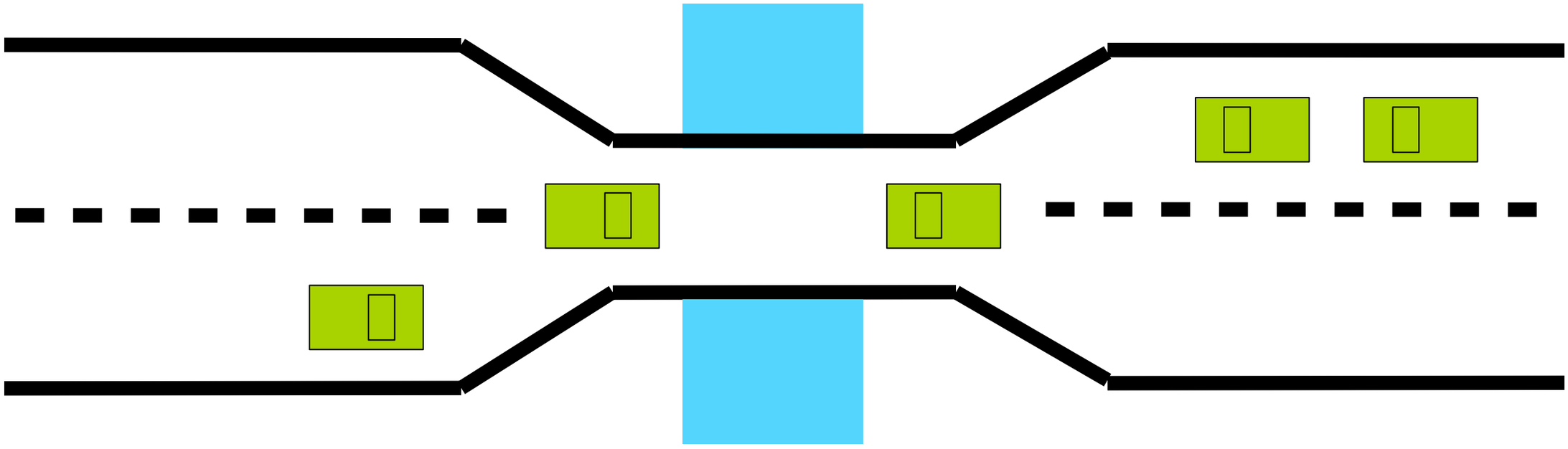
But neither of the processes will ever release the locks - they are blocked waiting for each other. The system has reached a **deadlock**.

Deadlock

When two or more tasks are waiting indefinitely for an event that can be caused by only one of the waiting tasks this is called deadlock.

Narrow bridge





- ★ On the bridge, traffic only in one direction at a time (**mutual exclusion**).
- ★ Each section of a bridge can be viewed as a **resource.**
- ★ If a **deadlock** occurs, it can be resolved if one car backs up (**preempt** resources and **rollback**).
- ★ Several cars may have to be backed up if a deadlock occurs.
- ★ **Starvation** is possible.

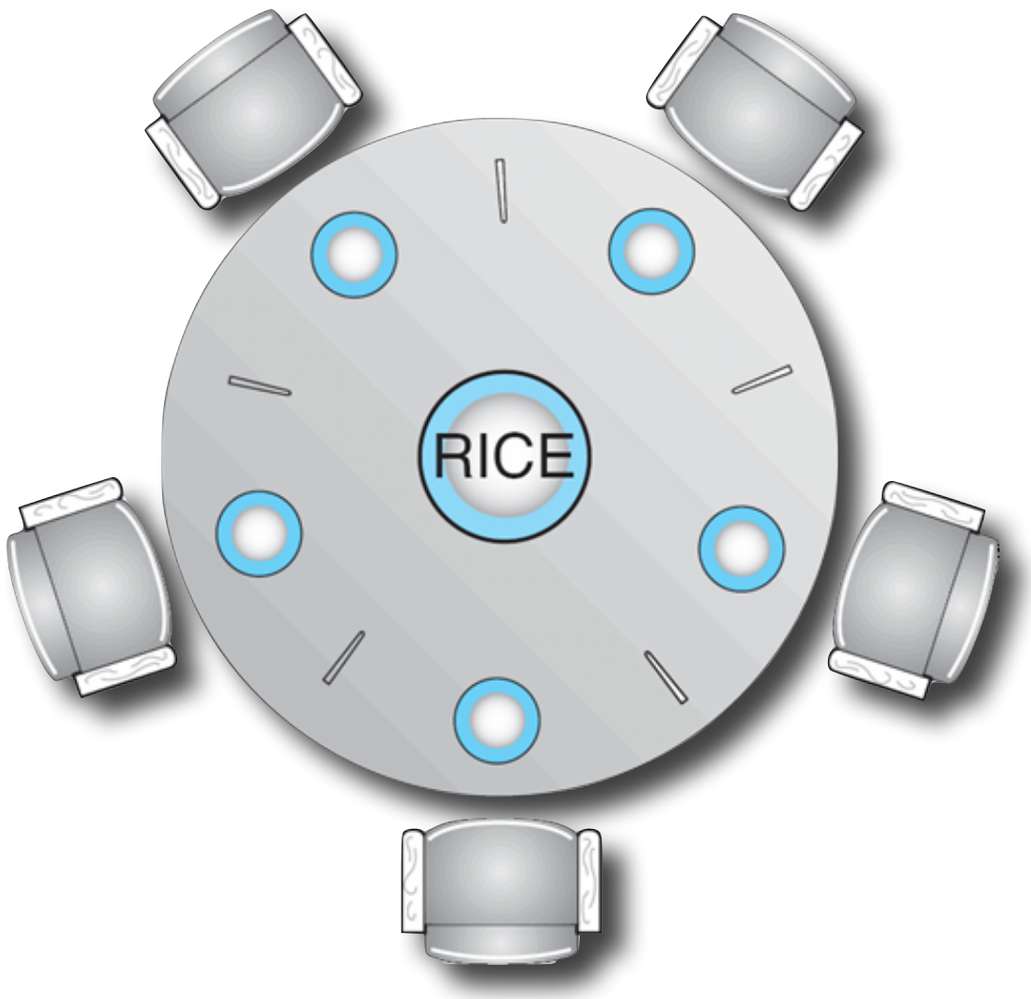
Dining philosophers

An **example problem** often used
in concurrent algorithm design to
illustrate synchronization issues and techniques
for resolving them.

Source: https://en.wikipedia.org/wiki/Dining_philosophers_problem 2018-02-12

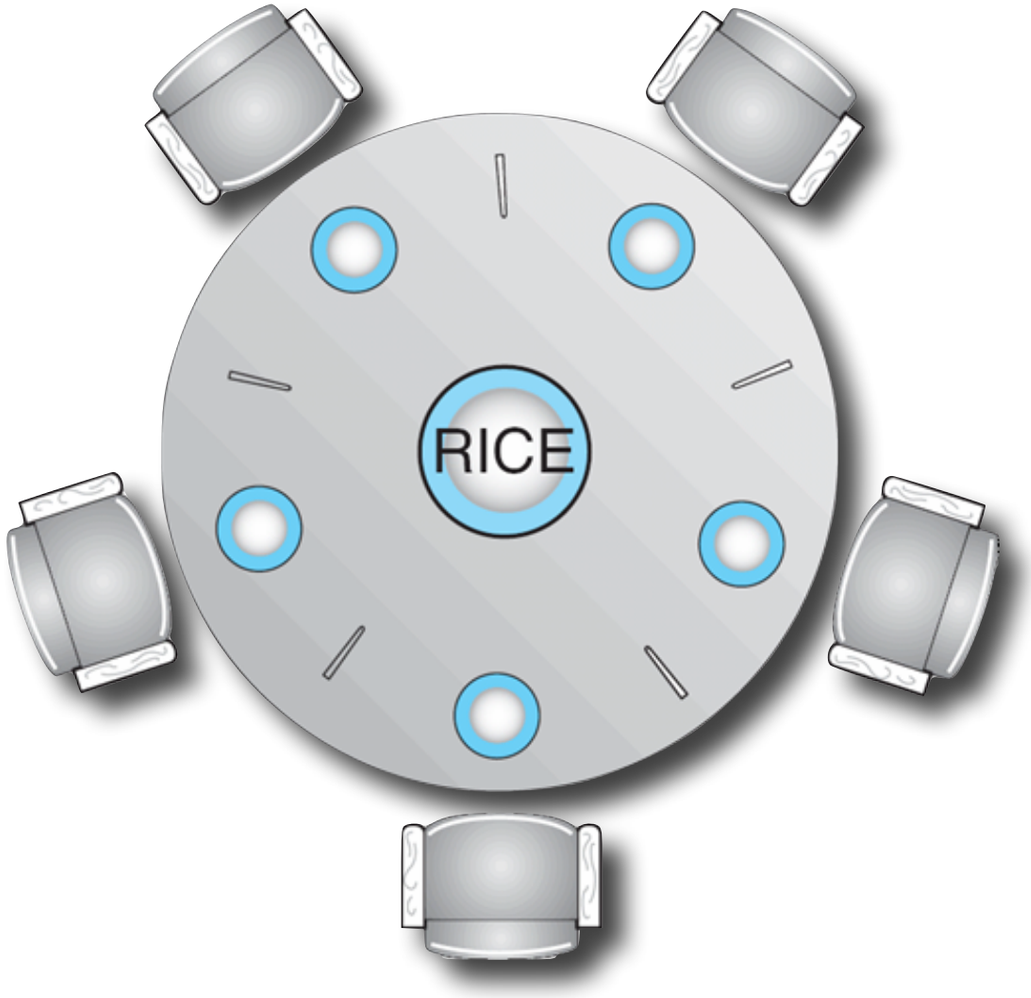
Dining philosophers (1)

Five silent philosophers sit at a round table around a bowl of rice.



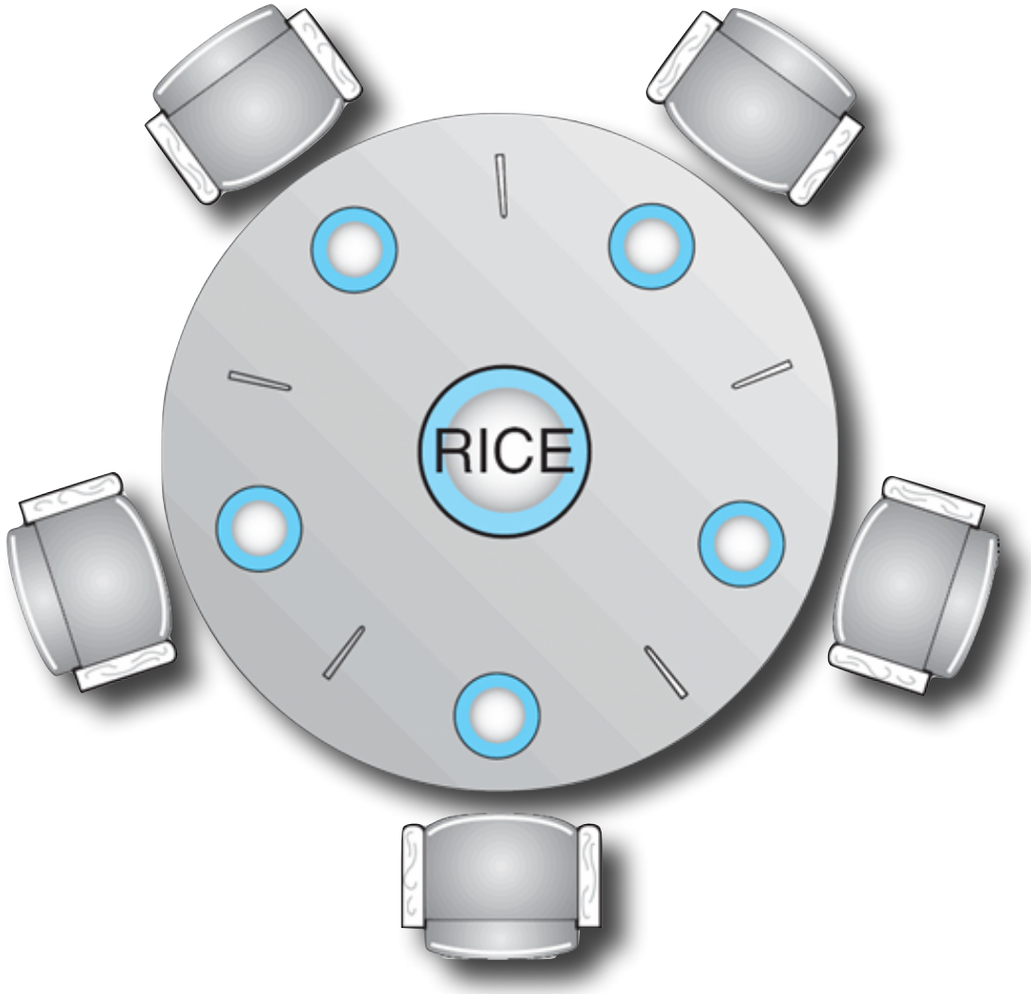
Dining philosophers (2)

A chopstick is placed between each pair of adjacent philosophers.



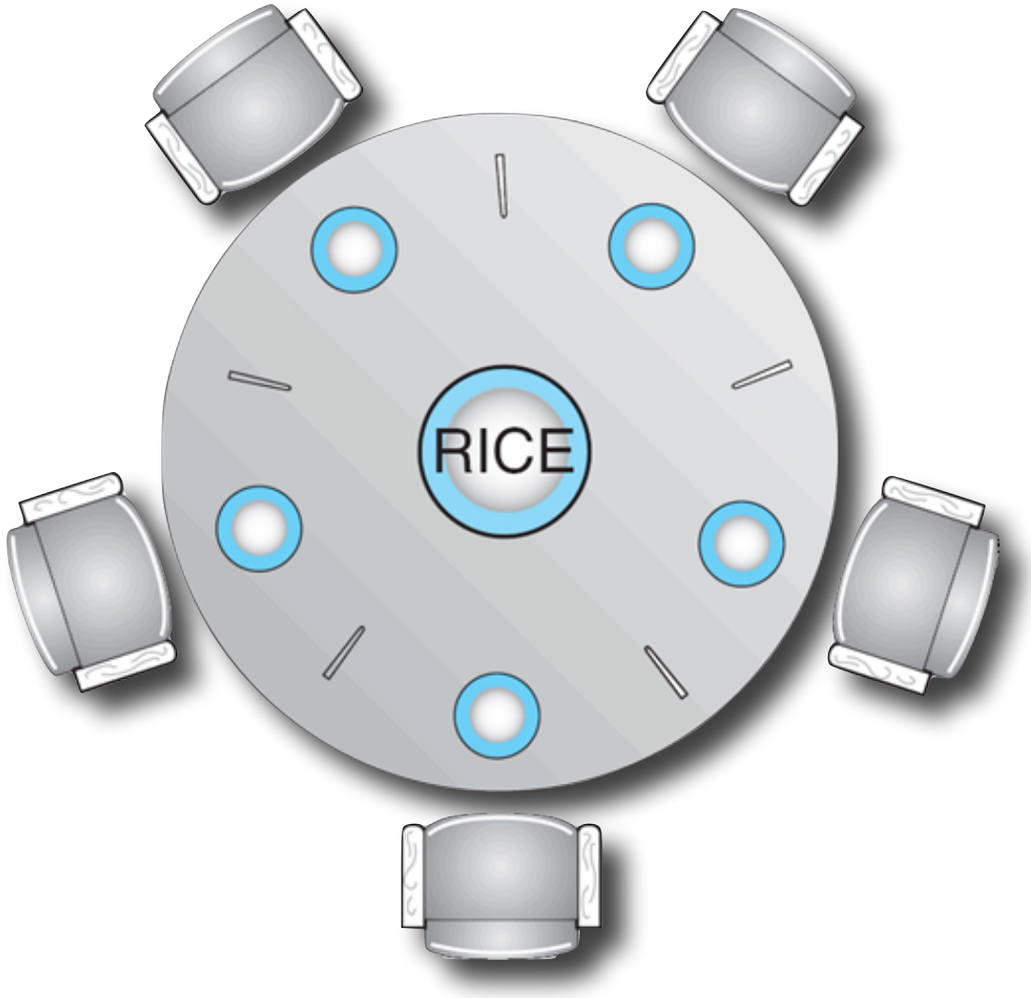
Dining philosophers (3)

Each philosopher must **alternately** **eat** and **think**.
However, a philosopher can only eat rice when he has both left and right chopsticks.



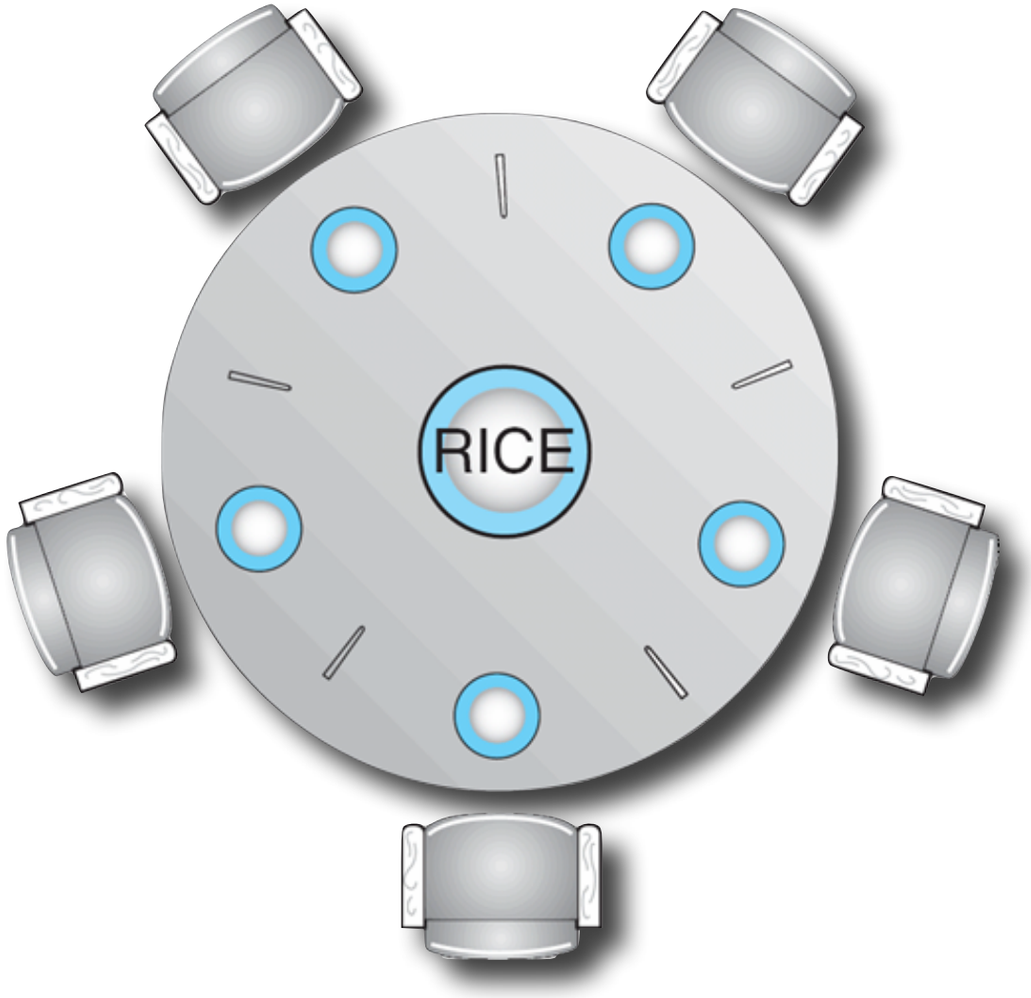
Dining philosophers (4)

Each chopstick can be held by only one philosopher and so a philosopher can use the chopstick only if it's not being used by another philosopher.



Dining philosophers (5)

After a philosopher finishes eating, he needs to put down both chopsticks so they become available to others. A philosopher can grab the chopstick on his right or the one on his left as they become available, but can't start eating before getting both of them.



Simulation of the dining philosophers

- Each **philosopher** is simulated by a **thread**.
- A philosopher first **waits** for the **right chopstick** to become available and then for the **left chopstick** to become available.
- After obtaining both the right and left chopstick the philosopher is able to **eat**.
- After eating a philosopher first **puts down** the **right chopstick** and then puts down the **left chopstick**.
- After putting down both chopsticks, the philosopher sits in silence and **thinks** for a while.

Shared resource

The philosophers all access the shared bowl of rice.

```
int rice = 127;

void eat(int *rice) {
  (*rice)--;
}
```

Mutual exclusion

A chopstick can only be held by one philosopher at the time.

Updating the amount of rice must be atomic.

Synchronization

- A philosopher must **wait** for the chopstick on his right and left to become available.
- Each **chopstick** can be represented by a **semaphore** initialised to 1.
- Keep all **chopstick semaphores** in an **array**.
- Use a single semaphore **mutex** initialised to 1 to protect the **critical section** when updating the amount of rice.

```c
#define N 5

sem_t mutex;
sem_t chopstick[N];

mutex = sem_init(1);

for (int i = 0; i < N; i++) {
  chopstick[i] = sem_init(1);
}
```

Mutual exclusion

Updating the amount of rice must be atomic.

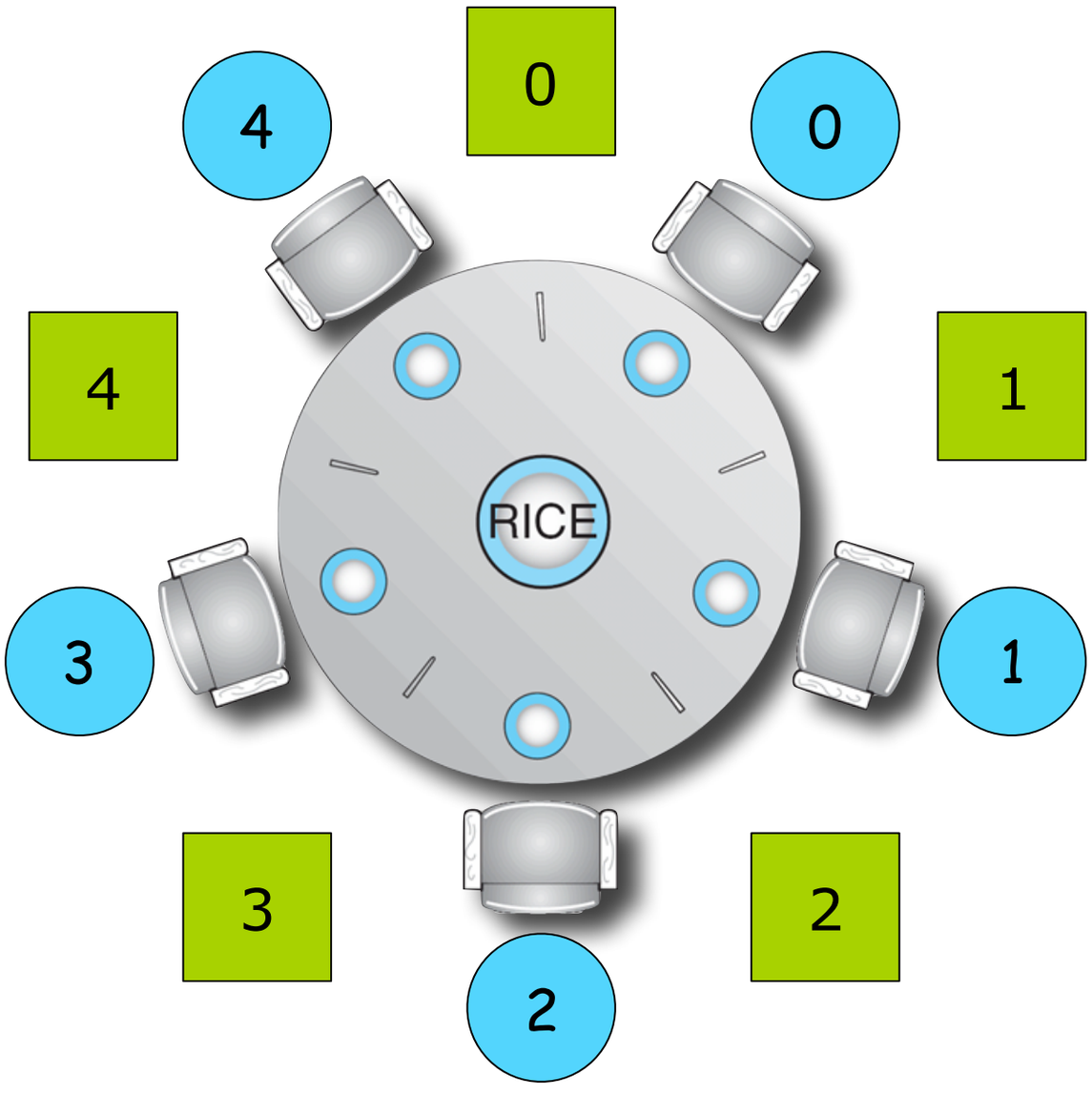
```
void eat(int *rice, sem_t mutex) {
    sem_wait(mutex);
    (*rice)--;
    sem_signal(mutex);
}
```

Graphical notation

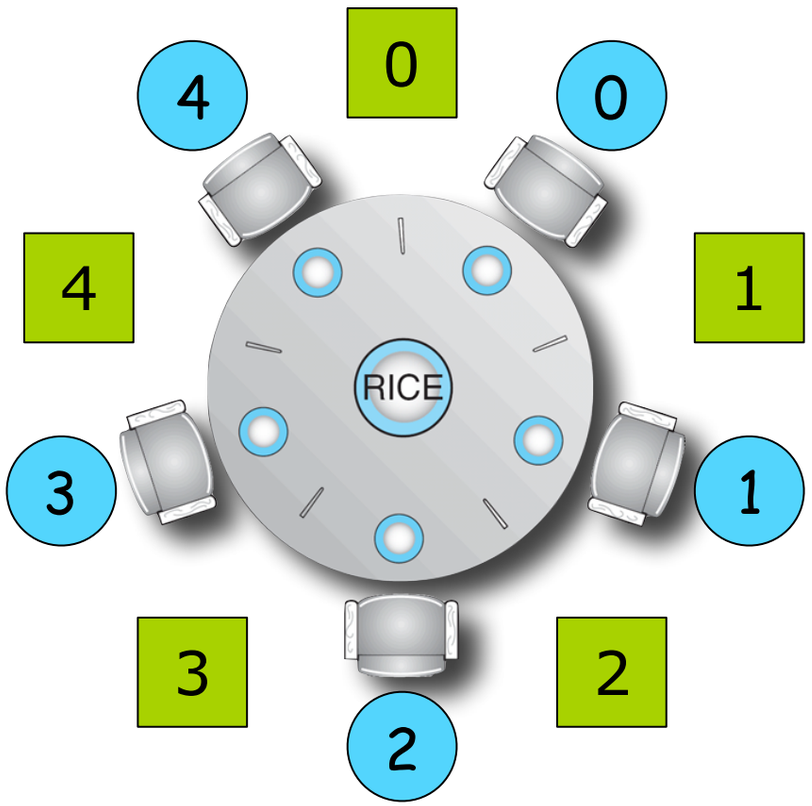
A philosopher is represented by a circle and a chopstick by a square.

Philosopher

Chopstick



Left and right

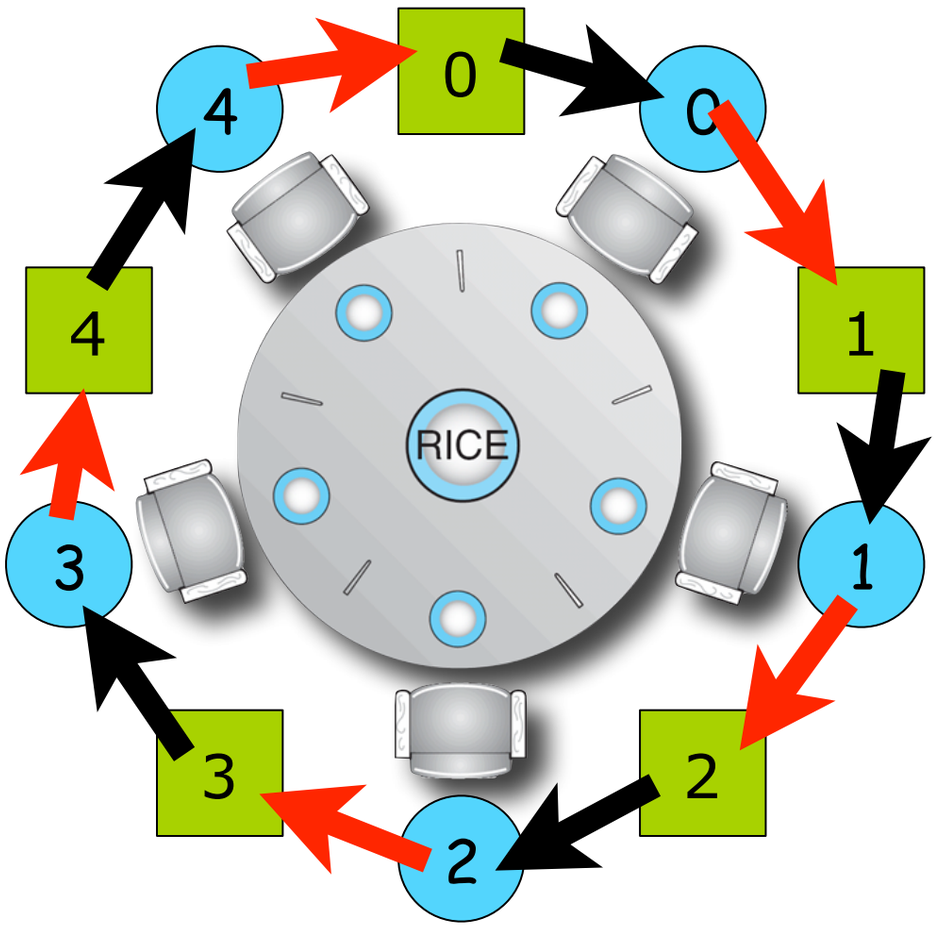



```
int right(int i) {
    return i;
}
```

```
int  left(int i) {
    return (i + 1) % N;
}
```

Philosopher

```
void philosopher(int i, int *rice, sem_t mutex,
                        sem_t chopstick[]) {
  do {
    sem_wait  (chopstick[right(i)]);
    sem_wait  (chopstick[ left(i)]);
    eat(rice, mutex);
    sem_signal(chopstick[right(i)]);
    sem_signal(chopstick[ left(i)]);
    think();
  } while (true);
}
```



If all philosophers holds their right chopstick ...

... and all waits for their left chopstick to become available

Deadlock due to **circular wait**

System model

Lets introduce some notation to make it possible to reason about resource management.

Resource types: R_1, R_2, . . ., R_m

- CPU cycles
- Memory space
- I/O devices
- Semaphores

Each resource type R_i has W_i **instances**.

Each process utilizes a resource as follows:

- **Request**
- **Use**
- **Release**

Deadlock characterization

Deadlock can arise **iff** these **four conditions** **holds simultaneously**.

Mutual exclusion: only one task at a time can use a resource instance.

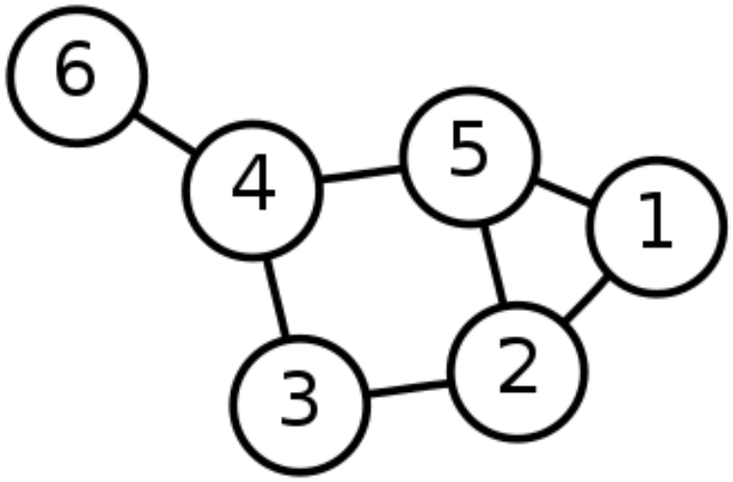
Hold and wait: a task holding at least one resource is waiting to acquire additional resources held by other tasks.

No preemption: a resource can be released only voluntarily by the task holding it, after that task has completed its task.

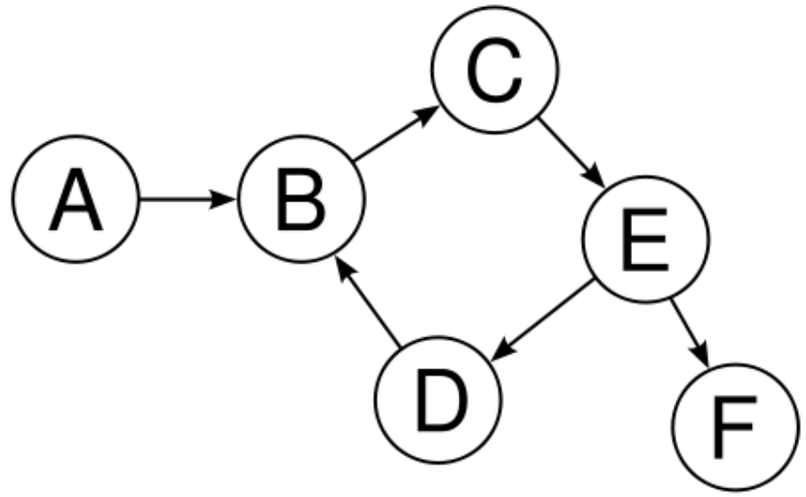
Circular wait: there exists a set $\{T_0, T_1, \ldots, T_0\}$ of waiting tasks such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by $T_2, \ldots, T_{n-1}$ is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

Graph theory

Graphs are mathematical structures used to **model** **pairwise relations** between objects.



An undirected graph



A directed graph

A graph in this context is made up of vertices, **nodes**, or points which are connected by **edges**, arcs, or lines.

A graph may be **undirected**, meaning that there is no distinction between the two vertices associated with each node, or its edges may be **directed** from one node to another.

Source:	https://en.wikipedia.org/wiki/Graph_theory	2018-02-12
Image:	https://en.wikipedia.org/wiki/Graph_theory#/media/File:6n-graf.svg	2019-03-02
Image:	https://stackoverflow.com/a/40930741/3834375	2019-03-02

Resource allocation graph

We can use a graph to study a system of tasks allocating a number of resources.

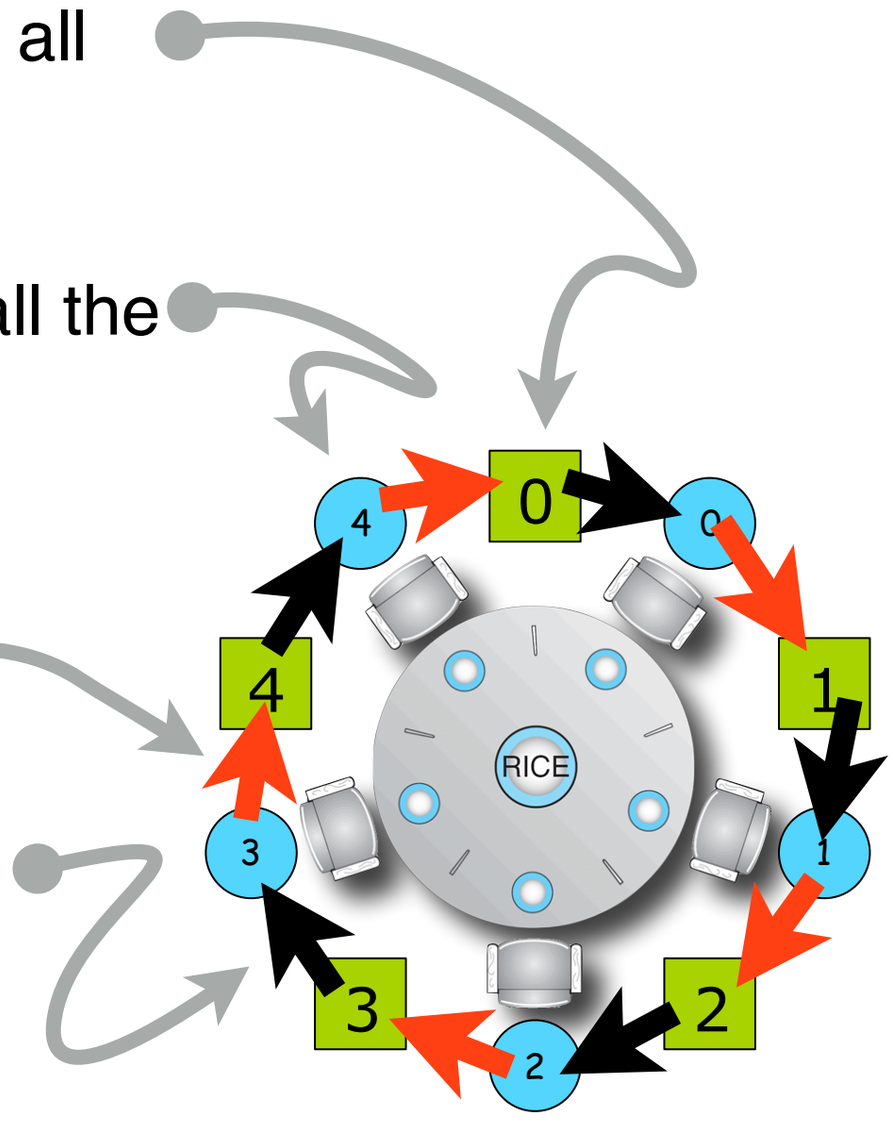
A set of **nodes** N and a set of **edges** E.

★ N is partitioned into two types:

- **R** = $\{R_1, R_2, \ldots, R_m\}$, the set consisting of all **resource types** in the system.
- **T** = $\{T_1, T_2, \ldots, T_n\}$, the set consisting of all the **tasks** (processes/threads) in the system.

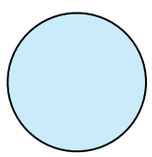
★ **request** edge – directed edge $T_i \rightarrow T_j$

★ **assignment** edge – directed edge $T_j \rightarrow T_i$

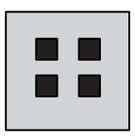


Graphical notation

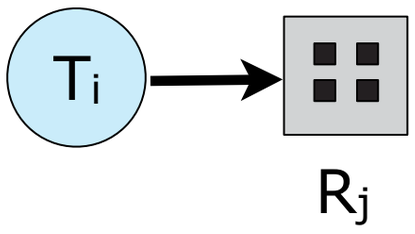
In the resource allocation graph (RAG), the following graphical notation is used.



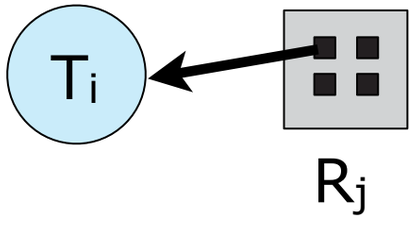
A task (process/thread).



A resource with four indistinguishable instances, for example four printers.

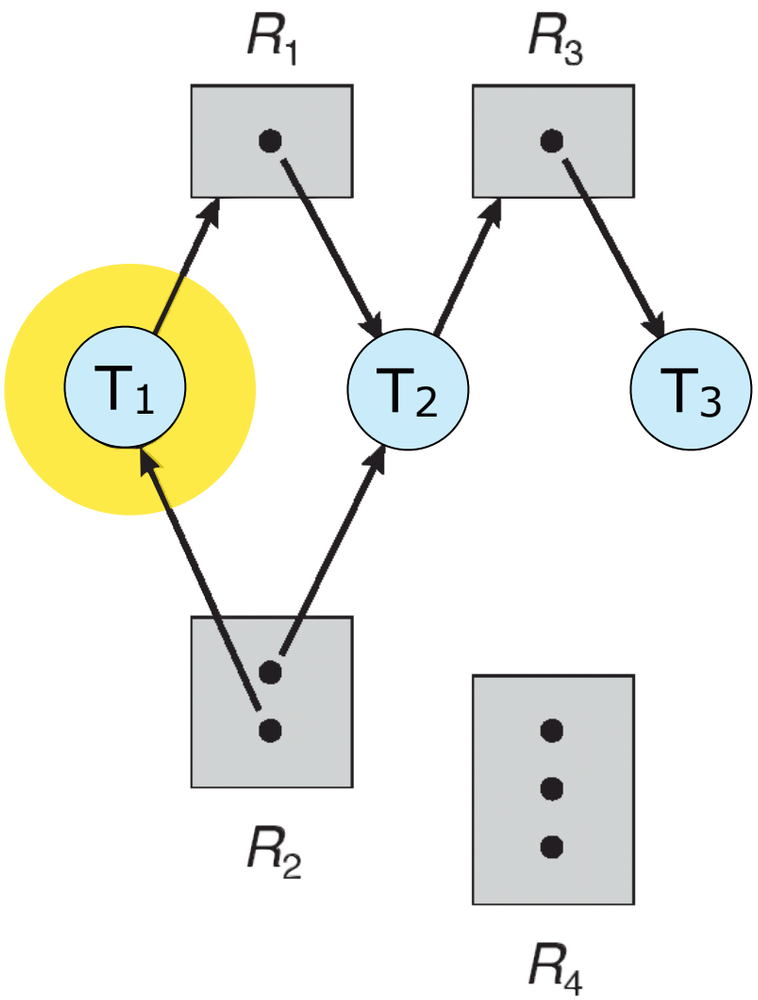


Task T_i requests one of the four instances of resource R_j, for example one of the four printers.



Task T_i is holding one of the four instances of resource R_j, for example one of the four printers.

Example of a resource allocation graph (RAG)



T_1

Holds one instance of the **R_2** resource. **Waits** for the **R_1** resource.

T_2

Holds the **R_1** resource and one instance of the **R_2** resource. **Waits** for the **R_3** resource.

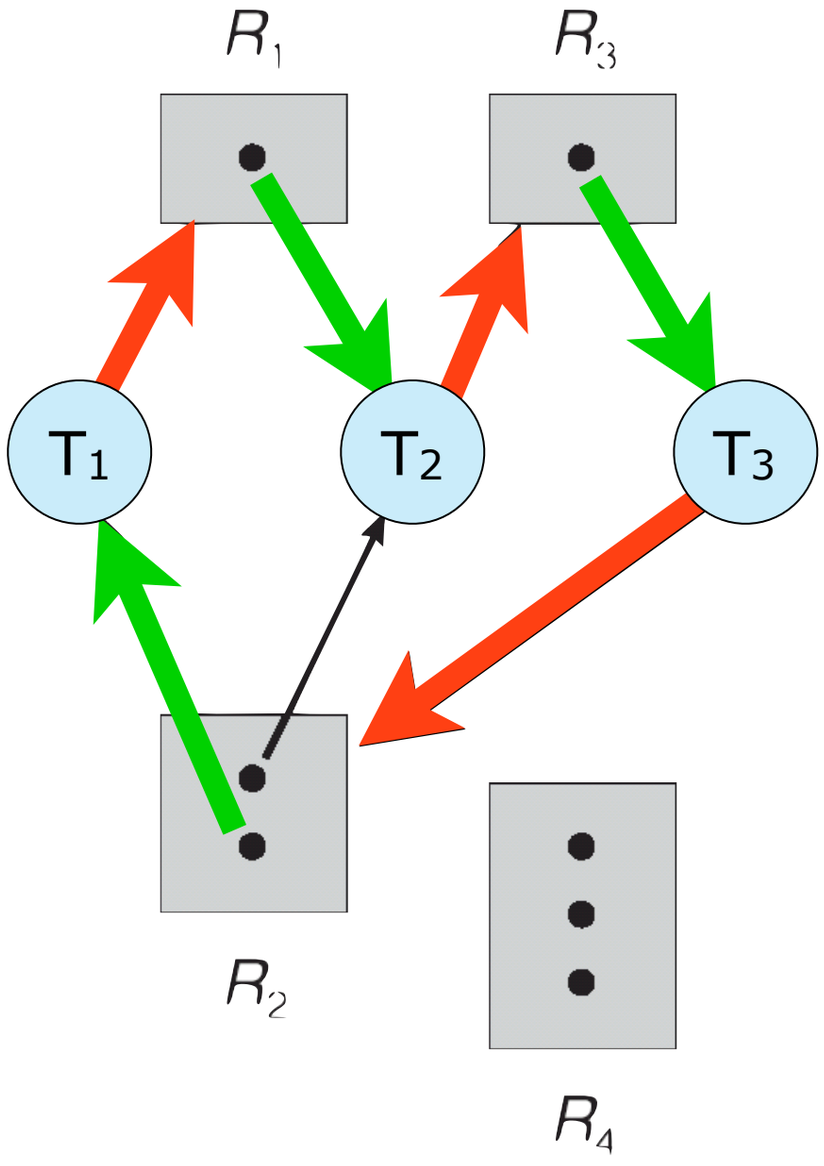
T_3

Holds the **R_3** resource.

Deadlock? **No deadlock!**

When P_3 releases the R_3 resource, T_2 will obtain R_3. Eventually T_2 will release R_1 and T_1 will be able to get access to R_1.

Resource allocation graph (RAG) with a deadlock



T_1

Holds one instance of the R_2 resource.

Waits for the R_1 resource held by T_2.

T_2

Holds the R_1 resource and one instance of the R_2 resource.

Waits for the R_3 resource held by T_3.

T_3

Holds the R_3 resource.

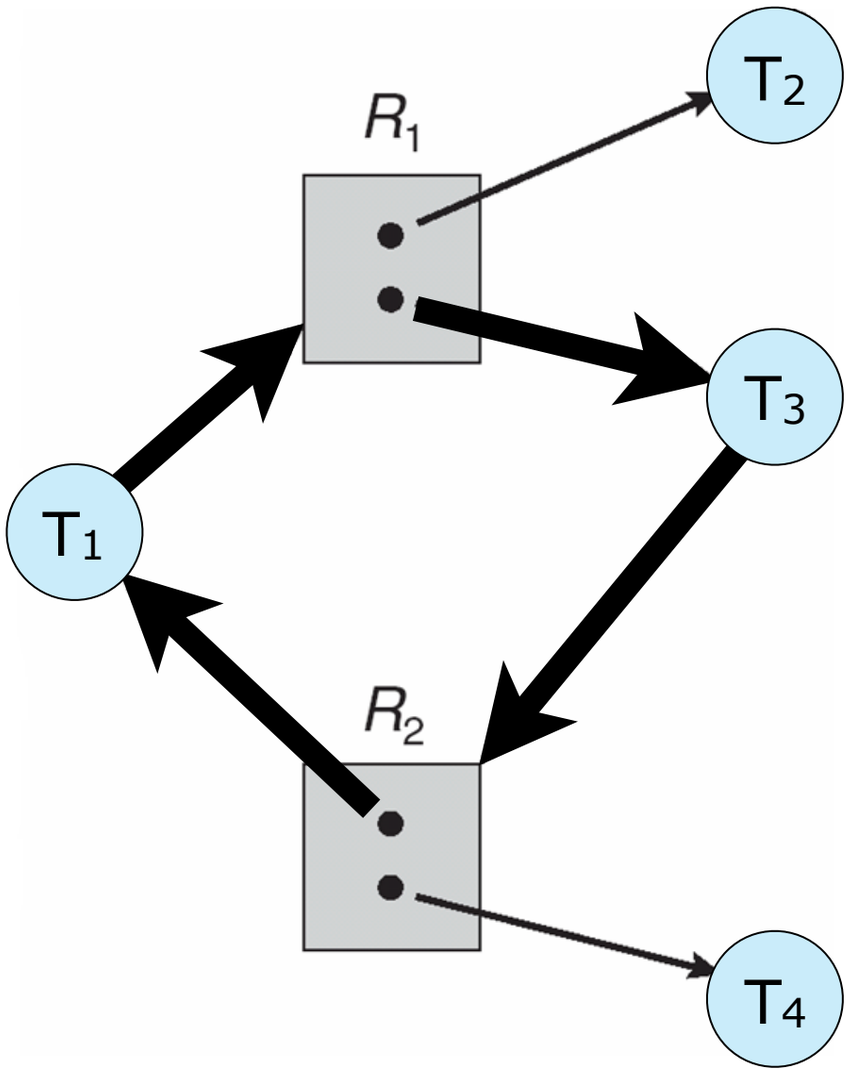
Waits for one instance of the two R_2 resources (one of which is held by T_1 and one which is held by T_2).

Deadlock?

Deadlock!

All processes are waiting for a resource held by another process also waiting for a resource ...

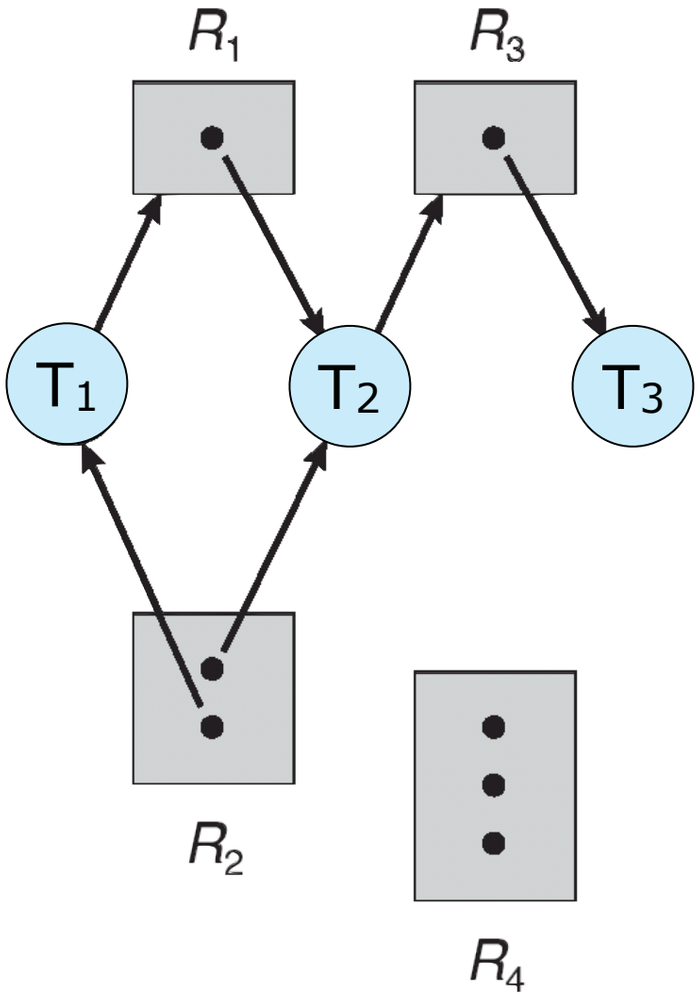
RAG with a cycle ... but no deadlock

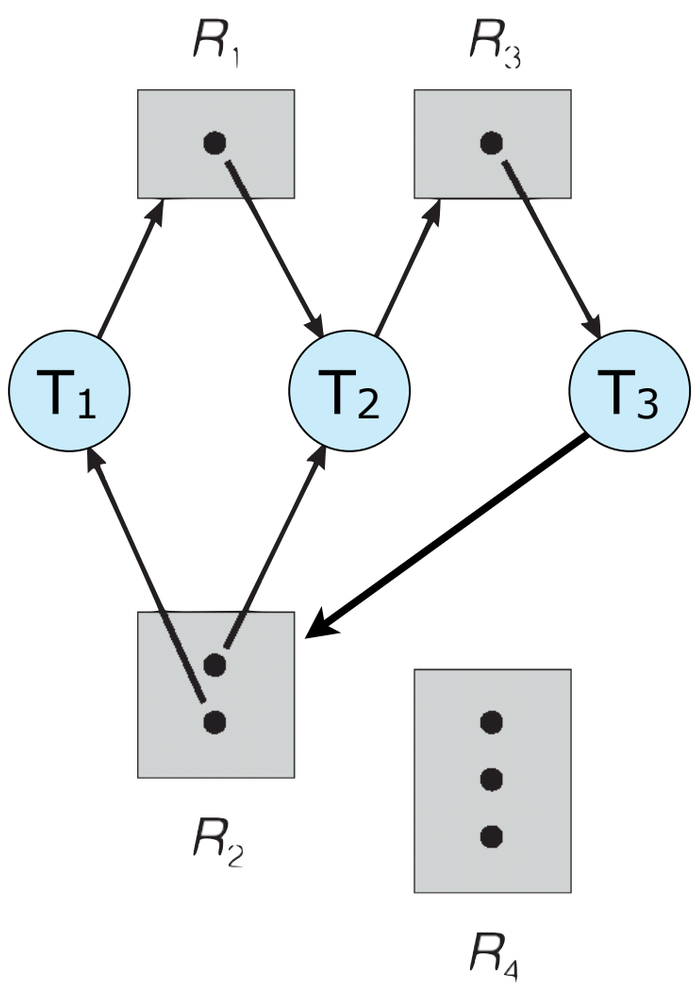


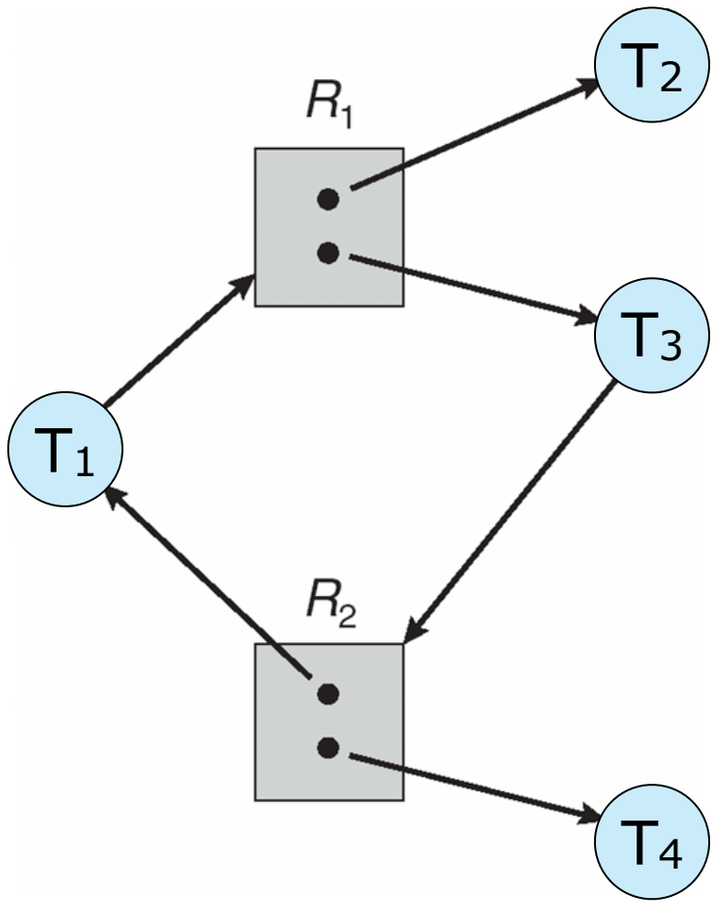
Deadlock? No deadlock!

There are several instances of the resources R_1 and R_2. Once T_2 releases R_1, process T_1 will be unblocked. Once T_4 releases R_2, process T_3 will be unblocked.

Resource allocation graph (RAG) facts

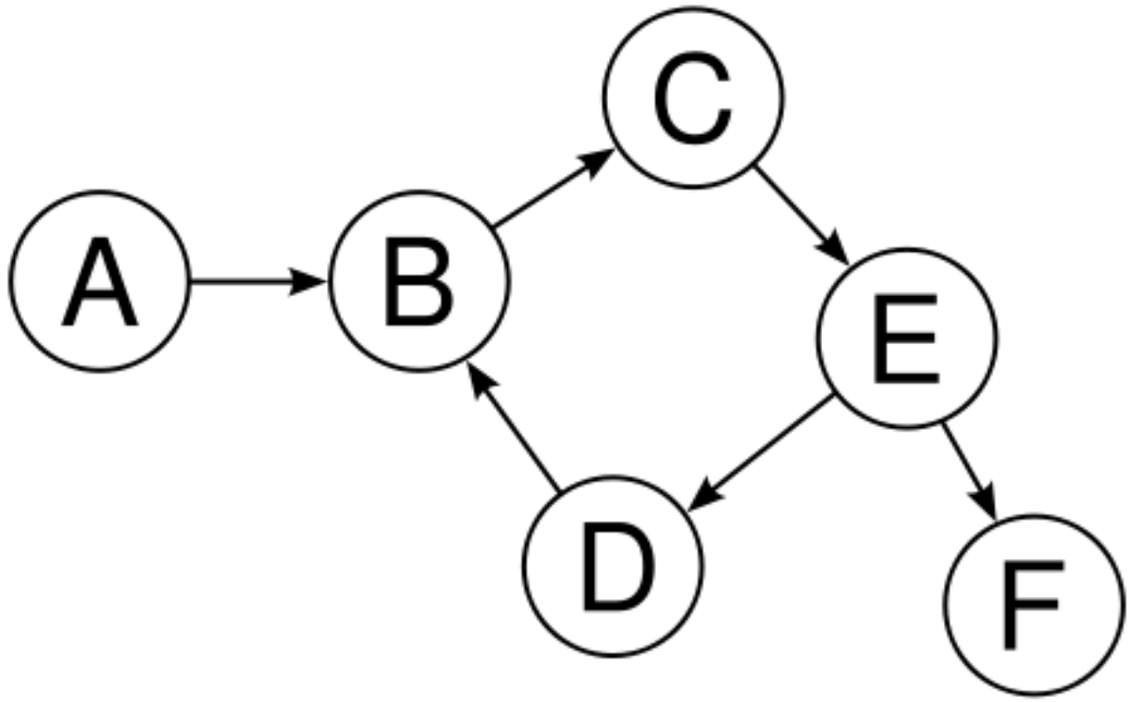






- If RAG contains **no cycles** ⇒ **no deadlock**
- If RAG contains **a cycle**:
  - if only **one instance** per resource type ⇒ **deadlock**
  - if **several instances** per resource type ⇒ **possibility of deadlock**

Cycle detection in a directed graph



For a directed graph with **V** vertices and **E** edges, using depth first traversal to detect cycles can be done with time complexity **O(V + E)**.

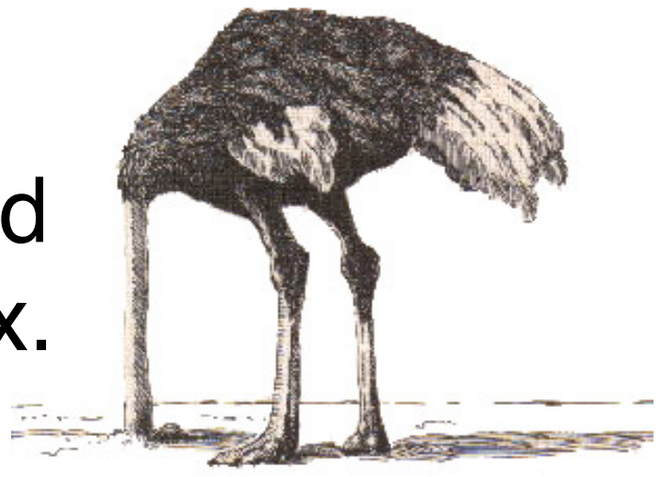
Image:	https://stackoverflow.com/a/40930741/3834375	2019-03-02
Source:	https://www.geeksforgeeks.org/detect-cycle-in-a-graph/	2019-03-02

Methods for handling deadlocks

- Use some protocol to **prevent** or **avoid** deadlocks to ensure that the system will never enter a deadlock state.

- **Allow** the system to enter a deadlock state, **detect** it, and then **recover.**

- **Ignore** the problem and pretend that deadlocks never occur in the system; used by most operating systems, including Unix.



The Ostrich algorithm

Deadlock prevention

Preventing deadlocks by constraining how requests for resources can be made in the system and how they are handled (**system design**).

The **goal** is to ensure that at least **one** of the four necessary **conditions** for deadlock can **never hold**.

Deadlock

avoidance

The system **dynamically** considers every request and **decides** whether it is **safe** to grant it at this point,

Requires additional **apriori information** regarding the overall potential use of each resource for each task.

Allows for more concurrency.

Deadlock prevention vs deadlock avoidance

The difference between deadlock prevention and deadlock avoidance is similar to the difference between a traffic light and a police officer directing traffic.

Prevention

Follow static rules strictly to stay out of trouble.

Avoidance

Dynamic and more adaptive to the actual state of the whole system.

Conditions for deadlock

Mutual exclusion: only one task at a time can use a resource instance.

Hold and wait: a task holding at least one resource is waiting to acquire additional resources held by other tasks.

No preemption: a resource can be released only voluntarily by the task holding it, after that task has completed its task.

Circular wait: there exists a set $\{T_0, T_1, \ldots, T_0\}$ of waiting tasks 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 $T_2, \ldots, T_{n-1}$ 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.

Deadlock prevention

Restrain the ways requests can be made:

- **Mutual exclusion** – not required for sharable resource instances;
  - Must hold for non-sharable resource instances.
- **Hold and wait** – must guarantee that whenever a task requests a resource, it does not hold any other resources:
  - Require processes to request and be allocated all its resources before it begins execution, or allow a process to request resources only when the process has none.
  - Low resource utilization; **starvation** possible.

Conditions for deadlock

Mutual exclusion: only one task at a time can use a resource instance.

Hold and wait: a task holding at least one resource is waiting to acquire additional resources held by other tasks.

No preemption: a resource can be released only voluntarily by the task holding it, after that task has completed its task.

Circular wait: there exists a set $\{T_0, T_1, \ldots, T_0\}$ of waiting tasks 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 $T_2, \ldots, T_{n-1}$ 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.

Deadlock prevention (continued)

Restrain the ways requests can be made:

★ **No preemption**

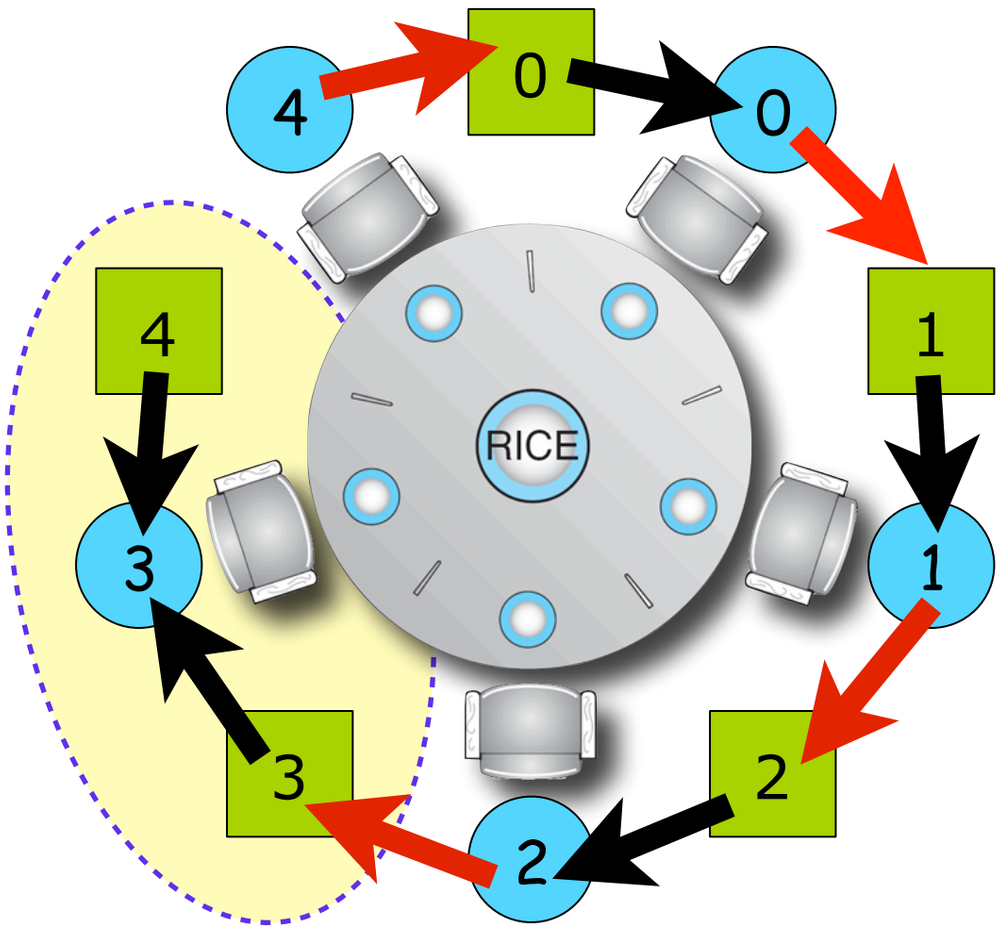
If a task that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released:

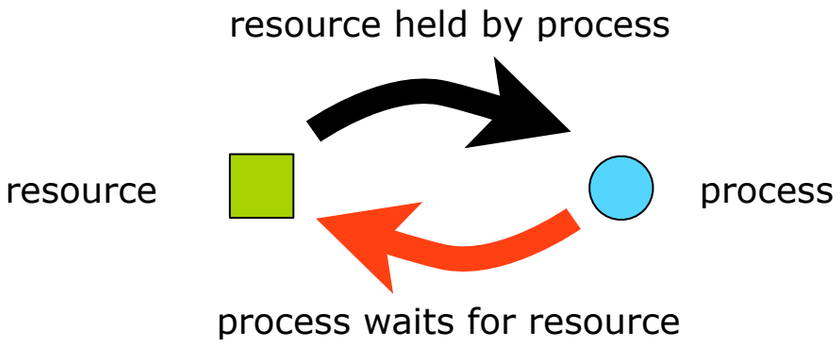
- Preempted resources are added to the list of resources for which the task is waiting.
- The task will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

★ **Circular wait**

Impose a **total ordering** of all resource types, and require that each task requests resources in an increasing order of enumeration.

Dining philosophers





Deadlock prevention: Make circular wait impossible by **impose a total ordering** of all resource types, and require that each process **requests** resources in an **increasing order of enumeration**.

Philosopher 0:

Grabs chopstick 0 , then waits for chopstick 1

Philosopher 1:

Grabs chopstick 1 , then waits for chopstick 2

Philosopher 2:

Grabs chopstick 2 , then waits for chopstick 3

Philosopher 3:

Grabs chopstick 3 , then grabs chopstick 4

Philosopher 4:

Waits for **Chopstick 0** ...

Imposing a total ordering prevents circular wait. In this scenario, philosopher 3 gets both chopsticks.

Deadlock avoidance

Requires that the system has some additional **a priori information** available.

A priori knowledge is independent of experience - information given to us before we start.

★ Simplest and most useful model requires that each task 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 potential **maximum** **demands** of the tasks.

Safe state

When a task requests an available resource, the system must decide if immediate allocation leaves the system in a safe state.

The system is in safe state if there exists a **sequence** $\langle T_1, T_2, \ldots, T_n \rangle$ of **all** the **tasks** in the systems such that for each T_i, the resources that T_i may still request can be satisfied by the currently available resources + resources held by all the T_j, with $j < i$.

That is:

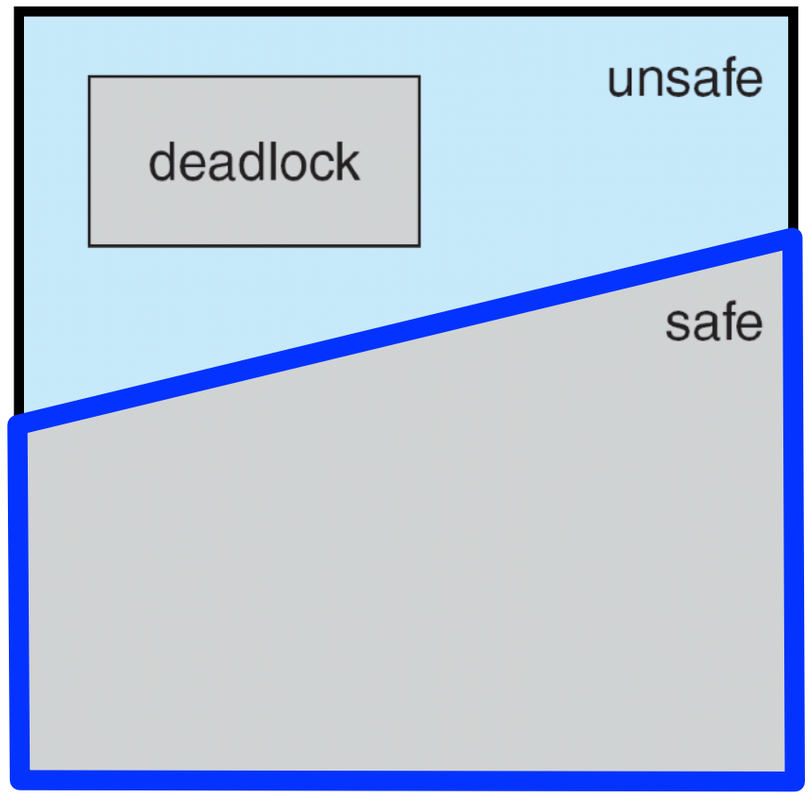
- If the resources needed by T_i are not immediately available, then T_i can wait until all T_j with $j < i$ have finished.
- When T_j is finished, T_i can obtain the needed resources, execute, return allocated resources, and terminate.
- When T_i terminates, T_{i+1} can obtain its needed resources, and so on.

Safe state and deadlock avoidance

★ If a system is in **safe state** ⇒ **no deadlocks**

★ If a system is in **unsafe state** ⇒ **possibility of deadlock**

★ **Avoidance** ⇒ ensure that a system will never enter an unsafe state.



Deadlock avoidance algorithms

- Single instance of each resource type
  - use a **resource allocation graph.**
- Multiple instances of a resource type
  - use **the Banker's algorithm.**

Deadlock

avoidance

Single instance of each resource

Resource allocation graph algorithm

Resource allocation graph algorithm

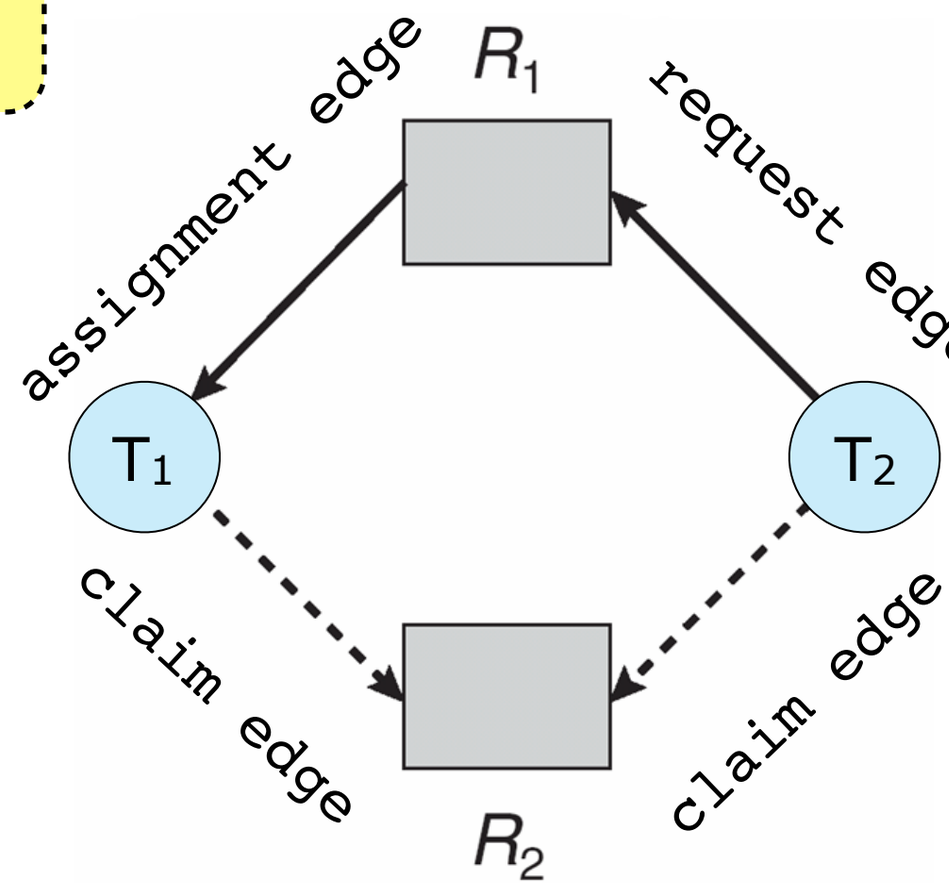
- ★ Resources must be claimed **a priori** in the system.
- ★ **Claim edge** $T_i \rightarrow R_j$ indicated that task T_j may request resource R_j; represented by a dashed line.

a priori (latin "from the earlier")

- ★ Claim edge converts to request edge when a process requests a resource.

- ★ **Request edge** converted to an assignment edge when the resource is allocated to the process.

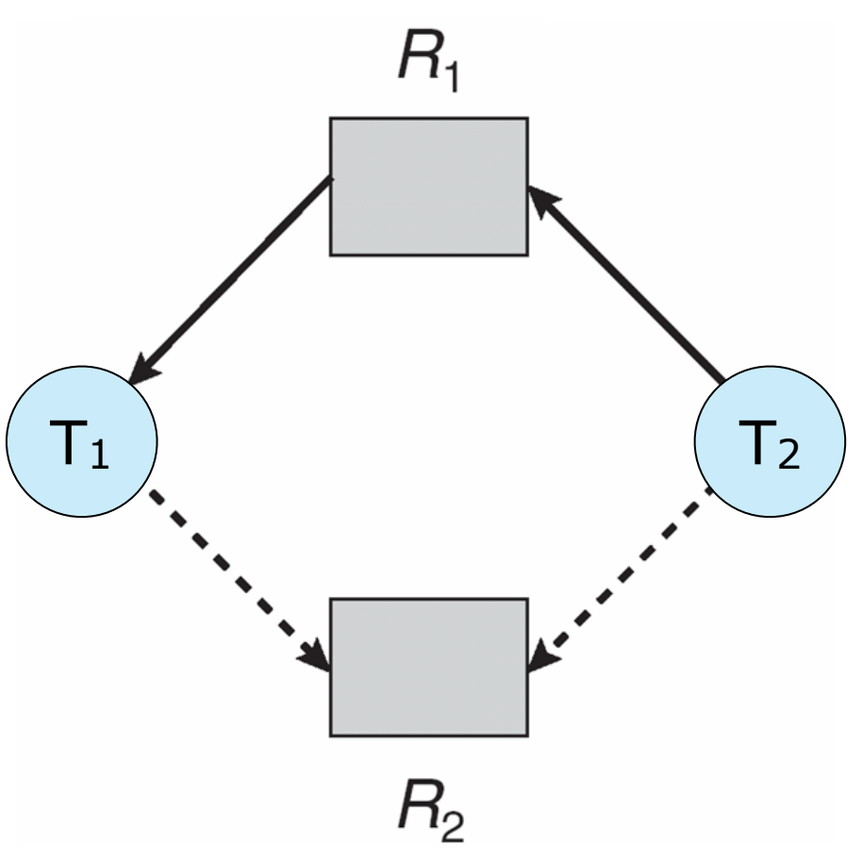
- ★ When a resource is released by a task, assignment edge reconverts to a claim edge.



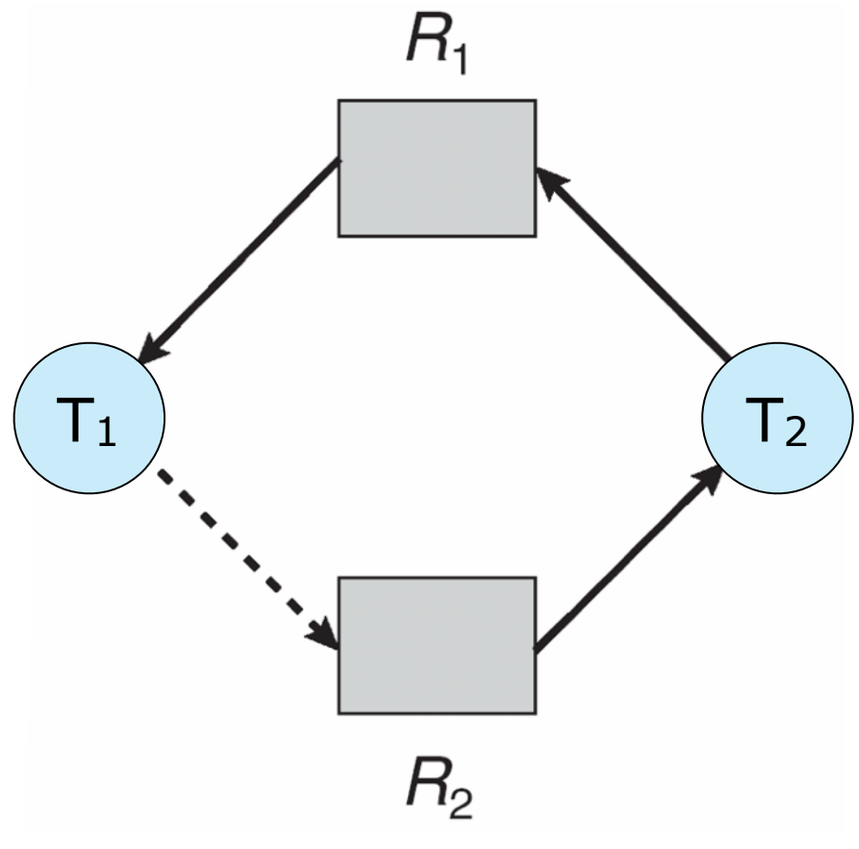
Resource allocation graph algorithm

★ Suppose that process P_i requests a resource R_j.

★ The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph.



If T_2 successfully allocates R_2.



This state is **safe**

This state is **not safe**

Deadlock

avoidance

Multiple instances of each resource

Banker's algorithm

Banker's algorithm

Banker's algorithm (1)

The Banker's algorithm is a resource allocation and **deadlock avoidance algorithm** developed by Edsger Dijkstra.

★ Tests for safety by **simulating** the **allocation** of **predetermined maximum possible amounts of all resources** ...

★ ... and then makes a "safe-state" check to **test for possible deadlock conditions for all other pending activities**, before deciding whether allocation should be allowed to continue.

Edsger Wybe Dijkstra
(1930 - 2002)

Banker's algorithm (2)

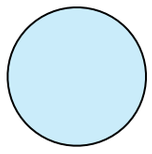
The Banker's algorithm is a resource allocation and **deadlock avoidance algorithm** developed by Edsger Dijkstra.

- ★ **Multiple instances** of each resource.
- ★ Each task must **a priori claim maximum use**.
  - **A priori** knowledge is independent of experience - information given to us before we start.
  - When a new task enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system
- ★ When a task **requests** a resource it **may have to wait**.
- ★ When a task gets all its resources it must return them in a finite amount of time.

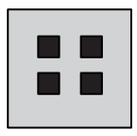
Resource types and instances

In the Banker's algorithm, each resource type is allowed to have multiple instances.

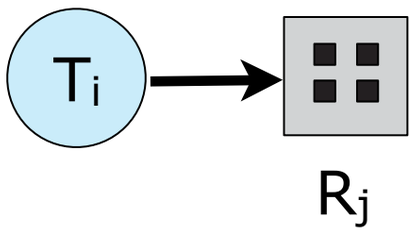
A task cannot distinguish between the instances of a resource.



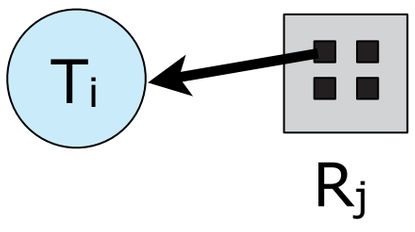
A task.



A resource with four indistinguishable instances, for example four printers.



Task T_i requests one of the four instances of resource R_j, for example one of the four printers.

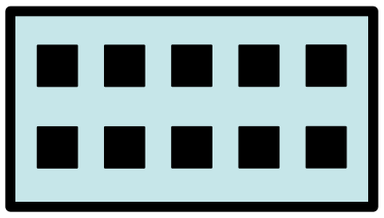


Task T_i is holding one of the four instances of resource R_j, for example one of the four printers.

Example of Banker's algorithm (1)

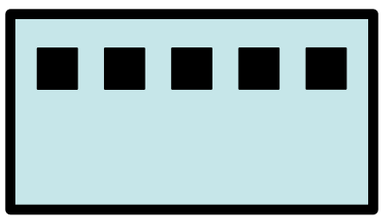
In the system there are 3 resource types: A (10 instances), B (5 instances), and C (7 instances).

A



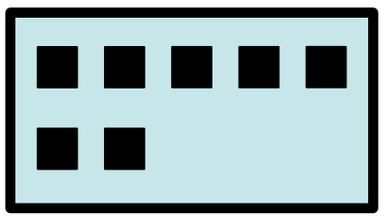
For example 10 available CPU cores.

B



For example 5 available color printers.

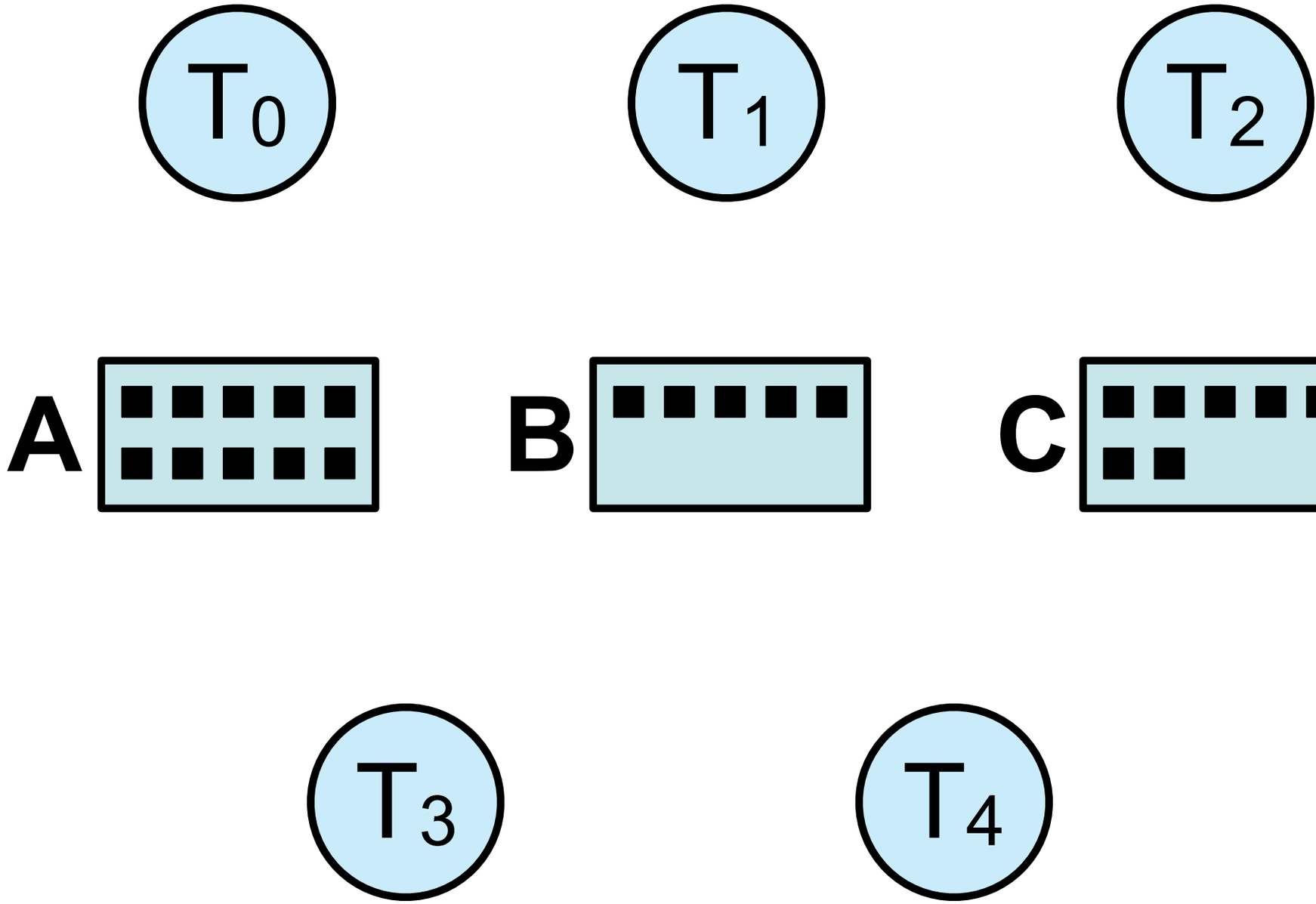
C



For example 7 available black and white printers.

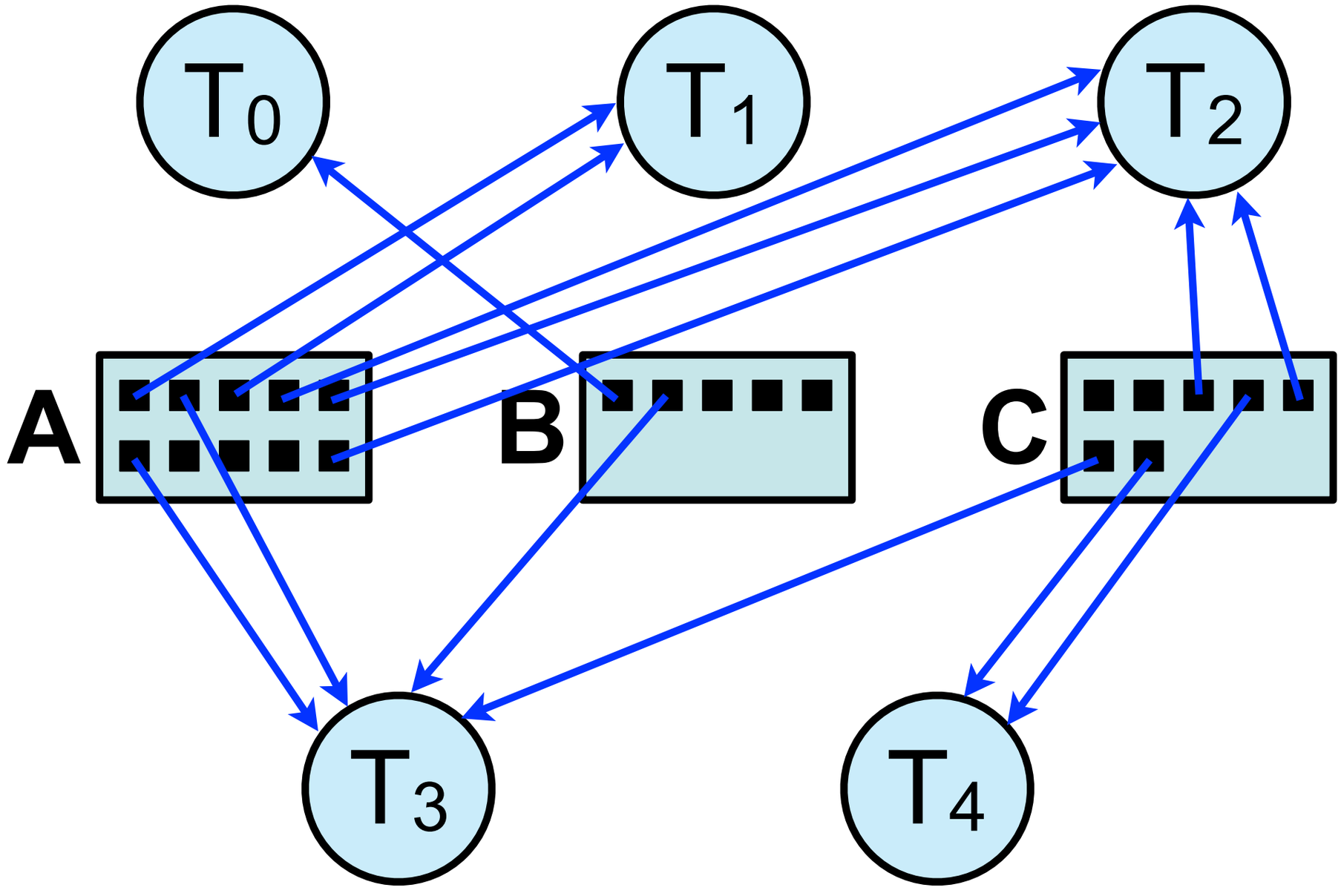
Example of Banker's algorithm (3)

In the system there are 5 tasks, T_0, T_1, ..., T_4.



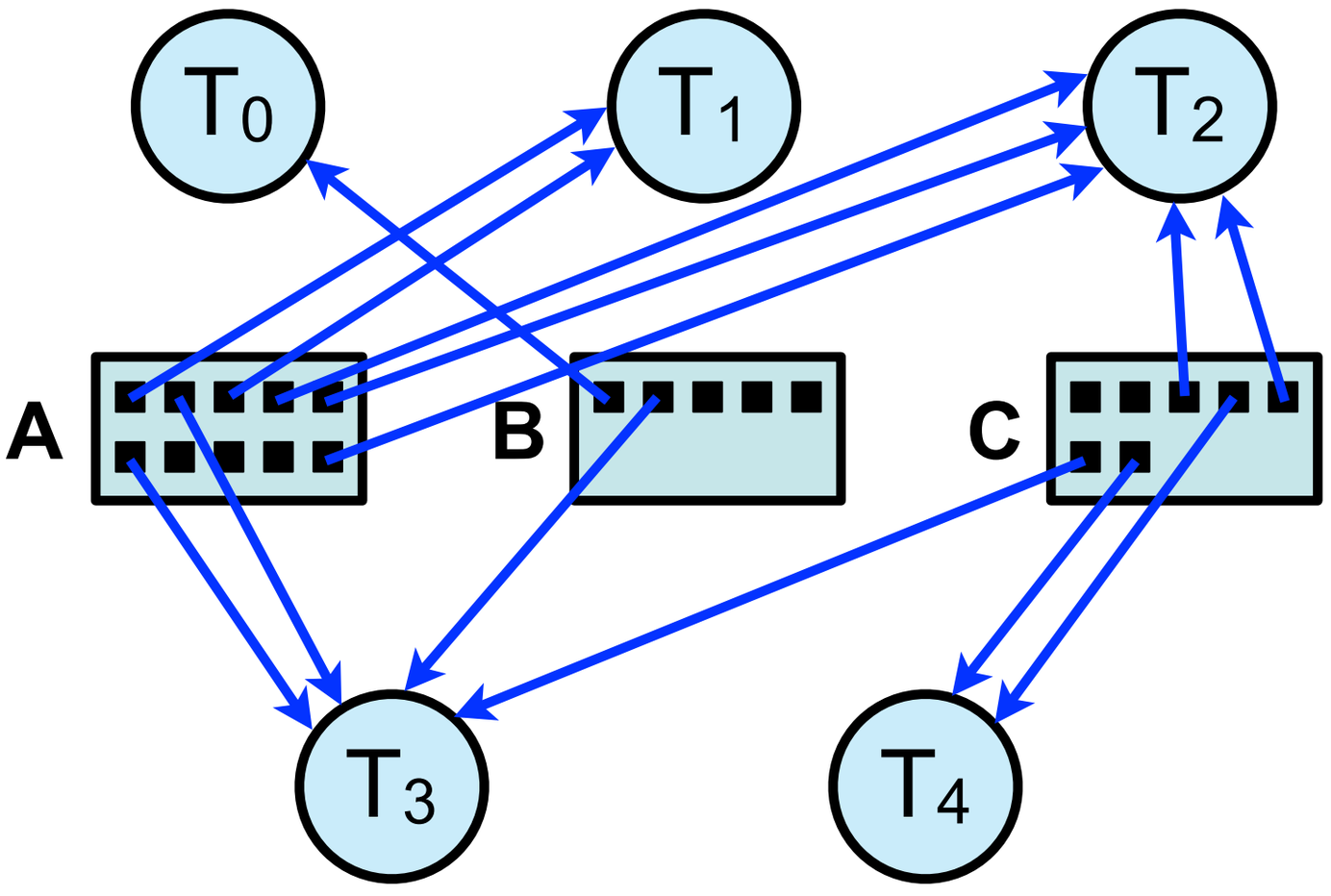
Example of Banker's algorithm (2)

Snapshot of the allocation state of the system at time t_0.



Example of Banker's algorithm (3)

Encode the resource allocation graph state using a resource allocation **matrix** and a **vector** with the available resource instances.



Non-allocated resource instances.

Available		
A	B	C
3	3	2

Allocated resource instances.

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

Example of Banker's algorithm (4)

When a new task enters a system, it must **a priori** declare the **maximum** number of instances of each resource type that it may ever **claim**.

A priori knowledge is independent of experience - information given to us before we start.

	Max		
	A	**B**	**C**
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

How many instances of each resource type does each task need at most?

Available		
A	**B**	**C**
3	3	2

Dynamic state

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

Example of Banker's algorithm (5)

Calculate the need matrix with information about how many instances of each resource type each task may request at most in addition to the currently allocated resource instances.

Available		
A	B	C
3	3	2

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	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

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

Need = Max - Allocation

Is the current state safe? (1)

Can we fulfil the need of one of the tasks?

Only one of T_1 or T_3 can be given the needed resources at once.

Before

Available		
A	B	C
3	3	2

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	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

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

Does it matter which of T_1 and T_3 we choose?

No it doesn't matter!

Is the current state safe? (2)

We pick T_1 to be granted the needed resources.

Before

Available		
A	**B**	**C**
3	3	2

	Max		
	A	**B**	**C**
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	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

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

Is the current state safe? (3)

We pick T_1 to be granted the needed resources.

Balance out the Available vector and the Allocation matrix.

Available decreases to (2, 1, 0).

	Before			Under		
	Available			Available		
	A	B	C	A	B	C
	3	3	2	2	1	0

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	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

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

Allocation for T_1 increases and now T_1 holds its max (3, 2, 2).

Is the current state safe? (4)

Once T_1 terminates the available resources instances increases.

$Available_{after} = Available_{under} + Allocation[T_1]_{under}$

$= (2, 1, 0) + (3, 2, 2) = (5, 3, 2)$

$= Available_{before} + Allocation[T_1]_{before}$

$= (3, 3, 2) + (2, 0, 0) = (5, 3, 2)$

	Before			Under			After		
	Available			Available			Available		
	A	B	C	A	B	C	A	B	C
	3	3	2	2	1	0	5	3	2

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	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

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

Is the current state safe? (5)

Tests for safety by **simulating** the allocation of predetermined maximum possible amounts of all resources for all tasks.

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

	Available				
Step	A	B	C	Done	Choice
1	3	3	2	-	T_1
2	5	3	2	T_1	T_3
3	7	4	3	T_3	T_4
4	7	4	5	T_4	T_2
5	10	4	7	T_2	T_0
6	10	5	7	T_0	★

All Done

During this simulation, we found a sequence $< T_1, T_3, T_4, T_2, T_0 >$ that allowed all tasks to get their maximum resource needs fulfilled - **state is safe**!

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 task 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]

Safety algorithm

The following safety algorithm is used to determine whether a state is safe or not for a system with **n** tasks and **m** resource types.

1) **Initialize:** Let Work and Finish be vectors of length m and n, respectively.
   - Work = Available (Initial number of available resources)
   - Finish[i] = false for i = 0, 1, …, n- 1 (Initially, no process has finished).

2) **Can a task be granted all requested resources?** Find and i such that both:
   - Finish[i] = false
   - $Need_i \leq Work$
   - If no such i exists, go to step 4, else continue to step 3.

3) **"Return resources":**
   - 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**

Dynamically invoke the bankers algorithm

When a task makes a resource allocation request, use the Banker's algorithm to ensure the system stays in a safe state.

1) If $\mathbf{Request_i \leq Need_i}$, go to step 2. Otherwise, raise error since the task has exceeded its maximum claim.

2) If $\mathbf{Request_i \leq Available}$ go to step 3. Otherwise T_i must wait, since the requested resources are not available.

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

   - $Available = Available - Request_i$
   - $Allocation_i = Allocation_i + Request_i$
   - $Need_i = Need_i - Request_i$
   - If state is **safe** ⇒ the **resources** are **allocated** to T_i
   - If state is **unsafe** ⇒ T_i must **wait**, and the old ”Before Request” resource-allocation state is restored

T_1 request (1,0,2)

1) If $\mathbf{Request_i \leq Need_i}$, go to step 2.

   Otherwise, raise error since the task has exceeded its maximum claim.

   (1, 0, 2) ≤ (1, 2, 2) OK

2) If $\mathbf{Request_i \leq Available}$ go to step 3.

   Otherwise T_i must wait, since resources are not available

   (1, 0, 2) ≤ (3, 3, 2) OK

Before request

Available		
A	B	C
3	3	2

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	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

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

T_1 request (1,0,2)

1) If $\mathbf{Request_i \leq Need_i}$, go to step 2.

   Otherwise, raise error since the task has exceeded its maximum claim.

   (1, 0, 2) ≤ (1, 2, 2) OK

2) If $\mathbf{Request_i \leq Available}$ go to step 3.

   Otherwise T_i must wait, since resources are not available

   (1, 0, 2) ≤ (3, 3, 2) OK

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

   - $Available = Available - Request_i$
     Available = (3, 3, 2) - (1, 0, 2) = (2, 3, 0)
   - $Allocation_i = Allocation_i + Request_i$
     $Allocation_1$ = (2, 0, 0) + (1, 0, 2) = (3, 0, 2)
   - $Need_i = Need_i - Request_i$
     $Need_1$ = (1, 2, 2) – (1, 0, 2) = (0, 2, 0)

   - If state is **safe** ⇒ the **resources** are **allocated** to T_i
   - If state is **unsafe** ⇒ T_i must **wait**, and the old ”Before Request” resource-allocation state is restored

Before request

Available		
A	B	C
3	3	2

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	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

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

”Pretend” request

Available		
A	B	C
2	3	0

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

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

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

Is the “pretended” state safe?

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

	Available				
Step	A	B	C	Done	Choice
1	2	3	0	-	T_1
2	5	3	2	T_1	T_3
3	7	4	3	T_3	T_4
4	7	4	5	T_4	T_2
5	10	4	7	T_2	T_0
6	10	5	7	T_0	★

All Done

During this simulation, we found a sequence < T_1, T_3, T_4, T_2, T_0> that allowed all tasks to get their maximum resource needs fulfilled - **state is safe**!

T_4 request (3,3,0)

1) If $Request_i \leq Need_i$ Needi go to step 2.

   Otherwise, raise error, since the task has exceeded its maximum claim.

   (3, 3, 0) ≤ (4, 3, 1) OK

2) If $Request_i \leq Available$, go to step 3.

   Otherwise T_i must wait, since resources are not available

   (3, 3, 0) ≤ (3, 3, 2) OK

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

   - $Available = Available - Request_i$
     $Available$ = (3, 3, 2) - (3, 3, 0) = (0, 0, 2)
   - $Allocation_i = Allocation_i + Request_i$
     $Allocation_1$ = (0, 0, 2) + (3, 3, 0) = (3, 3, 2)
   - $Need_i = Need_i - Request_i$
     $Need_1$ = (4, 3, 1) – (3, 3, 0) = (1, 0, 1)

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

Before Request

Available		
A	B	C
3	3	2

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	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

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

”Pretend” Request

Available		
A	B	C
0	0	2

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

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

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

Available (0,0,2) is not enough for any of the tasks to terminate. → **Not Safe!**

T_0 request (0,2,0)

1) If $Request_i \leq Need_i$ Needi go to step 2.

   Otherwise, raise error since the task has exceeded its maximum claim.

   (0, 2, 0) ≤ (7, 5, 3) OK

2) If $Request_i \leq Available$, go to step 3.

   Otherwise T_i must wait, since resources are not available

   (0, 2, 0) ≤ (3, 3, 2) OK

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

- $Available = Available - Request_i$
  $Available$ = (3, 3, 2) - (0, 2, 0) = (3, 1, 2)
- $Allocation_i = Allocation_i + Request_i$
  $Allocation_1$ = (0, 1, 0) + (0, 2, 0) = (0, 3, 0)
- $Need_i = Need_i - Request_i$
  $Need_1$ = (7, 4, 3) – (0, 2, 0) = (7, 2, 3)

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

Before Request

Available		
A	B	C
3	3	2

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

	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

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

”Pretend” Request

Available		
A	B	C
3	1	2

	Max		
	A	B	C
T_0	7	5	3
T_1	3	2	2
T_2	9	0	2
T_3	2	2	2
T_4	4	3	3

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

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

Safe?

Is the “pretended” state safe?

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

	Available				
Step	A	B	C	Done	Choice
1	3	1	2	-	T_3
2	5	2	3	T_3	T_1
3	7	2	3	T_1	T_0
4	7	5	3	T_0	T_2
5	10	5	5	T_2	T_4
6	10	5	7	T_4	★

All Done

During this simulation, we found a sequence < T_3, T_1, T_0, T_2, T_4> that allowed all tasks to get their maximun resource needs fulfilled - **state is safe!**

Time complexity of deadlock avoidance with Banker's algorithm

Deciding whether a state is safe or not using Banker's algorithm requires an order of **O(m x n^2)** operations, where

m = number of resources
n = number of tasks.

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 task 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]

Safety algorithm

The following safety algorithm is used to determine whether a state is safe or not for a system with **n** tasks and **m** resource types.

1) **Initialize:** Let Work and Finish be vectors of length m and n, respectively.
   - Work = Available (Initial number of available resources)
   - Finish[i] = false for i = 0, 1, …, n- 1 (Initially, no process has finished).

2) **Can a task be granted all requested resources?** Find and i such that both:
   - Finish[i] = false
   - $Need_i \leq Work$
   - If no such i exists, go to step 4, else continue to step 3.

3) **”Return resources”:**
   - 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**

Dynamically invokation of the bankers algorithm

When a task makes a resource allocation request, use the Banker's algorithm to ensure the system stays in a safe state.

1) If $\mathbf{Request_i \leq Need_i}$, go to step 2. Otherwise, raise error since the task has exceeded its maximum claim.

2) If $\mathbf{Request_i \leq Available}$ go to step 3. Otherwise T_i must wait, since the requested resources are not available.

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

   - $Available = Available - Request_i$
   - $Allocation_i = Allocation_i + Request_i$
   - $Need_i = Need_i - Request_i$
   - If state is **safe** ⇒ the **resources** are **allocated** to T_i
   - If state is **unsafe** ⇒ T_i must **wait**, and the old ”Before Request” resource-allocation state is restored

Deadlock

detection

Deadlock detection

If deadlocks occur - how can this be detected?

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

Deadlock detection

Single instance of each resource

Wait-for graph

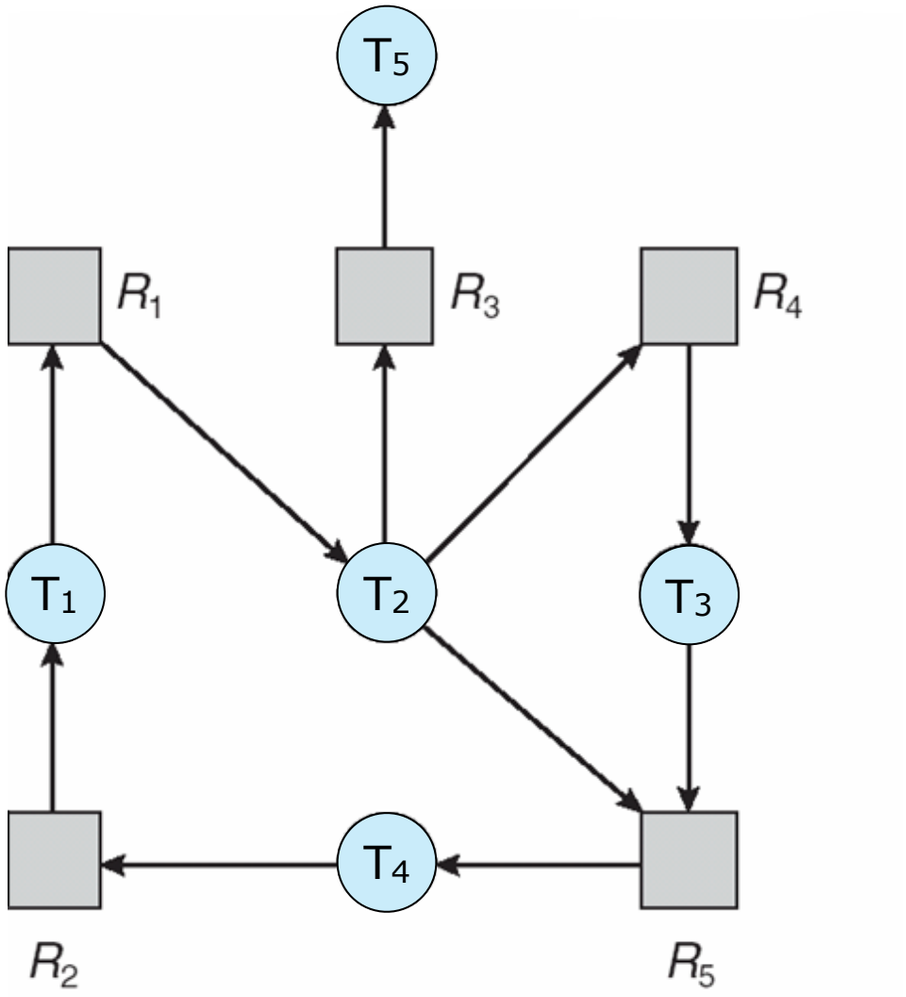
Single instance of each resource type

If all resources have only a single instance, a wait-for graph can be used to detect deadlocks.

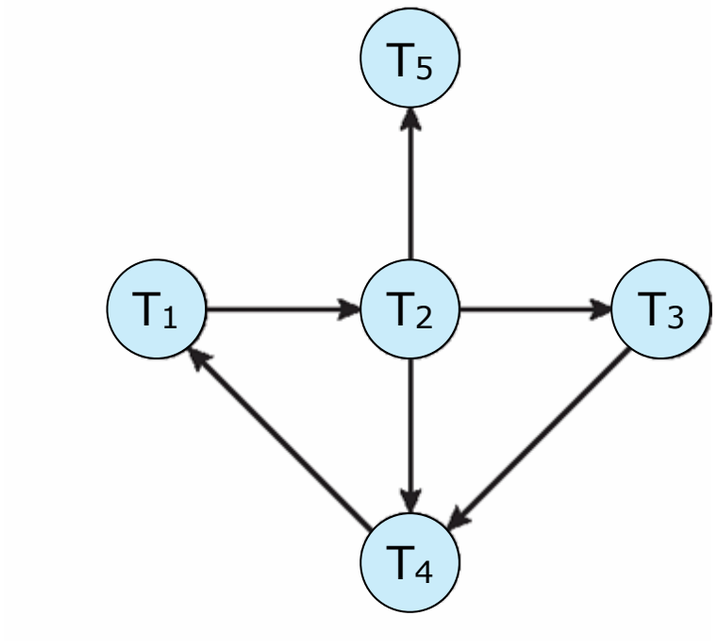
- ★ Maintain **wait-for graph**
  - Nodes are tasks
  - $T_i \rightarrow T_j$ if T_i is waiting for T_j
- ★ Periodically invoke an algorithm that searches for a cycle in the graph. **If** there is **a cycle**, there exists a **deadlock.**

★ An algorithm to detect a cycle in a graph requires an order of n^2 operations, where **n** is the number of nodes in the graph.

Resource allocation graph and wait-for graph



Resource allocation graph



Corresponding **wait-for graph**

Deadlock

detection

Multiple instances of each resource.

Variation of Banker´s algorithm

Deadlock detection

The wait-for graph scheme is not applicable to a system with multiple instances of each resource type. Deadlocks can be detected by an algorithm similar to the Banker's algorithm.

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

Note: This method don't use any a priori information regarding the maximal number of resources needed by each task.

Detection algorithm (example 1)

- ★ Five tasks T_0 through T_4
  - N = 5
- ★ Three resource types:
  - M = 3
    - A (7 instances)
    - B (2 instances)
    - C (6 instances)

- ★ Snapshot at time t_0:
  - Total = (7,2,6)
  - Allocated = (7,2,6)
  - Available = (0,0,0)
  - Work = (0,0,0)

	Allocation			Request			
	A	B	C	A	B	C	Finish
T_0	0	1	0	0	0	0	true
T_1	2	0	0	2	0	2	true
T_2	3	0	3	0	0	0	true
T_3	2	1	1	1	0	0	true
T_4	0	0	2	0	0	2	true
	7	2	6	**Total**			

	Work				
Step	A	B	C	Done	Choice
1	0	0	0	-	T_0
2	0	1	0	T_0	T_2
3	3	1	3	T_2	T_3
4	5	2	4	T_3	T_1
5	7	2	4	T_1	T_4
6	7	2	6	T_4	★

Initially, work = available

All Done

During this simulation, we found a sequence < T_0, T_2, T_3, T_1, T_4> that allowed all tasks to get their resources - **no deadlock!**

Detection algorithm (example 2)

- ★ Five tasks T_0 through T_4
  - N = 5
- ★ Three resource types:
  - M = 3
    - A (7 instances)
    - B (2 instances)
    - C (6 instances)

- ★ Snapshot at time t_0:
  - Total = (7,2,6)
  - Allocated = (7,2,6)
  - Available = (0,0,0)
  - Work = (0,0,0)
- ★ T_2 requests an additional instance of type C

	Allocation			Request			
	A	B	C	A	B	C	Finish
T_0	0	1	0	0	0	0	true
T_1	2	0	0	2	0	2	false
T_2	3	0	3	0	0	1	false
T_3	2	1	1	1	0	0	false
T_4	0	0	2	0	0	2	false
	7	2	6	**Total**			

	Work				
Step	A	B	C	Done	Choice
1	0	0	0	-	T_0
2	0	1	0	T_0	
3	?	?	?		
4	?	?	?		
5	?	?	?		
6	?	?	?		

Initially, work = available

Deadlock

Deadlock! Only T_0 can obtain the requested resources - T_1, T_2, T_3 and T_4 are all waiting for resources held by each other.

Detection algorithm

1) Let Work and Finish be vectors of length **m** (number of resource types) and **n** (number of tasks), respectively

   Initialize:

   - Work = Available
   - For i = 1,2, …, n,

     if Allocation[i] ≠ 0, then
         Finish[i] = false
     otherwise
         Finish[i] = true

2) Find an index i such that both:

   - Finish[i] == false
   - Request[i] ≤ Work

   - If no such i exists, go to step 4, else continue to step 3.

3) Work = Work + Allocation[i]
   Finish[i] = true
   go to step 2

4) If Finish[i] == false, for some i, 1 ≤ i ≤ n, then the system is in deadlock state.

   Moreover, if Finish[i] == false, then T_i is deadlocked

We know that T_i is not involved in a deadlock since Request[i] ≤ Work.

Thus we take an **optimistic attitude** and assume that T_i will require no more resources to complete it task; it will thus soon return all currently allocated resources to the system.

If our assumption is incorrect, a deadlock may occur later. That deadlock will be detected the next time the deadlock-detection algorithm is invoked.

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

Detection algorithm usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many tasks 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 tasks “caused” the deadlock

Recovery from deadlock

When a deadlock is detected, what actions can be taken?

- ★ **Abort** one or more **tasks** to break the circular wait.
- ★ **Preempt** some **resources** from one or more of the deadlocked tasks.

Process termination

The system reclaims all resources allocated to the terminated process.

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

  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Resource preemption

Preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

- **Selecting a victim** – minimize cost.

- **Rollback** – return to some safe state, restart process from that state.

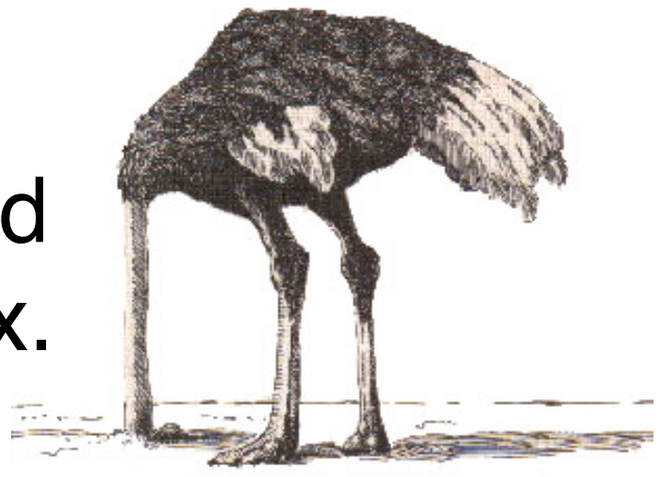
- **Starvation** – same process may always be picked as victim, include number of rollbacks in cost factor.

Methods for handling deadlocks

- Use some protocol to **prevent** or **avoid** deadlocks to ensure that the system will never enter a deadlock state.

- **Allow** the system to enter a deadlock state, **detect** it, and then **recover.**

- **Ignore** the problem and pretend that deadlocks never occur in the system; used by most operating systems, including Unix.



The Ostrich algorithm