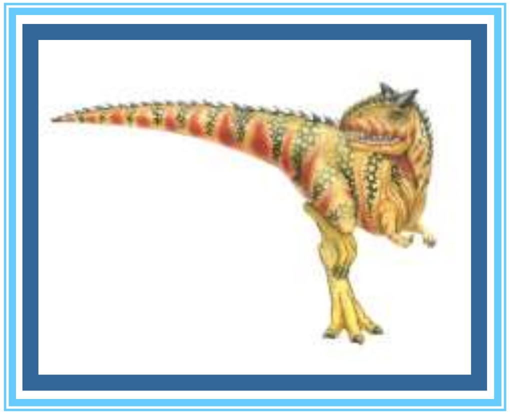
Chapter 7: Deadlocks



In a multiprogramming environment, several processes may compete for a finite number of resources.

A process requests resources; if the resources are not available at that time, the process enters a waiting state.

Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.

This situation is called a **deadlock**.

In this chapter, we describe methods that an operating system can use to prevent or deal with deadlocks

An Example

- 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
- A law passed by the Kansas legislature early in the 20th century

System Model

- System consists of a finite number of resources
- A number of competing processes
- Partitioned into several **resource types** R_1, R_2, . . ., R_m
  - *CPU cycles, memory space, I/O devices*
- Each resource type R_i has W_i instances.
- Synchronization tools, such as mutex locks and semaphores are also considered system resources
  - A common source of deadlock
- Each process utilizes a resource in only the sequence:
  - **request**
  - **use**
  - **release**

Deadlock with Mutex Locks

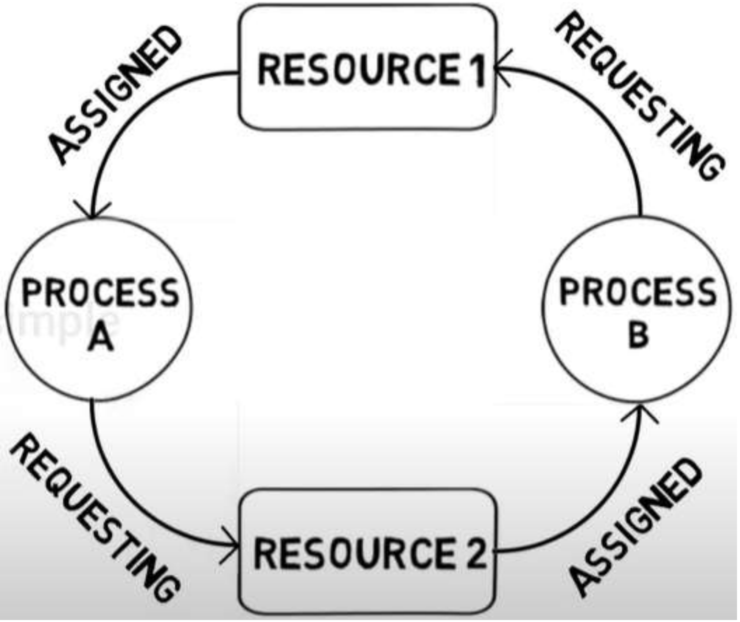
- The locking tools (e.g., mutex locks, semaphores) are designed to avoid race conditions. However, inappropriate usage of them can lead to deadlocks.

  Example?

- **Deadlock** – A set of processes is deadlocked when **every** process in the set is waiting for the resource that is currently allocated to another process in the set
- Let ***S*** and ***Q*** be two semaphores initialized to 1

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

Deadlock Example



Deadlock Characterization

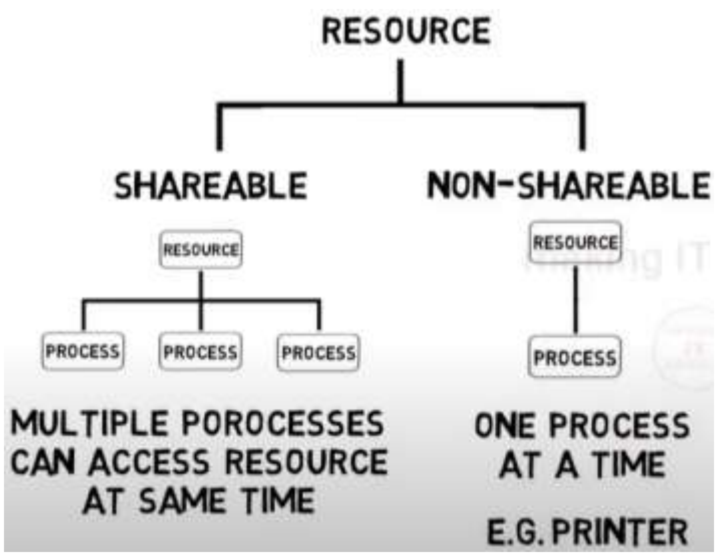
Deadlock can arise if four conditions hold simultaneously.

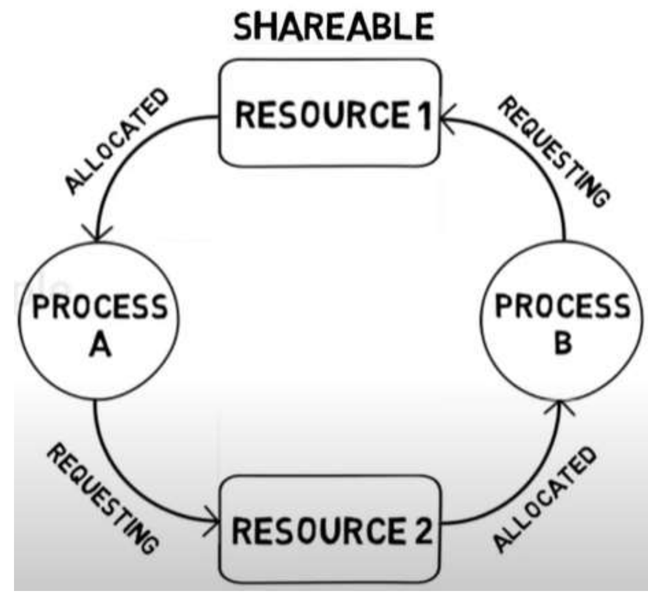
- **Mutual exclusion:** at least one resource must be held in a non-sharable mode; that is, only one process at a time can use a resource
- **Hold and wait:** a process must be holding at least one resource and waiting to acquire additional resources held by other processes
- **No preemption:** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** A set $\{P_0, P_1, \ldots, P_n\}$ 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, and P_n is waiting for a resource that is held by P_0.

The circular-wait condition implies the hold-and-wait condition

Mutual exclusion

- **Mutual exclusion:** at least one resource must be held in a nonsharable mode; that is, only one process at a time can use a resource



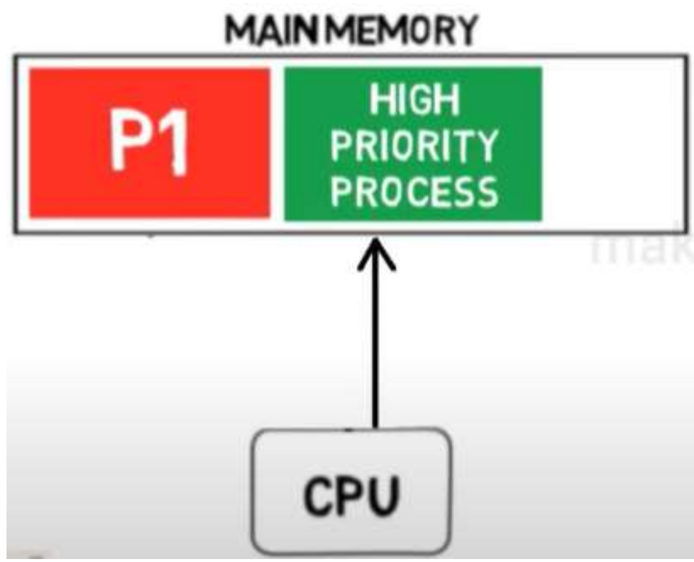


AS BOTH WILL GET EXECUTED
THERE WILL BE NO DEADLOCK

No preemption

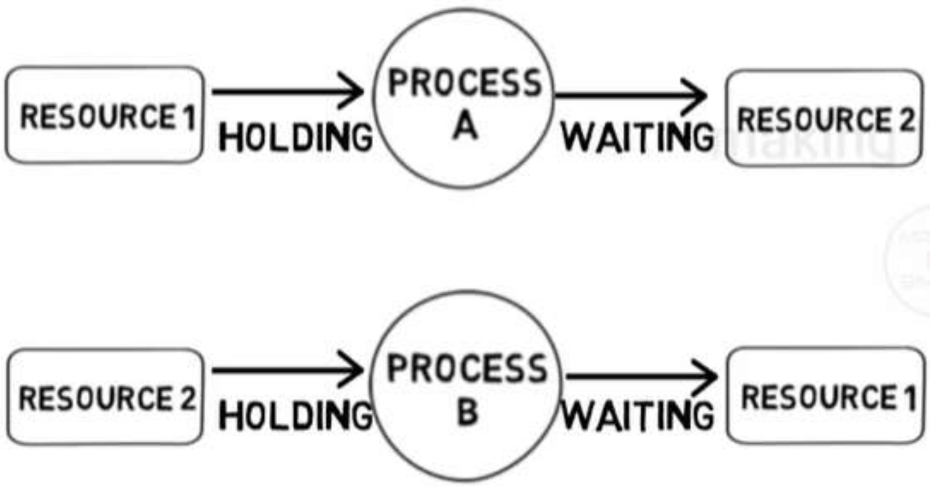
- **No preemption**: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task

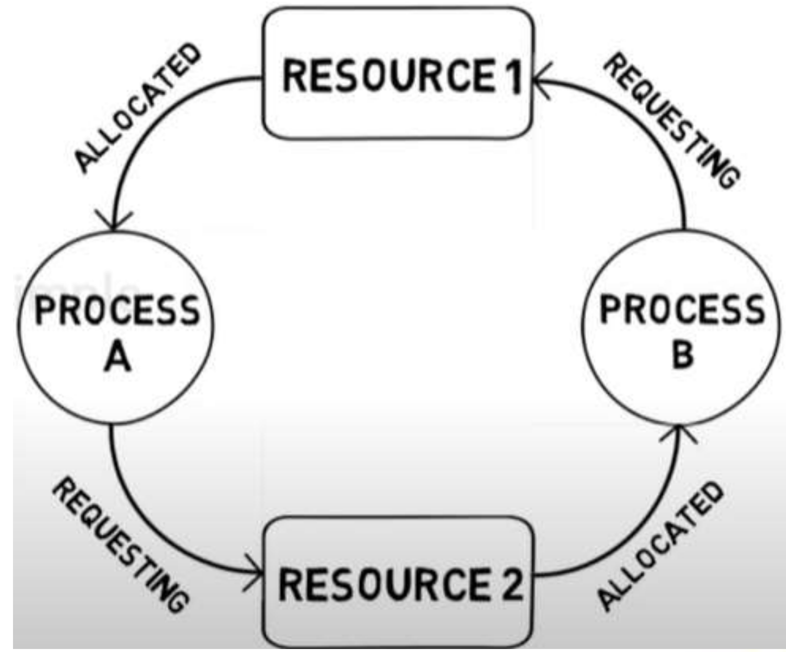
Preemption: force stopping a process



Hold and wait

- **Hold and wait:** a process must be holding at least one resource and waiting to acquire additional resources held by other processes

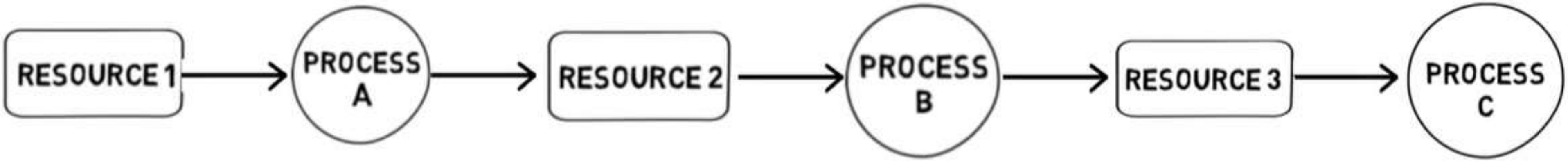


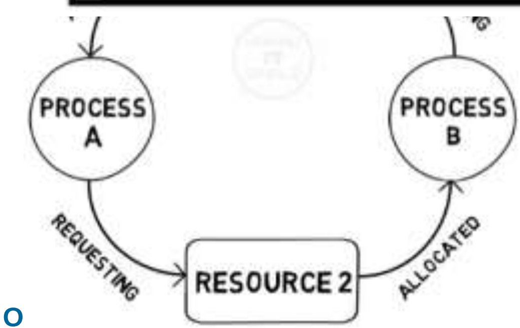


Circular wait

Deadlock can arise if four conditions hold simultaneously.

- **Circular wait:** A set $\{P_0, P_1, \ldots, P_n\}$ 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, and P_n is waiting for a resource that is held by P_0.





Resource-Allocation Graph

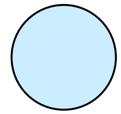
Deadlocks can be described more precisely in terms of a **directed graph** called a system resource-allocation graph

A set of vertices V and a set of edges E.

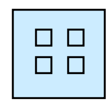
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

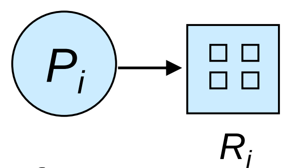
- Process



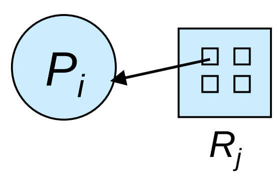
- Resource Type with 4 instances



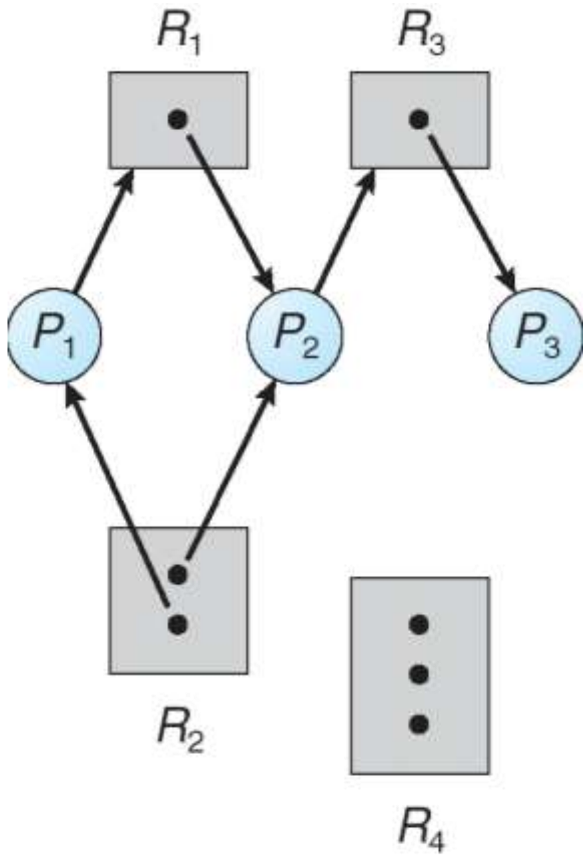
- P_i requests instance of R_j



- P_i is holding an instance of R_j

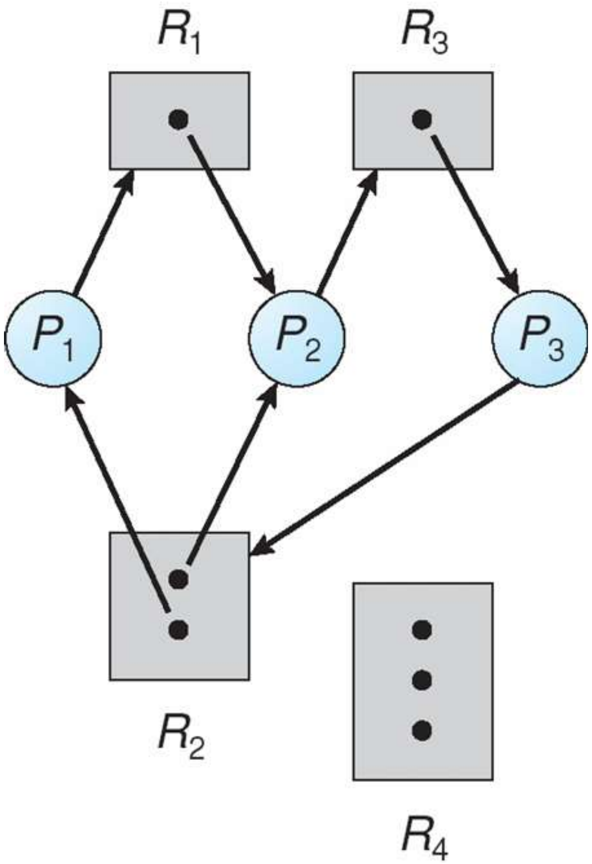


Example of a Resource Allocation Graph



- Do we have a deadlock here?
- It can be shown that if the graph contains no cycles, then no process in the system is deadlocked

Resource Allocation Graph With A Deadlock

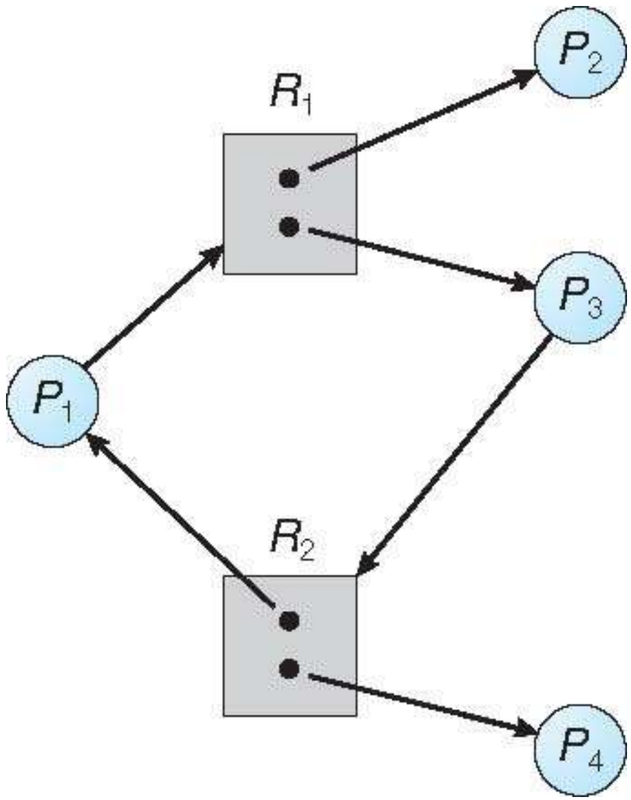


Two minimal cycles:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

Do we have a deadlock here?

Graph With A Cycle But No Deadlock



If the graph does contain a cycle, then a deadlock may exist

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

Do we have a deadlock here?

Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- **Deadlock prevention**
  - Ensure that at least one of the necessary conditions cannot hold
- **Deadlock avoidance**
  - Requires that the OS be given additional information in advance concerning which resources a process will request and use during its lifetime
- **Detection & Recovery**
  - Allow the system to enter a deadlock state, detect it, and then recover

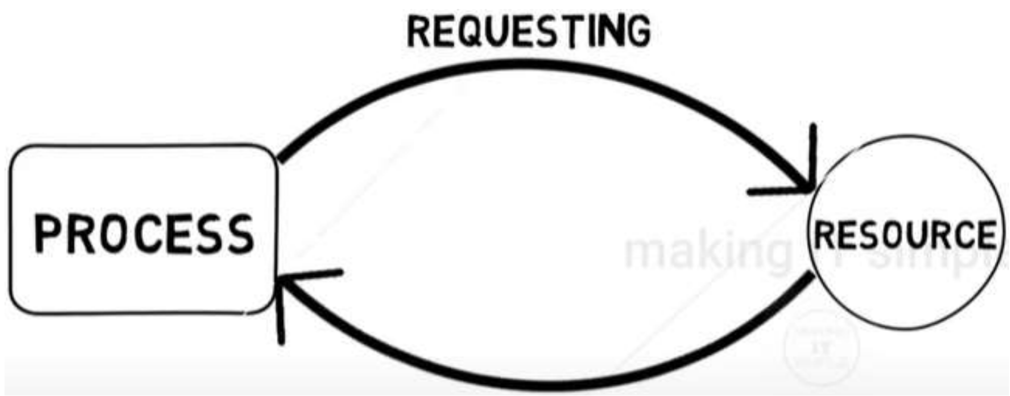
Deadlock Prevention

Prevention is better than cure

It's better to take preventive measures long before problem occurs

Idea: remove any one or all four conditions

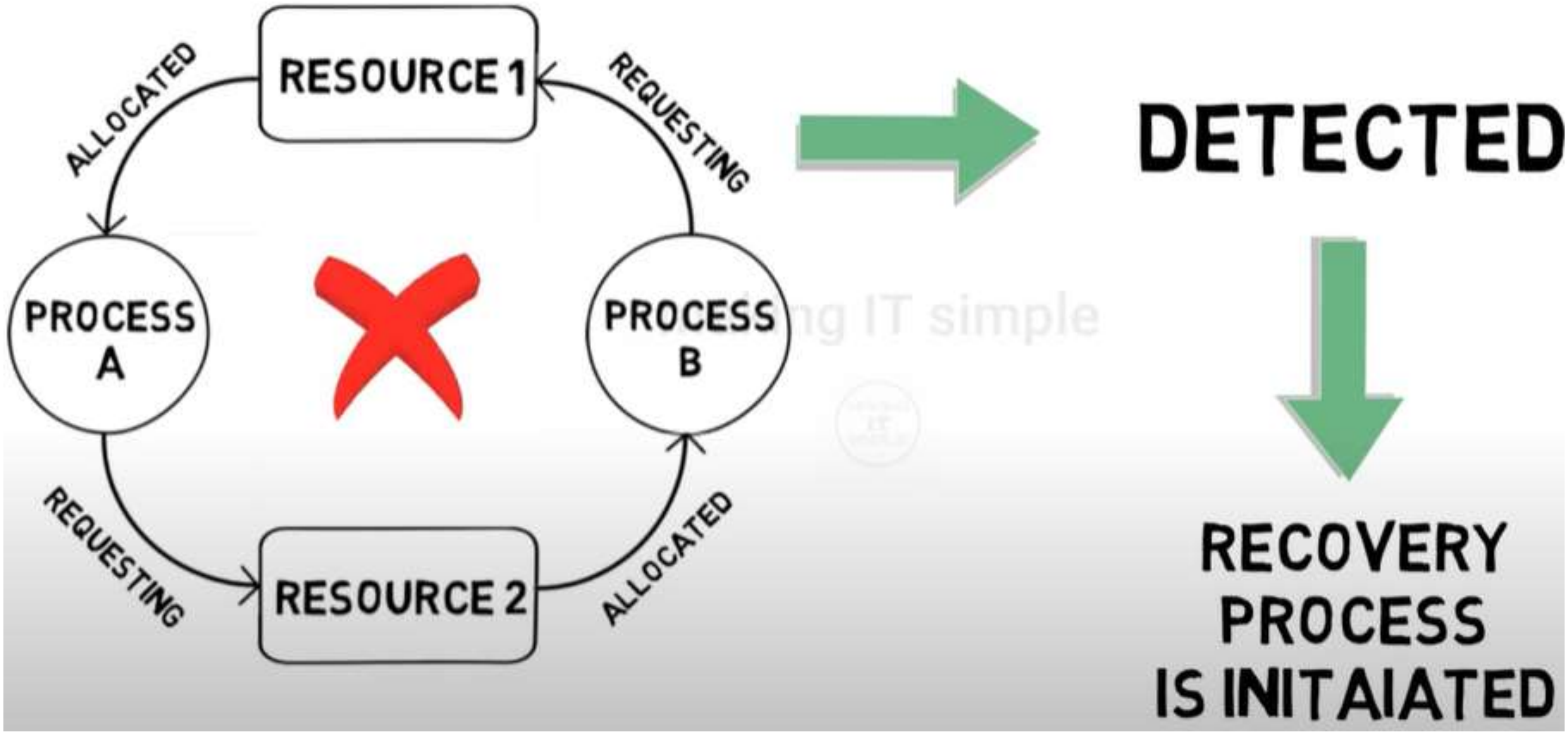
Deadlock Avoidance



IS THE REQUESTED RESOURCE
ALLOCATED TO ANY
OTHER PROCESS?

IF WE ALLOCATE THE
REQUESTED RESOURCE,
WILL IT LEAD TO DEADLOCK?

Detection & Recovery



Deadlock Prevention

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
  - We cannot prevent deadlocks by denying the mutual-exclusion condition

Deadlock Prevention

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Method 1: Require process to request and be allocated all its resources before it begins execution, disadvantage?
  - Method 2: Allow process to request resources only when the process has none allocated to it.
    - Before request, first release all
  - Consider a process that copies data from a DVD driver to a file on disk, sorts the file, and then prints the results to a printer
    - Method 1: request all three resources in the beginning
    - Method 2: request DVD and disk file first; copy from DVD to disk file; release both DVD and file; and then request disk file and printer
  - Low resource utilization; starvation possible

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 **preempted** (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

Deadlock Prevention (Cont.)

- **Circular Wait** – impose a total **ordering** of all **resource types**, and require that each process requests resources **in an increasing order** of enumeration
  - F(tape drive) = 1; F(disk drive) = 5; F(printer) = 12
  - A process can initially request any number of instances of a resource type---say, R_i. After that, the process can request instances of resource type R_j if and only if F(R_j) > F(R_i)
  - Alternatively, we can require that a process requesting an instance of resource type R_j must have released any resources R_i such that F(R_i) > F(R_j)

If these two protocols are used, then the circular-wait condition cannot hold

Deadlock Example

```
/* 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);
}

/* 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);
}
```

If the lock ordering was:

F(first_mutex) = 1
F(second_mutex) = 5

The thread_two could not request the locks out of order

Deadlock Avoidance

Requires that the system has some **additional information** available, about how resources are to be request

- Simplest and most useful model requires that each process 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

Need priori information

Deadlock Avoidance

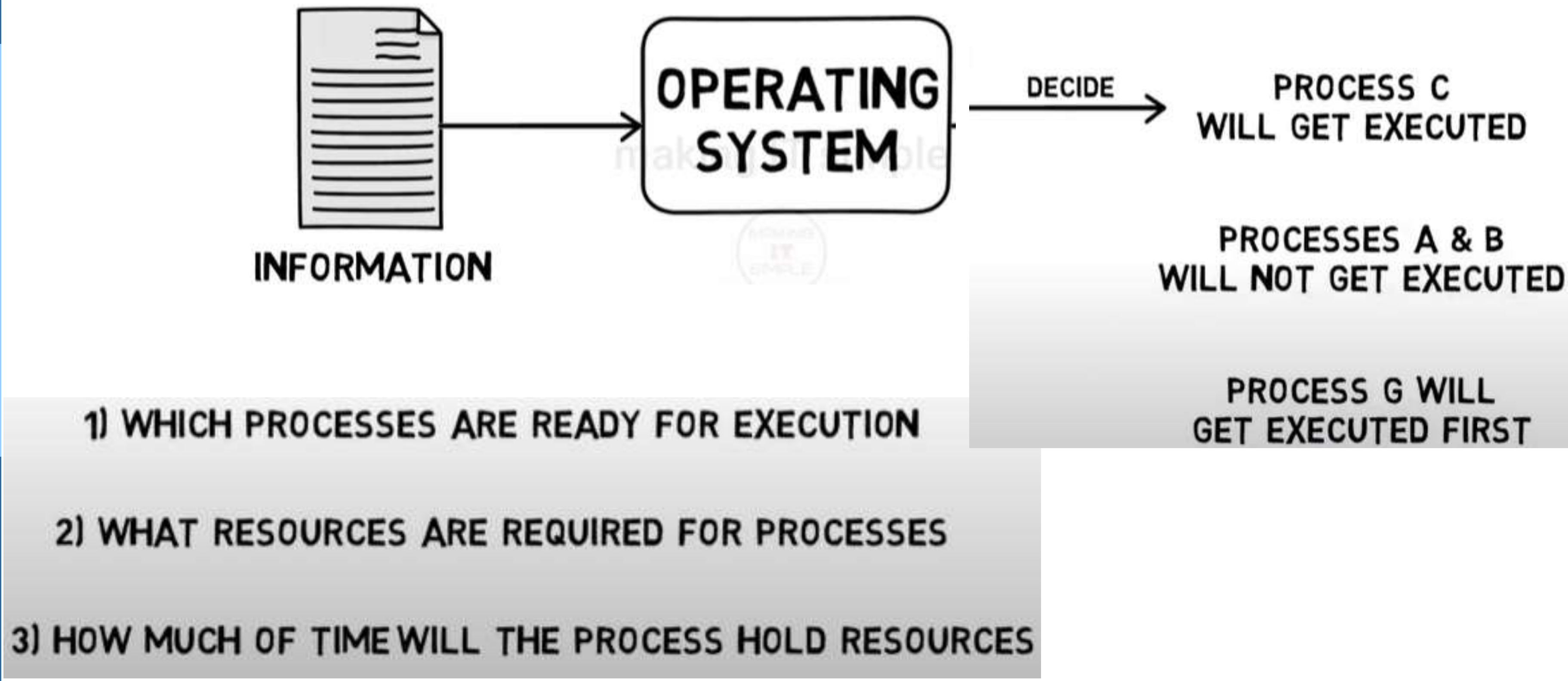
Process P will request first the tape drive and then the printer before releasing both resources?

Whereas process Q will request first the printer and then the tape drive?

With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

Deadlock Avoidance

Just avoiding problem when it is about to happens



Deadlock Avoidance Algorithm:

Banker’s Algorithm

Banker's Algorithm

- A resource allocation and deadlock avoidance algorithm
  - Tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources
  - Then makes a check to test for possible activities, before deciding whether allocation should be allowed to continue.

Banker's Algorithm

- Considering a system with 5 processes, and 3 resources of type A, B, C
  - A: 10 instances
  - B: 5 instances
  - C: 7 instances
- Suppose at time t_0 following snapshot of the system has been taken:

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

Banker's Algorithm

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

Question1. What will be the content of the **Need matrix**?

Need = Max – Allocation

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

Banker's Algorithm

Question2. Is the system in a safe state? If Yes, then what is the safe sequence?

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

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

For P_0: need = 7 4 3

available = 3 3 2

need > available

P_0 must wait

Banker's Algorithm

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

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

For P_1: need = 1 2 2

available = 3 3 2

need < available

P_1 can be allocated to resources

After P1 is finished, all allocated resources must be released

available = 3 3 2 + 2 0 0 = 5 3 2

Banker's Algorithm

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

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

For P_2: need = 6 0 0

available = 5 3 2

need > available

P_2 must wait

Banker's Algorithm

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

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

For P_3: need = 0 1 1

available = 5 3 2

need < available

P_3 can be allocated to resources

After P3 is finished, all allocated resources must be released

available = 5 3 2 + 2 1 1 = 7 4 3

Banker's Algorithm

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

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

For P_4: need = 4 3 1

available = 7 4 3

need < available

P_4 can be allocated to resources

After P4 is finished, all allocated resources must be released

available = 7 4 3 + 0 0 2 = 7 4 5

Banker's Algorithm

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

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

For P_0: need = 7 4 3

available = 7 4 5

need < available

P_0 can be allocated to resources

After P0 is finished, all allocated resources must be released

available = 7 4 5 + 0 1 0 = 7 5 5

Banker's Algorithm

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

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

For P_2: need = 6 0 0

available = 7 5 5

need < available

P_2 can be allocated to resources

After P2 is finished, all allocated resources must be released

available = 7 5 5 + 3 0 2 = 10 5 7

Banker's Algorithm

Hence, the system is in safe state!

The safe sequence is P1, P3, P4, P0, P2

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
  - An algorithm that examines the state of the system to determine whether a deadlock has occurred
- Recovery scheme
  - Option 1: Process Termination------abort one or more processes to break the circular wait
  - Option 2: Process Preemption------preempt some resources from one or more of the deadlocked processes

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
  - What's drawback?
  - Break, but at great expense
- Abort one process at a time until the deadlock cycle is eliminated
  - What's drawback?
  - Incurs considerable overhead, deadlock-detection algorithm invoked frequently
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken
- **Selecting a victim** – minimize cost
  - Number of resources is holding; amount of time has thus far consumed
- **Rollback** – return to some safe state, restart process from that state
  - Hard to determine, so total rollback
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

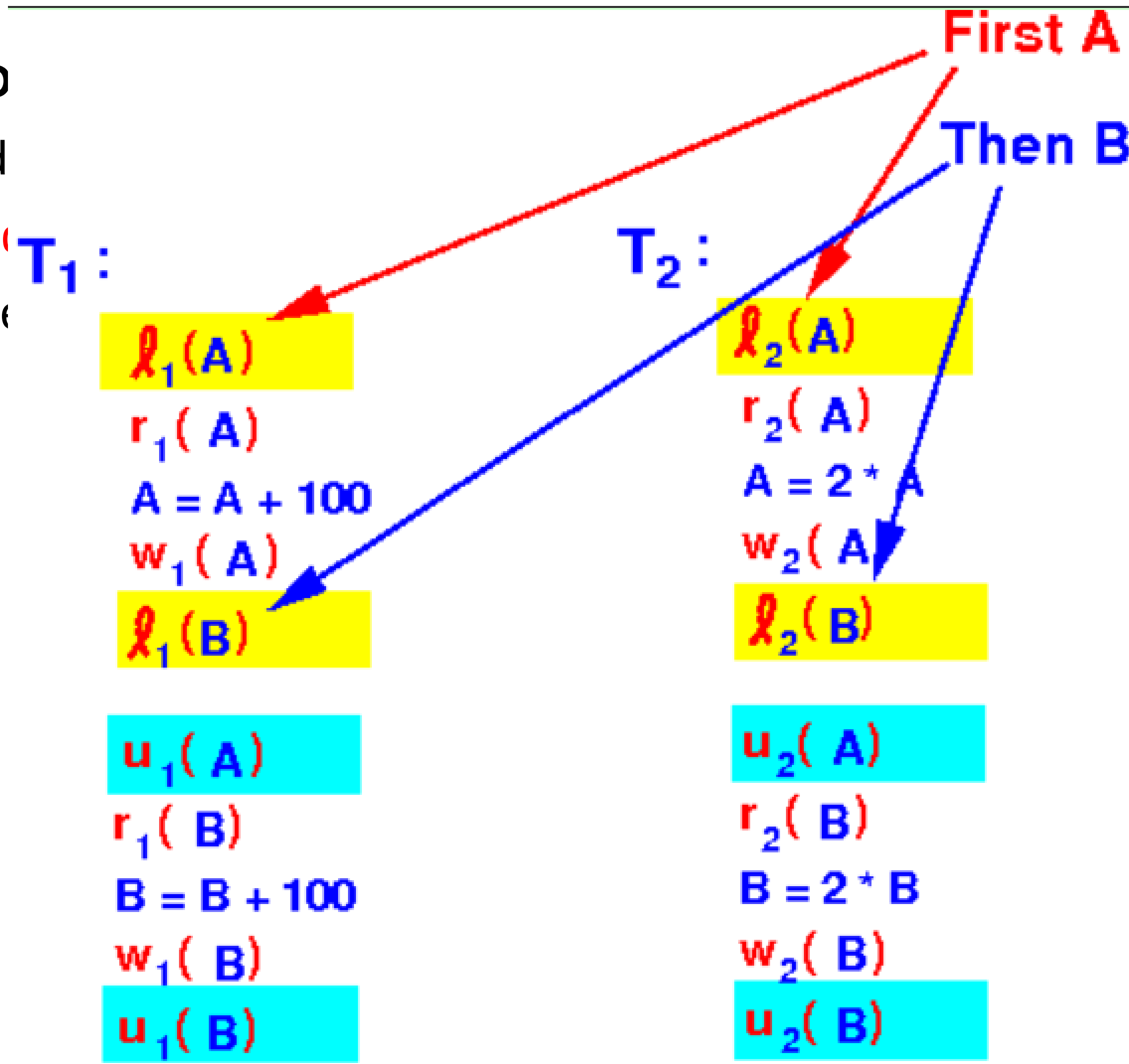
Deadlock prevention

Deadlock prevention by ordering the resource requesting

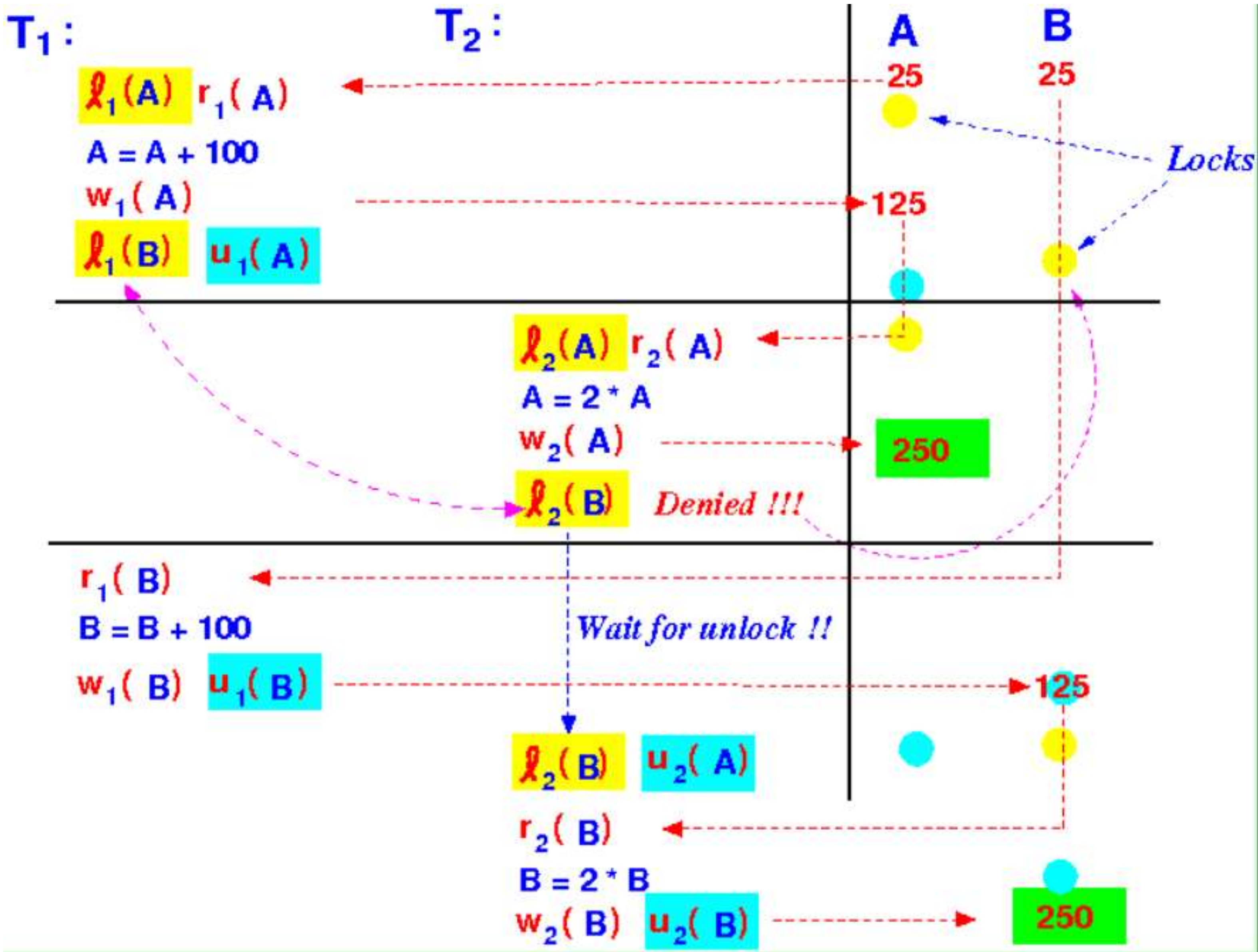
Deadlock prevention

Topic: D … **esting**

- Dead … tions
- "Goo
  Proc … same order:



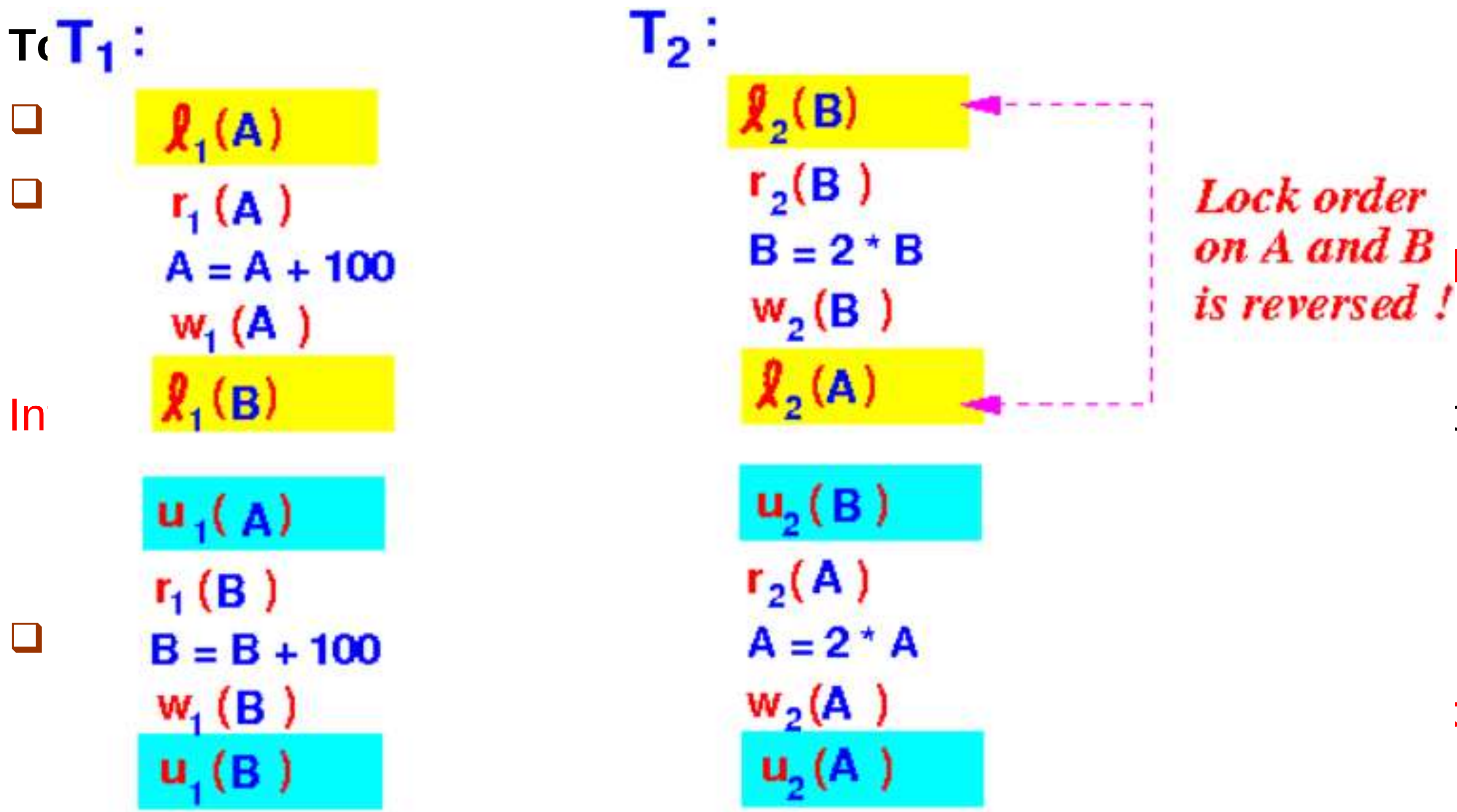
Deadlock prevention



er:

k:

Deadlock prevention



To

In

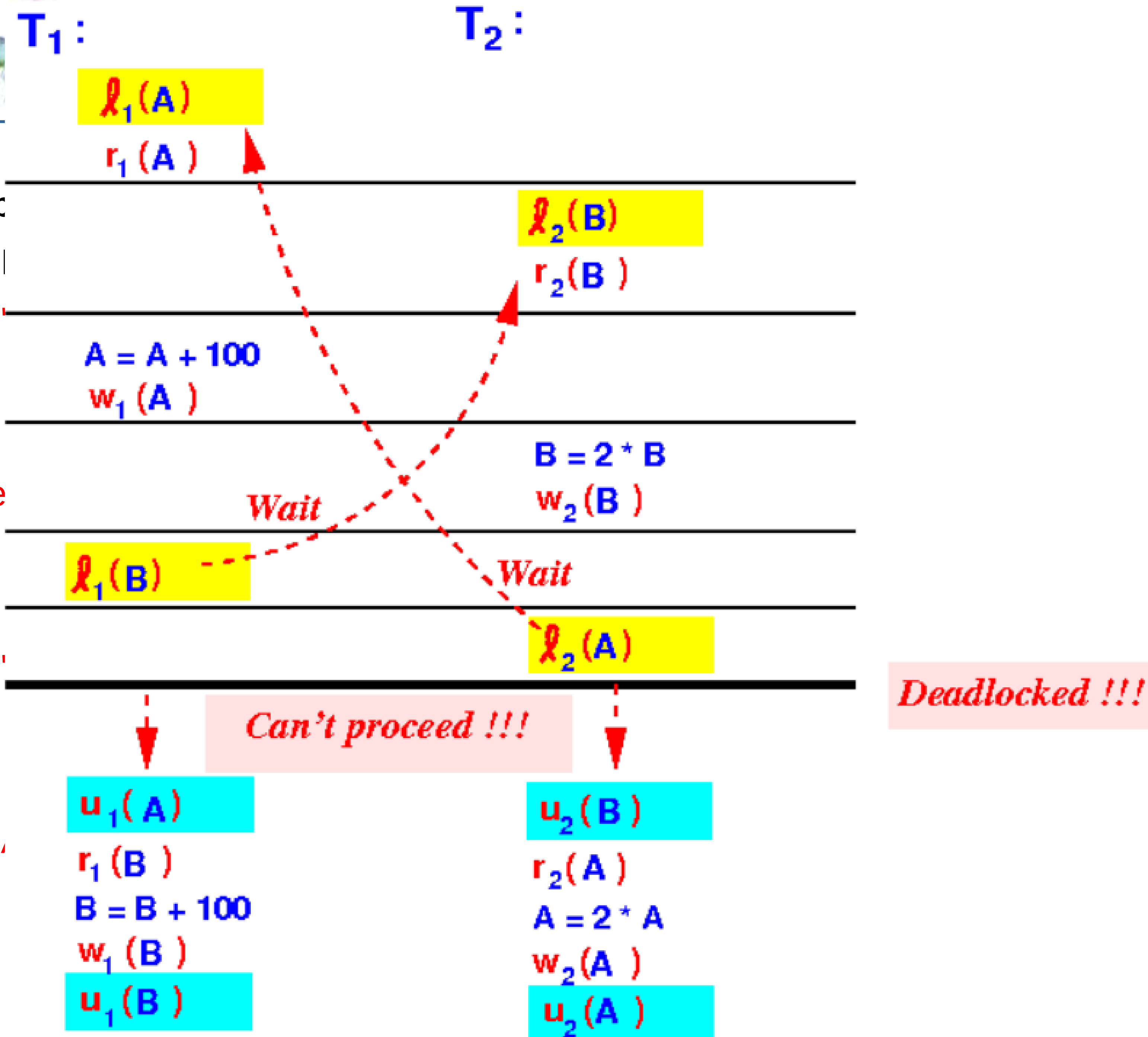
ler:

ck:

e order:

Top

Inte



T1 :
T2 :
l1(A)
r1(A)
l2(B)
r2(B)
A = A + 100
w1(A)
B = 2 * B
w2(B)
Wait
l1(B)
Wait
l2(A)
Can't proceed !!!
Deadlocked !!!
u1(A)
r1(B)
B = B + 100
w1(B)
u1(B)
u2(B)
r2(A)
A = 2 * A
w2(A)
u2(A)

ler:

Deadlock prevention

Topic: Deadlock prevention by ordering the resource requesting

- The cause of the deadlock:

> The transactions T1 and T2 have each locked the shared variables in reverse order:
>
> - T_1 locks **A ⇒ B**
> - T_2 locks **B ⇒ A**

- Preventing deadlock by imposing an ordering on the shared variables

  Example ordering:

  - Consider shared variables (resources) as a binary number

  Example ordering:

  - Processes must request locks on shared variables in the **ordering** of variables

Deadlock prevention

Topic: Deadlock prevention by ordering the resource requesting

- Preventing deadlock by imposing an ordering on the shared variables

  Example ordering:

  - Consider shared variables (resources) as a binary number

  Exam

  - Pr ... **ng** of
    va

A process T wants to lock the shared resources:

Printer 567
Disk 789
File 123

Lock request order made by transaction T:

File 123
Printer 567
Disk 789

Deadlock prevention

- **Theorem: Ordered locking on shared resources will prevent deadlock**

  Claim:

  If all processes request locks on shared resources in the order of the resources, then **deadlock is not possible.**

Proof: by contradiction:

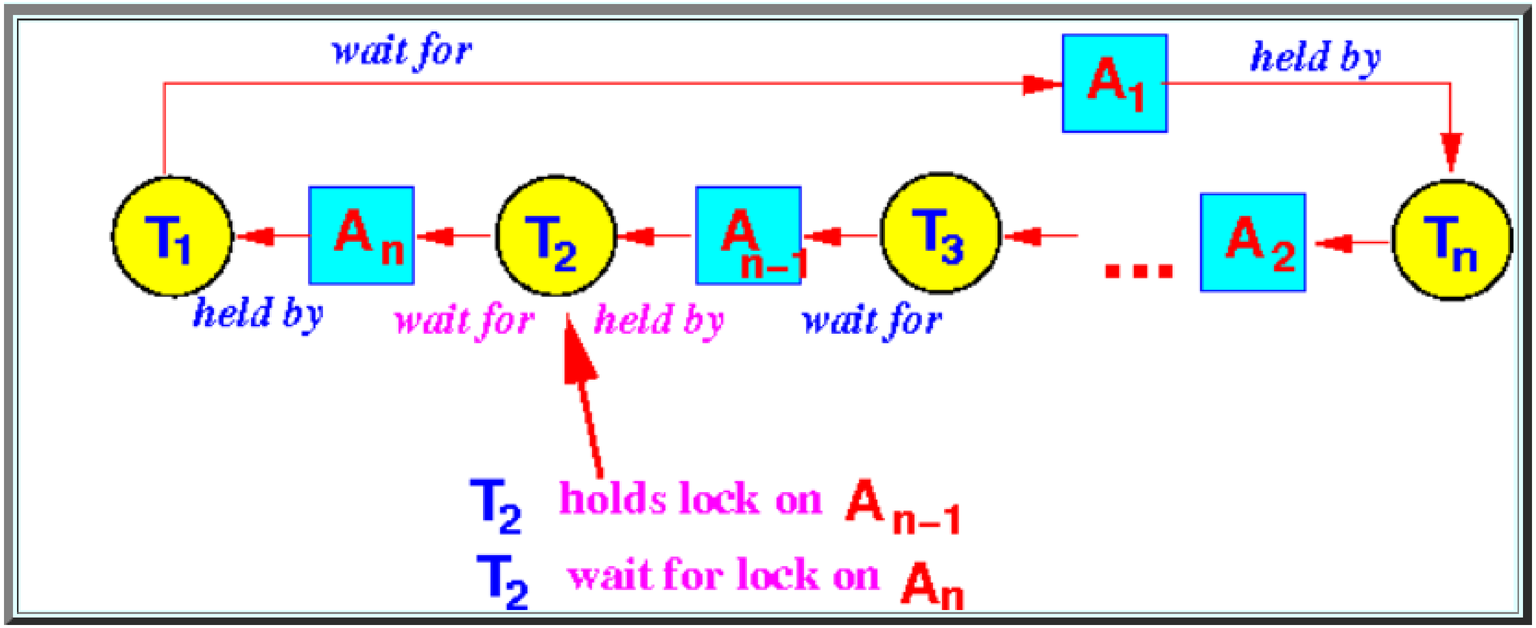
Suppose that:

- Processes T1, T2, ..., Tn request locks on shared resources according to the ordering of the database elements and
- Transactions T1, T2, ..., Tn are deadlocked

Deadlock prevention

Without lost of generality, let's assume that

The **wait-for graph**

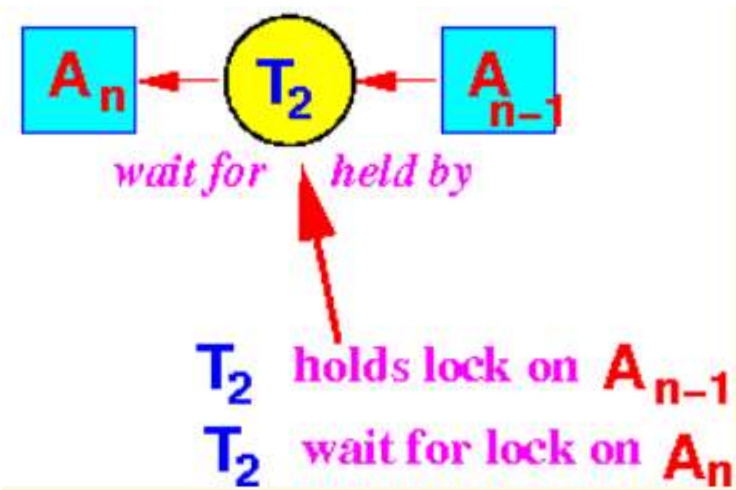


- T1 has locked some variable An and T2 is waiting for a lock on An
- T2 has locked some variable An-1 and T3 is waiting for a lock on An-1
- T3 has locked some variable An-2 and T4 is waiting for a lock on An-2
- ...
- Tn−1 has locked some variable A2 and Tn is waiting for a lock on A2
- Tn has locked some variable A1 and T1 is waiting for a lock on A1

Deadlock prevention

Consider the shared variables that **process T2 locks**:

- T2 has locked the variable An-1 (first) and
- T2 is waiting (trying to lock) for the variable An (next)



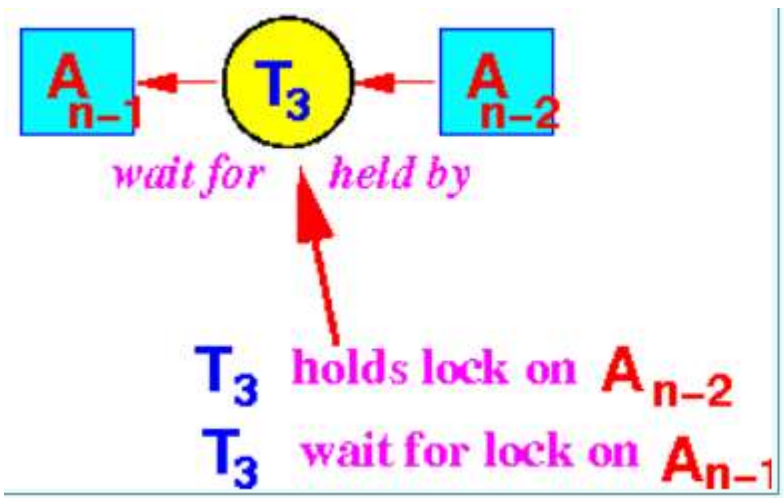
Since process T2 will lock variable according to their **order**, we conclude that:

- $A_{n-1} < A_n$

Deadlock prevention

Consider the shared variables that **process T3 locks**:

- T3 has locked the variable An-2 (first) and
- T3 is waiting (trying to lock) for the variable An-1 (next)



Since process T3 will lock variable according to their **order**, we conclude that:

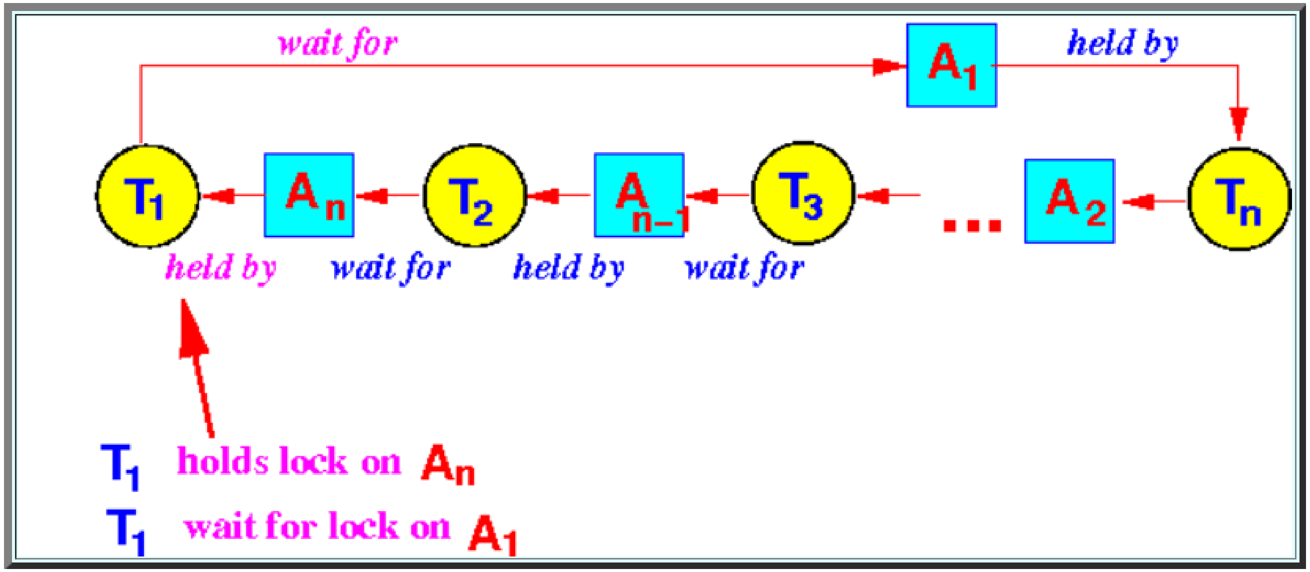
- $A_{n-2} < A_{n-1}$

Therefore, we have that:

- $A_1 < A_2 < \ldots\ldots < A_{n-1} < A_n$

Deadlock prevention

But we also have:



- T_1 has **locked** the A_n **and**
- T_1 is *waiting* for the A_1

we conclude that:

- $A_n < A_1$

Deadlock prevention

Therefore, we have that:

- $A_1 < A_2 < \ldots\ldots < A_{n-1} < A_n$

we conclude that:

- $A_n < A_1$

- $A_n < A_1 < A_2 < \ldots\ldots < A_{n-2} < A_{n-1} < A_n$

And we have a contradiction:

- $A_n < A_n$

Therefore:

- The assumption that n processes can be involved in a deadlock is wrong
- So deadlock is not possible

Deadlock prevention

Preventing deadlock with timestamps: the wait-die method

Wait-Die Method

- **Timestamped** processes:
  - Each process is assigned a unique increasing timestamp
    - $T_1, T_2, T_3, \ldots$
  - Important fact:

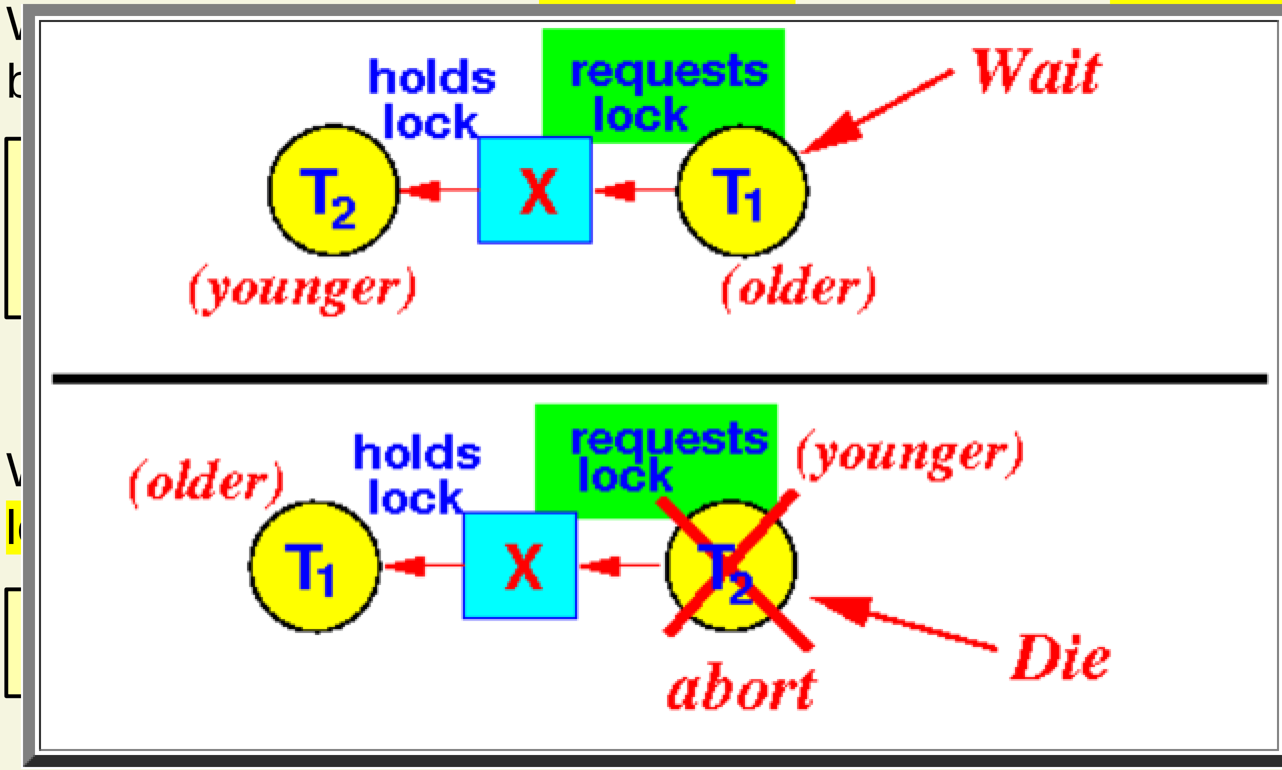
    An earlier process receives a smaller timestamp

- Purpose of the time stamp

To give older processes (= processes with smaller timestamps) a **higher priority** over "younger" processes

Wait-Die Method

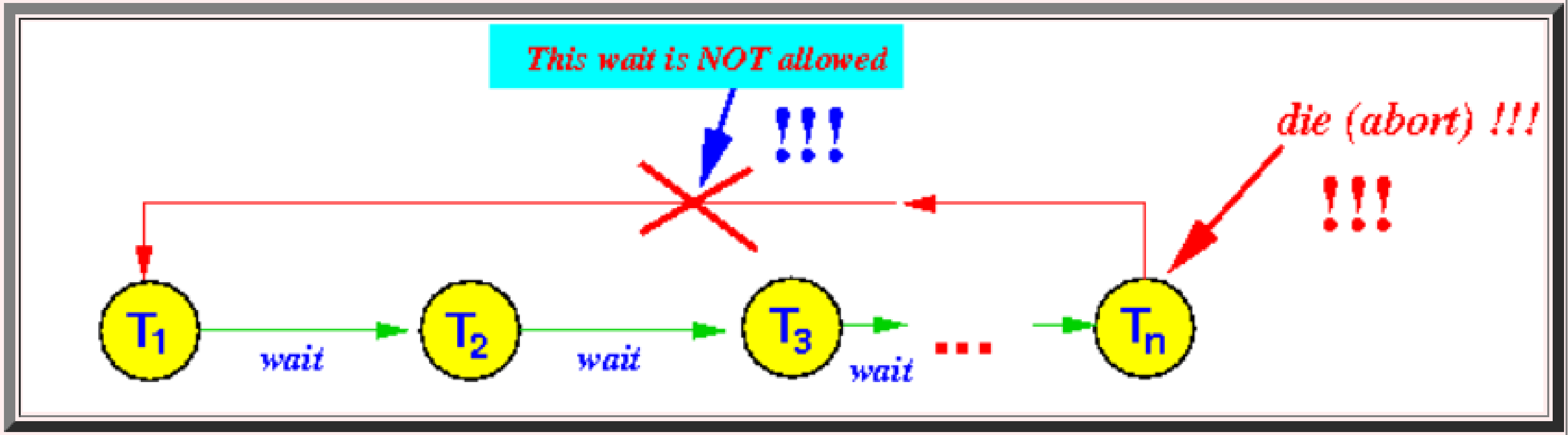
- The **Wait-die** locking rule:



Wait-Die Method

- The **Wait-die** locking rule will prevent the development of deadlock because:

  - T_n cannot wait for T_1:



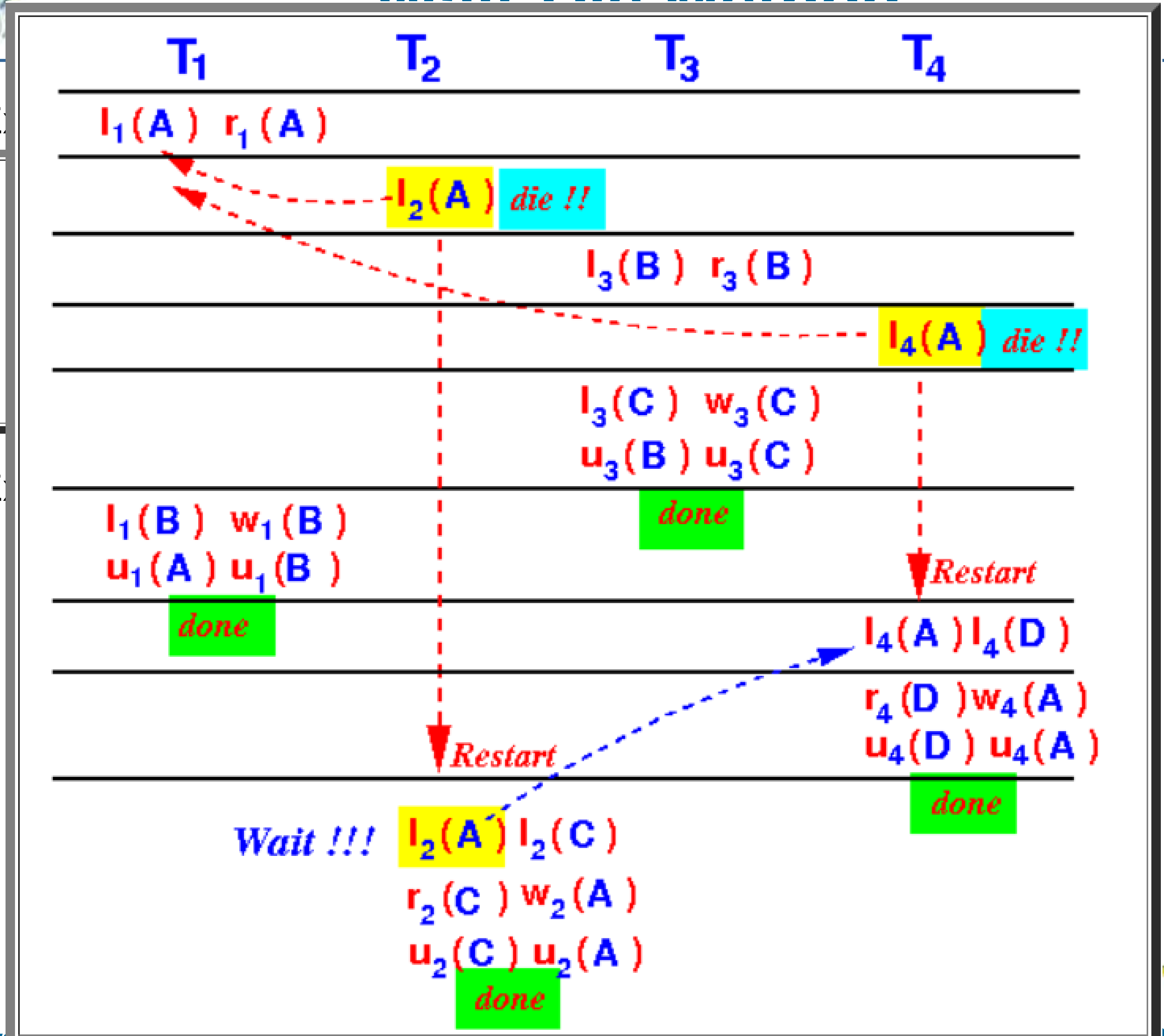
So there cannot be a cycle in the wait-for graph:

- No deadlock possible !!!

Wait-Die Method

- Ex

- Ex



Wait-Die Method

- Important note:

  - When a process is **aborted** and **restarts**

    The process will retain its (old) timestamp !!!

  - Therefore:

    Eventually, a process will become the "oldest" process and will complete execution !!!

Revisit: banker's algorithm

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

What are safe sequences?

Revisit: banker's algorithm

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

1. The Content of the need matrix can be calculated by using the formula given below:

Need = Max – Allocation

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

Revisit: banker's algorithm

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

For process P0, Need = (3, 2, 1) and
Available = (2, 1, 0)
Need <=Available = False
So, the system will move to the next process.

Revisit: banker's algorithm

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

For Process P1, Need = (1, 1, 0)
Available = (2, 1, 0)
Need <= Available = True
Request of P1 is granted.
Available = Available +Allocation
= (2, 1, 0) + (2, 1, 2)
= (4, 2, 2) (New Available)

Revisit: banker's algorithm

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

For Process P2, Need = (5, 0, 1)
Available = (4, 2, 2)
Need <=Available = False
So, the system will move to the next process.

Revisit: banker's algorithm

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

For Process P3, Need = (7, 3, 3)
Available = (4, 2, 2)
Need <=Available = False
So, the system will move to the next process.

Revisit: banker's algorithm

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

For Process P4, Need = (0, 0, 0)
Available = (4, 2, 2)
Need <= Available = True
Request of P4 is granted.
Available = Available + Allocation
= (4, 2, 2) + (1, 1, 2)
= (5, 3, 4) now, (New Available)

Revisit: banker's algorithm

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

Now again check for Process P2, Need = (5, 0, 1)
Available = (5, 3, 4)
Need <= Available = True
Request of P2 is granted.
Available = Available + Allocation
= (5, 3, 4) + (4, 0, 1)
= (9, 3, 5) now, (New Available)

Revisit: banker's algorithm

Processes	Allocation A B C	Max A B C	Available A B C
P0	1 1 2	4 3 3	2 1 0
P1	2 1 2	3 2 2	
P2	4 0 1	9 0 2	
P3	0 2 0	7 5 3	
P4	1 1 2	1 1 2	

Process	Need		
	A	B	C
P_0	3	2	1
P_1	1	1	0
P_2	5	0	1
P_3	7	3	3
P_4	0	0	0

The system allocates all the needed resources to each process. So, we can say that the system is in a safe state.

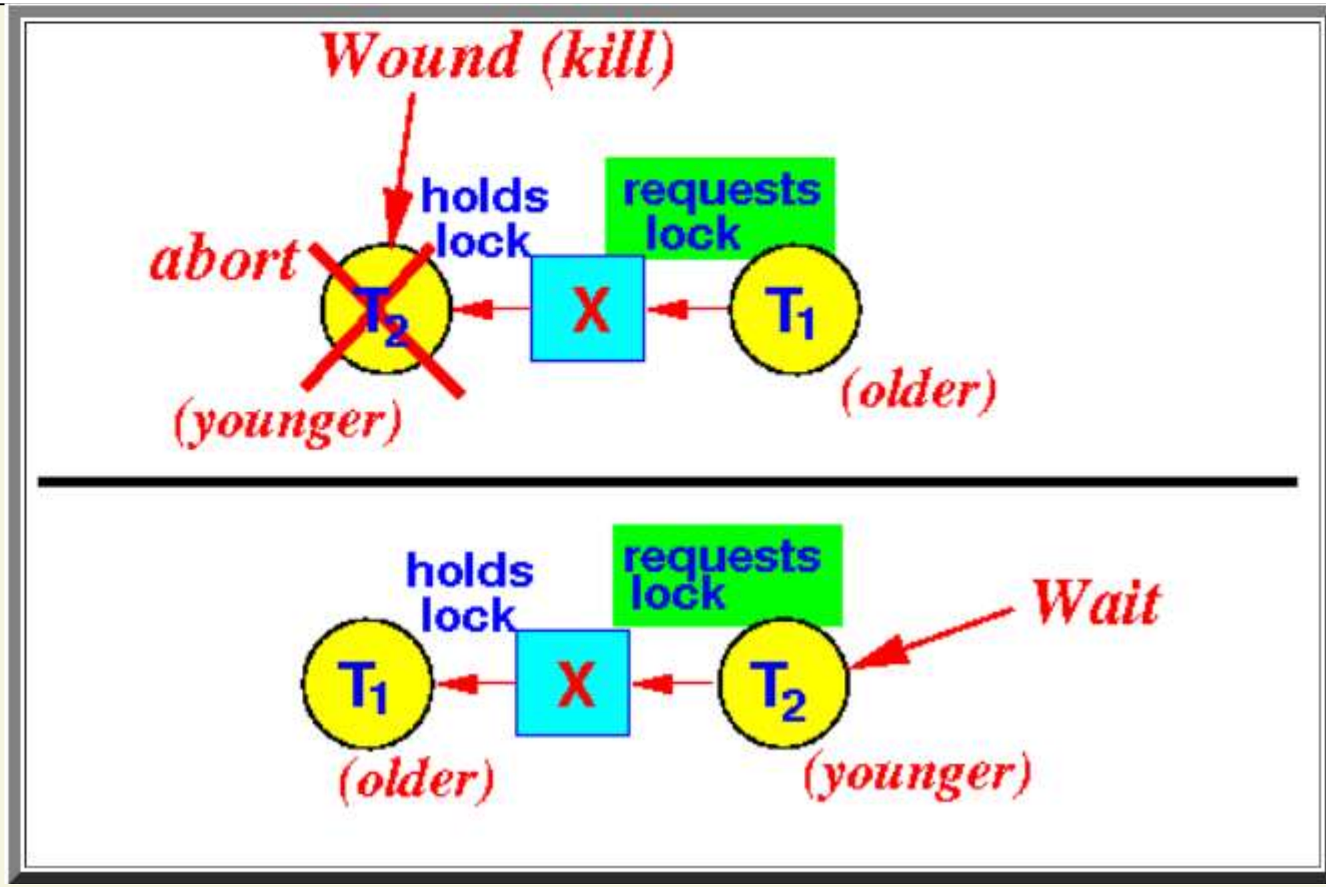
Now again check for Process P0, = Need (3, 2, 1)
= Available (9, 5, 5)
Need <= Available = True
So, the request will be granted to P0.
Safe sequence: < P1, P4, P2, P3, P0>

Deadlock prevention

Preventing deadlock with timestamps: the wound-wait method

Wound-Wait Method

- The **Wound-wait** locking rule:
  - locked
  - n
  - the



Wound-Wait Method

- The **Wound-wait** locking rule will prevent (forbid) the development of deadlock because:

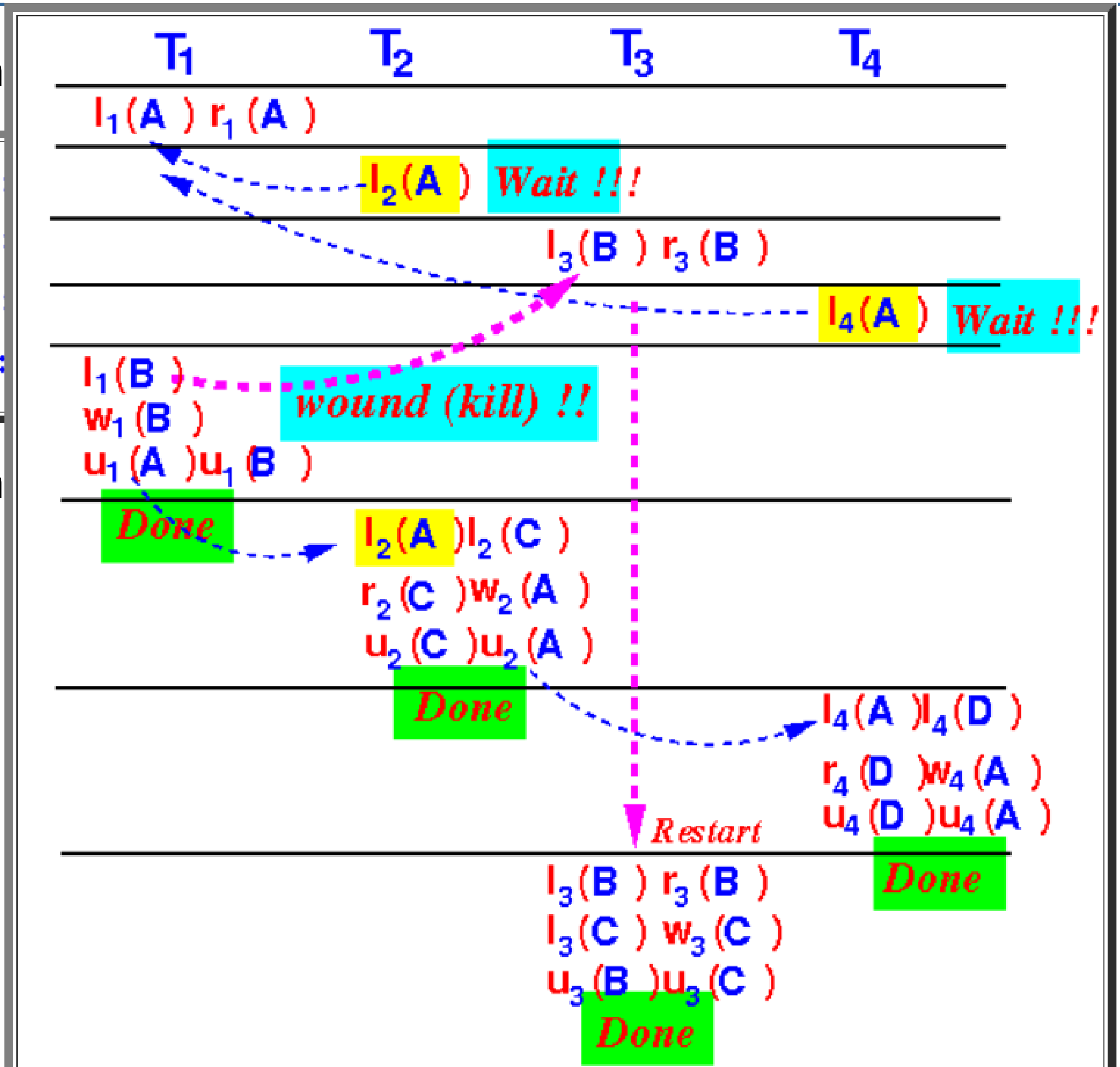
  - T_1 will **wound (= kill/abort)** the (younger) T_n:

(The **wait** ***direction*** in **wound-wait** is **opposite** from a **wait-die locking rule** !!!)

- In both schemes (wait-die and wound-wait), the younger process will get **aborted**
- The ***older*** **transaction** is ***not*** be **aborted** !!!

Wound-Wait Method

- Exam
- Exam



Deadlock prevention

Comparing the wait-die and wound-wait schemes

Comparing

- Similarity:
  - In both the wait-die and the wound-wait scheme

    The older process will "win" over the younger process

  - Both schemes are fair

    When processes restart, they keep their timestamps

    Eventually, the aborted (younger) processes will become the oldest processes in the system

Comparing

- Difference:

  - Wait-die:

    The younger processes are killed when:

    It (= the younger process) makes a request for a lock being held by an older process

  - Wound-die:

    The younger processes are killed when:

    An older process makes a request for a lock being held by the younger processes

Comparing

- The cost of aborting (younger/older) transactions:

  - Younger processes (having just started running) will typically:

    hold fewer number of locks and

    has read/written fewer number of variables

  - Older processes (having run longer) will typically

    hold higher number of locks and

    has read/written higher number of variables

  Older processes (= that started earlier) are more expensive (costly) to be aborted !!!!

Comparing

Which method has higher aborting rate?

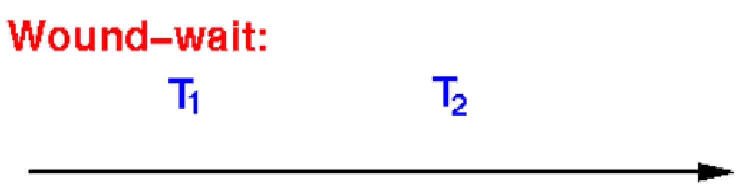
Comparing

- Comparing the **abort rates** of the wait-die and would-wait schemes:

  > - The number of processes that will be aborted in the wound-wait method will be
  >
  >   lower than number of processes that will be aborted in the wait-die method

Reasons:

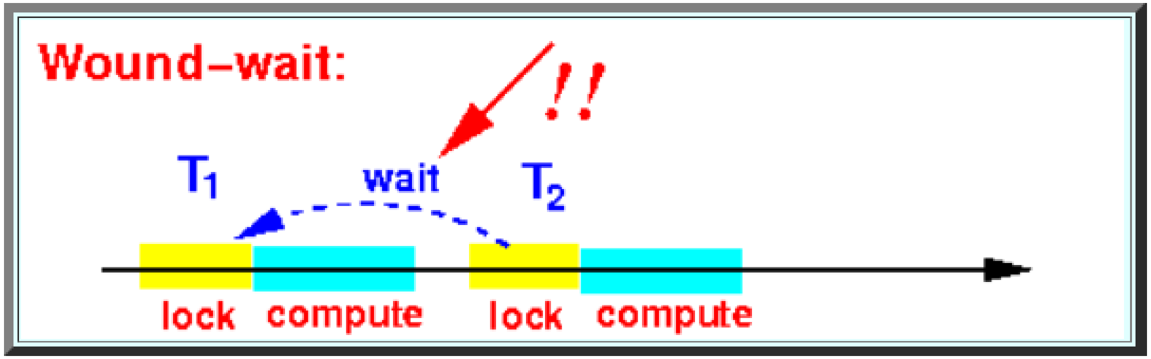
- A younger process starts later than an older process:



Comparing

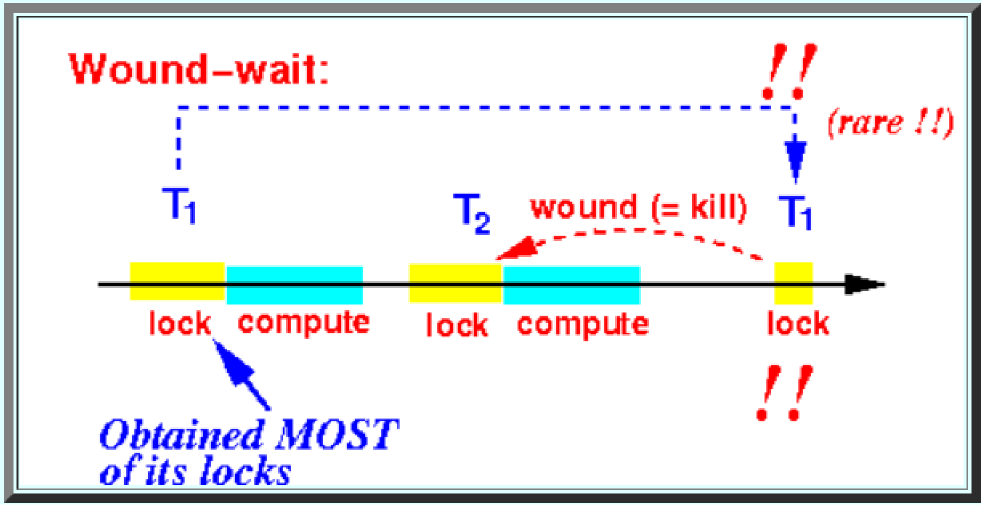
- By the time that the younger process T2 starts, the older process T1 will have obtained most of their locks:

- A younger (just starting) process T2 will make many lock requests !!
- In the wound-wait method, a younger (just starting) process T2 will wait (and not abort) if T2 requests a lock held by the older process T1:



Comparing

- An older process (T1) that requests a lock can **abort** a younger process (T2)



However:

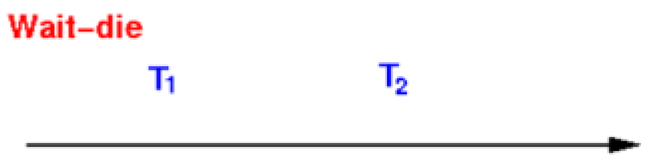
- The event that older processes (T1) will request a lock is **rare**
- (because older processes (T1) have obtained most of their locks already)

Therefore:

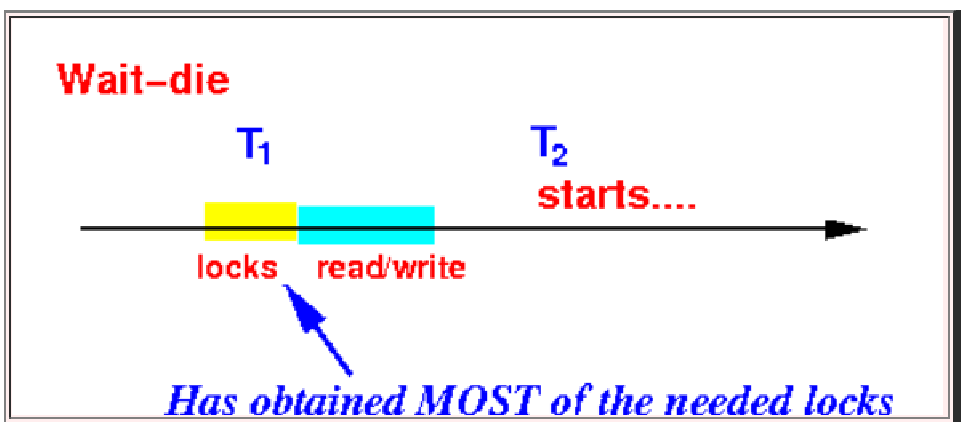
- The number of aborts in wound-wait is relatively low

Comparing

- Wait-die method will have more aborts:
- A younger process starts later than an older process:



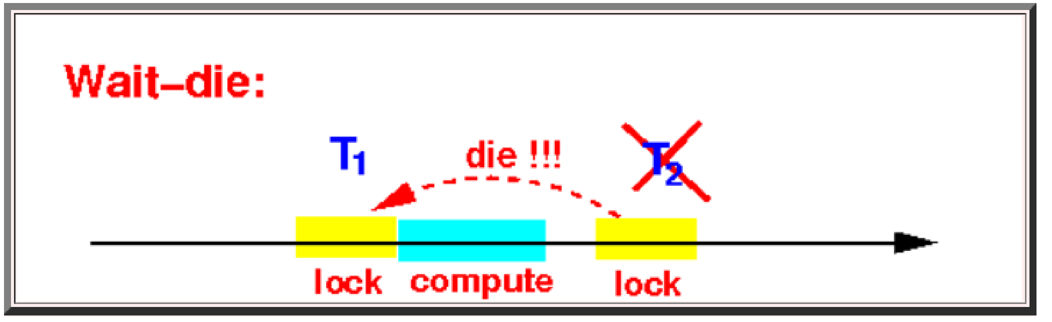
- By the time that T2 starts, the older process T1 will have obtained most of their locks:



- A younger (just starting) process T2 will make many lock requests !!

Comparing

- In the wait-die method, a younger (just starting) process T2 will die (= abort) if T2 requests a lock held by the older process T1:



- Since younger processes makes many lock requests:

  - There will be **more aborts** (of young processes) in the wait-die method

- Comment:

  - It looks like that the wound-wait method will have better performance...

Comparing

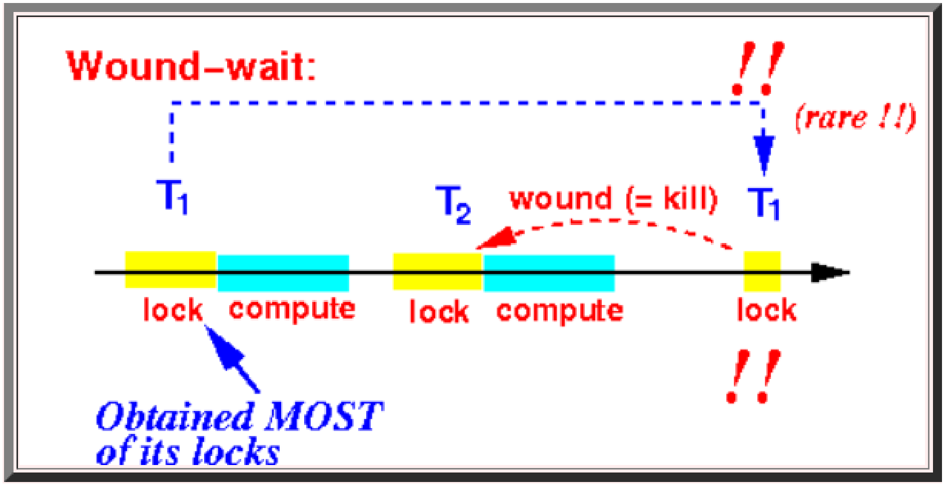
- Comparing the amount of waste work in **aborted process** in wait-die and wound-wait

- The wound-wait method may have fewer aborts, however; it's likely that:

  - An aborted (younger) process in the wound-wait method has performed more work than an aborted (younger) process in the wait-die method !!!

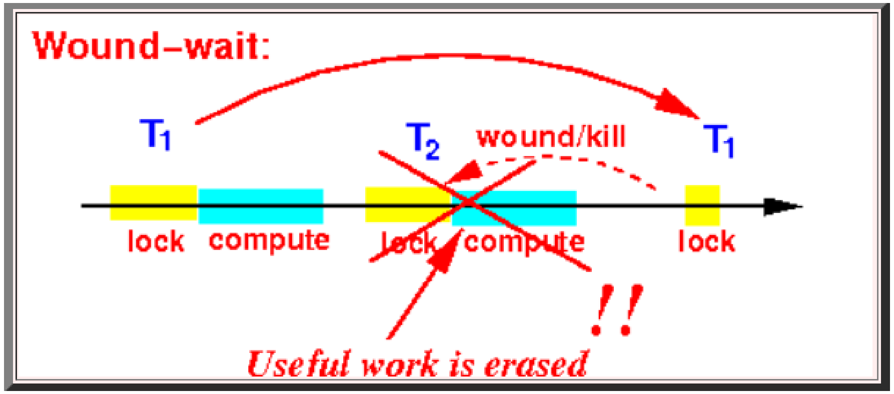
Comparing

- Consider a process in the wound-wait method that gets aborted:



Notice that the aborted process **must** hold some locks !!!

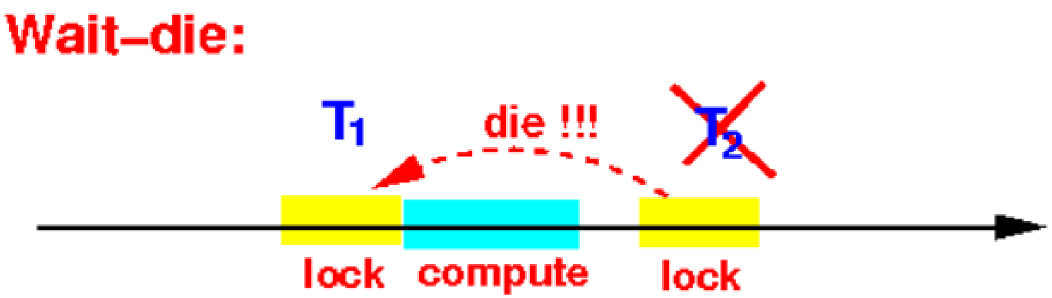
- I.e.: the aborted process has performed some read/write operations (= useful work):



The useful work is erased (rolled back) !!!

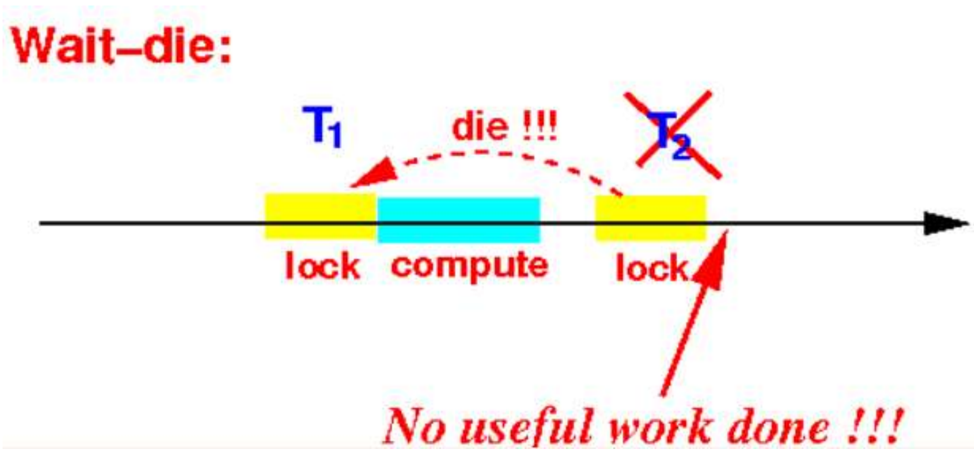
Comparing

- Consider a process in the wait-die method that gets aborted:



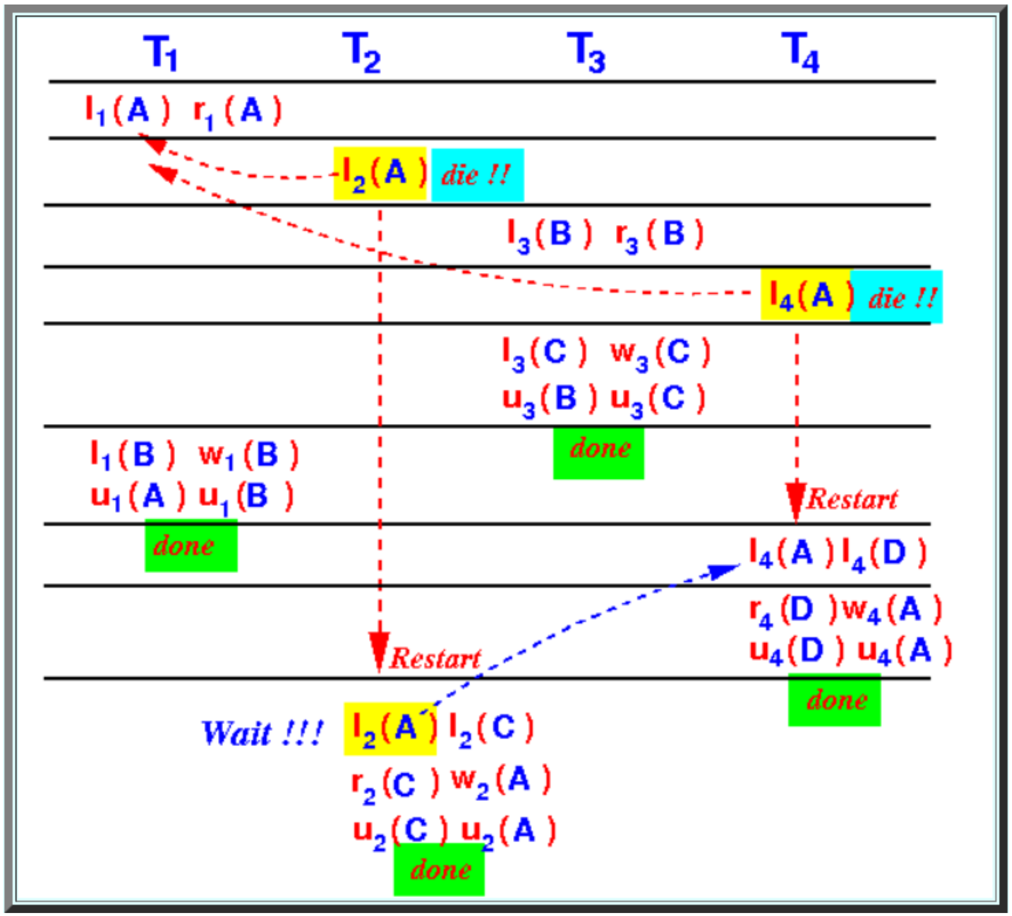
Notice that the aborted process may **not** hold any lock at all !!!

- I.e.: the aborted process has **not** performed any read/write operations (= no useful work):



Comparing

- Consider the previous examples:
- **Wait-die** locking method



There are 2 transaction aborts (that's 50% of the # processes !!!)

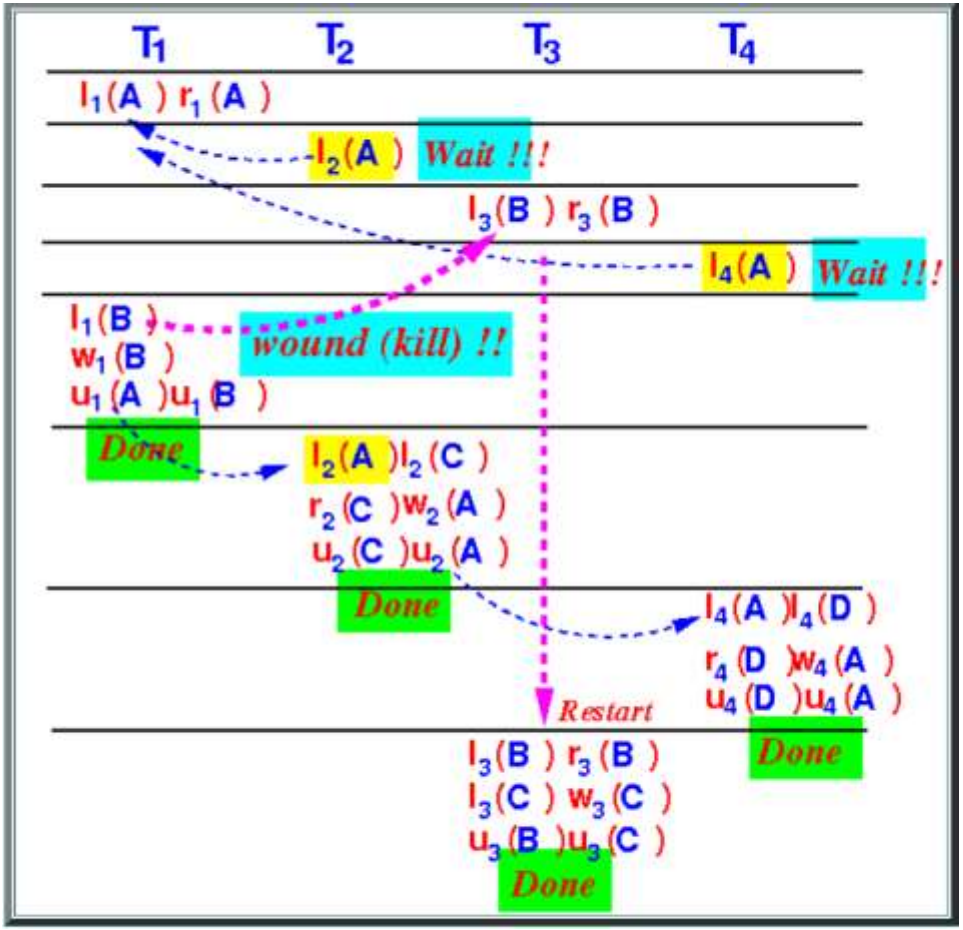
However:

Both aborted processes has done no useful work !!!

(The aborted processes has **not** performed any read/write operation !!!)

Comparing

- **Wound-wait** locking method



There is only 1 process aborts (less than in wait-die !)

However:

The aborted processes has done some useful work !!!

(Process T3 has read B !!!)

Comparing

- Summary
  - Wait-die has more roll backs; but the processes have performed little to no work
  - Wound-wait has less roll backs; but the processes that got aborted has done some work !!!

Deadlock Detection

Using timeout to detect deadlock

Deadlock detection

- Fact:
  - When a process T is involved in a deadlock, the transaction T will not finish

- Simple way to detect if a process is a candidate to be involved in a deadlock:
  - Set a **timer** for each operation
  - When the timer expires, the transaction is assumed to be involved in a deadlock

- Note:
  - This method is quite **reliable** because most operations (read/write) does not take very long !!!

Deadlock detection

- Example:
  - MySQL uses timeout to detect deadlocks

Experiment:

User 1	User 2

Don't do anything...	`START TRANSACTION;`
	
	`UPDATE employee`
	`SET salary=salary + 0`
	`WHERE lname='Smith';`
	
	`>> 1 row updated`

`START TRANSACTION;`
 	
`UPDATE employee`
`SET salary=salary + 0`
`WHERE lname='Smith';`
 	
`>> Update HANGS !!!`
 	
`(WAIT about 1 min TIME)`
 	
`ERROR 1205: Lock wait timeout exceeded;`
`try restarting transaction`
	Don't do anything...

- **Open** 2 windows and run `sqlroot` and log in as `cheung`:

- `use companyDB`
- **Employee:**

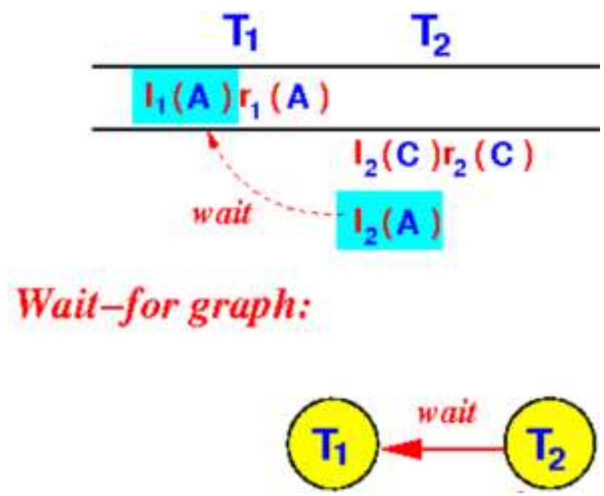
fname	lname	salary
John	Smith	20900.00
Frankl	Wong	50000.00
Alicia	Zelaya	25000.00
Jennif	Wallace	43000.00
Ramesh	Narayan	38000.00
Joyce	English	25000.00
Ahmad	Jabbar	25000.00
James	Borg	55000.00

Deadlock Detection

Detecting deadlocks using wait-for graphs

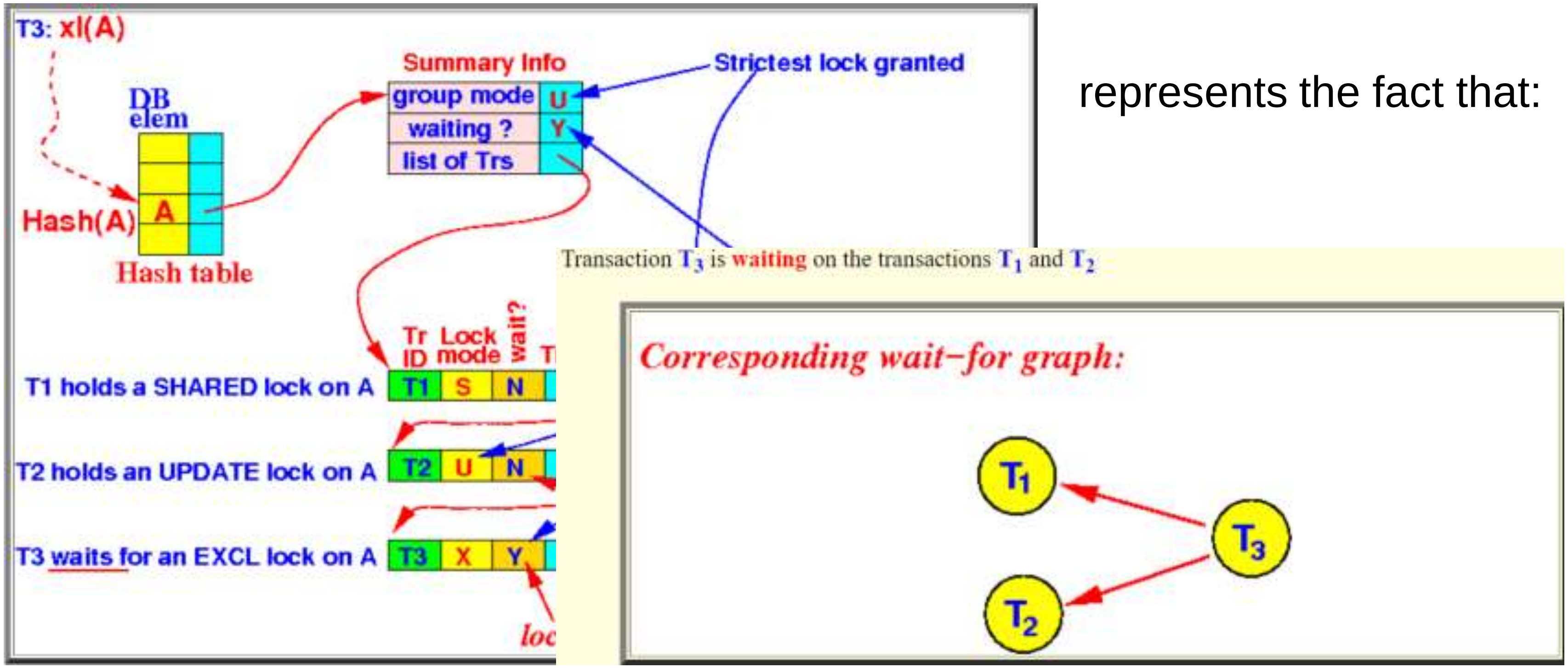
Deadlock detection

- Wait-for graph is a graph where::
  - Node represents a process
  - Edge i ⇒ j represents: the process i is waiting for a lock held by the process j



Deadlock detection

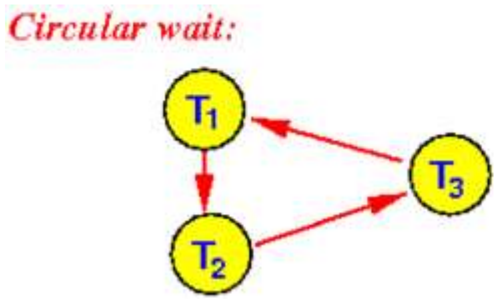
- The wait-for graph can be constructed using the information stored in the lock table entries



represents the fact that:

Deadlock detection

- Deadlock occurs in a circular wait situation

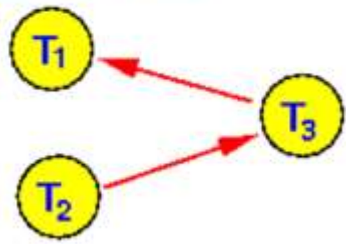


- Each process is waiting for some process to complete
- **Result: No transaction can proceed forward**

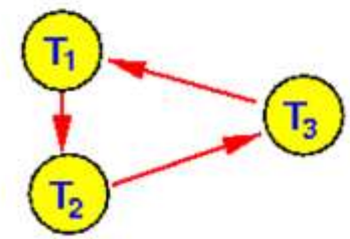
Deadlock detection

- If the wait-for graph has no cycles, then:
  - There is no deadlock

Wait–for graph with no cycle:



Circular wait:

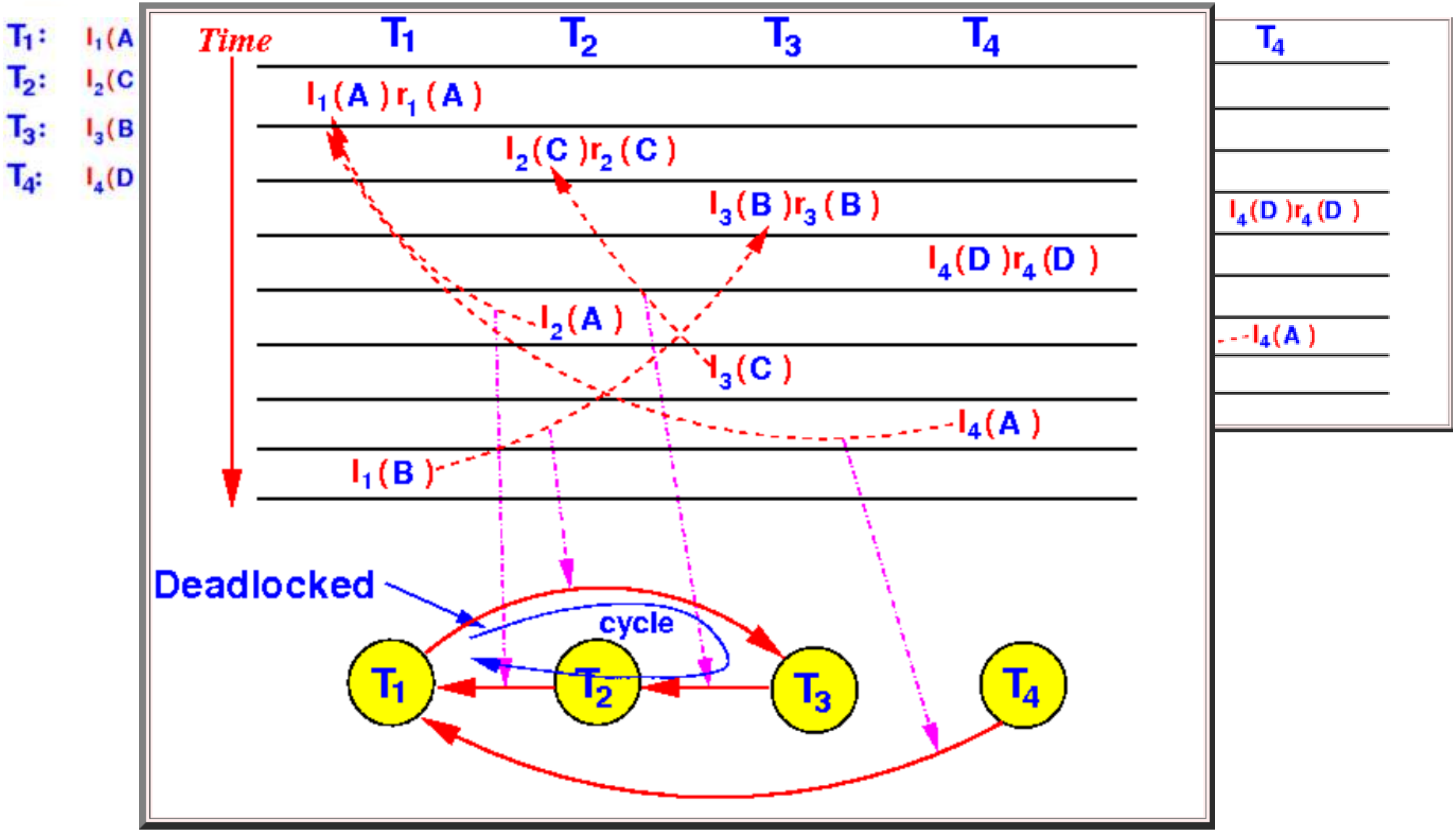


- Completion order:
  - T1 will complete first (and releases its locks)
  - Then T3 can obtain its locks after T1 unlocks and complete next (and releases its locks)
  - Finally, T2 can obtain its locks after T3 unlocks and complete next (and releases its locks)

Deadlock detection

- Deadlock detect in a scheduler
  - A process scheduler that uses locking to ensure serializability must
    - Maintain a wait-for graph on all processes
    - If the wait-for graph contains a cycle:
      - The scheduler will **abort** one of the processes in the cycle

Deadlock detection



The process manager will now have to abort some process T1, T2 and T3 that are involved in the deadlock

End of Chapter 7