R

بسم الله الرحمن الرحيم

سیستم عامل

جلسه دهم – تشخیص و جلوگیری از بن‌بست

Original slides by: Jonathan Walpole & Hadi Foroughmand Arabi – Some slides/images by Silberschatz, Galvin and Gagne

جلسه‌ی گذشته

مسائل هم‌روندی و مانیتور

System Model

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

Definition of Deadlock

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

- Usually, the event is the release of a currently held resource
- None of the processes can …
  - *Be awakened*
  - *Run*
  - *Release its resources*

Deadlock Conditions

- A deadlock situation can occur *if and only if* the following conditions hold simultaneously
  - ***Mutual exclusion*** *condition – resource assigned to one process only*
  - ***Hold and wait*** *condition – processes can get more than one resource*
  - ***No preemption*** *condition*
  - ***Circular wait*** *condition – chain of two or more processes (must be waiting for resource from next one in chain)*

Resource-Allocation Graph

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

- V is partitioned into two types:
  - *$T = \{T_1, T_2, \ldots, T_n\}$, the set consisting of all the threads 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 $T_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow T_i$

Dealing With Deadlock

- Ignore the problem
- Detect it and recover from it
- Dynamically avoid is via careful resource allocation
- Prevent it by attacking one of the four necessary conditions

جلسه‌ی جدید

بن‌بست

DEADLOCK PREVENTION

Deadlock Prevention

Prevent it by attacking one of the four necessary conditions

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

Deadlock Prevention (Cont.)

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

Deadlock Prevention

- Attacking mutual exclusion?
  - *Some resource types resource could be corrupted*
  - *May be OK if a resource can be partitioned*
- Attacking no preemption?
  - *Some resources could be left in an inconsistent state*
  - *May work with support of checkpointing and rollback of idempotent operations*

Deadlock Prevention

- Attacking hold and wait?
  - *Require processes to request all resources before they start*
  - *Processes may not know what they need ahead of time*
  - *When problems occur a process must release all its resources and start again*

Deadlock Prevention

- Attacking circular waiting?
  - *Number each of the resources*
  - *Require each process to acquire lower numbered resources before higher numbered resources*
  - *More precisely: A process is not allowed to request a resource whose number is lower than the highest numbered resource it currently holds*

Example

Thread A:

```
acquire (resource_1)
acquire (resource_2)
use resources 1 & 2
release (resource_2)
release (resource_1)
```

Thread B:

```
acquire (resource_2)
acquire (resource_1)
use resources 1 & 2
release (resource_1)
release (resource_2)
```

Assume that resources are ordered:

1. Resource_1
2. Resource_2
3. ...etc...

Example

Thread A:

```
acquire (resource_1)
acquire (resource_2)
use resources 1 & 2
release (resource_2)
release (resource_1)
```

Thread B:

```
acquire (resource_2)
acquire (resource_1)
use resources 1 & 2
release (resource_1)
release (resource_2)
```

- Assume that resources are ordered:
- 1. Resource_1
- 2. Resource_2
- 3. ...etc...
- Thread B violates the ordering!

Resource Ordering

- Assume deadlock has occurred.
- Process A
  - *holds X*
  - *requests Y*
- Process B
  - *holds Y*
  - *requests Z*
- Process C
  - *holds Z*
  - *requests X*

Resource Ordering

- Assume deadlock has occurred.
- Process A
  - -- *holds X*
  - -- *requests Y*

$X < Y$

- Process B
  - -- *holds Y*
  - -- *requests Z*
- Process C
  - -- *holds Z*
  - -- *requests X*

Resource Ordering

- Assume deadlock has occurred.
- Process A
  - -- *holds X*
  - -- *requests Y*

$X < Y$

- Process B
  - -- *holds Y*
  - -- *requests Z*

$Y < Z$

- Process C
  - -- *holds Z*
  - -- *requests X*

Resource Ordering

- Assume deadlock has occurred.
- Process A
  - -- *holds X*
  - -- *requests Y*

$X < Y$

- Process B
  - -- *holds Y*
  - -- *requests Z*

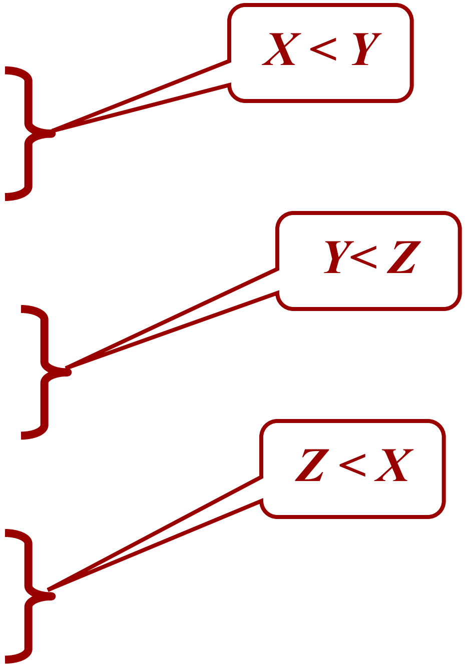
$Y < Z$

- Process C
  - -- *holds Z*
  - -- *requests X*

$Z < X$

Resource Ordering

- Assume deadlock has occurred.
- Process A
  - -- *holds X*
  - -- *requests Y*
- Process B
  - -- *holds Y*
  - -- *requests Z*
- Process C
  - -- *holds Z*
  - -- *requests X*



$X < Y$

$Y < Z$

$Z < X$

This is impossible!

Resource Ordering

- The chief problem:
  - *It may be hard to come up with an acceptable ordering of resources!*

Starvation

- Starvation and deadlock are different
  - *With deadlock – no work is being accomplished for the processes that are deadlocked, because processes are waiting for each other. Once present, it will not go away.*
  - *With starvation – work (progress) is getting done, however, a particular set of processes may not be getting any work done because they cannot obtain the resource they need*

تشخیص و بازیابی بن‌بست

تشخیص اینکه الآن بن‌بست هست یا نه.
نه اینکه امکان بن‌بست داریم یا نداریم.

Deadlock Detection

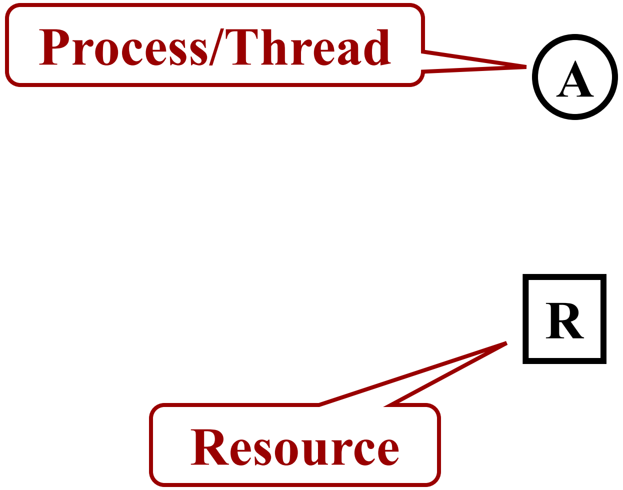
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

قدم ۱: ساده‌سازی مسئله

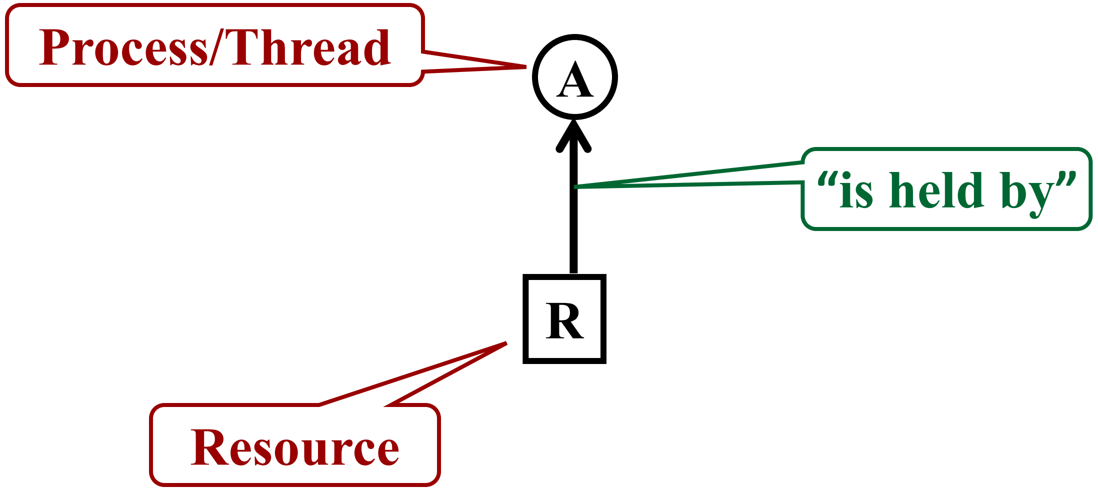
ساده‌سازی

- فعلا فرض کنیم از هر نوع منبع فقط یکی داریم.

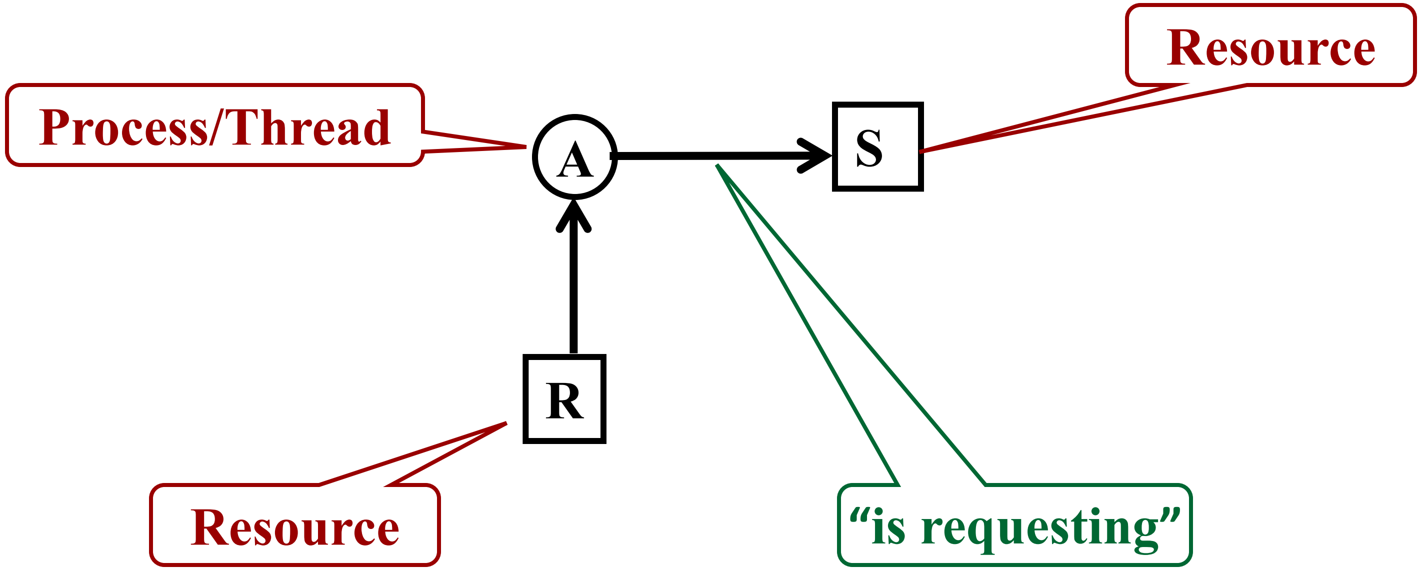
Resource Allocation Graphs



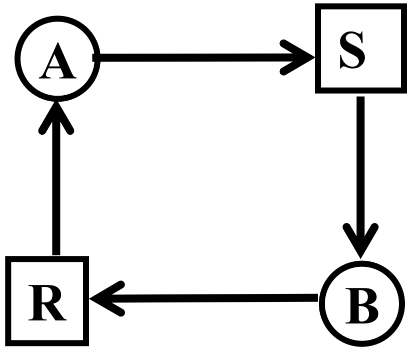
Resource Allocation Graphs



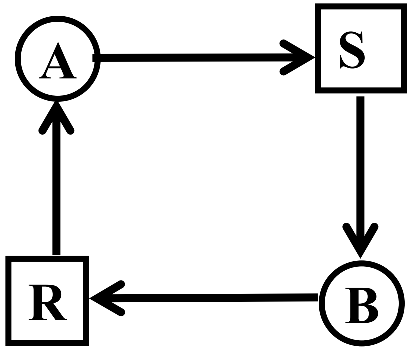
Resource Allocation Graphs



Resource Allocation Graphs

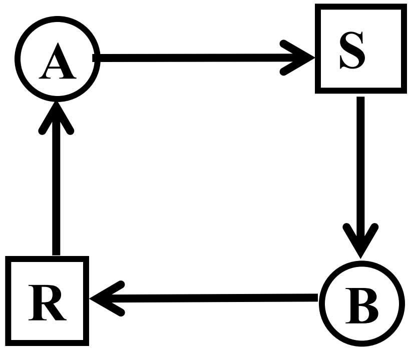


Resource Allocation Graphs



Deadlock

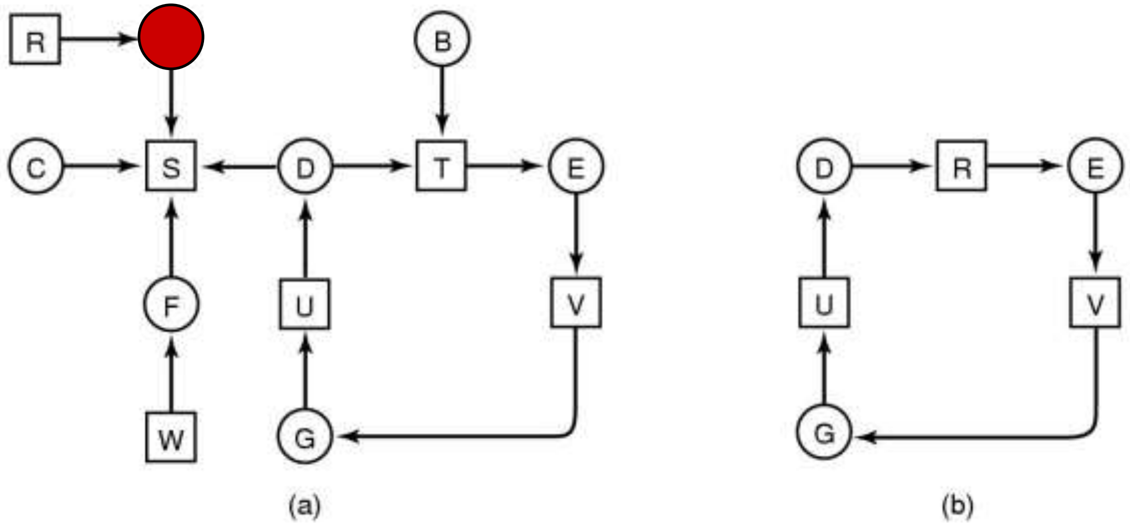
Resource Allocation Graphs



Deadlock = a cycle in the graph

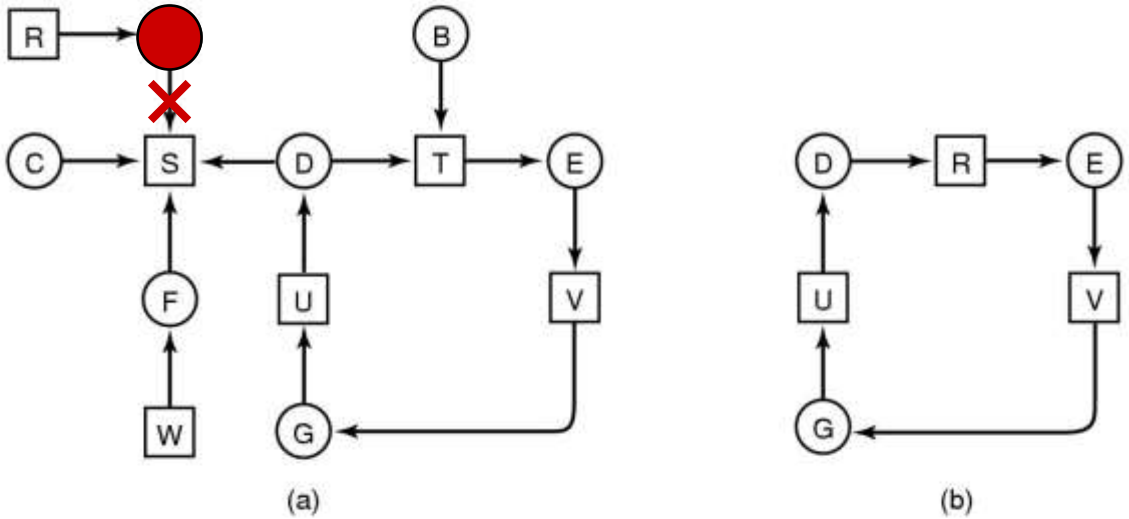
Deadlock Detection

- Do a depth-first-search on the resource allocation graph



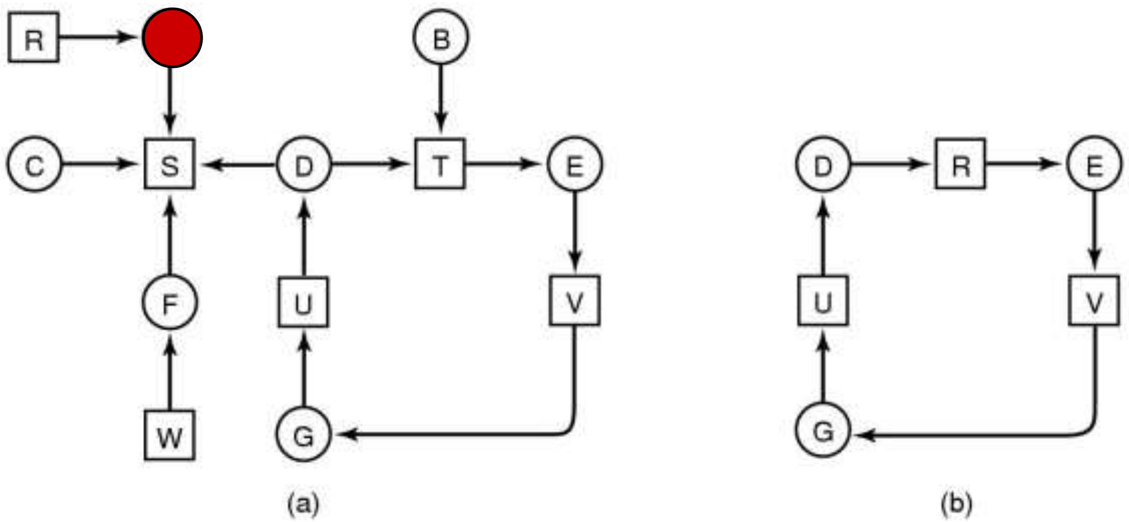
Deadlock Detection

- Do a depth-first-search on the resource allocation graph



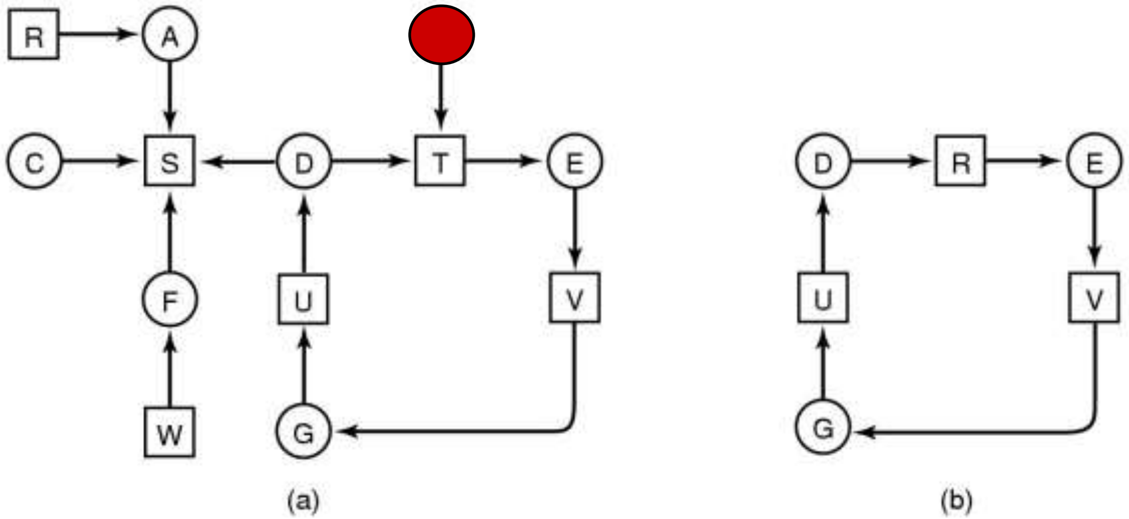
Deadlock Detection

- Do a depth-first-search on the resource allocation graph



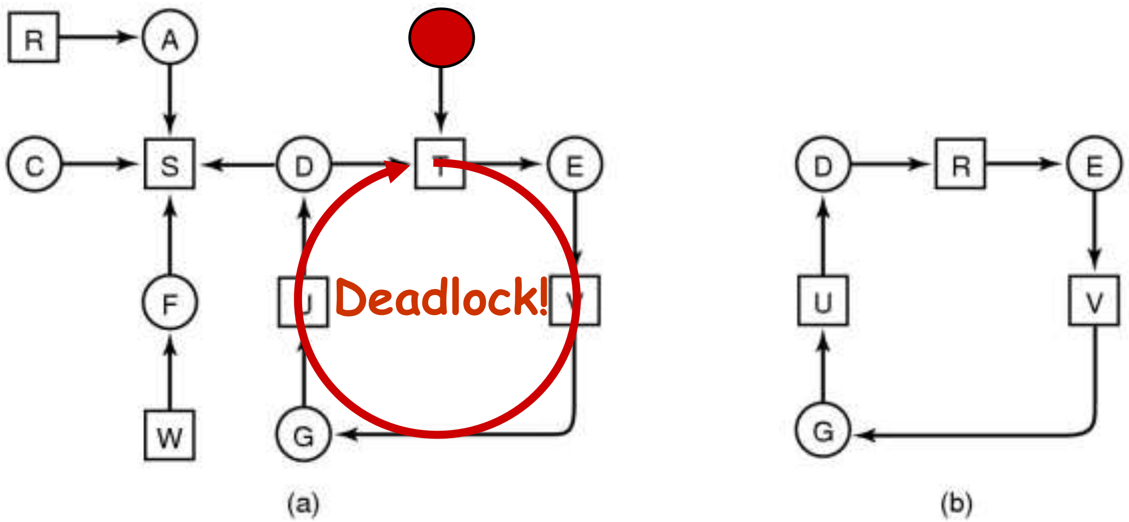
Deadlock Detection

- Do a depth-first-search on the resource allocation graph



Deadlock Detection

- Do a depth-first-search on the resource allocation graph



تشخیص بن‌بست

حذف فرض اضافه‌ی داشتن یک نسخه از منبع

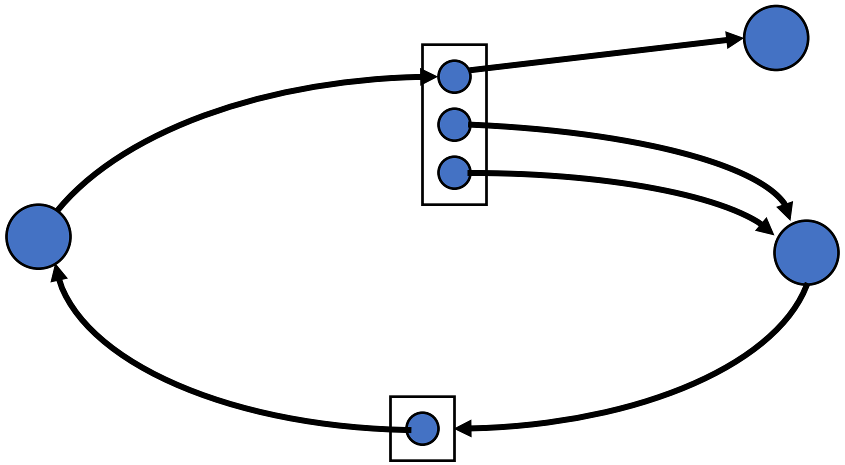
Multiple Instances of a Resource

- Some resources have only one instance
  - *i.e. a lock or a printer*
  - *Only one thread at a time may hold the resource*
- Some resources have several instances
  - *i.e. Page frames in memory*
  - *All units are considered equal; any one will do*

Multiple Instances of a Resource

- **Theorem:** *If a graph does not contain a cycle then no processes are deadlocked*
  - *A cycle in a RAG is a necessary condition for deadlock*
  - *Is it a sufficient condition?*

Multiple Instances of a Resource



Deadlock Detection Issues

- How often should the algorithm run?
  - *On every resource request?*
  - *Periodically?*
  - *When CPU utilization is low?*
  - *When we suspect deadlock because some thread has been asleep for a long period of time?*

Several Instances of a Resource Type

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

Detection Algorithm

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

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

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

Detection Algorithm (Cont.)

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

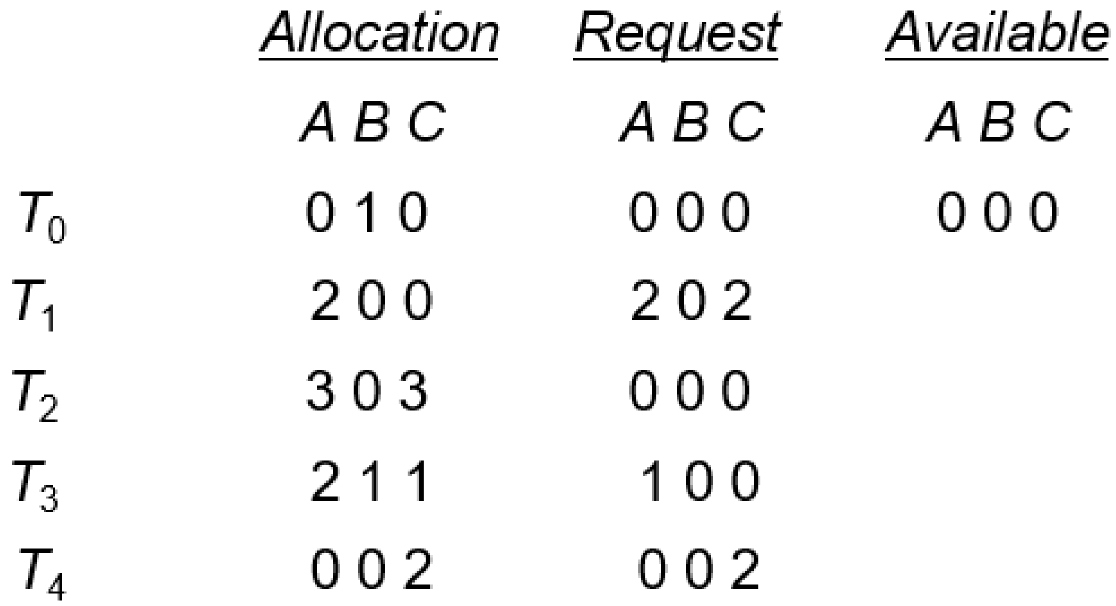
4. If ***Finish[i] == false***, for some ***i***, $1 \leq \boldsymbol{i} \leq \boldsymbol{n}$, then the system is in deadlock state. Moreover, if ***Finish***[*i*] **==** ***false***, then $\boldsymbol{T}_i$ is deadlocked

Detection Algorithm (Cont.)

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

Example of Detection Algorithm

- Five threads $\boldsymbol{T_0}$ through $\boldsymbol{T_4}$; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time $\boldsymbol{T_0}$:



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

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

Example (Cont.)

- $\boldsymbol{T_2}$ requests an additional instance of type ***C***

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

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

Deadlock Detection Issues

- How often should the algorithm run?
  - *On every resource request?*
  - *Periodically?*
  - *When CPU utilization is low?*
  - *When we suspect deadlock because some thread has been asleep for a long period of time?*

Recovery From Deadlock

- If we detect deadlock, what should be done to recover?
  - *Abort deadlocked processes and reclaim resources*
  - *Abort one process at a time until deadlock cycle is eliminated*
- Where to start?
  - *Lowest priority process?*
  - *Shortest running process?*
  - *Process with fewest resources held?*
  - *Batch processes before interactive processes?*
  - *Minimize number of processes to be terminated?*

Deadlock Recovery

- How do we prevent resource corruption
  - *For example, shared variables protected by a lock?*
- Recovery through preemption and rollback
  - *Save state periodically (ie. at start of critical section)*
  - *Take a checkpoint of memory*
  - *Start computation again from checkpoint*
  - *Can also make long-lived computation systems resilient*

DEADLOCK AVOIDANCE

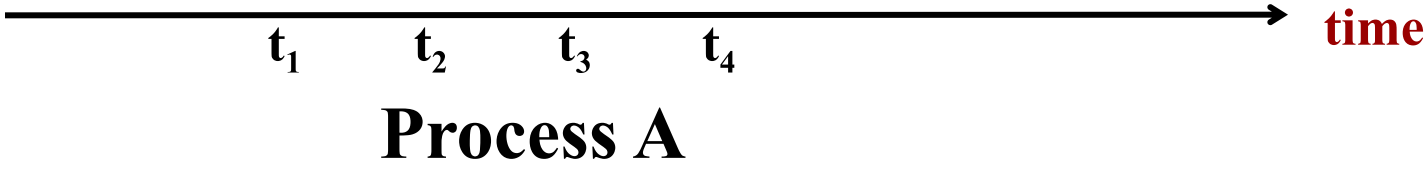
Deadlock Avoidance

- Detection – *optimistic* approach
  - *Allocate resources*
  - *Break system to fix the problem if necessary*

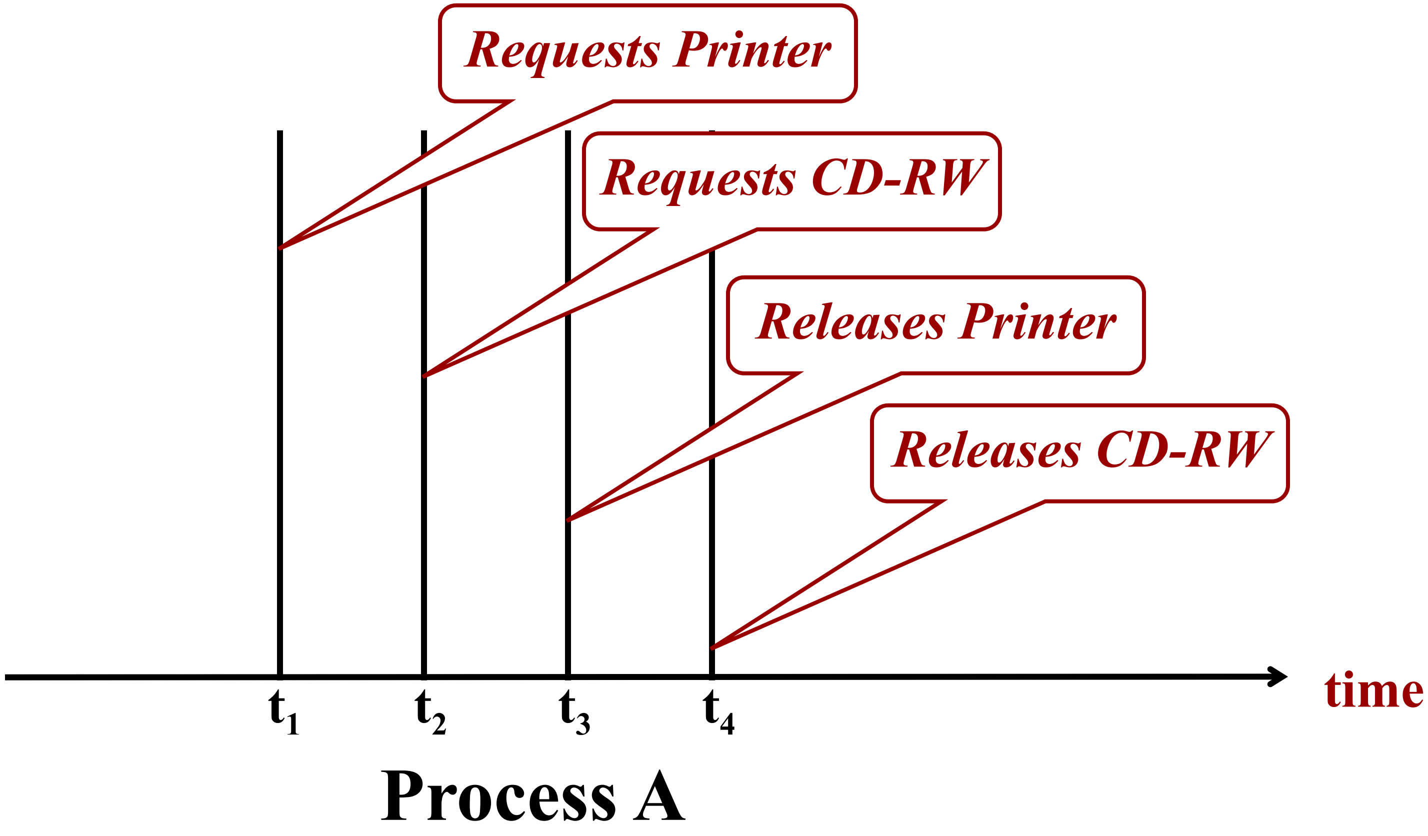
- Avoidance – *pessimistic* approach
  - *Don’t allocate resource if it may lead to deadlock*
  - *If a process requests a resource make it wait until you are sure it’s OK*

- Which one to use depends upon the application and how easy is it to recover from deadlock!

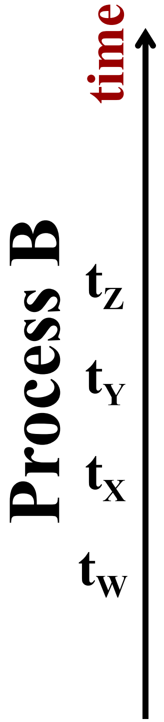
Deadlock Avoidance



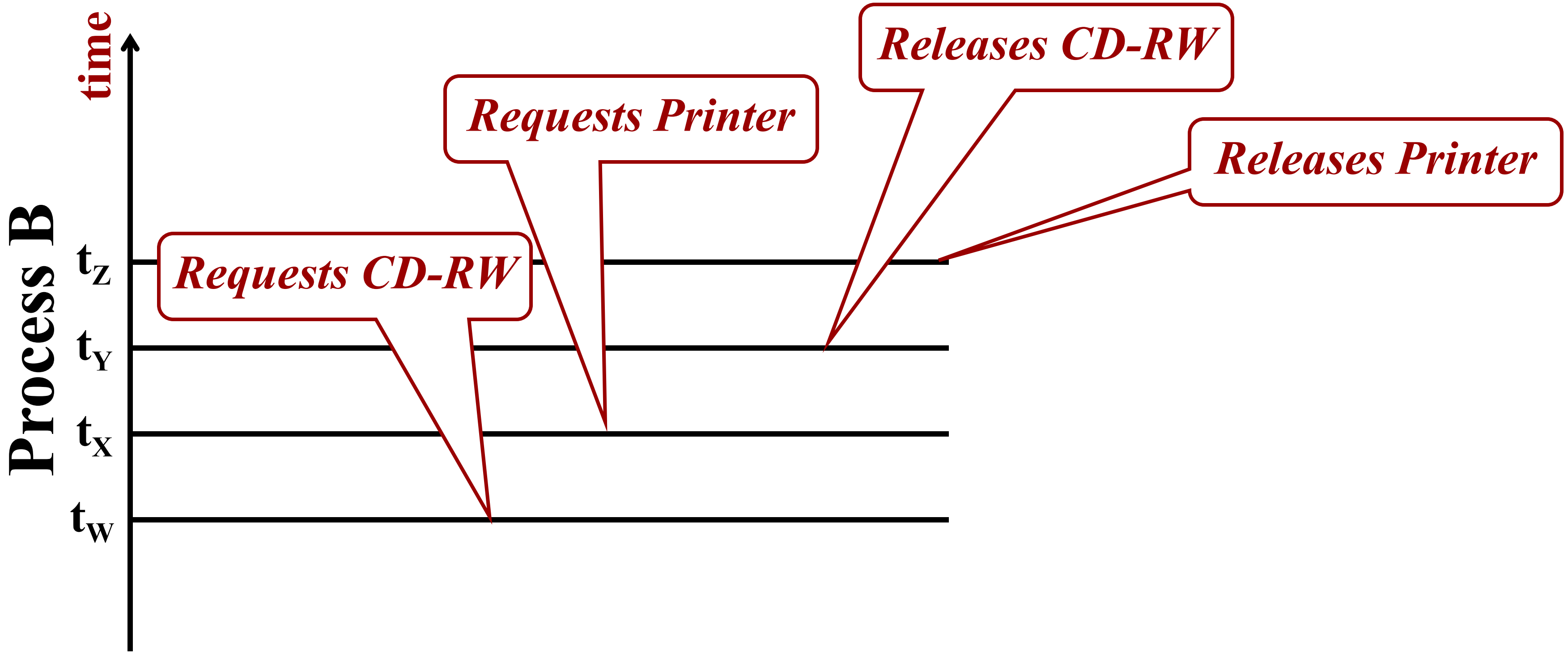
Deadlock Avoidance



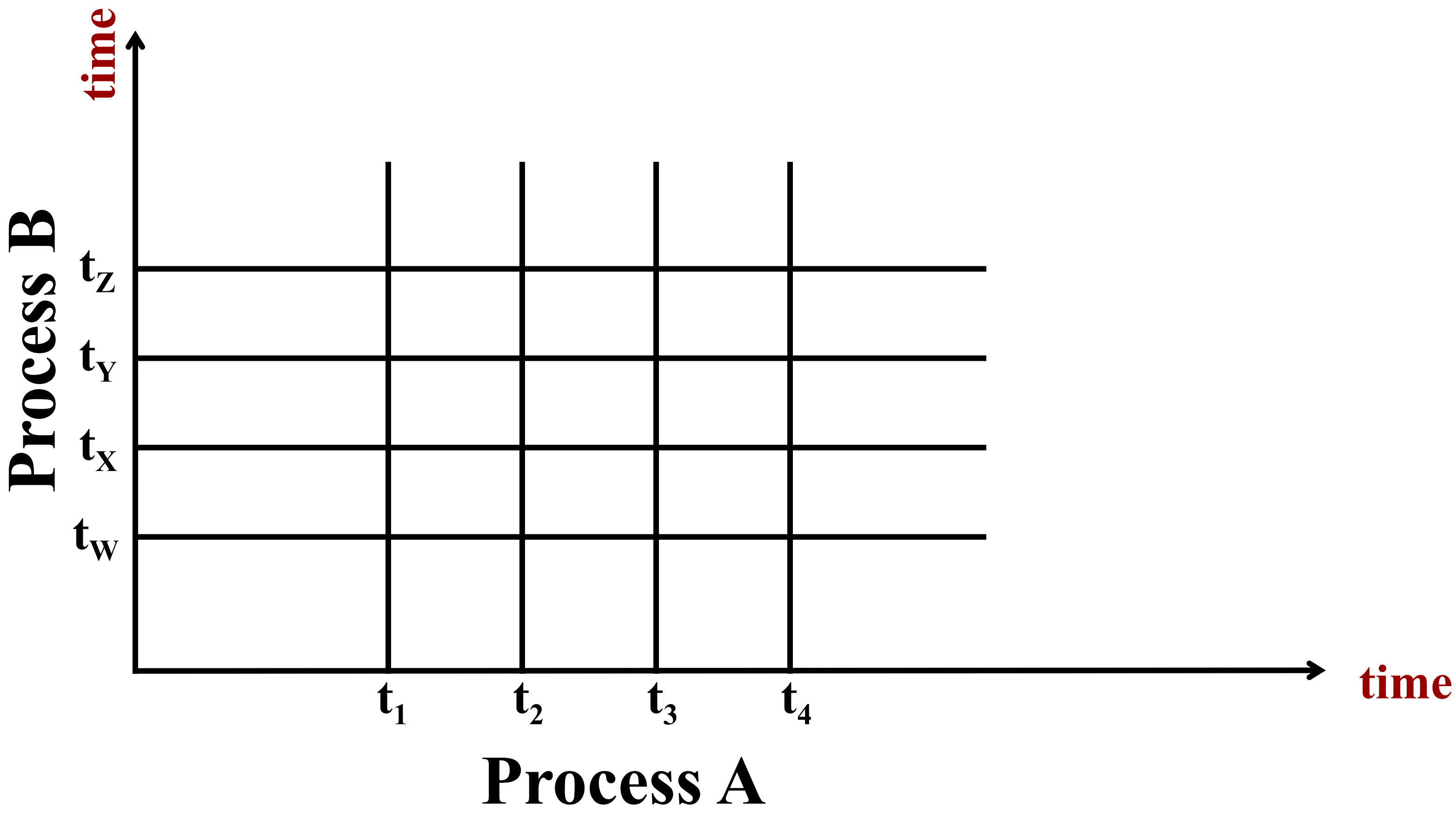
Deadlock Avoidance



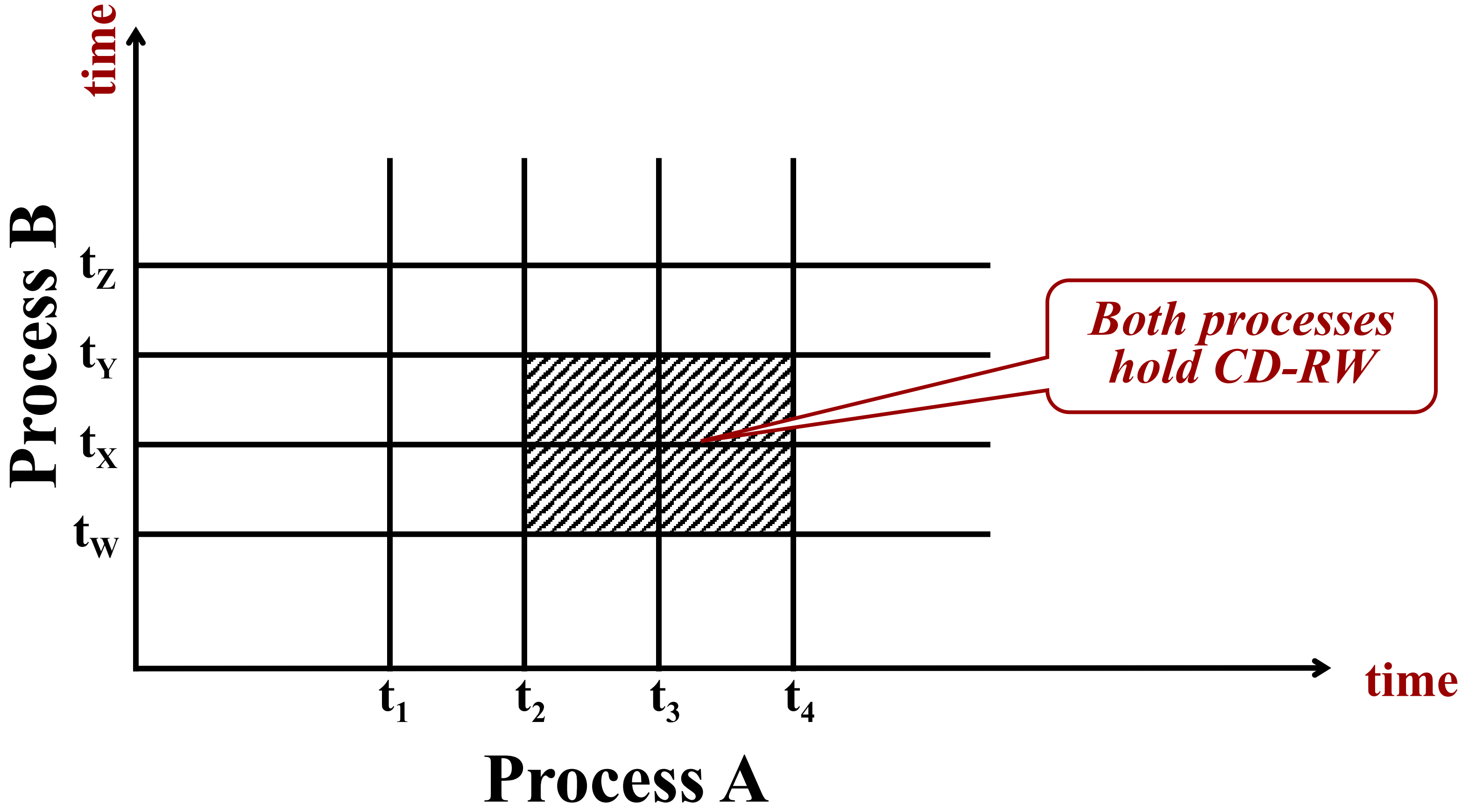
Deadlock Avoidance



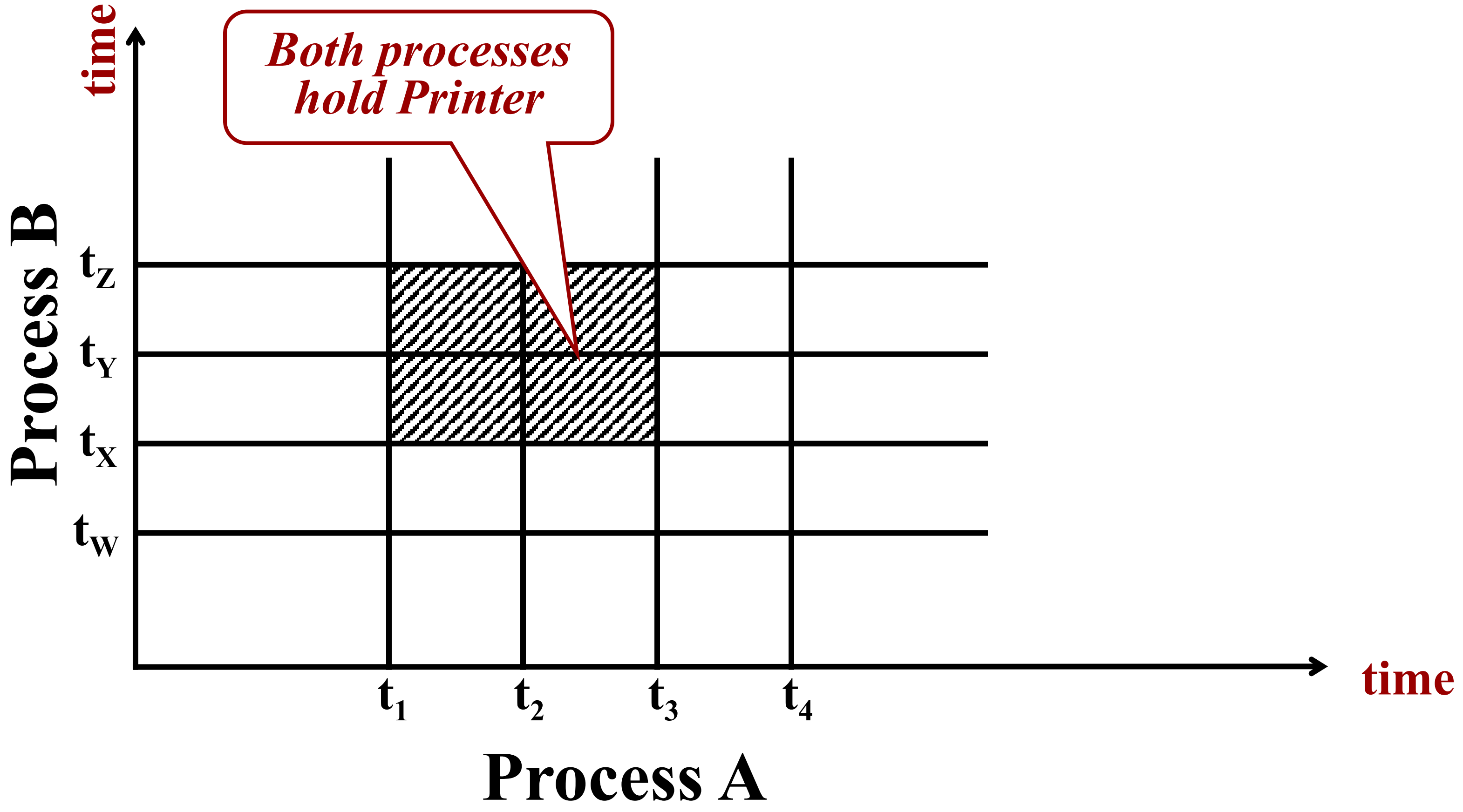
Deadlock Avoidance



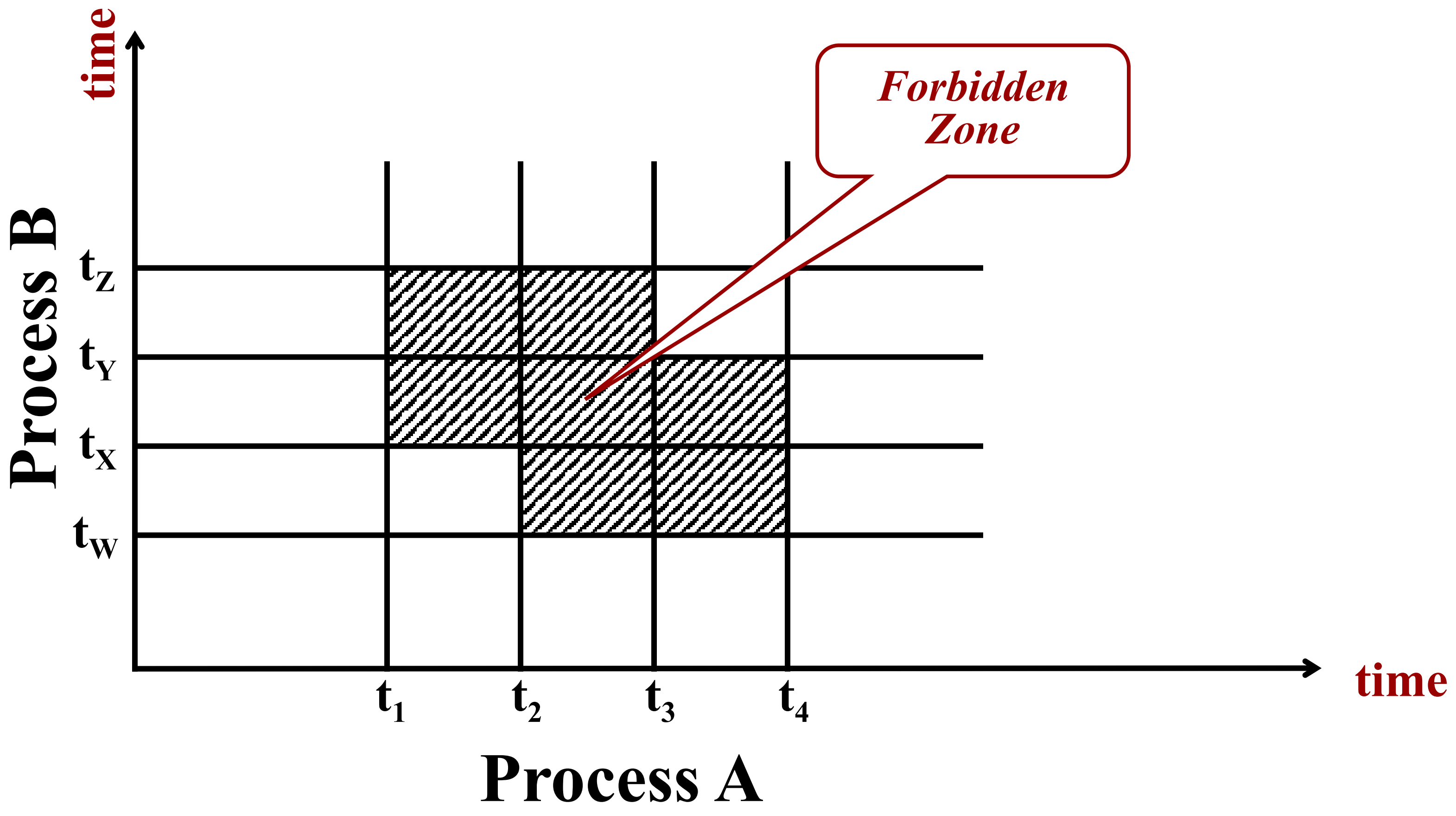
Deadlock Avoidance



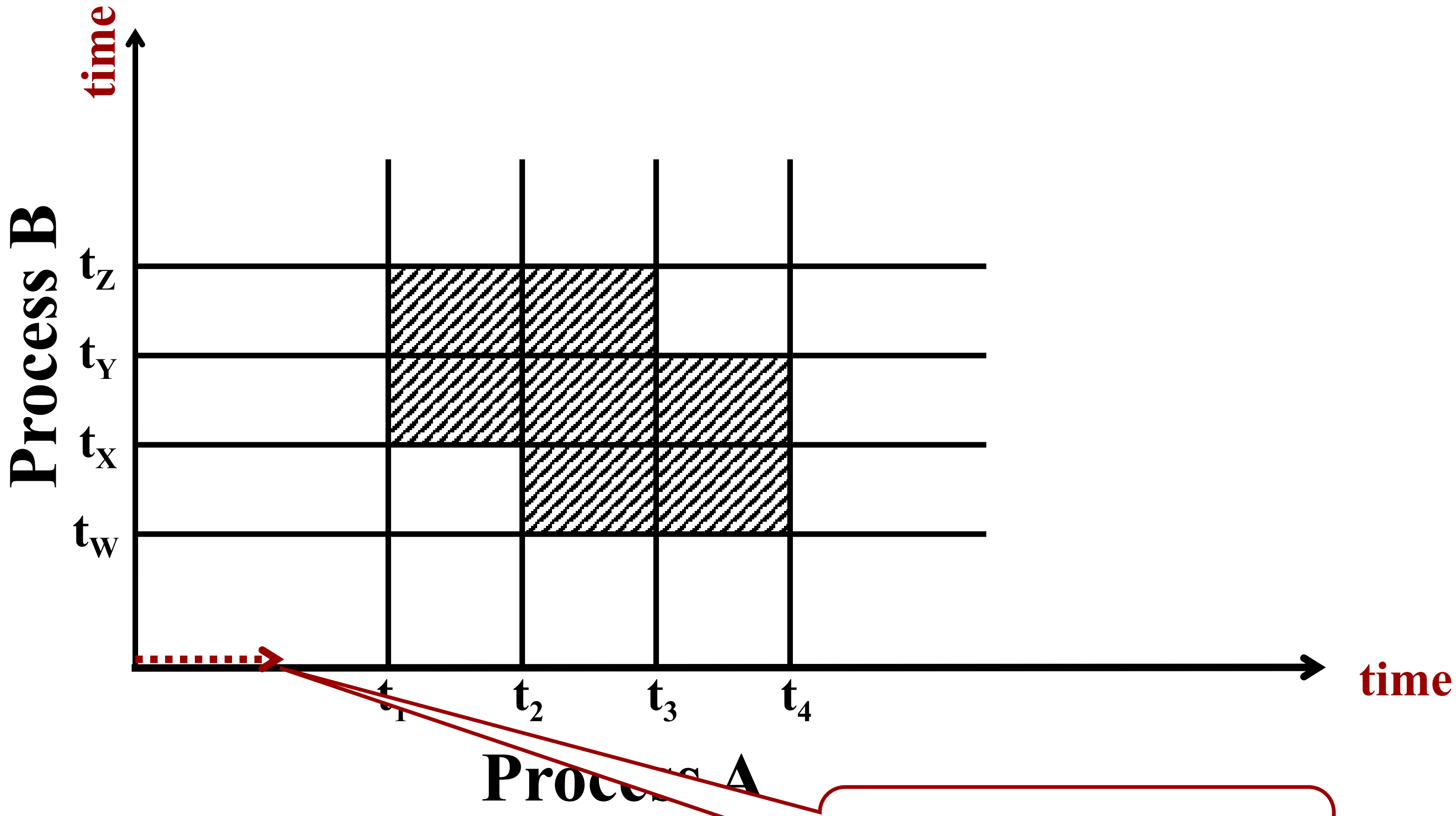
Deadlock Avoidance



Deadlock Avoidance

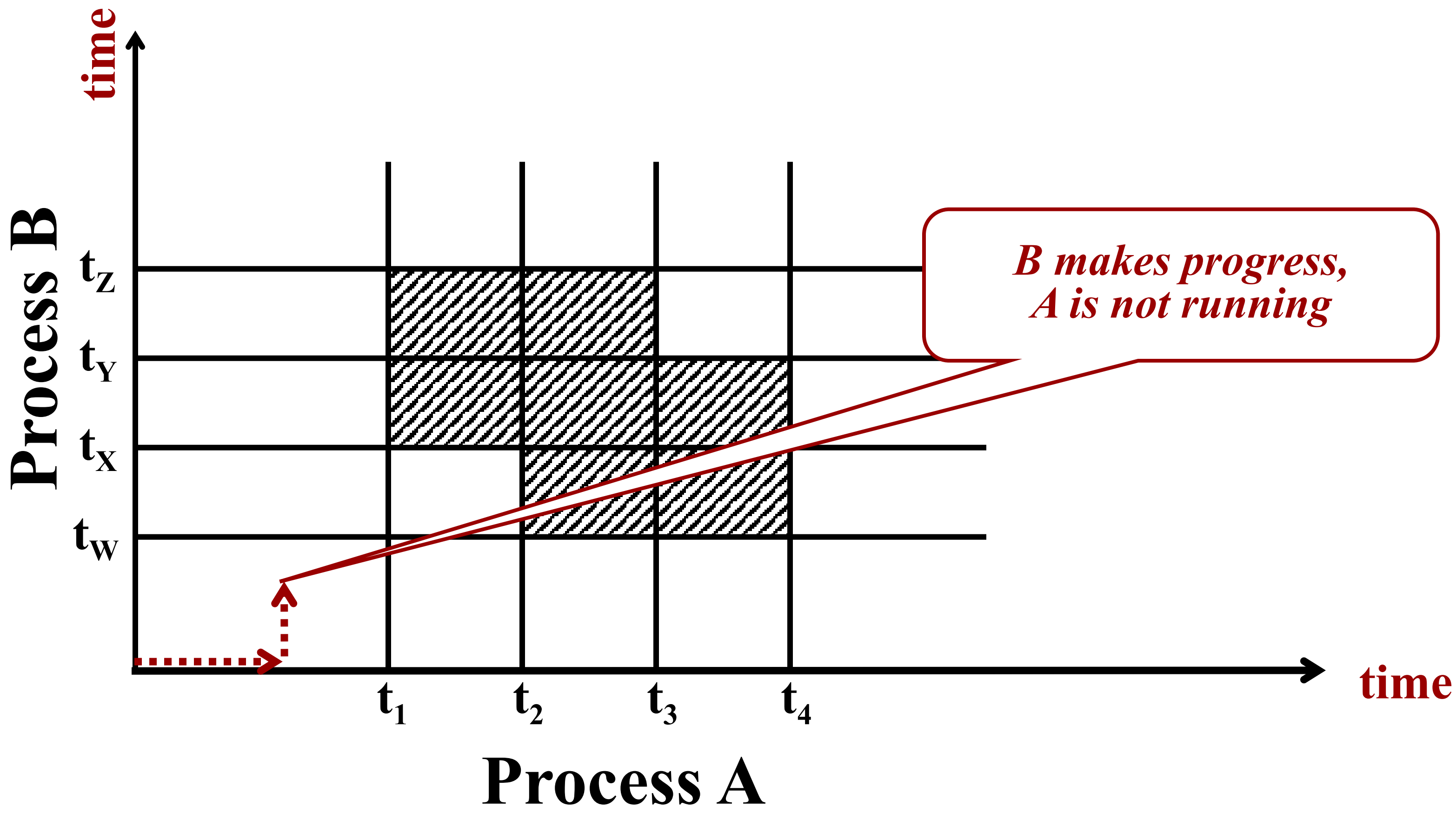


Deadlock Avoidance



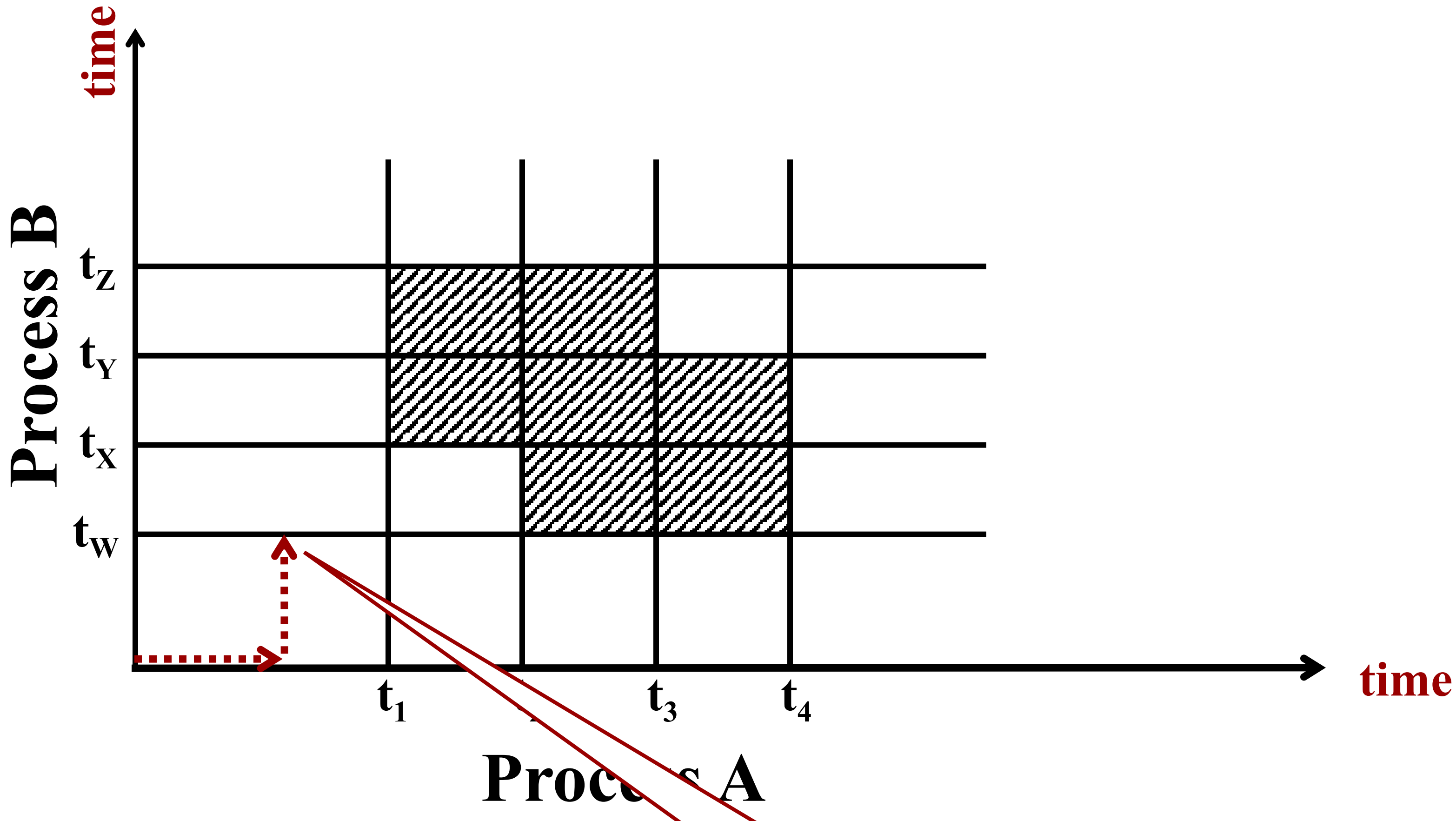
Trajectory showing system progress

Deadlock Avoidance



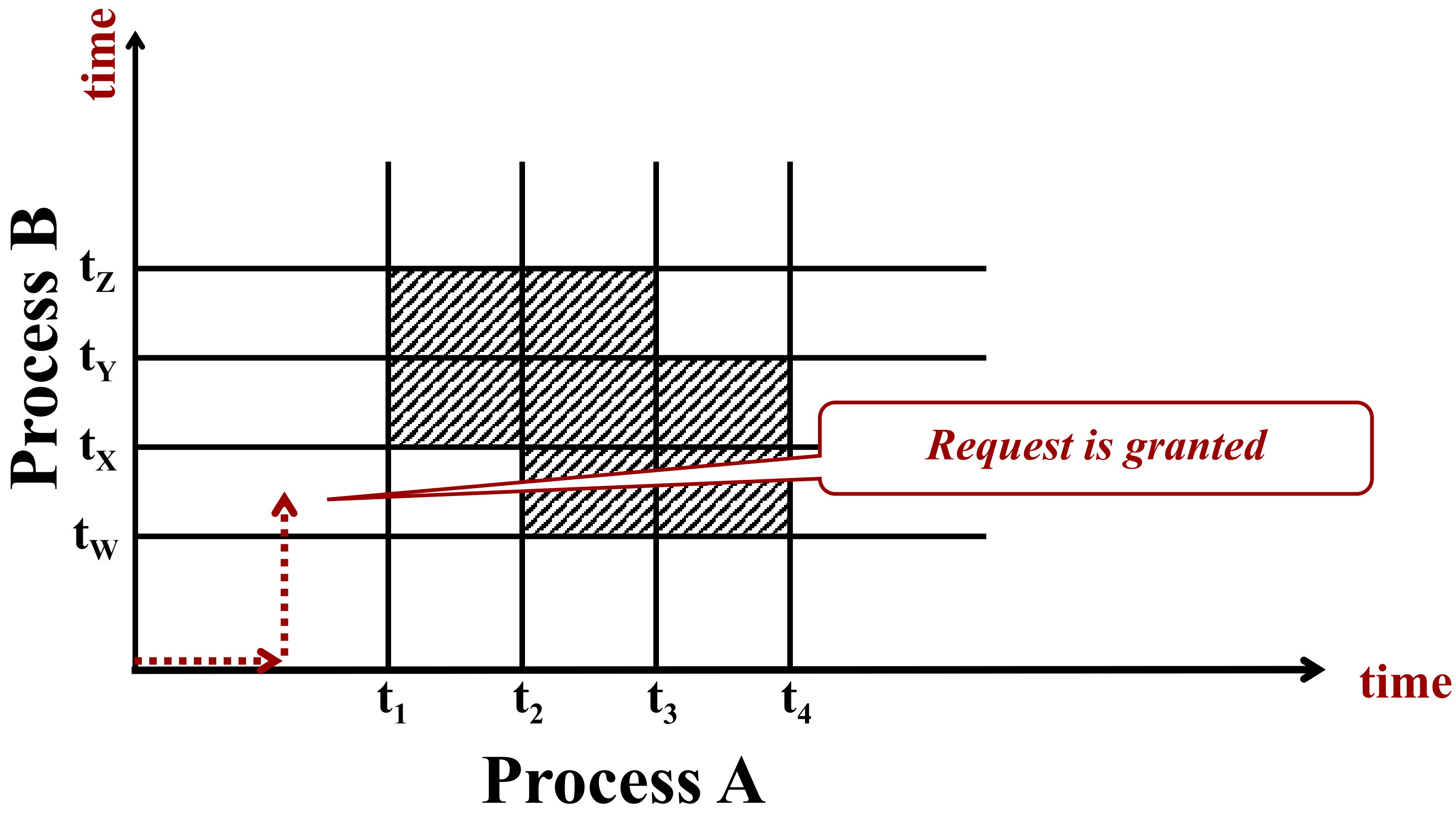
B makes progress,
A is not running

Deadlock Avoidance

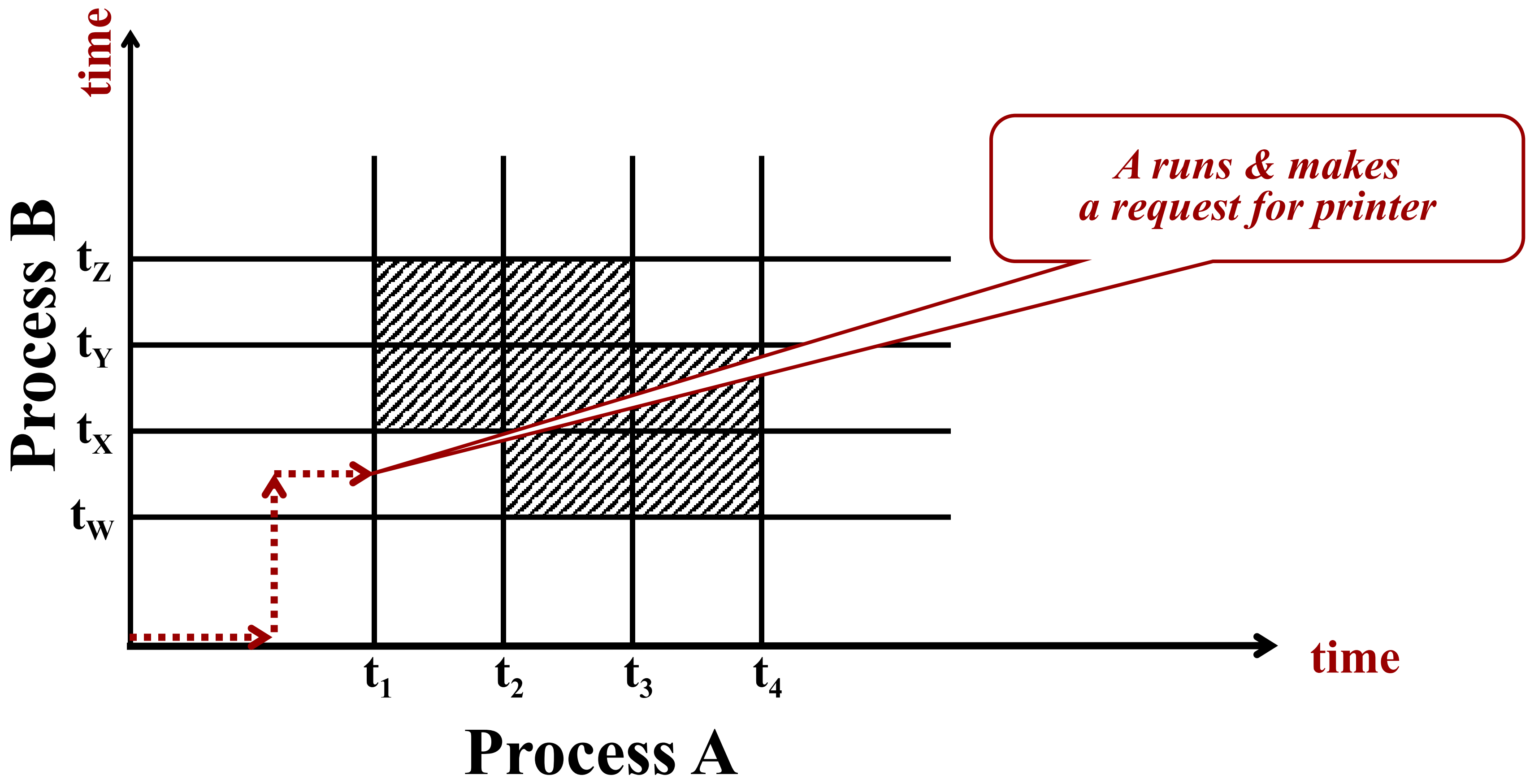


B requests the CD-RW

Deadlock Avoidance

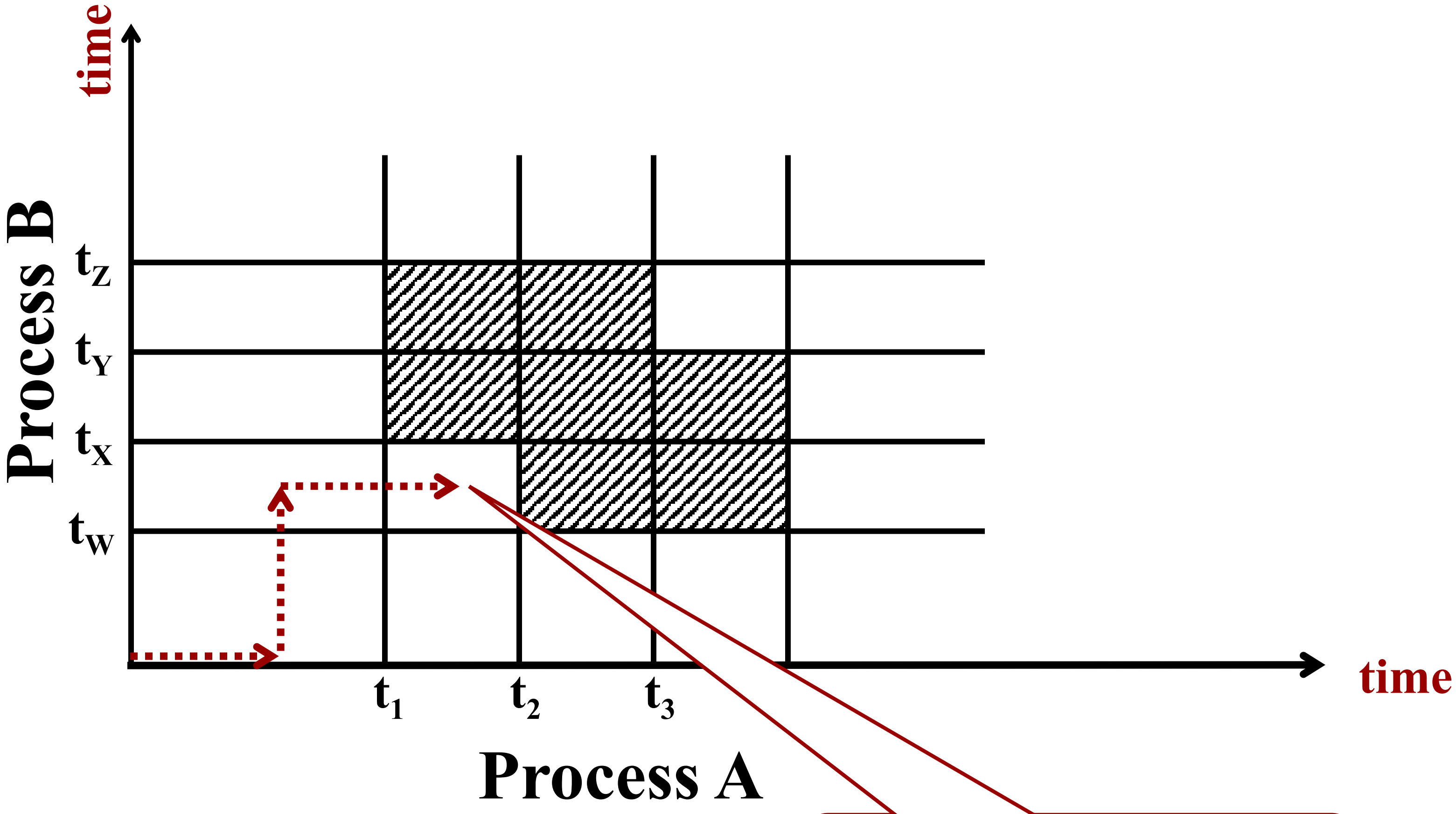


Deadlock Avoidance



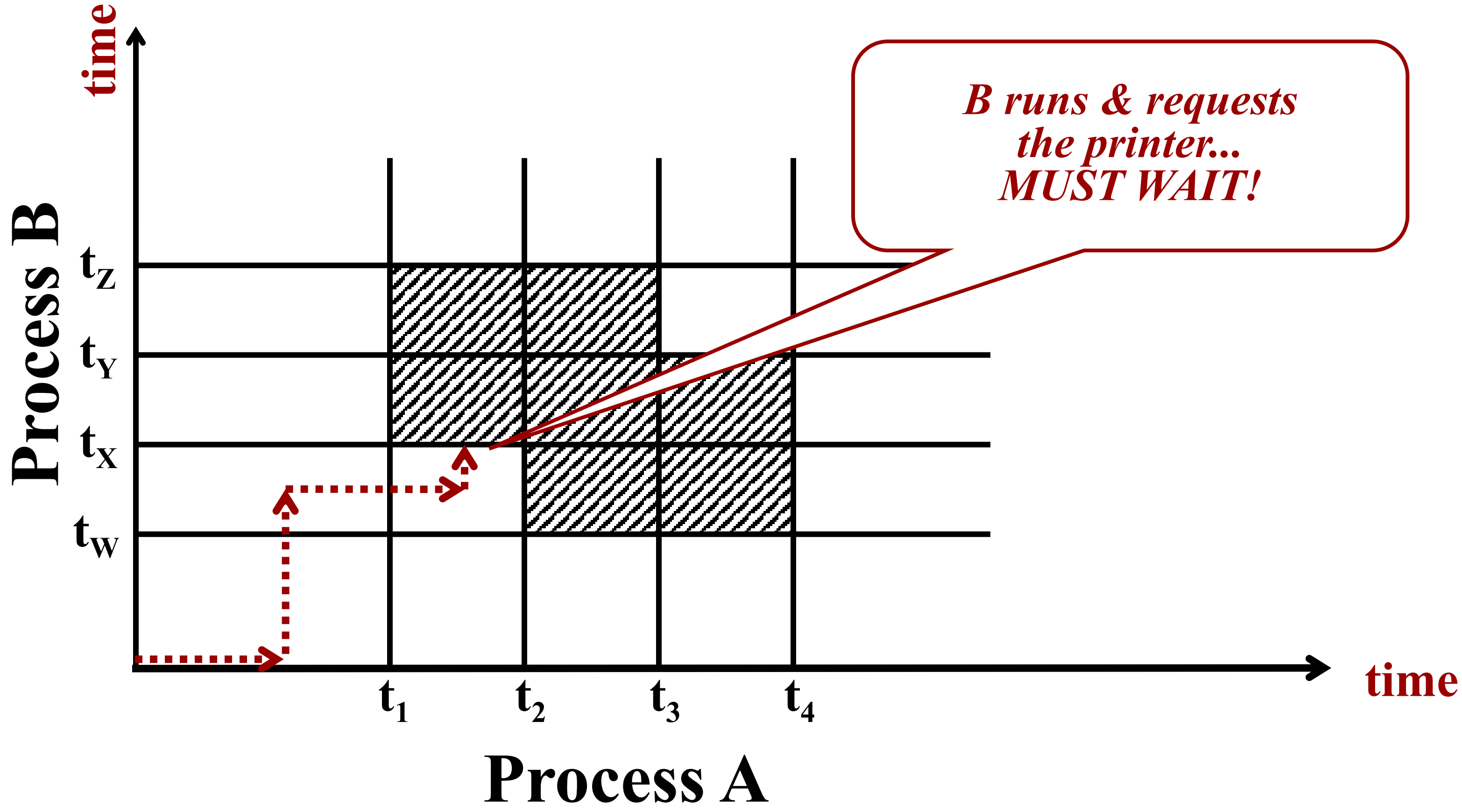
A runs & makes a request for printer

Deadlock Avoidance



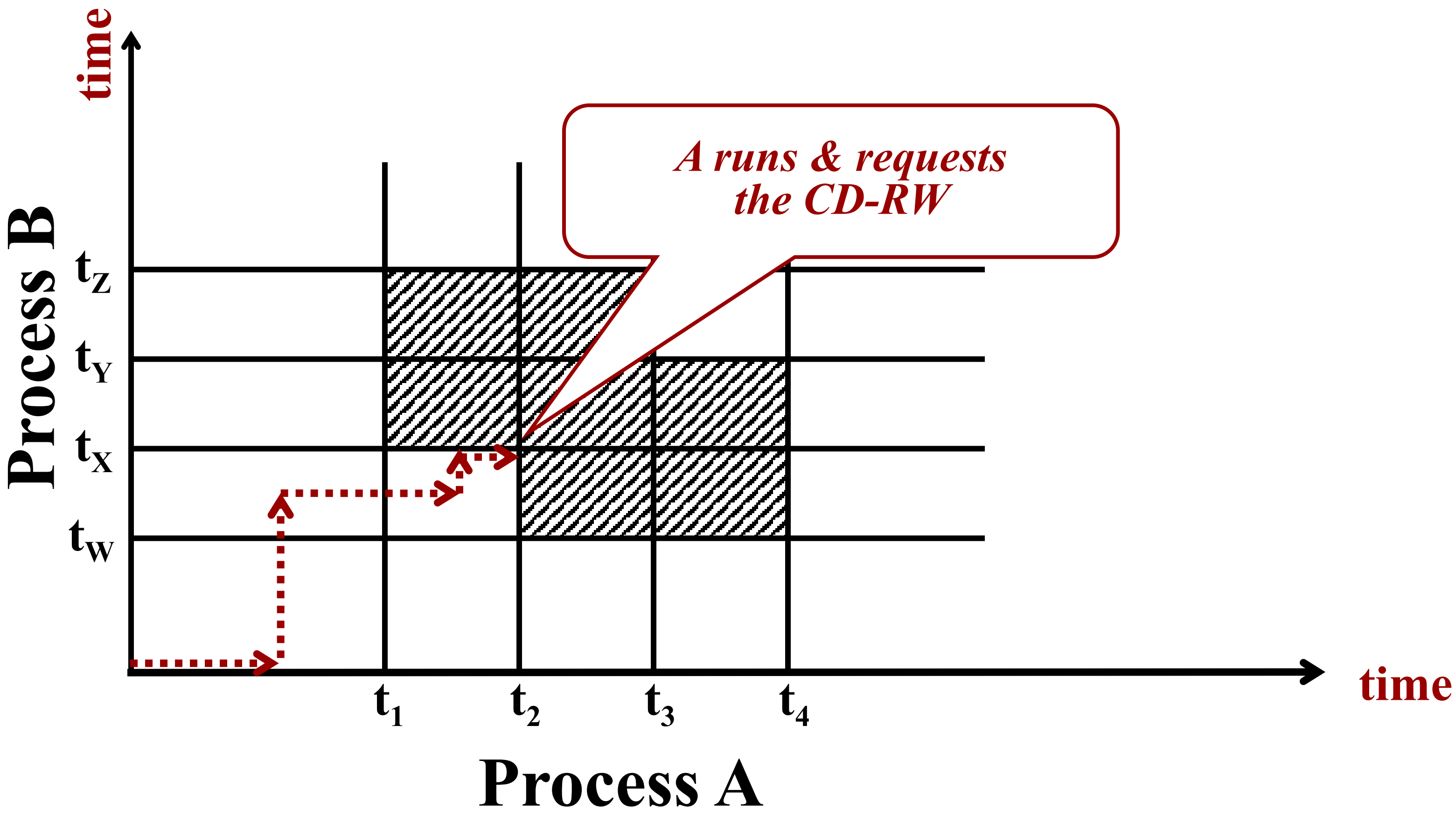
Request is granted;
A proceeds

Deadlock Avoidance

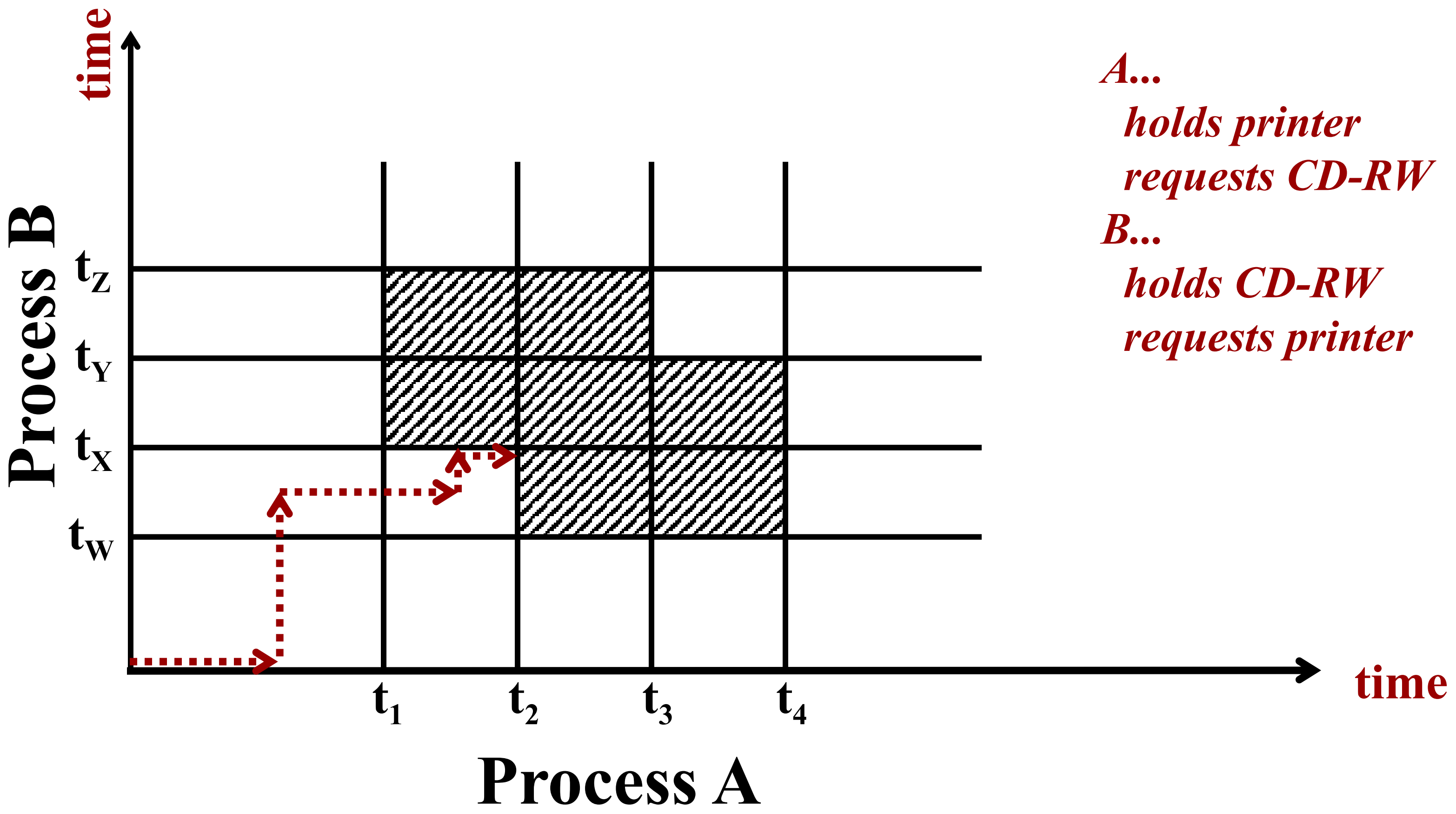


B runs & requests the printer... MUST WAIT!

Deadlock Avoidance



Deadlock Avoidance



A...

holds printer

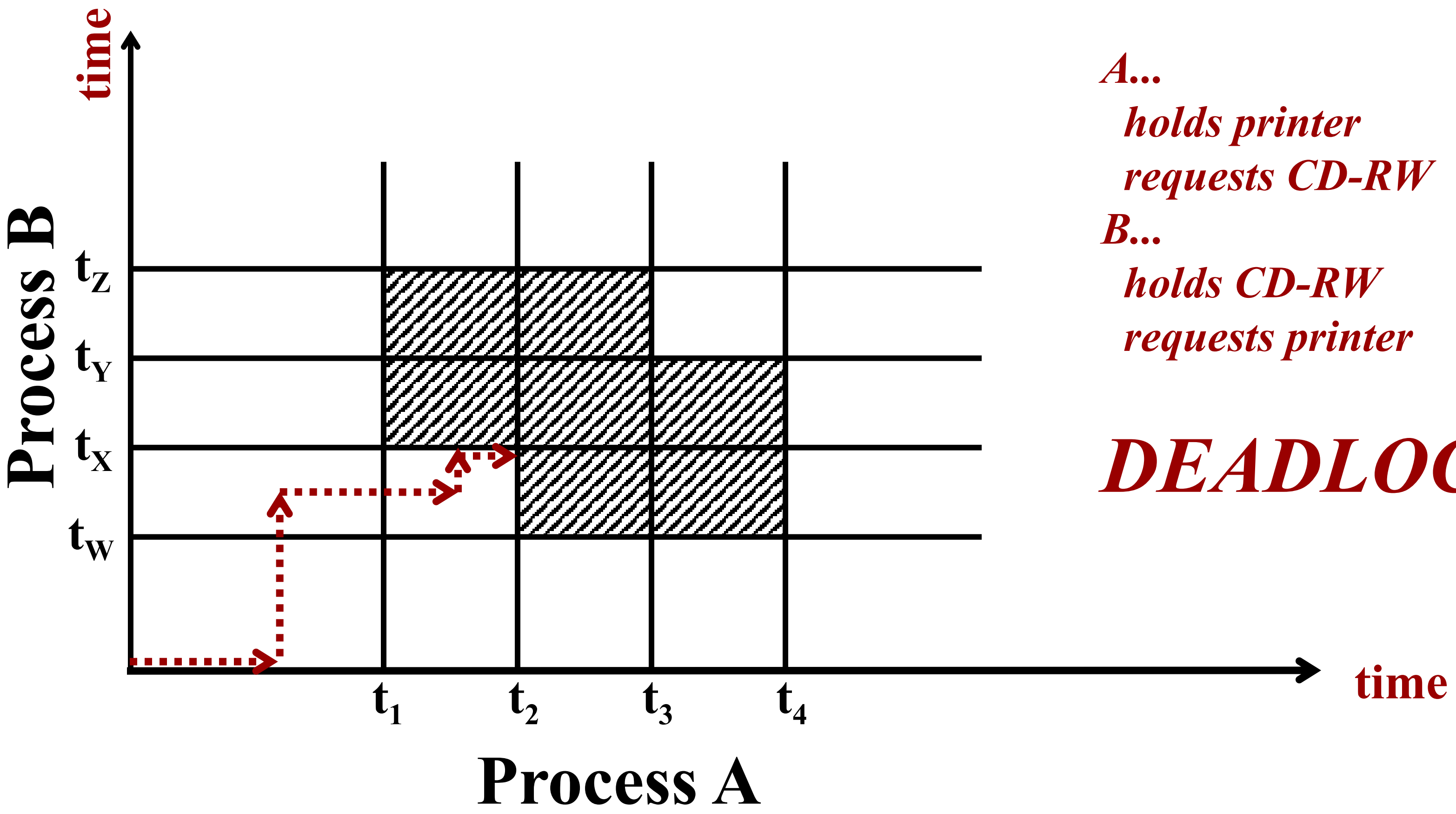
requests CD-RW

B...

holds CD-RW

requests printer

Deadlock Avoidance



A...

holds printer

requests CD-RW

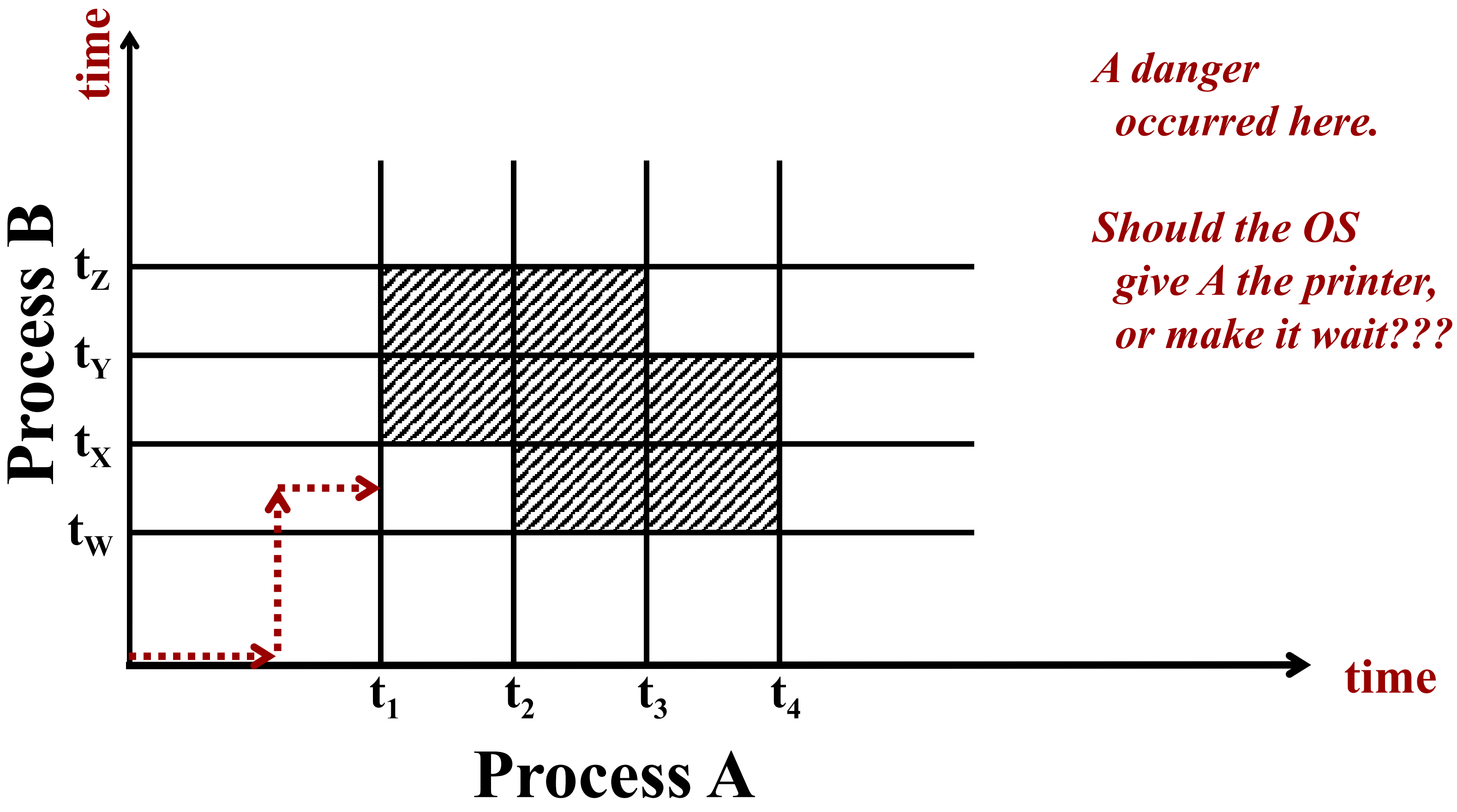
B...

holds CD-RW

requests printer

DEADLOCK!

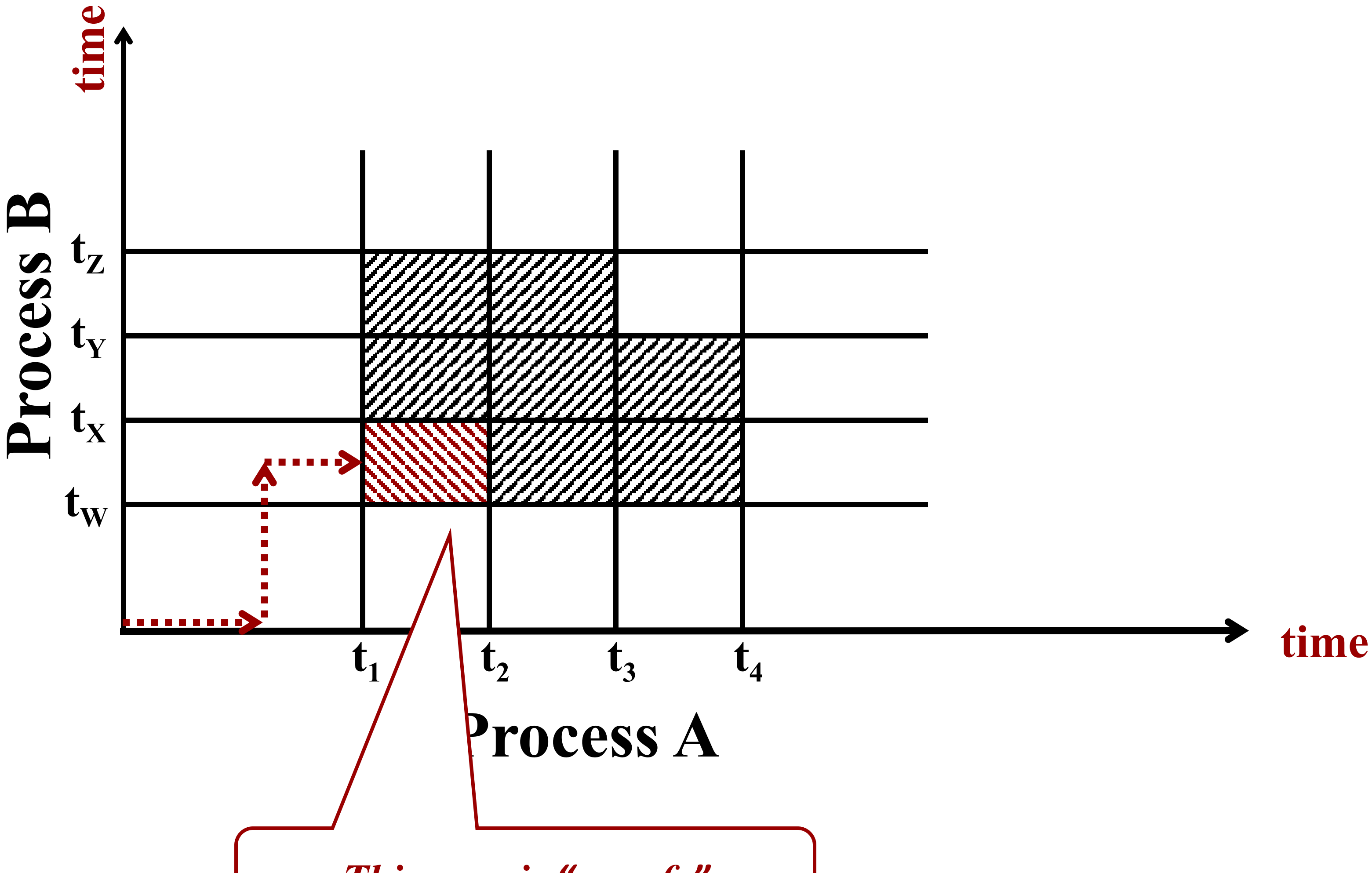
Deadlock Avoidance



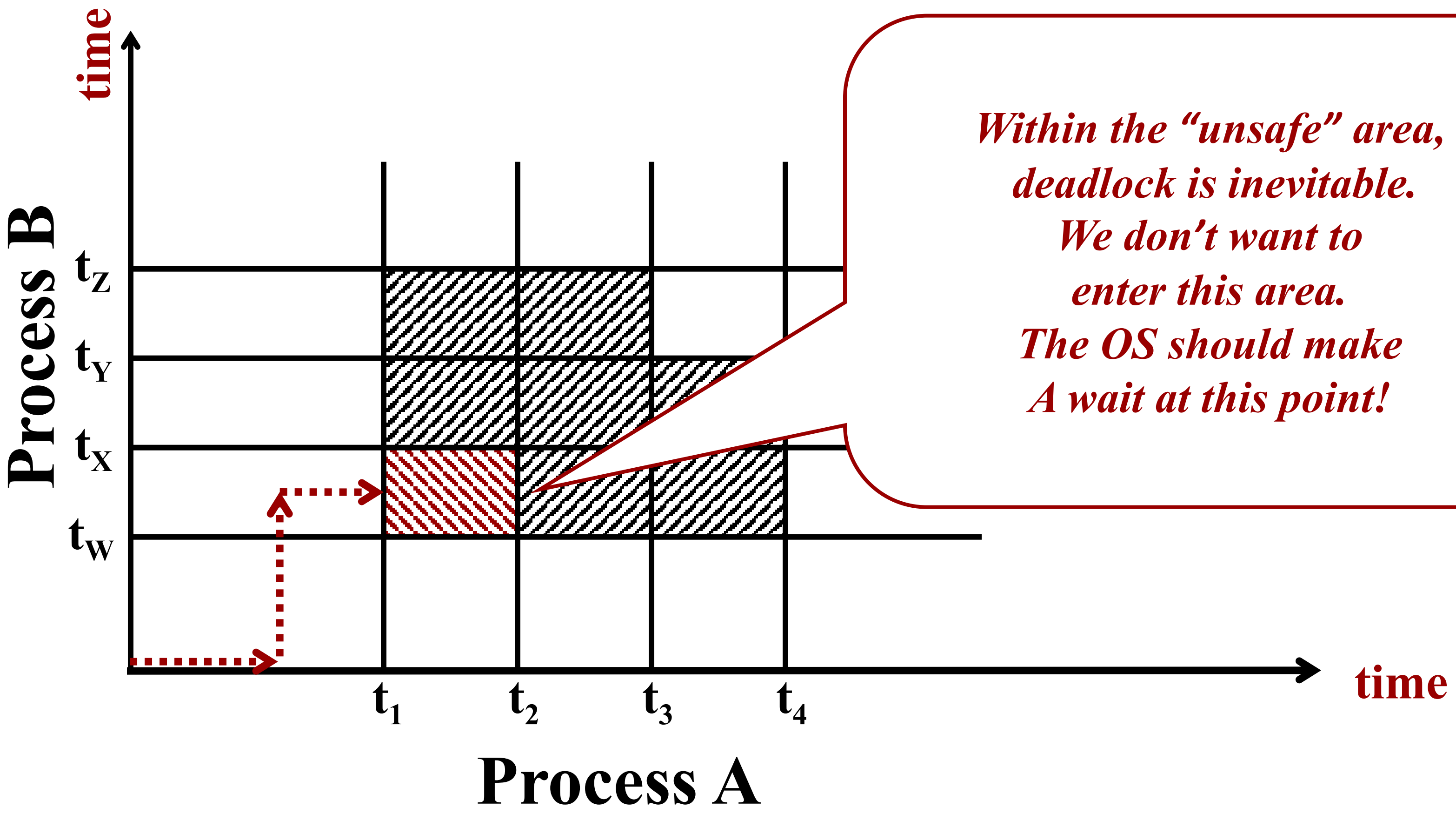
A danger occurred here.

Should the OS give A the printer, or make it wait???

Deadlock Avoidance

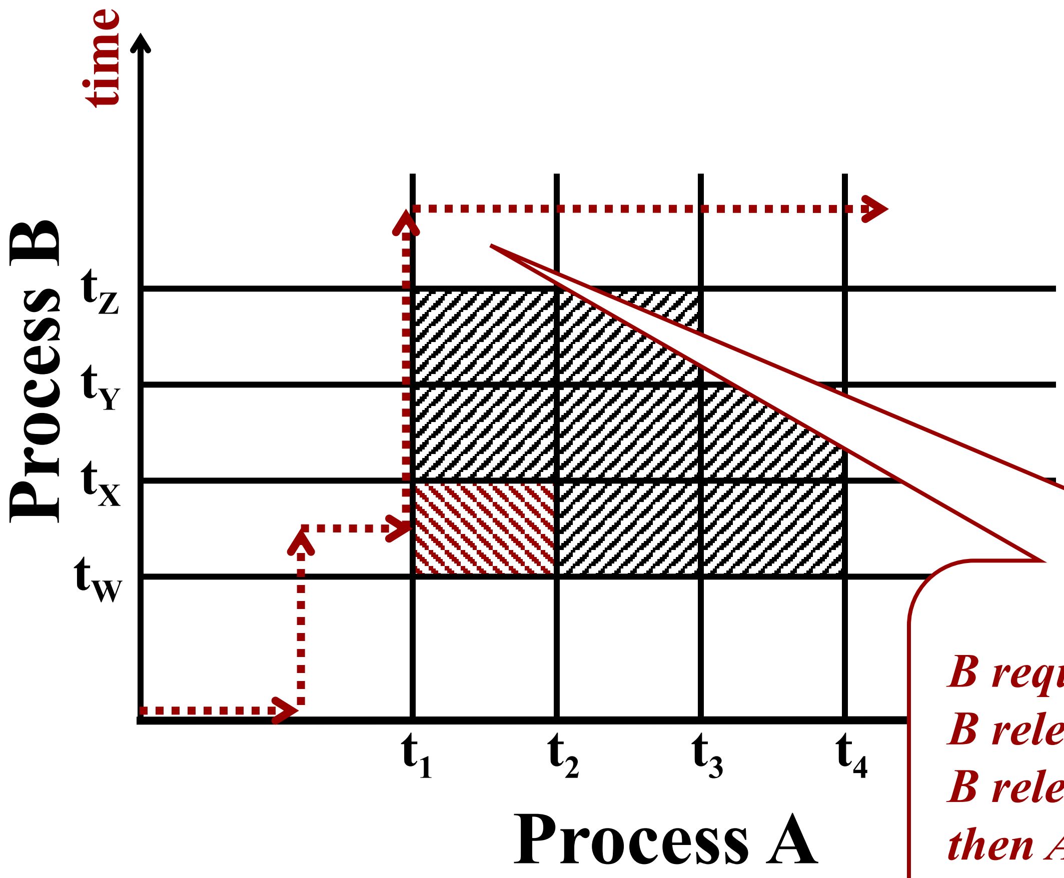


Deadlock Avoidance



Within the "unsafe" area, deadlock is inevitable. We don't want to enter this area. The OS should make A wait at this point!

Deadlock Avoidance



B requests the printer,
B releases CD-RW,
B releases printer,
then A runs to completion!

Safe States

- The current state: which processes hold which resources
- A "safe" state:
  - *No deadlock, and*
  - *There is some scheduling order in which every process can run to completion even if all of them request their maximum number of units immediately*
- <u>The Banker's Algorithm:</u>
  - *Goal: Avoid unsafe states!!!*
  - *When a process requests more units, should the system grant the request or make it wait?*

Avoidance - Multiple Resources

Total resource vector

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation
to process n

Available resource vector

Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Maximum Request Vector

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 might need

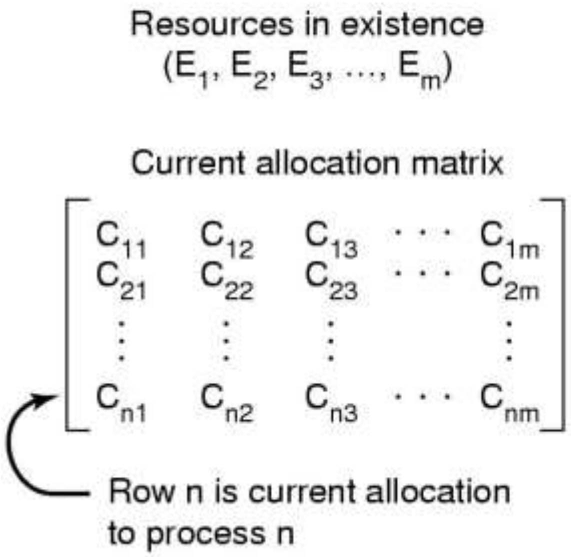
Note: These are the max. possible requests, which we assume are known ahead of time!

Banker’ s Algorithm

- Look for a row, *R*, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion
- Assume the process of the row chosen requests all the resources that it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to A vector
- Repeat steps 1 and 2, until either all process are marked terminated, in which case the initial state was safe, or until deadlock occurs, in which case it was not

Avoidance - Multiple Resources

Total resource vector



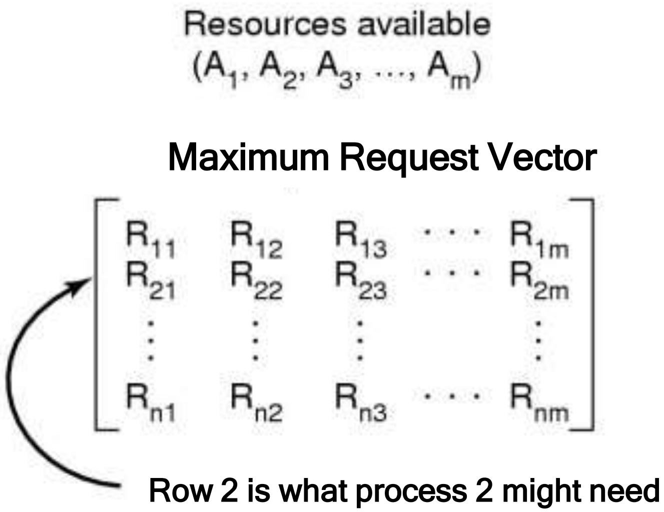
Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$

Current allocation matrix

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Row n is current allocation to process n

Available resource vector



Resources available
$(A_1, A_2, A_3, \ldots, A_m)$

Maximum Request Vector

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Row 2 is what process 2 might need

Run algorithm on every resource request!

Avoidance - Multiple Resources

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,4 & 2 & 3 & 1\,) \end{array}$$

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,2 & 1 & 0 & 0\,) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Max request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Avoidance - Multiple Resources

$$\begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ E = (& 4 & 2 & 3 & 1\ ) \end{array}$$

$$\begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ A = (& 2 & 1 & 0 & 0\ ) \end{array}$$

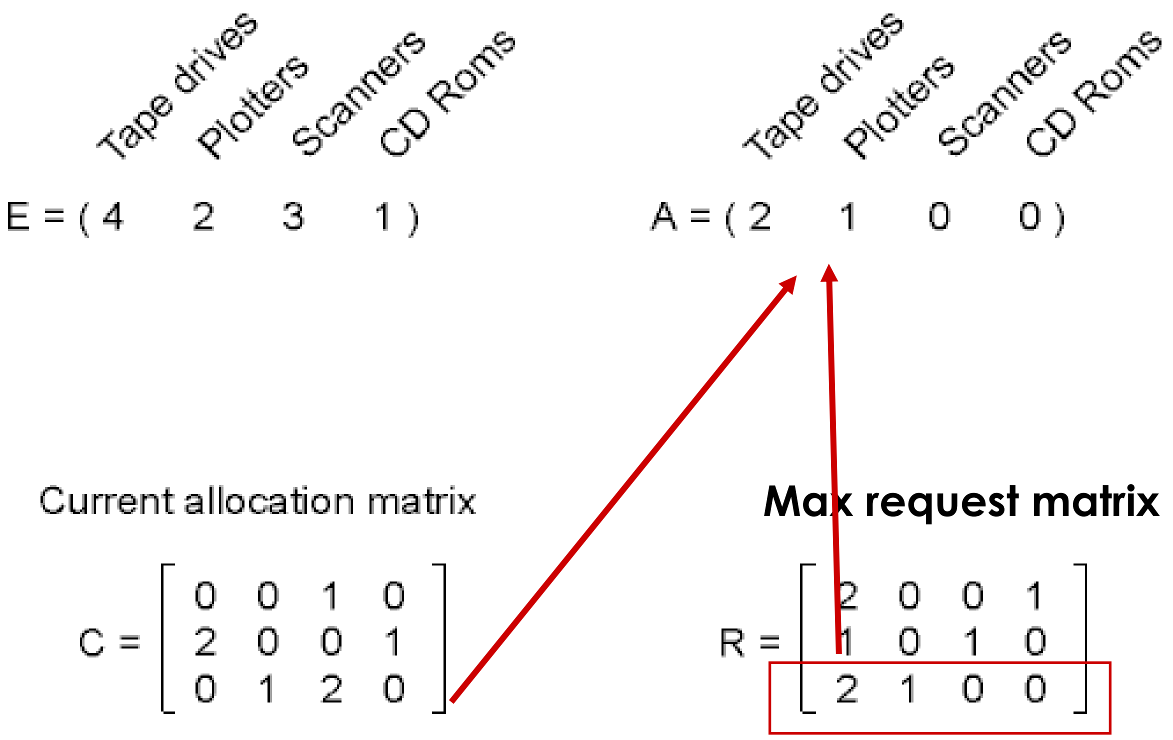
Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Max request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Avoidance - Multiple Resources



$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,4 & 2 & 3 & 1\,) \end{array}$$

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,2 & 1 & 0 & 0\,) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Max request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Avoidance - Multiple Resources

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,4 & 2 & 3 & 1\,) \end{array}$$

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,2 & 1 & 0 & 0\,) \\ \mathbf{2} & \mathbf{2} & \mathbf{2} & \mathbf{0} \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Max request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Avoidance - Multiple Resources

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,4 & 2 & 3 & 1\,) \end{array}$$

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,2 & 1 & 0 & 0\,) \\ \mathbf{2} & \mathbf{2} & \mathbf{2} & \mathbf{0} \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ \cancel{0} & \cancel{1} & \cancel{2} & \cancel{0} \end{bmatrix}$$

Max request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ \cancel{2} & \cancel{1} & \cancel{0} & \cancel{0} \end{bmatrix}$$

Avoidance - Multiple Resources

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,4 & 2 & 3 & 1\,) \end{array}$$

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (\,2 & 1 & 0 & 0\,) \\ \mathbf{2} & \mathbf{2} & \mathbf{2} & \mathbf{0} \\ \mathbf{4} & \mathbf{2} & \mathbf{2} & \mathbf{1} \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ \cancel{2} & \cancel{0} & \cancel{0} & \cancel{1} \\ \cancel{0} & \cancel{1} & \cancel{2} & \cancel{0} \end{bmatrix}$$

Max request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ \cancel{1} & \cancel{0} & \cancel{1} & \cancel{0} \\ \cancel{2} & \cancel{1} & \cancel{0} & \cancel{0} \end{bmatrix}$$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work = Available

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

2. Find an ***i*** such that both:

   (a) ***Finish [i] = false***

   (b) ***Need$_i$ ≤ Work***

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

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

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

Resource-Request Algorithm for Process P_i

$\textbf{\textit{Request}}_i$ = request vector for process $\textbf{\textit{T}}_i$. If $\textbf{\textit{Request}}_i\,[j] = k$ then process $\textbf{\textit{T}}_i$ wants ***k*** instances of resource type $\textbf{\textit{R}}_j$

1. *If* $\textbf{\textit{Request}}_i \leq \textbf{\textit{Need}}_i$ *go to step 2. Otherwise, raise error condition*
2. *If* $\textbf{\textit{Request}}_i \leq \textbf{\textit{Available}}$*, go to step 3. Otherwise* $\textbf{\textit{T}}_i$ *must wait, since resources are not available*
3. *Pretend to allocate requested resources to* $\textbf{\textit{T}}_i$ *by modifying the state as follows:*

$$\textbf{\textit{Available}} = \textbf{\textit{Available}} - \textbf{\textit{Request}}_i;$$

$$\textbf{\textit{Allocation}}_i = \textbf{\textit{Allocation}}_i + \textbf{\textit{Request}}_i;$$

$$\textbf{\textit{Need}}_i = \textbf{\textit{Need}}_i - \textbf{\textit{Request}}_i;$$

- If safe $\Rightarrow$ the resources are allocated to $\textbf{\textit{T}}_i$
- If unsafe $\Rightarrow$ $\textbf{\textit{T}}_i$ must wait, and the old resource-allocation state is restored

What Real OSes Use

- **Windows** and **Linux** focus on a mix of **detection** and **recovery** mechanisms, with minimal prevention techniques to handle deadlocks.
- **Prevention techniques** include guidelines for resource acquisition orders, mutual exclusion best practices, and occasional resource preemption.
- **Detection** is done at a low level using tools and techniques like Wait-For Graphs in Windows or kernel-level debugging tools in Linux.
- **Recovery** typically involves terminating or restarting the process that caused the deadlock or forcing the release of held resources.

جلسه‌ی بعد

- زمان‌بندی پردازه‌ها