

CS 333

Introduction to Operating Systems

Class 7 - Deadlock

Jonathan Walpole
Computer Science
Portland State University

Monitors

... from class 6

Monitors

- ❑ **It is difficult to produce correct programs using semaphores**
 - ❖ correct ordering of wait and signal is tricky!
 - ❖ avoiding race conditions and deadlock is tricky!
 - ❖ boundary conditions are tricky!
- ❑ **Can we get the compiler to generate the correct semaphore code for us?**
 - ❖ what are suitable higher level abstractions for synchronization?

Monitors

- ❑ Related shared objects are collected together
- ❑ Compiler enforces encapsulation/mutual exclusion
 - ❖ Encapsulation:
 - Local data variables are accessible only via the monitor's entry procedures (like methods)
 - ❖ Mutual exclusion
 - A monitor has an associated mutex lock
 - Threads must acquire the monitor's mutex lock before invoking one of its procedures

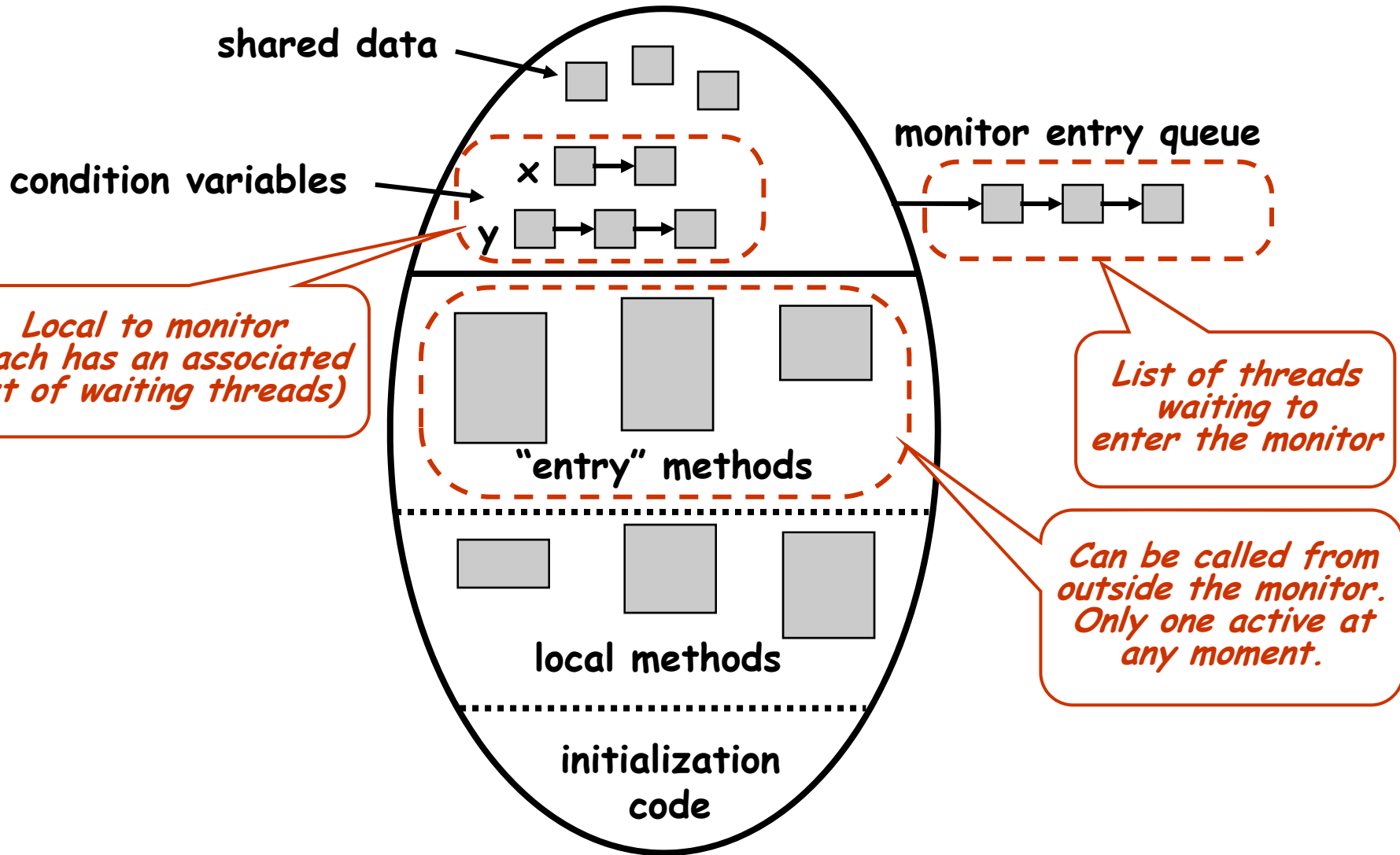
Monitors and condition variables

- But we need two flavors of synchronization
 - ❖ Mutual exclusion
 - Only one at a time in the critical section
 - Handled by the monitor's mutex
 - ❖ Condition synchronization
 - Wait until a certain condition holds
 - Signal waiting threads when the condition holds

Monitors and condition variables

- Condition variables (cv) for use within monitors
 - ❖ `cv.wait(mon-mutex)`
 - thread blocked (queued) until condition holds
 - Must not block while holding mutex!
 - monitor mutex must be released!
 - ❖ `cv.signal()`
 - signals the condition and unblocks (dequeues) a thread

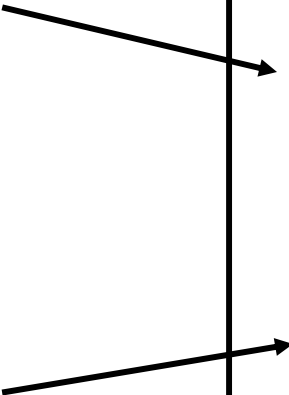
Monitor structures



Monitor example for mutual exclusion

```
process Producer
begin
  loop
    <produce char "c">
    BoundedBuffer.deposit(c)
  end loop
end ProducerL
```

```
process Consumer
begin
  loop
    BoundedBuffer.remove(c)
    <consume char "c">
  end loop
end Consumer
```



```
monitor: BoundedBuffer
var  buffer : ...;
    nextIn, nextOut :... ;

entry deposit(c: char)
begin
  ...
end

entry remove(var c: char)
begin
  ...
end

end BoundedBuffer
```


Observations

- ❑ That's much simpler than the semaphore-based solution to producer/consumer (bounded buffer)!
- ❑ ... but where is the mutex?
- ❑ ... and what do the bodies of the monitor procedures look like?

Monitor example with condition variables

```
monitor : BoundedBuffer
var buffer          : array[0..n-1] of char
    nextIn,nextOut   : 0..n-1 := 0
    fullCount        : 0..n    := 0
    notEmpty, notFull : condition
```

```
entry deposit(c:char)
begin
    if (fullCount = n) then
        wait(notFull)
    end if

    buffer[nextIn] := c
    nextIn := nextIn+1 mod n
    fullCount := fullCount+1

    signal(notEmpty)
end deposit
```

```
entry remove(var c: char)
begin
    if (fullCount = n) then
        wait(notEmpty)
    end if

    c := buffer[nextOut]
    nextOut := nextOut+1 mod n
    fullCount := fullCount-1

    signal(notFull)
end remove
```

```
end BoundedBuffer
```

Condition variables

“Condition variables allow processes to synchronize based on some state of the monitor variables.”

Condition variables in producer/consumer

“NotFull” condition

“NotEmpty” condition

- Operations **Wait()** and **Signal()** allow synchronization within the monitor
- When a producer thread adds an element...
 - ❖ A consumer may be sleeping
 - ❖ Need to wake the consumer... **Signal**

Condition synchronization semantics

- *"Only one thread can be executing in the monitor at any one time."*
- **Scenario:**
 - ❖ Thread A is executing in the monitor
 - ❖ Thread A does a **signal** waking up thread B
 - ❖ What happens now?
 - ❖ Signaling and signaled threads can not both run!
 - ❖ ... so which one runs, which one blocks, and on what queue?

Monitor design choices

- ❑ **Condition variables introduce a problem for mutual exclusion**
 - ❖ Q1: only one process active in the monitor at a time, so what to do when a process is unblocked on **signal**?
 - ❖ Q2: must not block holding the mutex, so what to do when a process blocks on **wait**?
- ❑ **Should signals be stored/remembered?**
 - ❖ signals are not stored (unlike signals on semaphores)
 - ❖ if signal occurs before wait, signal is lost!
- ❑ **Should condition variables count?**

Monitor design choices

- **Choices when A signals a condition that unblocks B**
 - ❖ Opt1: A waits for B to exit the monitor or block again
 - ❖ Opt2: B waits for A to exit the monitor or block
 - ❖ Opt3: Signal causes A to immediately exit the monitor or block (... but awaiting what condition?)
- **Choices when A signals a condition that unblocks B & C**
 - ❖ Opt1: B is unblocked, but C remains blocked
 - ❖ Opt2: C is unblocked, but B remains blocked
 - ❖ Opt3: Both B & C are unblocked and compete for the mutex?
- **Choices when A calls wait and blocks**
 - ❖ a new external process is allowed to enter
 - ❖ but which one?

Design 1: Hoare semantics

- **What happens when a Signal is performed?**
 - ❖ signaling thread (A) is suspended
 - ❖ signaled thread (B) wakes up and runs immediately
- **Result:**
 - ❖ B can assume the condition it waited for now holds
 - ❖ Hoare semantics give strong guarantees
 - ❖ Easier to prove correctness
- **When B leaves monitor, A can run.**
 - A might resume execution immediately
 - Or another thread (C) may be allowed to slip in!

Design 2: MESA Semantics (Xerox PARC)

- **What happens when a Signal is performed?**
 - ❖ the signaling thread (A) continues.
 - ❖ the signaled thread (B) waits.
 - ❖ when A leaves the monitor B can run
- **Issue: What happens while B is waiting?**
 - ❖ can the condition that caused A to generate the signal be changed before B runs?
- **In MESA semantics a signal is more like a hint**
 - ❖ Requires B to recheck the condition on which it waited to see if it can proceed or must wait some more

Code for the "deposit" entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
    nextIn, nextOut: int = 0
    cntFull: int = 0
    notEmpty: Condition
    notFull: Condition

  entry deposit(c: char)
    if cntFull == N
      notFull.Wait()
    endIf
    buffer[nextIn] = c
    nextIn = (nextIn+1) mod N
    cntFull = cntFull + 1
    notEmpty.Signal()
  endEntry

  entry remove()
    ...

endMonitor
```

} Hoare Semantics

Code for the “deposit” entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
    nextIn, nextOut: int = 0
    cntFull: int = 0
    notEmpty: Condition
    notFull: Condition

  entry deposit(c: char)
    while cntFull == N
      notFull.Wait()
    endWhile
    buffer[nextIn] = c
    nextIn = (nextIn+1) mod N
    cntFull = cntFull + 1
    notEmpty.Signal()
  endEntry

  entry remove()
    ...

endMonitor
```

} MESA Semantics

Code for the "remove" entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
    nextIn, nextOut: int = 0
    cntFull: int = 0
    notEmpty: Condition
    notFull: Condition

  entry deposit(c: char)
    ...

  entry remove()
    if cntFull == 0
      notEmpty.Wait()
    endIf
    c = buffer[nextOut]
    nextOut = (nextOut+1) mod N
    cntFull = cntFull - 1
    notFull.Signal()
  endEntry

endMonitor
```

} Hoare Semantics

Code for the "remove" entry routine

```
monitor BoundedBuffer
  var buffer: array[n] of char
    nextIn, nextOut: int = 0
    cntFull: int = 0
    notEmpty: Condition
    notFull: Condition

  entry deposit(c: char)
    ...

  entry remove()
    while cntFull == 0
      notEmpty.Wait()
    endWhile
    c = buffer[nextOut]
    nextOut = (nextOut+1) mod N
    cntFull = cntFull - 1
    notFull.Signal()
  endEntry

endMonitor
```

} MESA Semantics

"Hoare Semantics"

What happens when a Signal is performed?

The signaling thread (A) is suspended.

The signaled thread (B) wakes up and runs immediately.

B can assume the condition is now true/satisfied

From the original Hoare Paper:

"No other thread can intervene [and enter the monitor] between the signal and the continuation of exactly one waiting thread."

"If more than one thread is waiting on a condition, we postulate that the signal operation will reactivate the longest waiting thread. This gives a simple neutral queuing discipline which ensures that every waiting thread will eventually get its turn."

Implementing Hoare Semantics

- ❑ Thread A holds the monitor lock
- ❑ Thread A **signals** a condition that thread B was waiting on
- ❑ Thread B is moved back to the ready queue?
 - ❖ B should run immediately
 - ❖ Thread A must be suspended...
 - ❖ the monitor lock must be passed from A to B
- ❑ When B finishes it releases the monitor lock
- ❑ Thread A must re-acquire the lock
 - ❖ A is blocked, waiting to re-aquire the lock

Implementing Hoare Semantics

- **Problem:**
 - ❖ Possession of the monitor lock must be passed **directly** from A to B and then eventually back to A

Implementing Hoare Semantics

□ Implementation Ideas:

- ❖ Consider a signaling thread like *A* to be “urgent” after it hands off the monitor lock to *B*
 - Thread *C* trying to gain initial entry to the monitor is not “urgent” so *A* should have priority
- ❖ Consider two wait lists associated with each *MonitorLock* (so now this is not exactly a mutex)
 - UrgentlyWaitingThreads
 - NonurgentlyWaitingThreads
- ❖ Want to wake up urgent threads first, if any
- ❖ Alternatively, *A* could be added to the front of the monitor lock list instead of the back

Implementing Hoare Semantics

- **Recommendation for Project 4 implementation:**
 - ❖ Do not modify the mutex methods provided, because future code will use them
 - ❖ Create new classes:
 - **MonitorLock** -- similar to Mutex
 - **HoareCondition** -- similar to Condition

Brinch-Hansen Semantics

□ Hoare Semantics

- ❖ On signal, allow signaled process to run
- ❖ Upon its exit from the monitor, signaling process continues.

□ Brinch-Hansen Semantics

- ❖ Signaler must immediately exit following any invocation of signal
- ❖ Restricts the kind of solutions that can be written
- ❖ ... but monitor implementation is easier

Review of a Practical Concurrent Programming Issue - Reentrant Functions

Reentrant code

- A function/method is said to be **reentrant** if...

A function that has been invoked may be invoked again before the first invocation has returned, and will still work correctly

- In the context of concurrent programming...

A reentrant function can be executed simultaneously by more than one thread, with no ill effects

Reentrant Code

- Consider this function...

```
var count: int = 0  
function GetUnique () returns int  
    count = count + 1  
    return count  
endFunction
```

- What if it is executed by different threads concurrently?

Reentrant Code

- Consider this function...

```
var count: int = 0  
function GetUnique () returns int  
    count = count + 1  
    return count  
endFunction
```

- What if it is executed by different threads concurrently?
 - ❖ The results may be incorrect!
 - ❖ This routine is not reentrant!

When is code reentrant?

- ❑ **Some variables are**
 - ❖ "local" -- to the function/method/routine
 - ❖ "global" -- sometimes called "static"
- ❑ **Access to local variables?**
 - ❖ A new stack frame is created for each invocation
 - ❖ Each thread has its own stack
- ❑ **What about access to global variables?**
 - ❖ **Must use synchronization!**

Does this work?

```
var count: int = 0
```

```
    myLock: Mutex
```

```
function GetUnique ( ) returns int
```

```
    myLock.Lock( )
```

```
    count = count + 1
```

```
    myLock.Unlock( )
```

```
    return count
```

```
endFunction
```

What about this?

```
var count: int = 0
```

```
    myLock: Mutex
```

```
function GetUnique ( ) returns int
```

```
    myLock.Lock( )
```

```
    count = count + 1
```

```
    return count
```

```
    myLock.Unlock( )
```

```
endFunction
```

Making this function reentrant

```
var count: int = 0
```

```
    myLock: Mutex
```

```
function GetUnique () returns int
```

```
    var i: int
```

```
    myLock.Lock()
```

```
    count = count + 1
```

```
    i = count
```

```
    myLock.Unlock()
```

```
    return i
```

```
endFunction
```

Message Passing

Message Passing

- ❑ **Interprocess Communication**
 - ❖ via shared memory
 - ❖ across machine boundaries
- ❑ **Message passing can be used for synchronization or general communication**
- ❑ **Processes use *send* and *receive* primitives**
 - ❖ *receive* can block (like *waiting* on a Semaphore)
 - ❖ *send* unblocks a process blocked on *receive* (just as a *signal* unblocks a *waiting* process)

Producer-consumer with message passing

□ The basic idea:

- ❖ After producing, the producer sends the data to consumer in a message
- ❖ The system buffers messages
 - The producer can out-run the consumer
 - The messages will be kept in order
- ❖ But how does the producer avoid overflowing the buffer?
 - We need some kind of flow-control
 - After consuming the data, the consumer sends back an “empty” message
- ❖ A fixed number of messages ($N=100$)
- ❖ The messages circulate back and forth

Producer-consumer with message passing

```
const N = 100                -- Size of message buffer
var em: char
for i = 1 to N                -- Initialize the system by
  Send (producer, &em)        -- sending N empty messages
endFor
```

```
thread consumer
  var c, em: char
  while true
    Receive(producer, &c)      -- Wait for a char
    Send(producer, &em)        -- Send empty message back
    // Consume char...
  endwhile
end
```

Producer-consumer with message passing

```
thread producer
  var c, em: char
  while true
    // Produce char c...
    Receive(consumer, &em)    -- Wait for an empty msg
    Send(consumer, &c)         -- Send c to consumer
  endwhile
end
```


Design choices for message passing

□ Option 1: Mailboxes

- ❖ System maintains a buffer of sent, but not yet received, messages
- ❖ Must specify the size of the mailbox ahead of time
- ❖ Sender will be blocked if the buffer is full
- ❖ Receiver will be blocked if the buffer is empty

Design choices for message passing

- ❑ Option 2: No buffering
 - ❖ If Send happens first, the sending thread blocks
 - ❖ If Receiver happens first, the receiving thread blocks
 - ❖ Sender and receiver must Rendezvous (ie. meet)
 - ❖ Both threads are ready for the transfer
 - ❖ The data is copied / transmitted
 - ❖ Both threads are then allowed to proceed

Resources and deadlocks

- Processes need access to resources in order to make progress
- Examples of computer resources
 - ❖ printers
 - ❖ disk drives
 - ❖ kernel data structures (scheduling queues ...)
 - ❖ locks/semaphores to protect critical sections
- Suppose a process holds resource A and requests resource B
 - ❖ at the same time another process holds B and requests A
 - ❖ both are blocked and remain so ... **this is deadlock**

Deadlock modeling: resource usage model

- **Sequence of events required to use a resource**
 - ❖ **request** the resource (like acquiring a mutex lock)
 - ❖ **use** the resource
 - ❖ **release** the resource (like releasing a mutex lock)

- **Must **wait** if request is denied**
 - ❖ block
 - ❖ busy wait
 - ❖ fail with error code

Preemptable vs nonpreemptable resources

- **Preemptable resources**
 - ❖ can be taken away from a process with no ill effects
- **Nonpreemptable resources**
 - ❖ will cause the holding process to fail if taken away
 - ❖ May corrupt the resource itself
- **Deadlocks occur when processes are granted exclusive access to non-preemptable resources and wait when the resource is not available**

Definition of deadlock

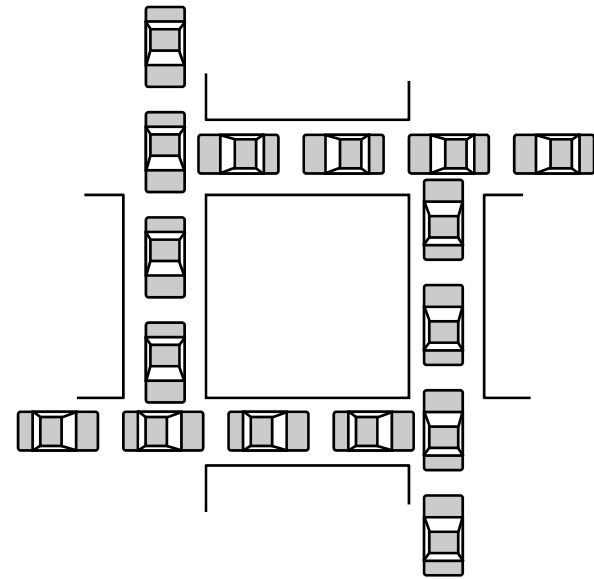
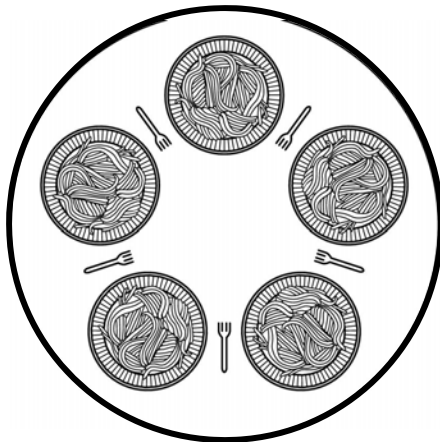
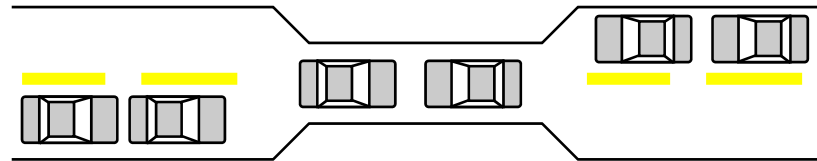
A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

- Usually the event is the release of a currently held resource
- None of the processes can ...
 - ❖ be awakened
 - ❖ run
 - ❖ release 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)

Examples of deadlock



Resource acquisition scenarios

Thread A:

```
acquire (resource_1)  
use resource_1  
release (resource_1)
```

Example:

```
var r1_mutex: Mutex  
...  
r1_mutex.Lock()  
Use resource_1  
r1_mutex.Unlock()
```

Resource acquisition scenarios

Thread A:

```
acquire (resource_1)
use resource_1
release (resource_1)
```

Another Example:

```
var r1_sem: Semaphore
r1_sem.Signal()
...
r1_sem.Wait()
Use resource_1
r1_sem.Signal()
```

Resource acquisition scenarios

Thread A:

```
acquire (resource_1)  
use resource_1  
release (resource_1)
```

Thread B:

```
acquire (resource_2)  
use resource_2  
release (resource_2)
```

Resource acquisition scenarios

Thread A:

```
acquire (resource_1)
use resource_1
release (resource_1)
```

Thread B:

```
acquire (resource_2)
use resource_2
release (resource_2)
```

No deadlock can occur here!

Resource acquisition scenarios: 2 resources

Thread A:

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

Thread B:

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

Resource acquisition scenarios: 2 resources

Thread A:

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

Thread B:

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

No deadlock can occur here!

Resource acquisition scenarios: 2 resources

Thread A:

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

Thread B:

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

Resource acquisition scenarios: 2 resources

Thread A:

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

Thread B:

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

No deadlock can occur here!

Resource acquisition scenarios: 2 resources

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)
```

Resource acquisition scenarios: 2 resources

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)
```

Deadlock is possible!

Consequences of deadlock

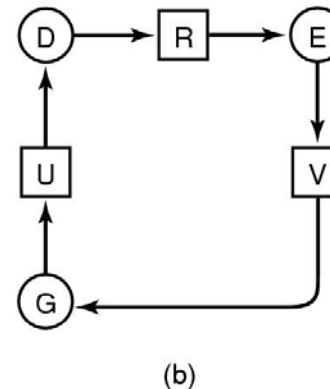
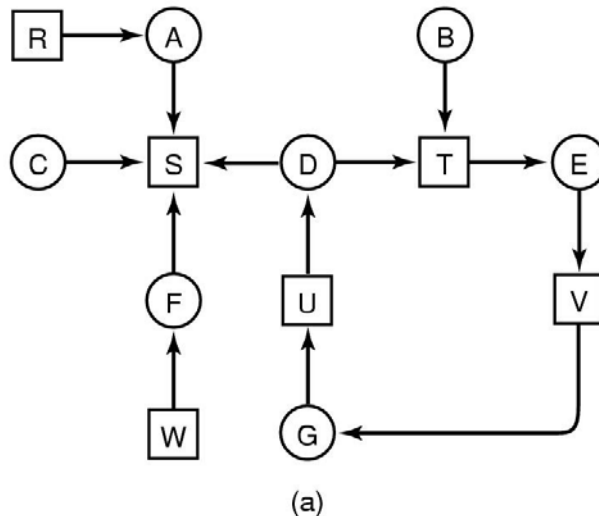
- ❑ **Deadlock occurs in a single program**
 - ❖ Programmer creates a situation that deadlocks
 - ❖ Kill the program and move on
 - ❖ Not a big deal
- ❑ **Deadlock occurs in the Operating System**
 - ❖ Spin locks and locking mechanisms are mismanaged within the OS
 - ❖ Threads become frozen
 - ❖ System hangs or crashes
 - ❖ Must restart the system and kill all applications

Dealing with deadlock

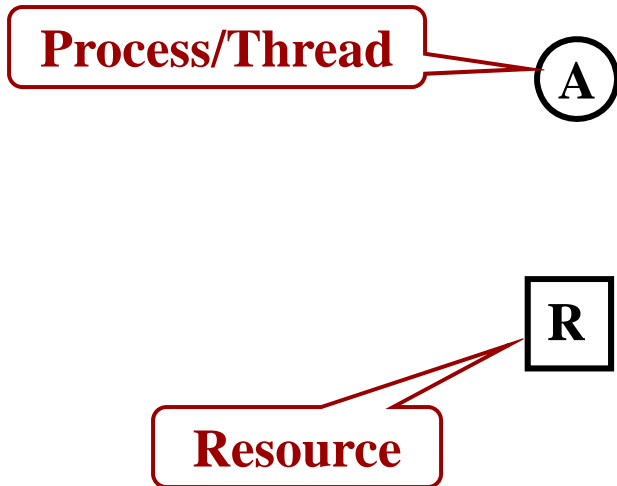
- **Four general strategies**
 - ❖ Ignore the problem
 - Hmm... advantages, disadvantages?
 - ❖ Detection and recovery
 - ❖ Dynamic avoidance via careful resource allocation
 - ❖ Prevention, by structurally negating one of the four necessary conditions

Deadlock detection

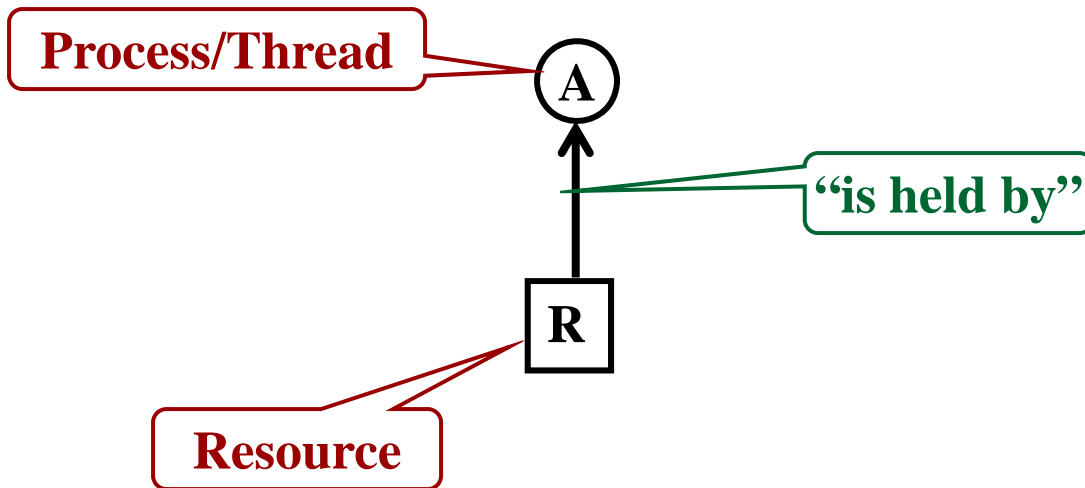
- ❑ Let the problem happen, then recover
- ❑ How do you know it happened?
- ❑ Do a depth-first-search on the resource allocation graph



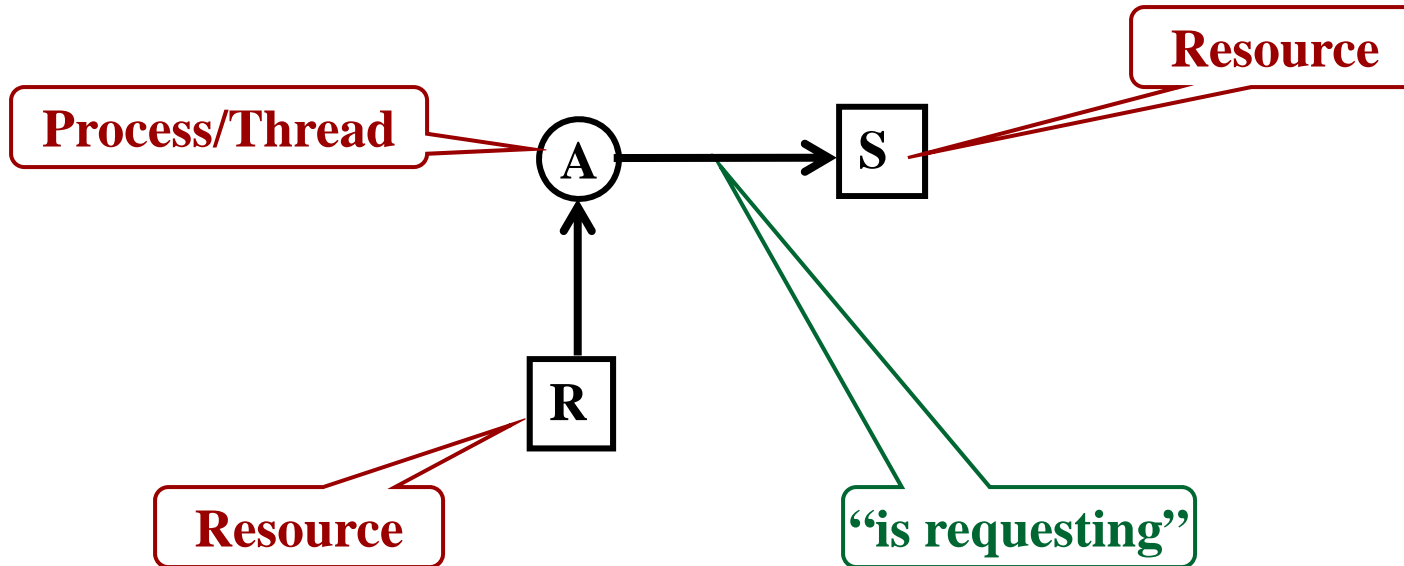
Detection: Resource Allocation Graphs



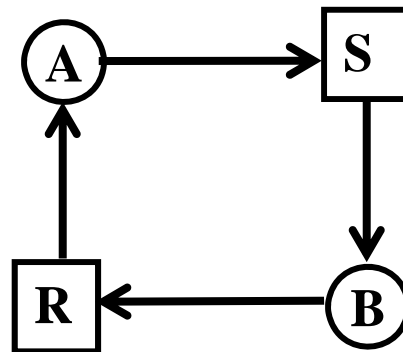
Detection: Resource Allocation Graphs



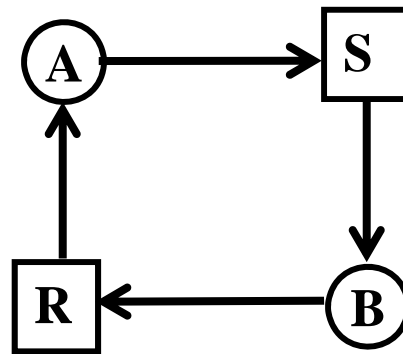
Detection: Resource Allocation Graphs



Detection: Resource Allocation Graphs

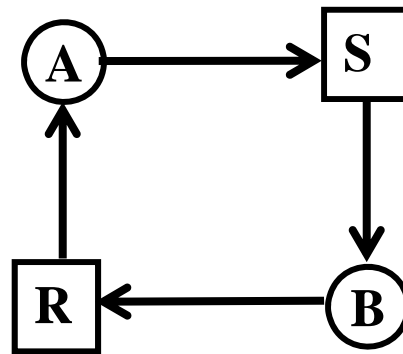


Detection: Resource Allocation Graphs



Deadlock

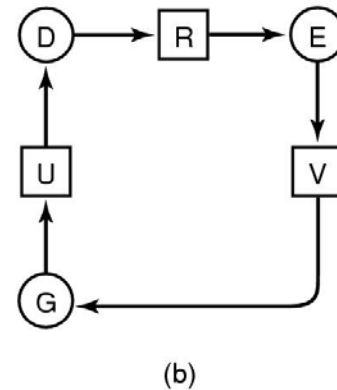
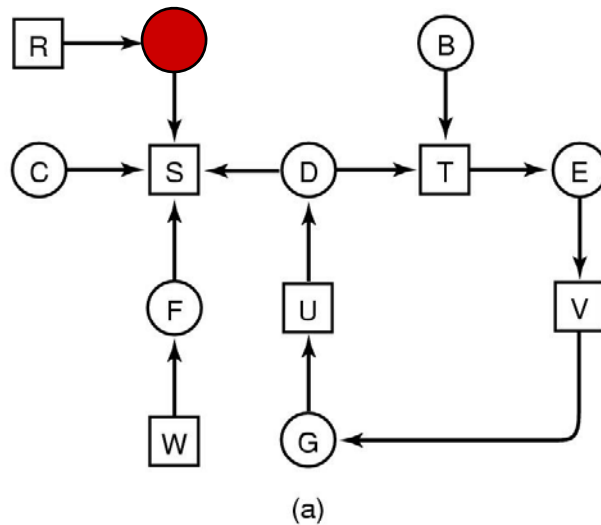
Detection: Resource Allocation Graphs



Deadlock = a cycle in the graph

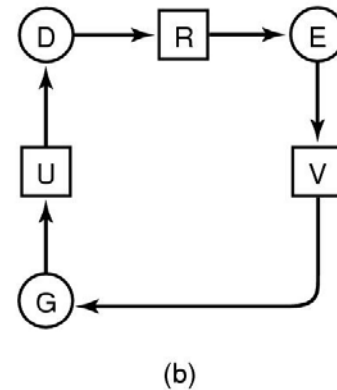
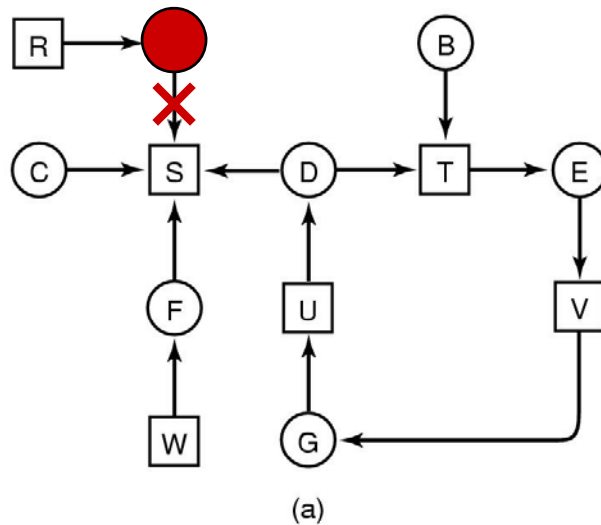
Deadlock detection (1 resource of each)

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



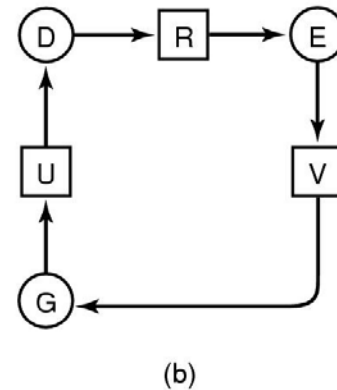
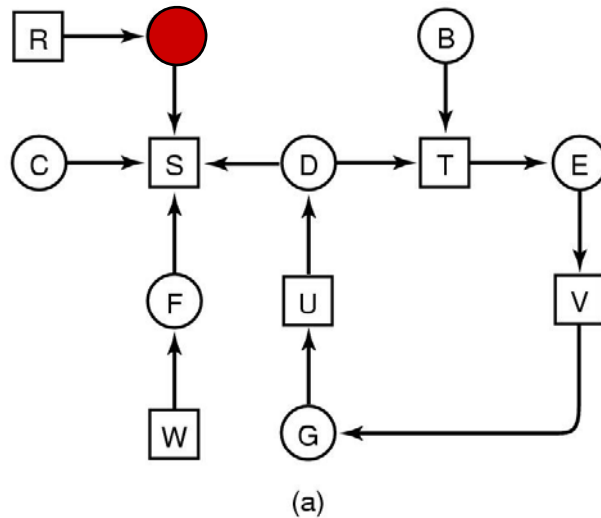
Deadlock detection (1 resource of each)

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



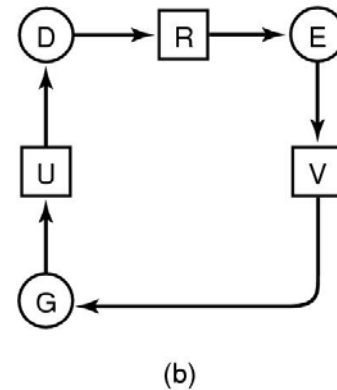
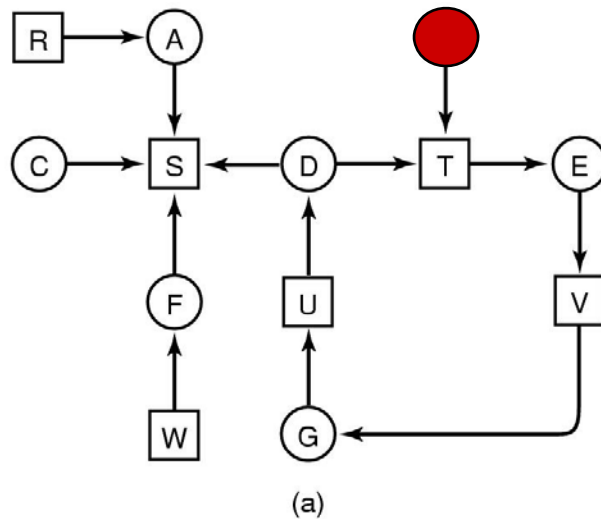
Deadlock detection (1 resource of each)

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



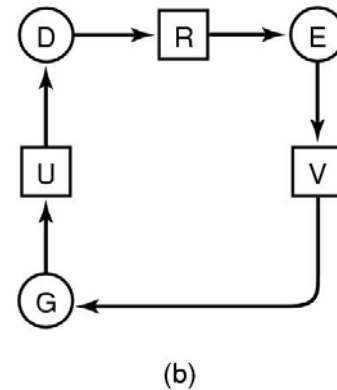
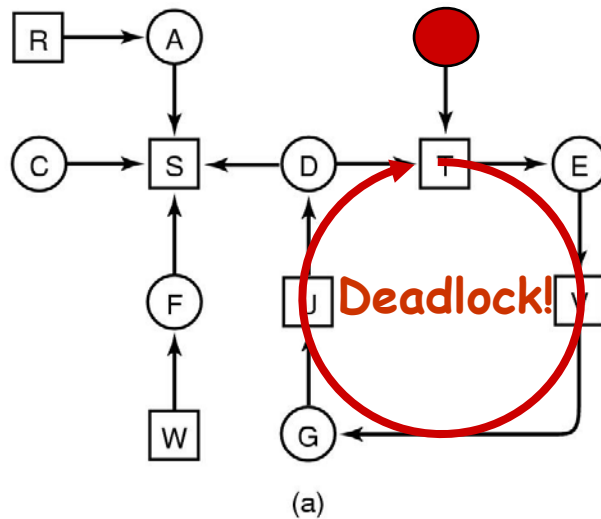
Deadlock detection (1 resource of each)

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



Deadlock detection (1 resource of each)

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



Multple units/instances of a resource

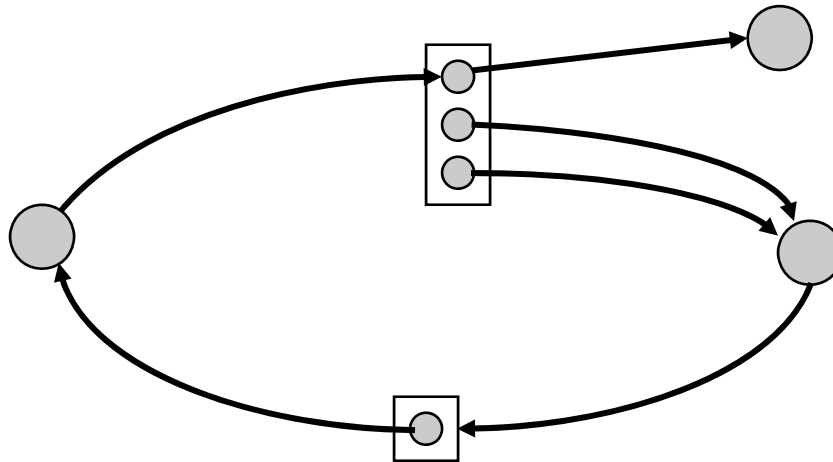
- ❑ **Some resources have only one “unit”.**
 - ❖ Only one thread at a time may hold the resource.
 - Printer
 - Lock on ReadyQueue
- ❑ **Some resources have several units.**
 - ❖ All units are considered equal; any one will do.
 - Page Frames
 - Dice in the Gaming Parlor problem
 - ❖ A thread requests “k” units of the resource.
 - ❖ Several requests may be satisfied simultaneously.

Deadlock modeling with multiple resources

- 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?

Deadlock modeling with multiple resources

- **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?



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?

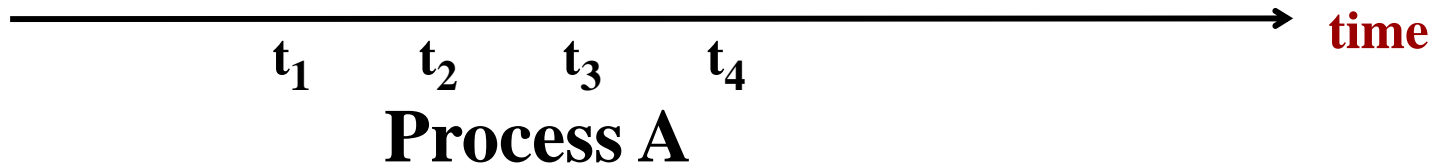
Other deadlock recovery techniques

- ❑ **How do we prevent the resource becoming corrupted**
 - ❖ For example, shared variables protected by a lock?
- ❑ **Recovery through preemption and rollback**
 - ❖ Save state periodically (at start of critical section)
 - take a checkpoint of memory
 - start computation again from checkpoint
 - Checkpoint must be prior to resource acquisition!
 - ❖ Useful for long-lived computation systems

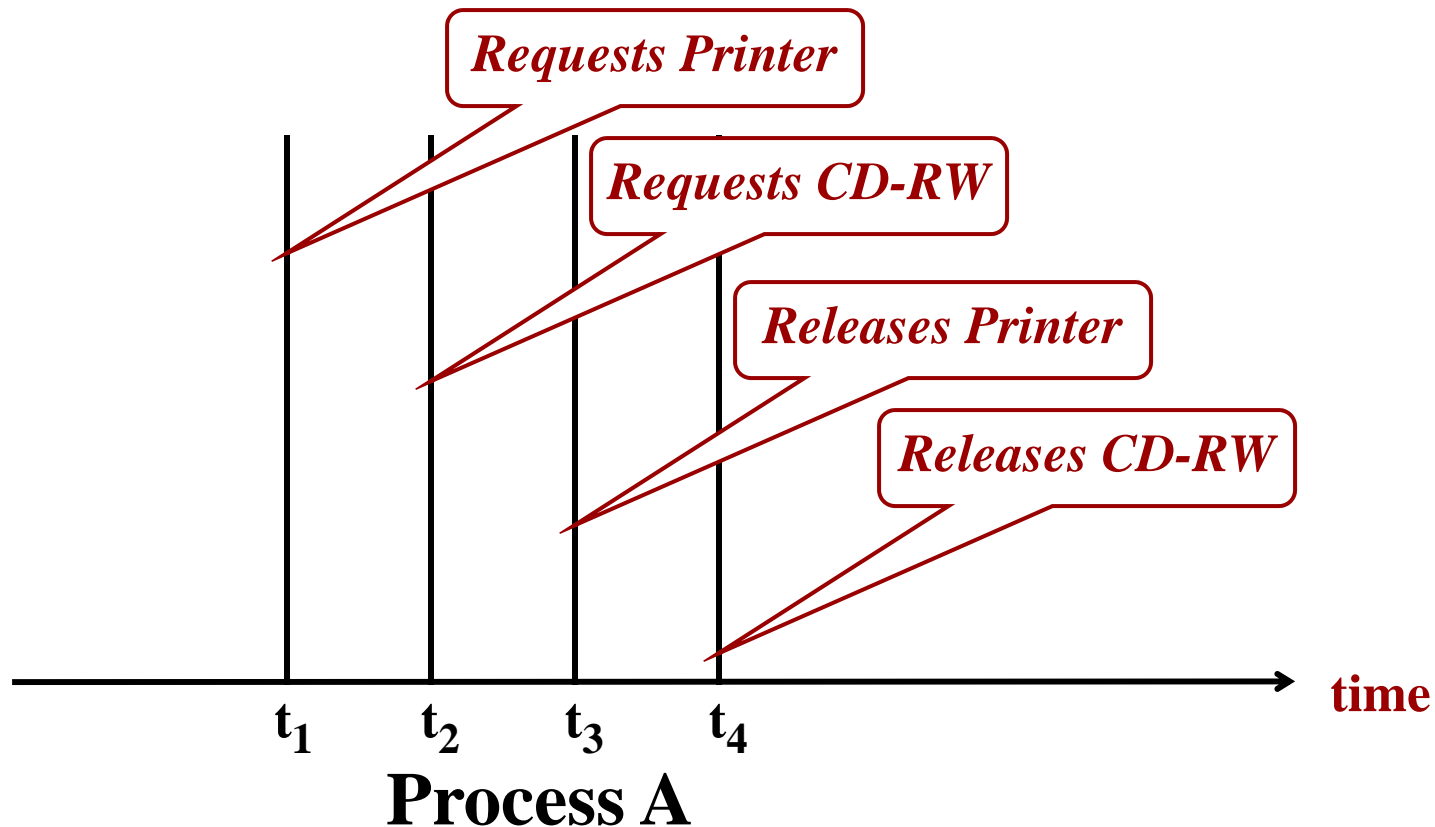
Deadlock avoidance

- Detection vs. 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!

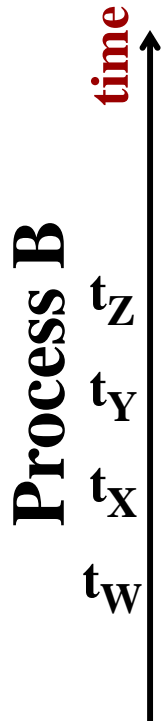
Avoidance using process-resource trajectories



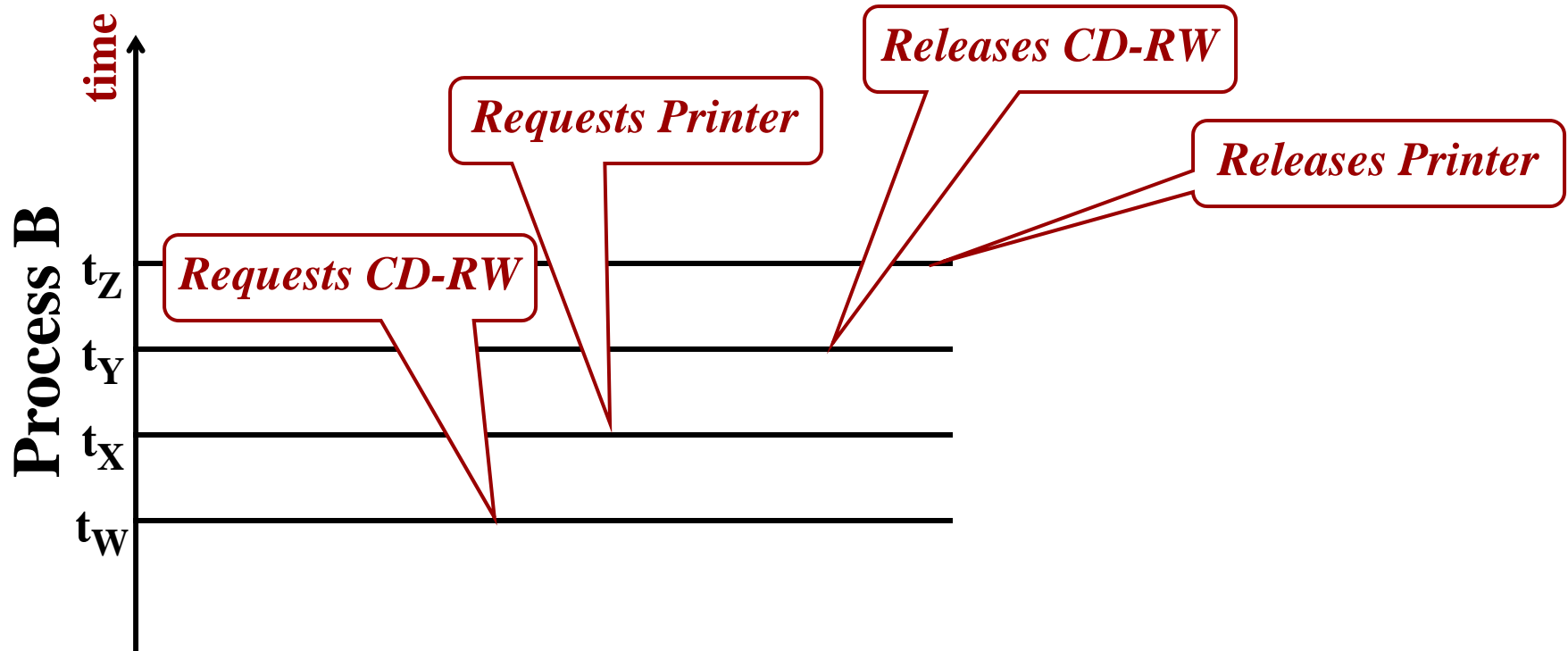
Avoidance using process-resource trajectories



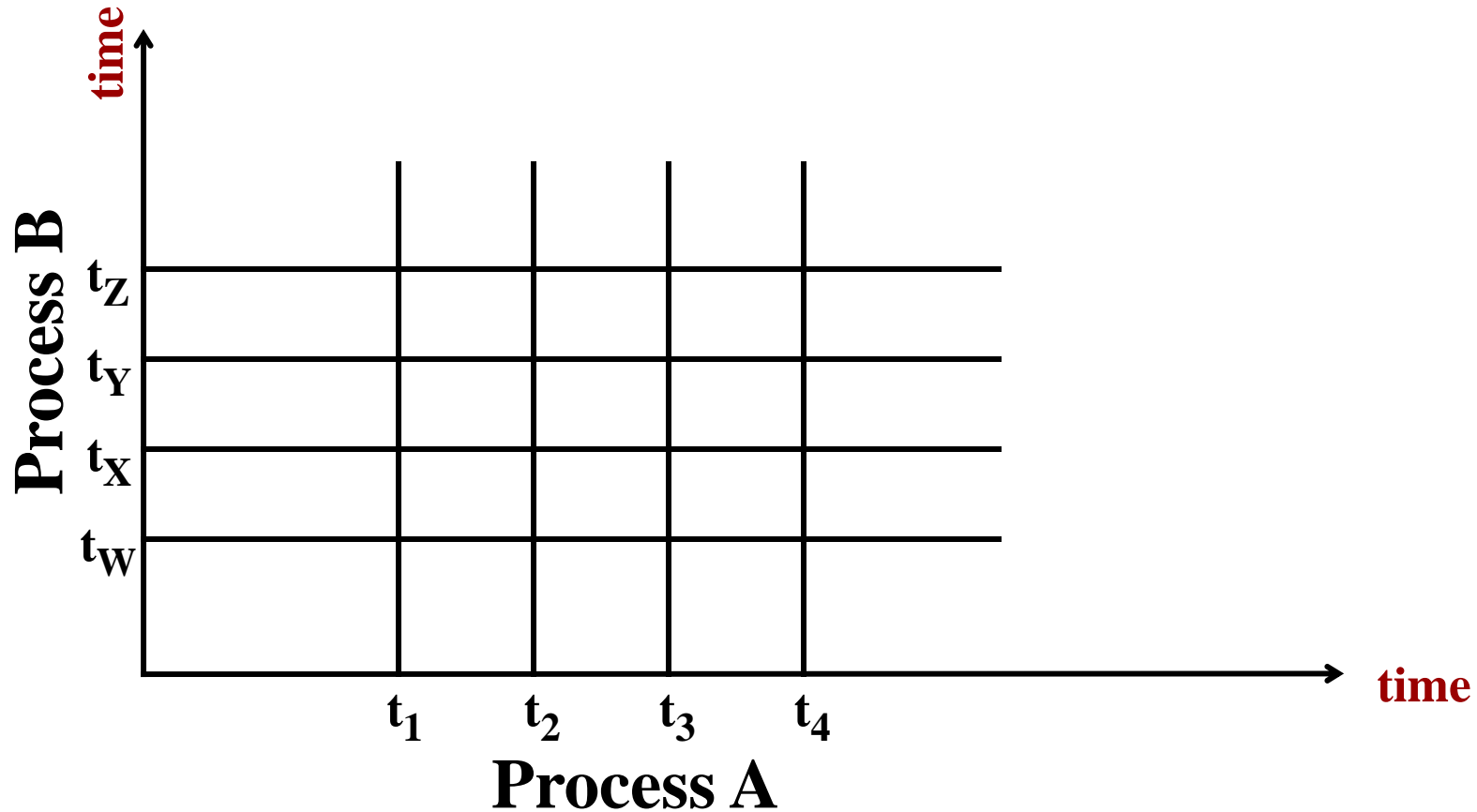
Avoidance using process-resource trajectories



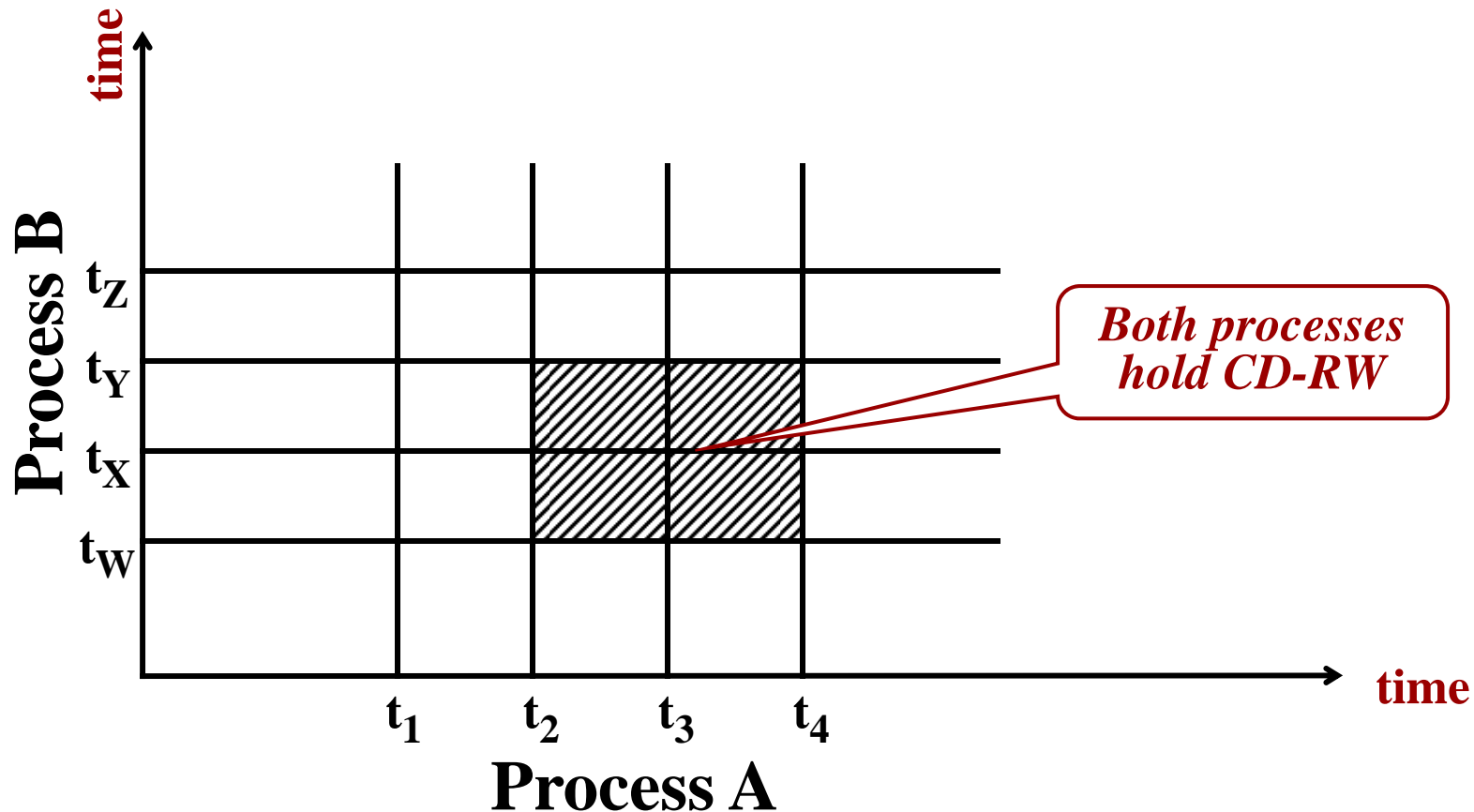
Avoidance using process-resource trajectories



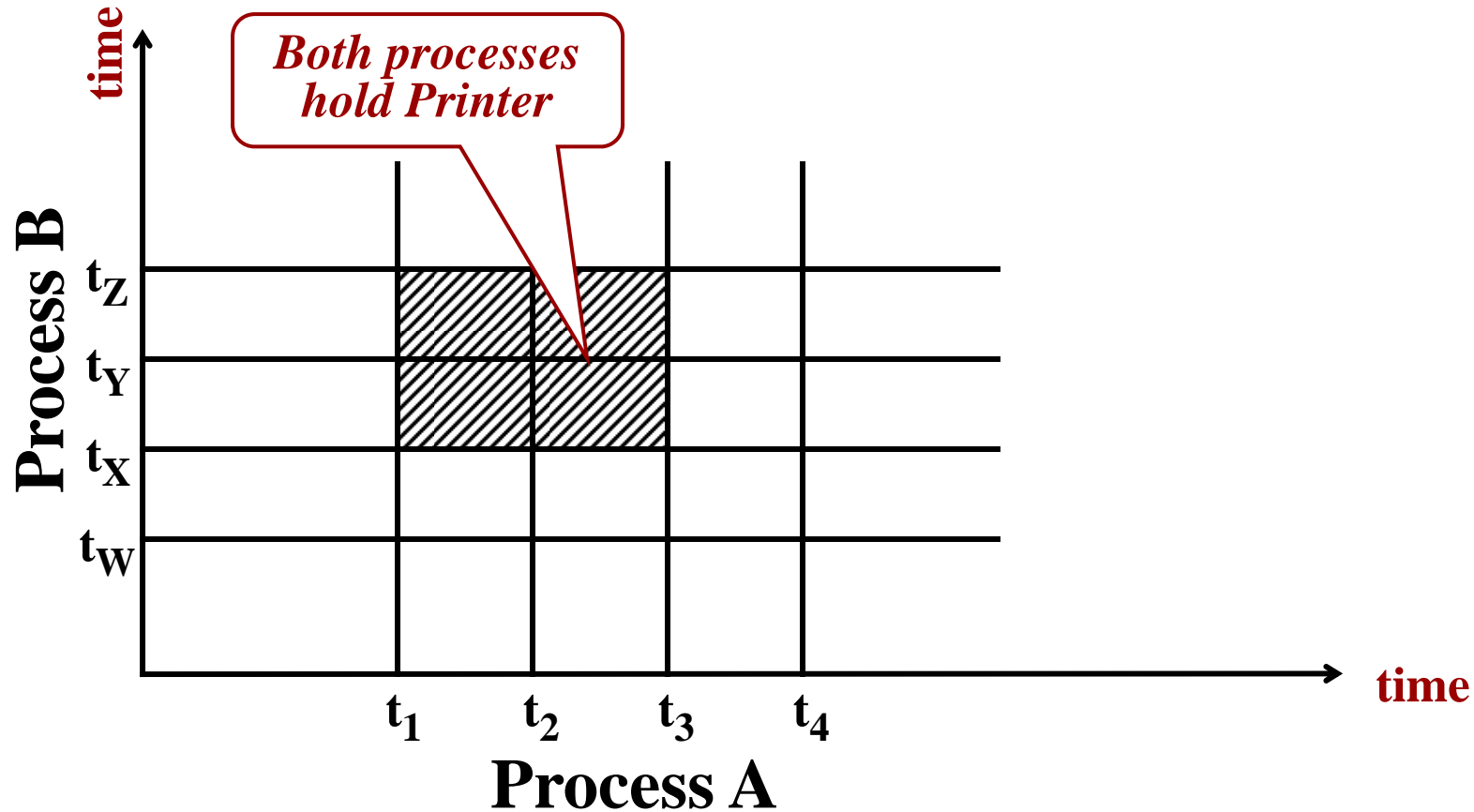
Avoidance using process-resource trajectories



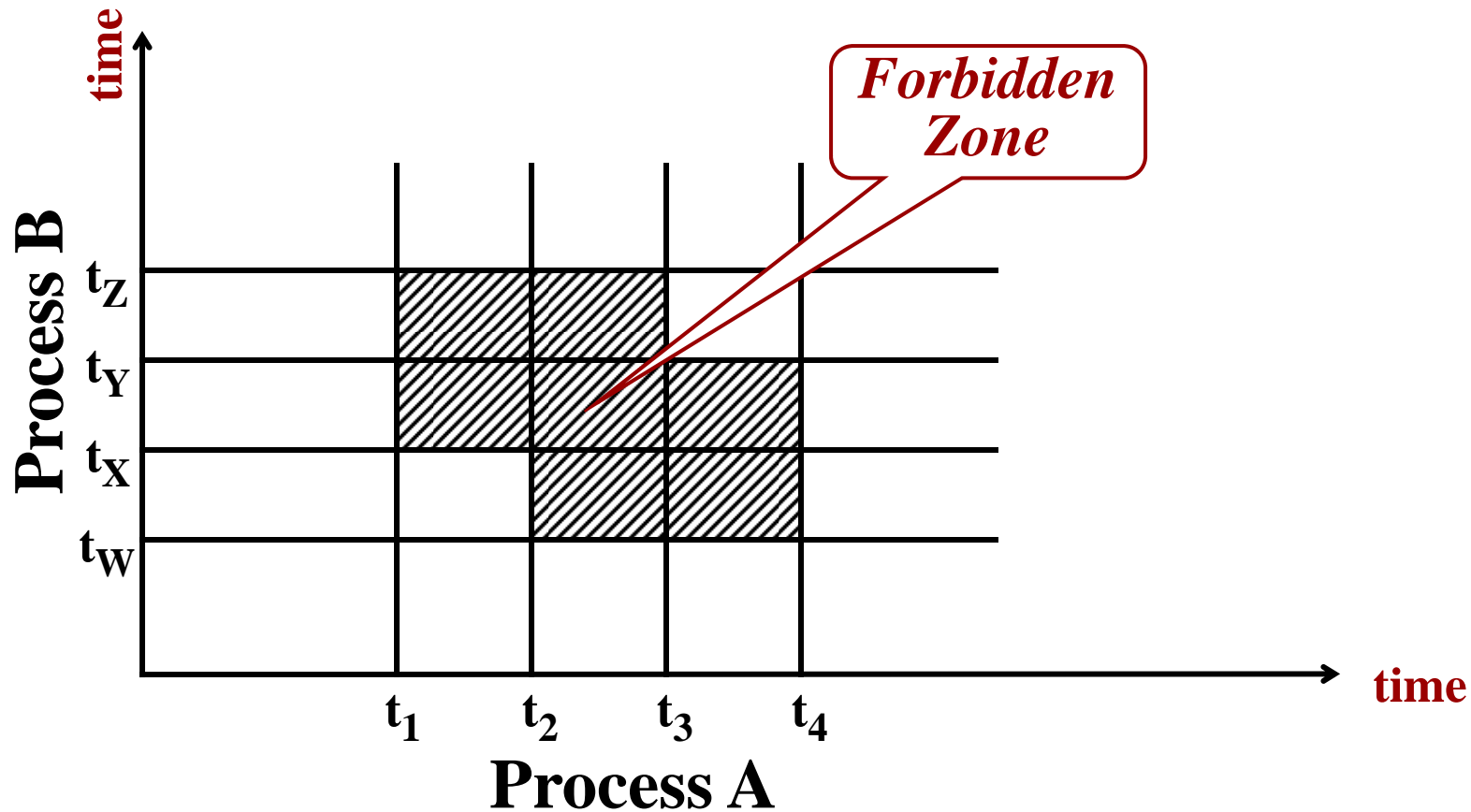
Avoidance using process-resource trajectories



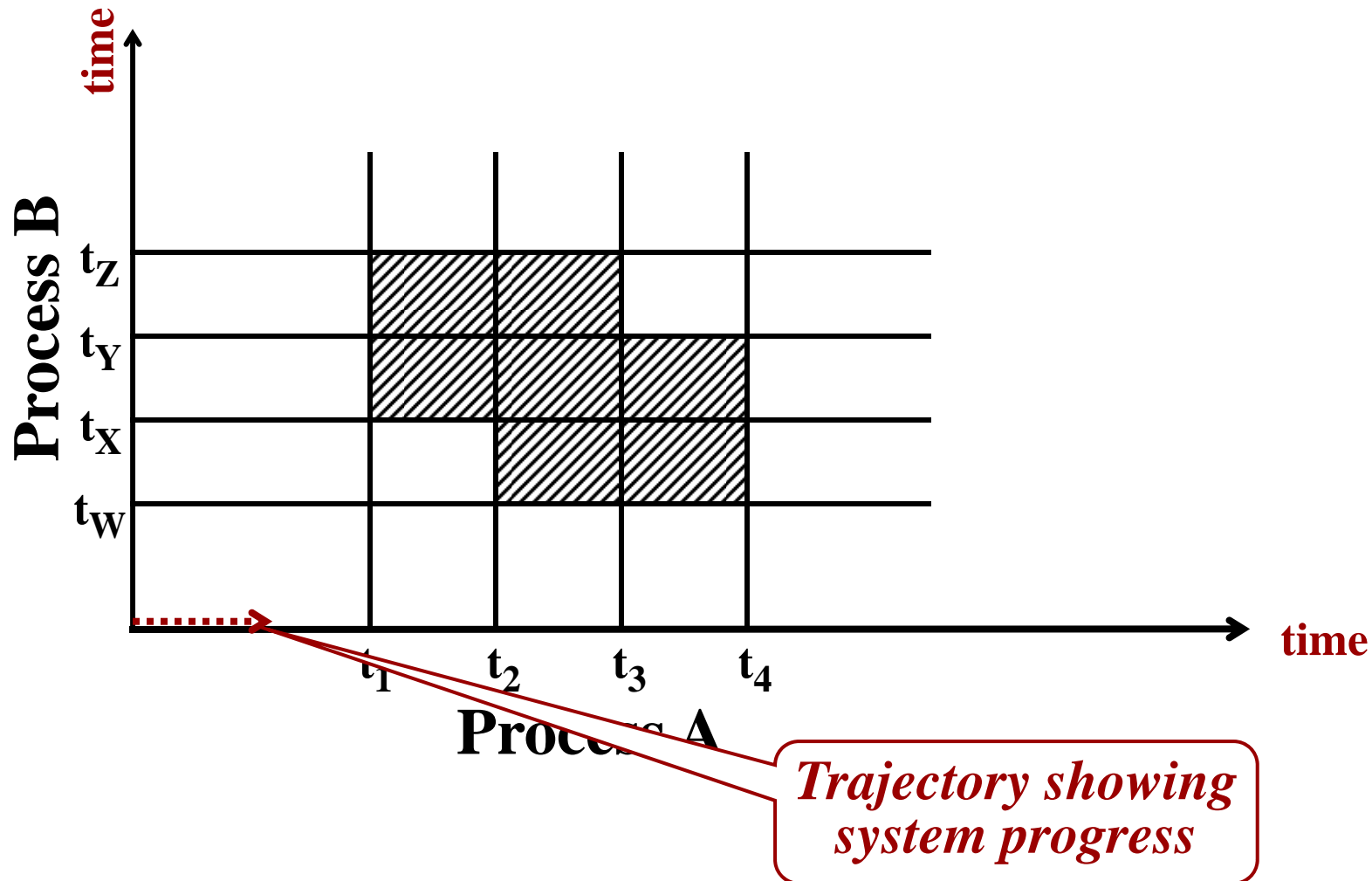
Avoidance using process-resource trajectories



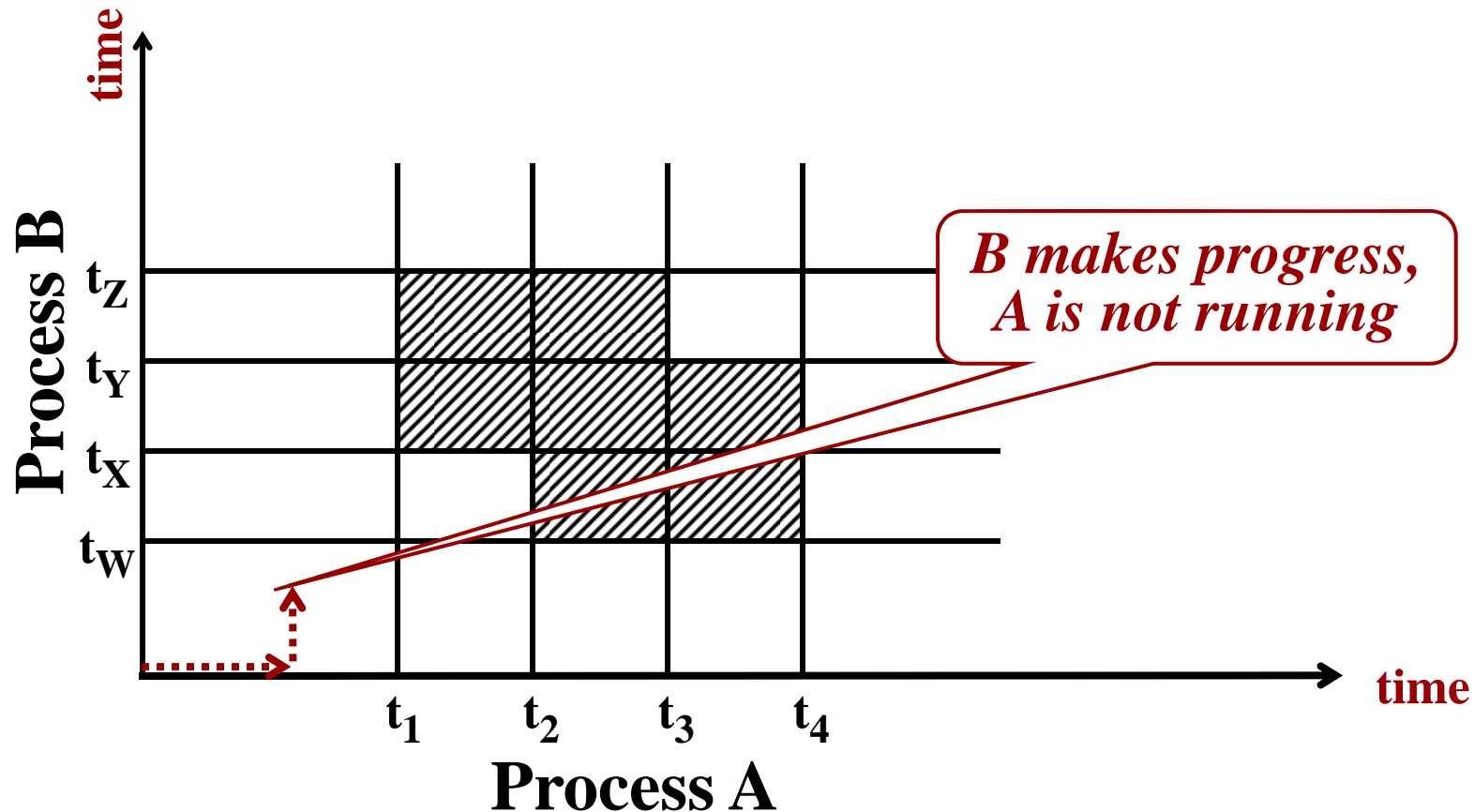
Avoidance using process-resource trajectories



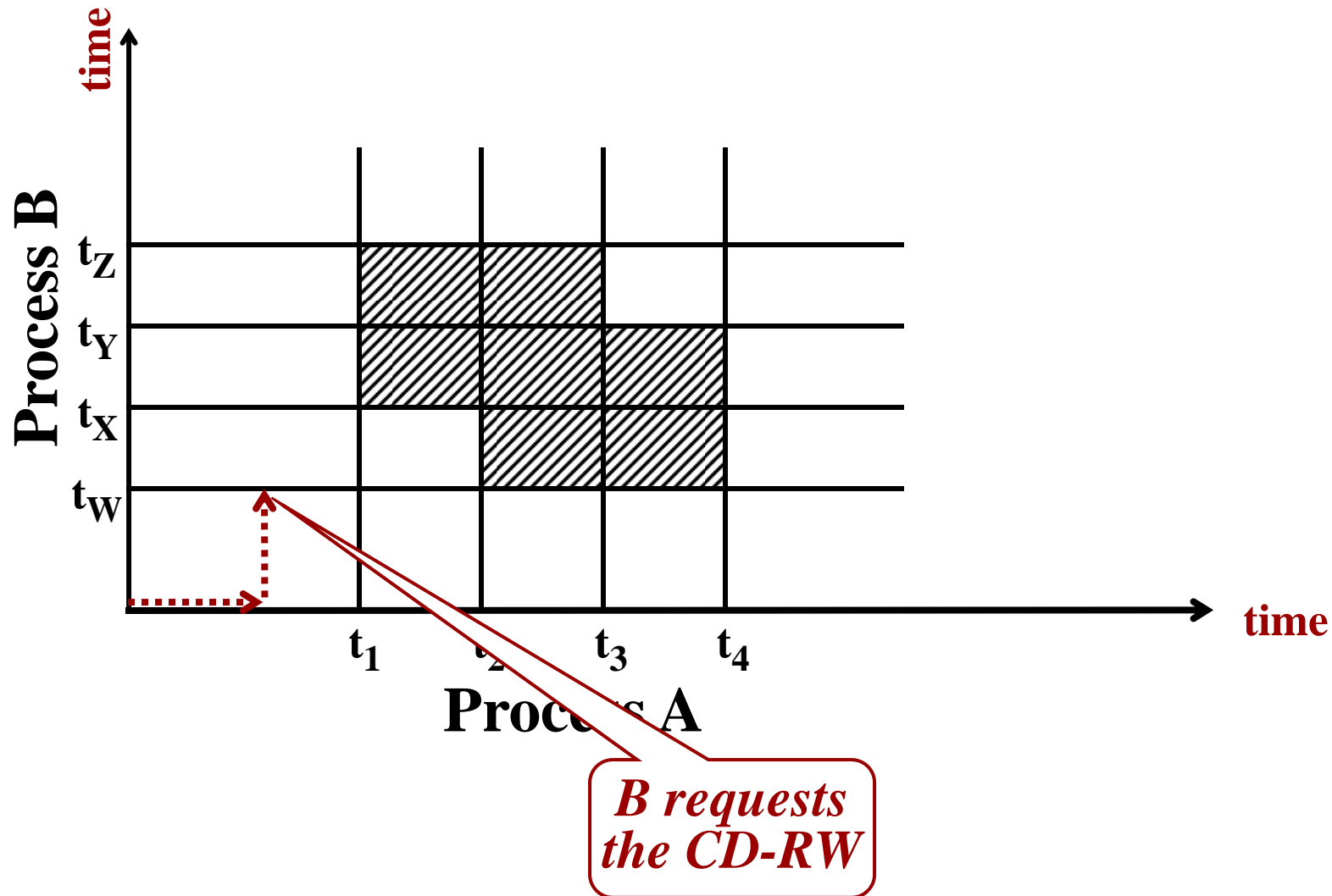
Avoidance using process-resource trajectories



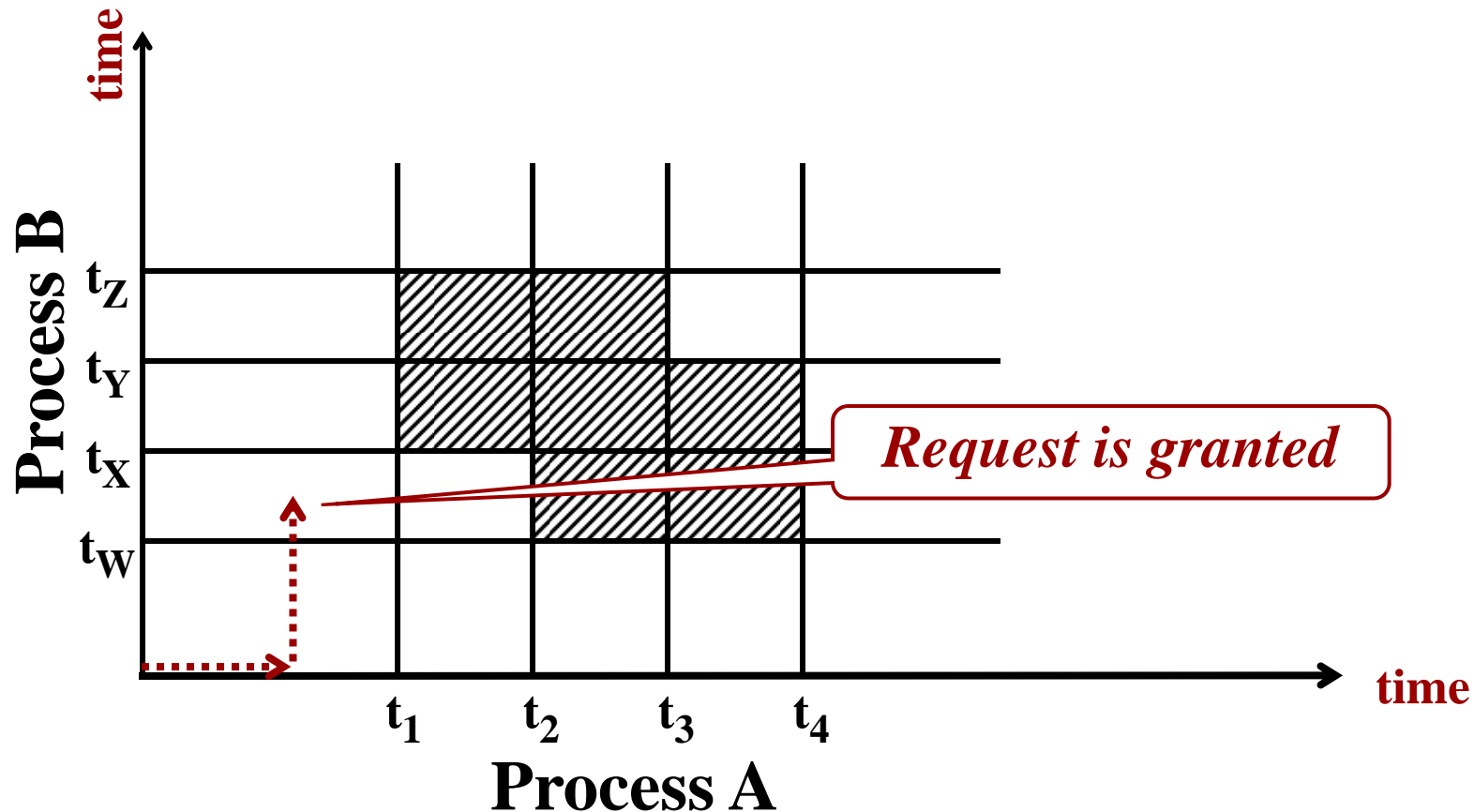
Avoidance using process-resource trajectories



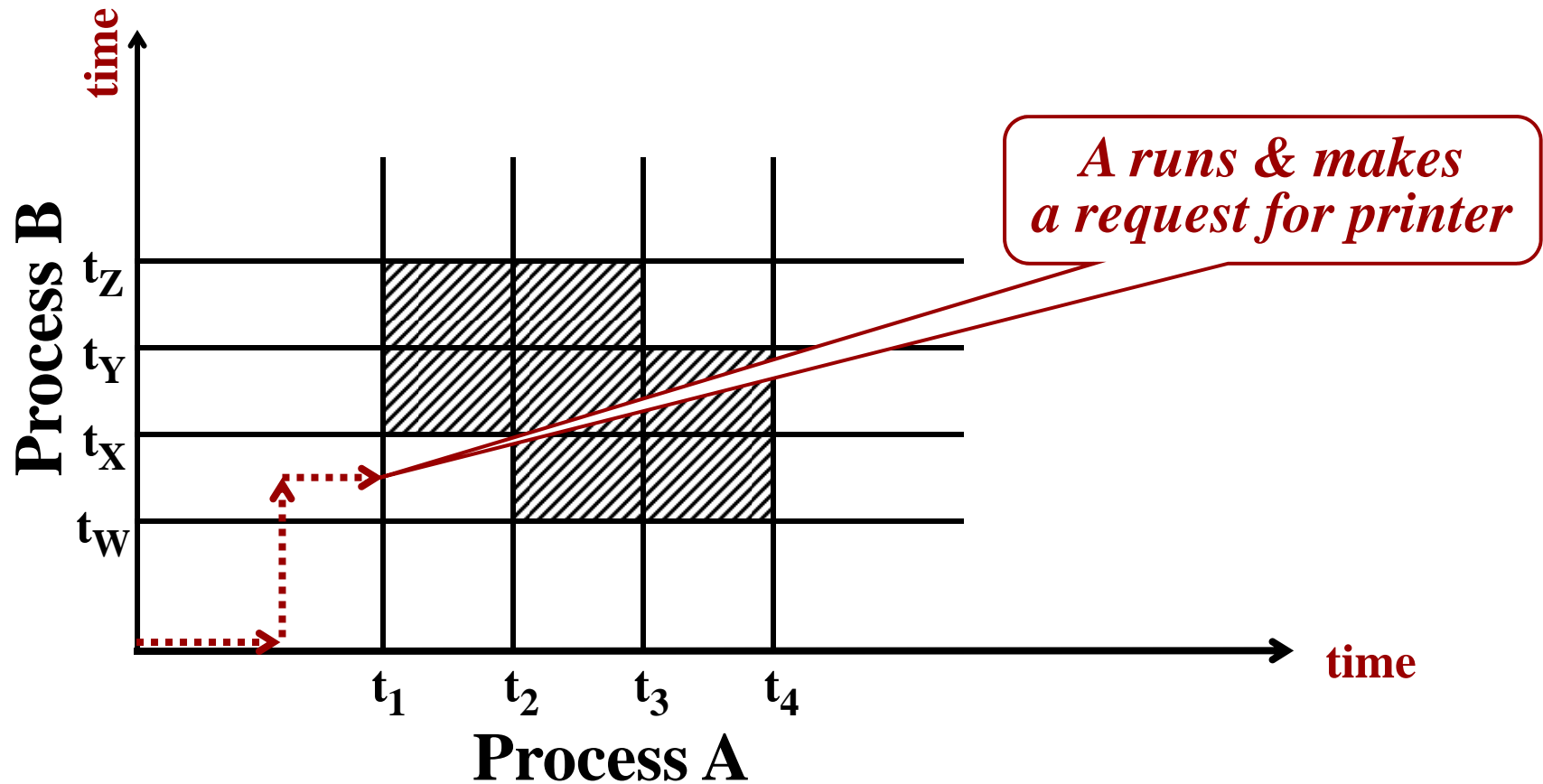
Avoidance using process-resource trajectories



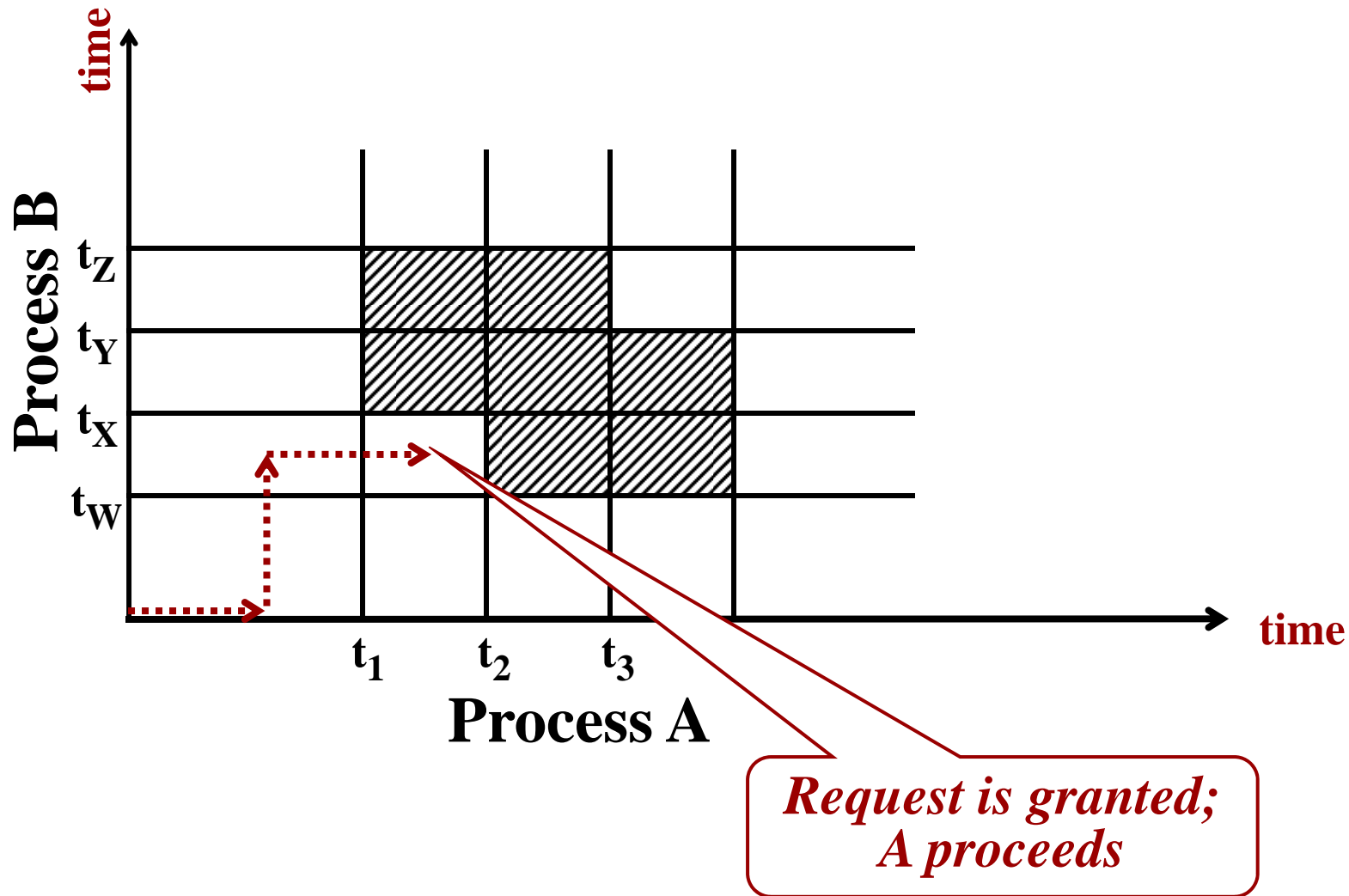
Avoidance using process-resource trajectories



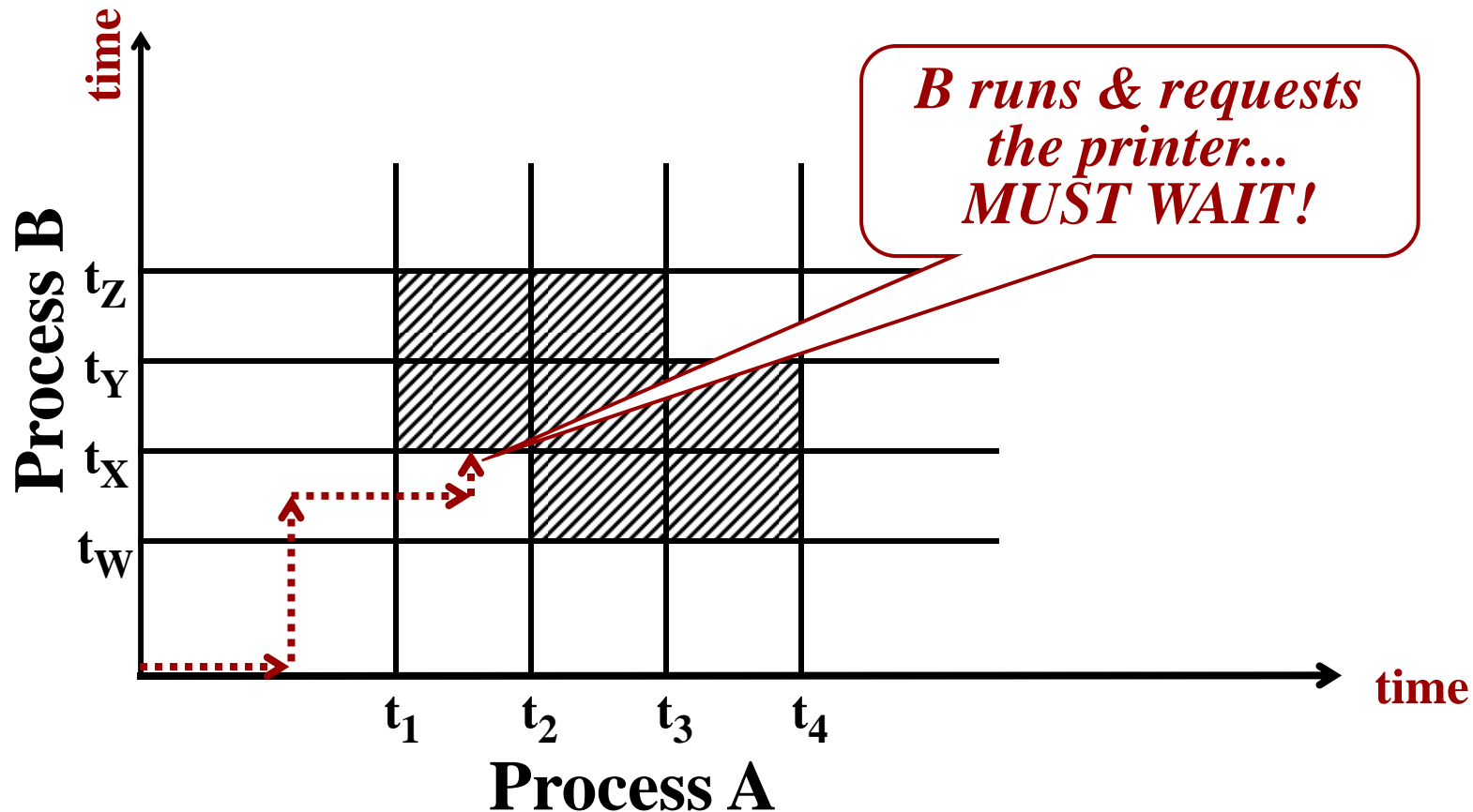
Avoidance using process-resource trajectories



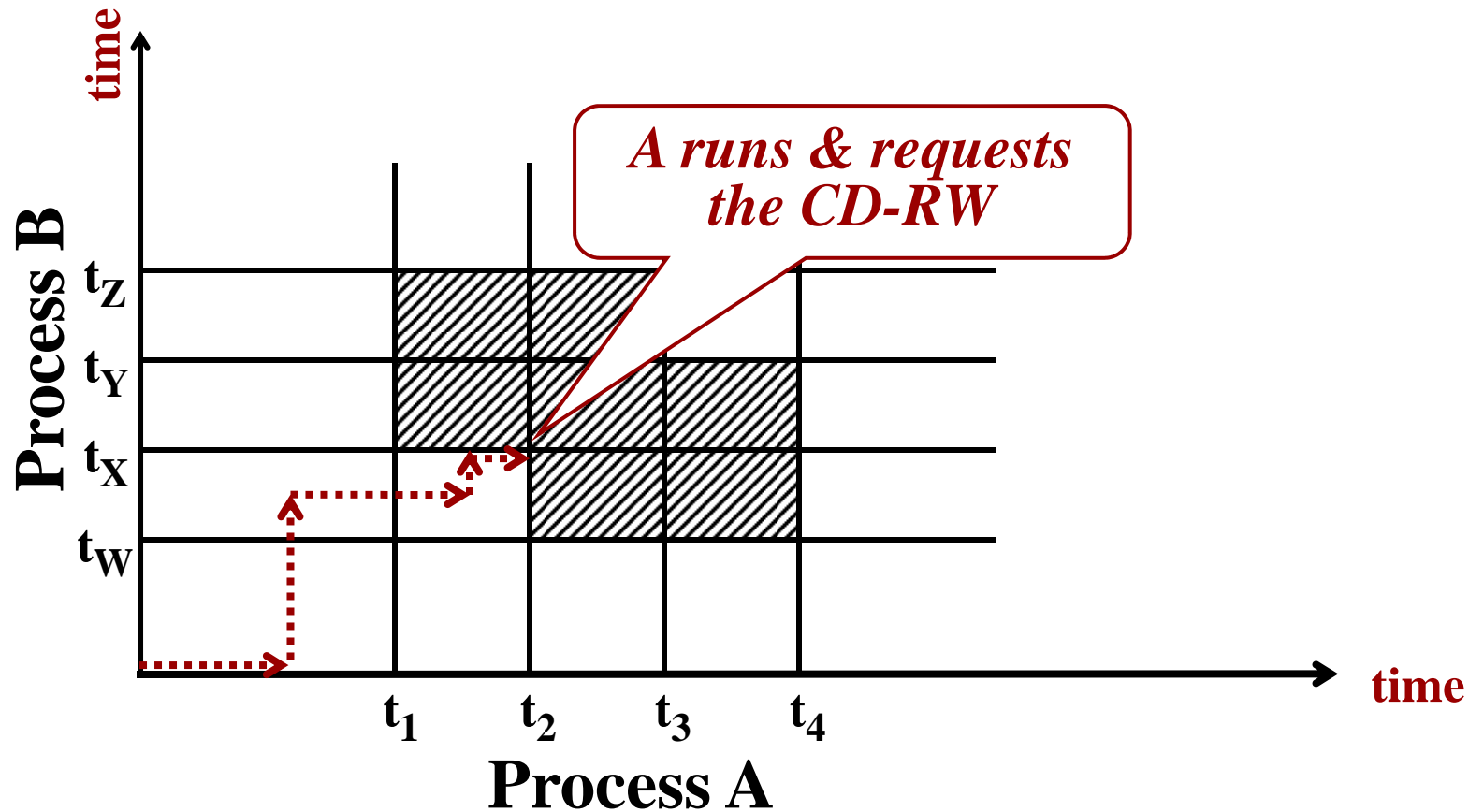
Avoidance using process-resource trajectories



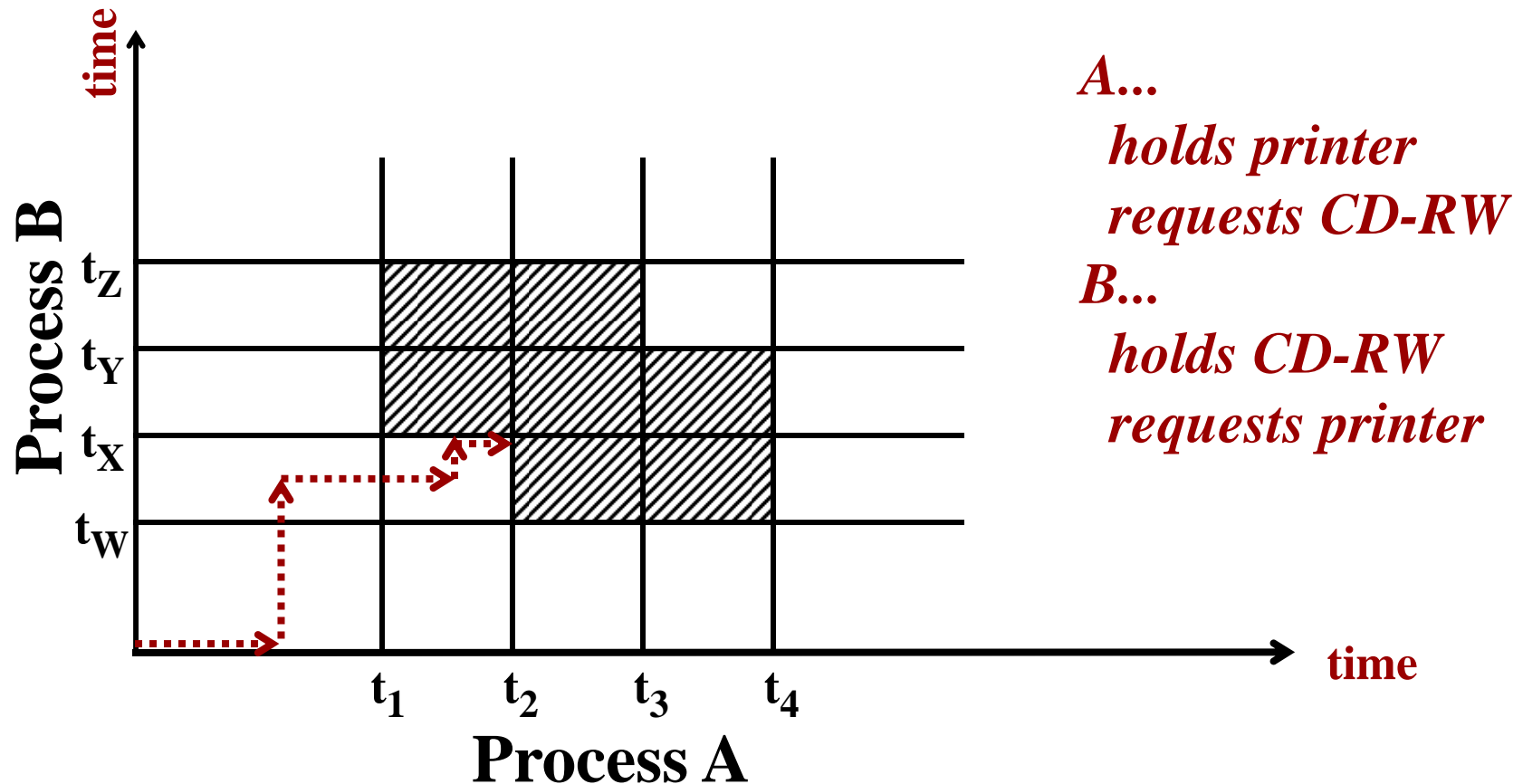
Avoidance using process-resource trajectories



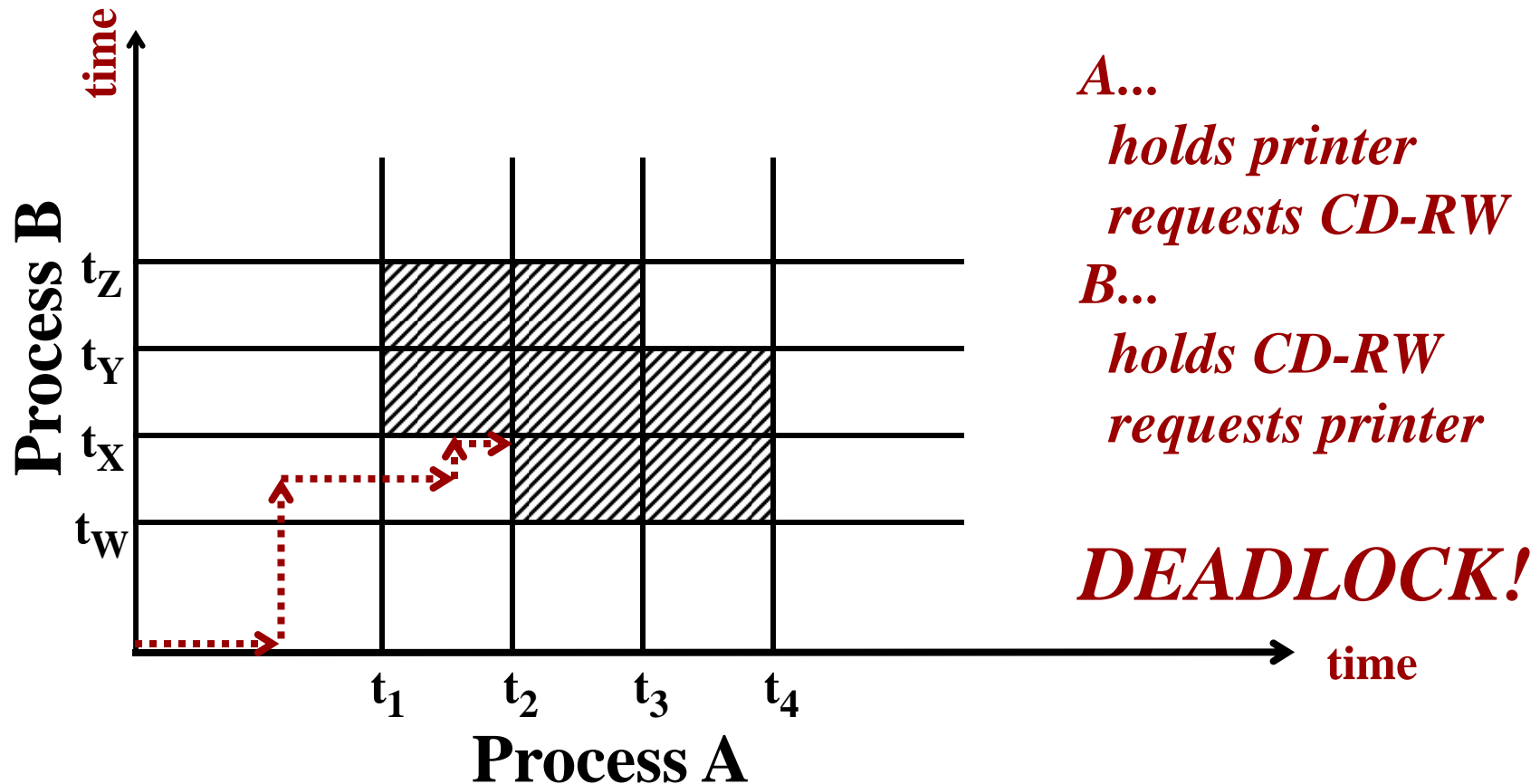
Avoidance using process-resource trajectories



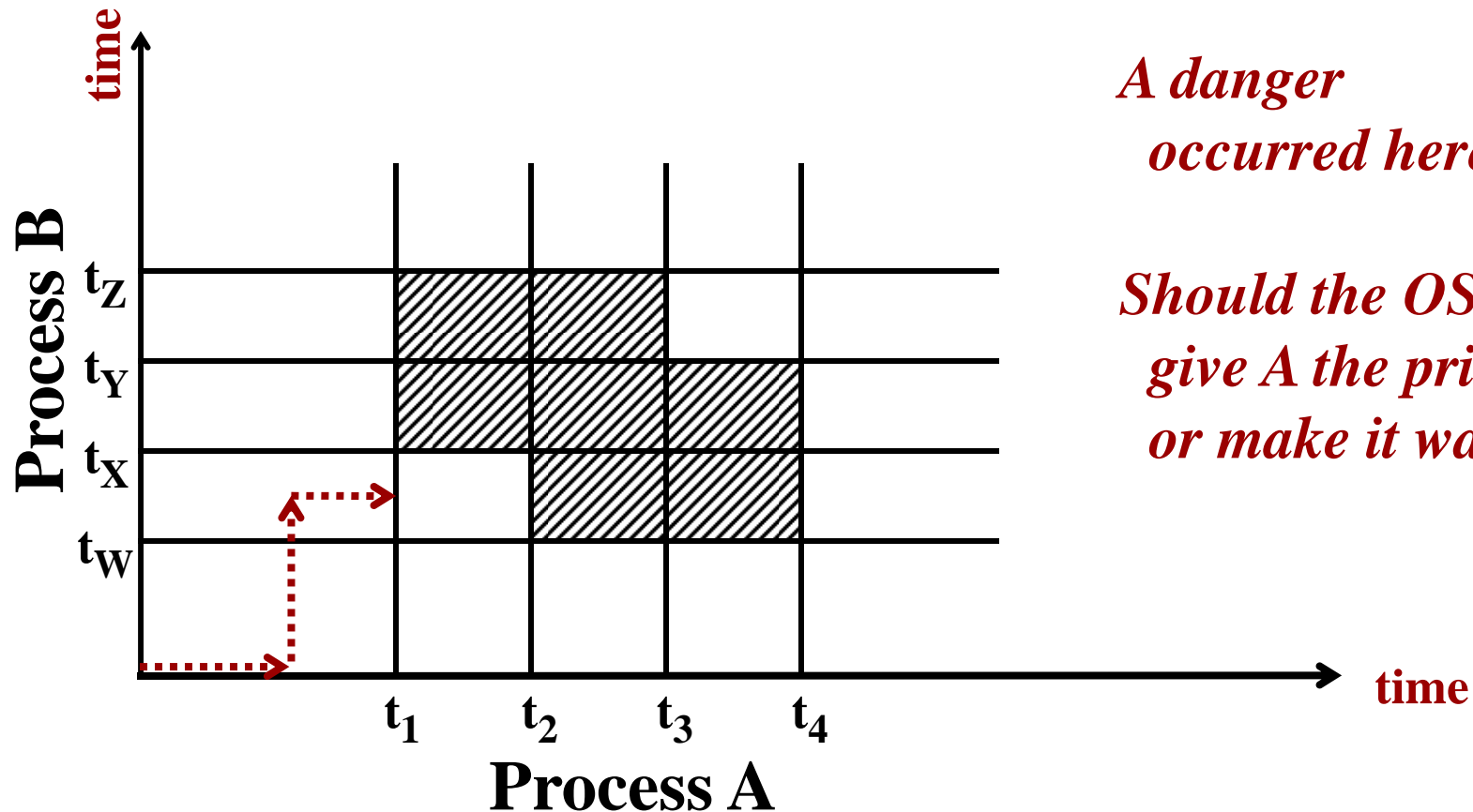
Avoidance using process-resource trajectories



Avoidance using process-resource trajectories



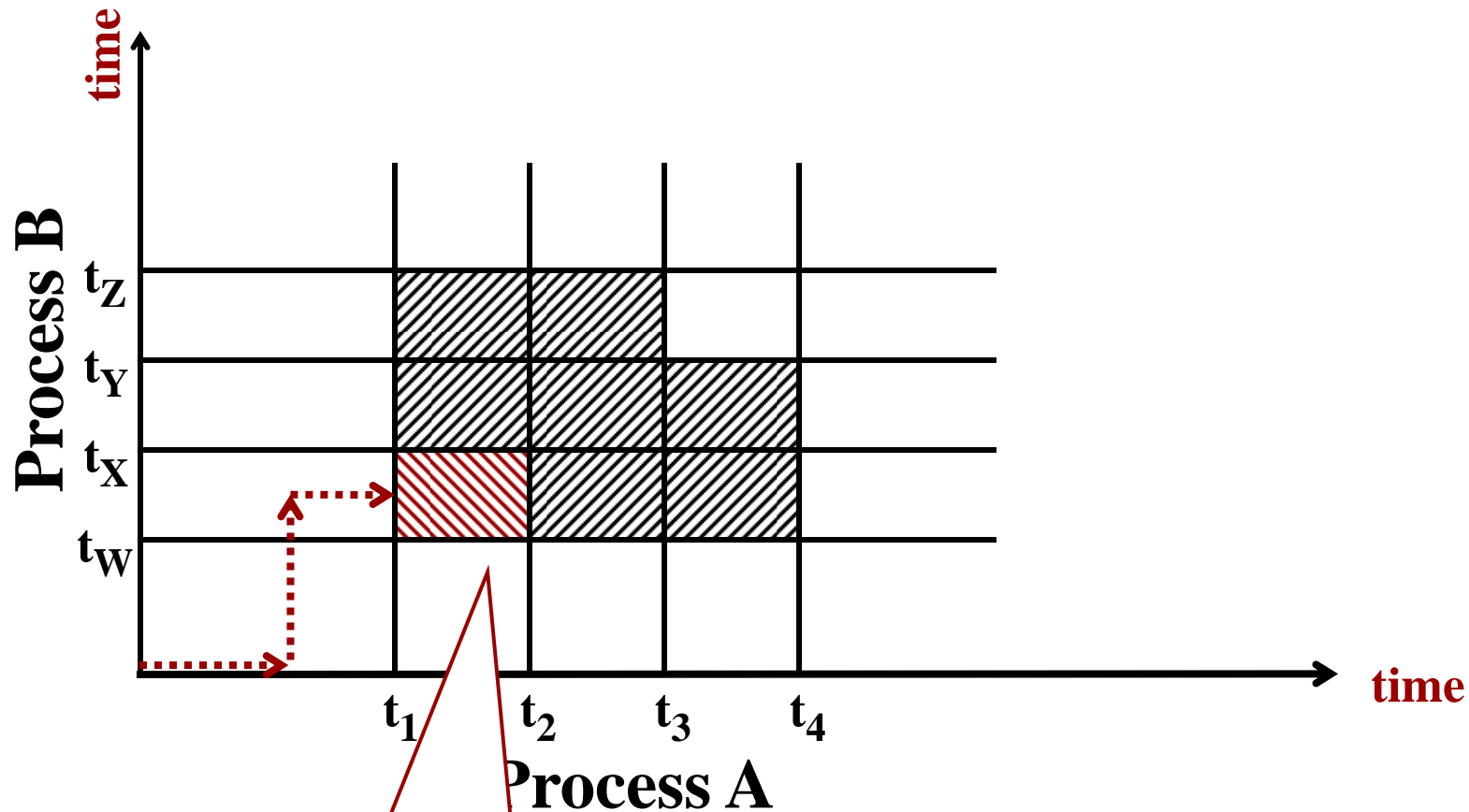
Avoidance using process-resource trajectories



*A danger
occurred here.*

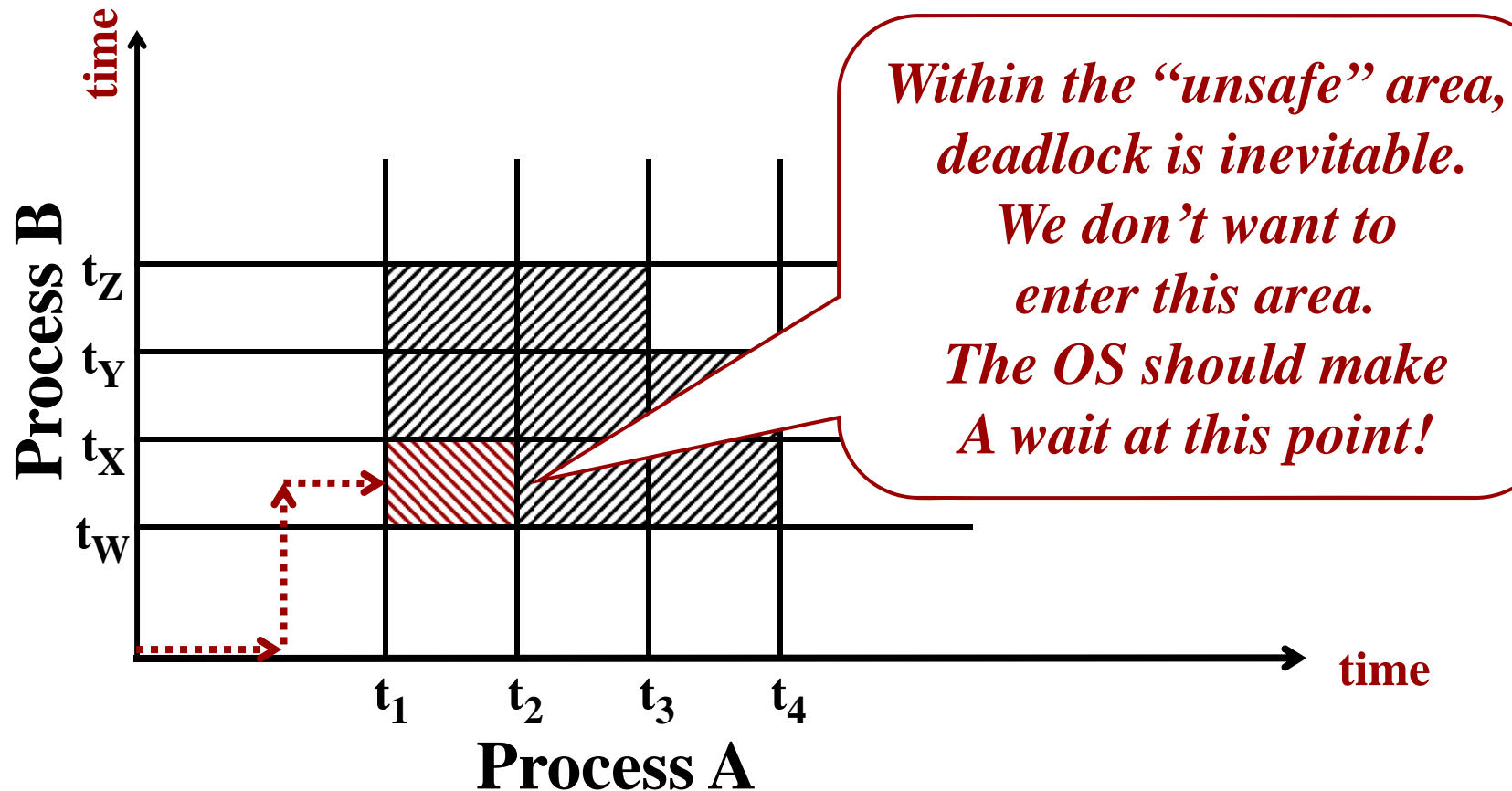
*Should the OS
give A the printer,
or make it wait???*

Avoidance using process-resource trajectories

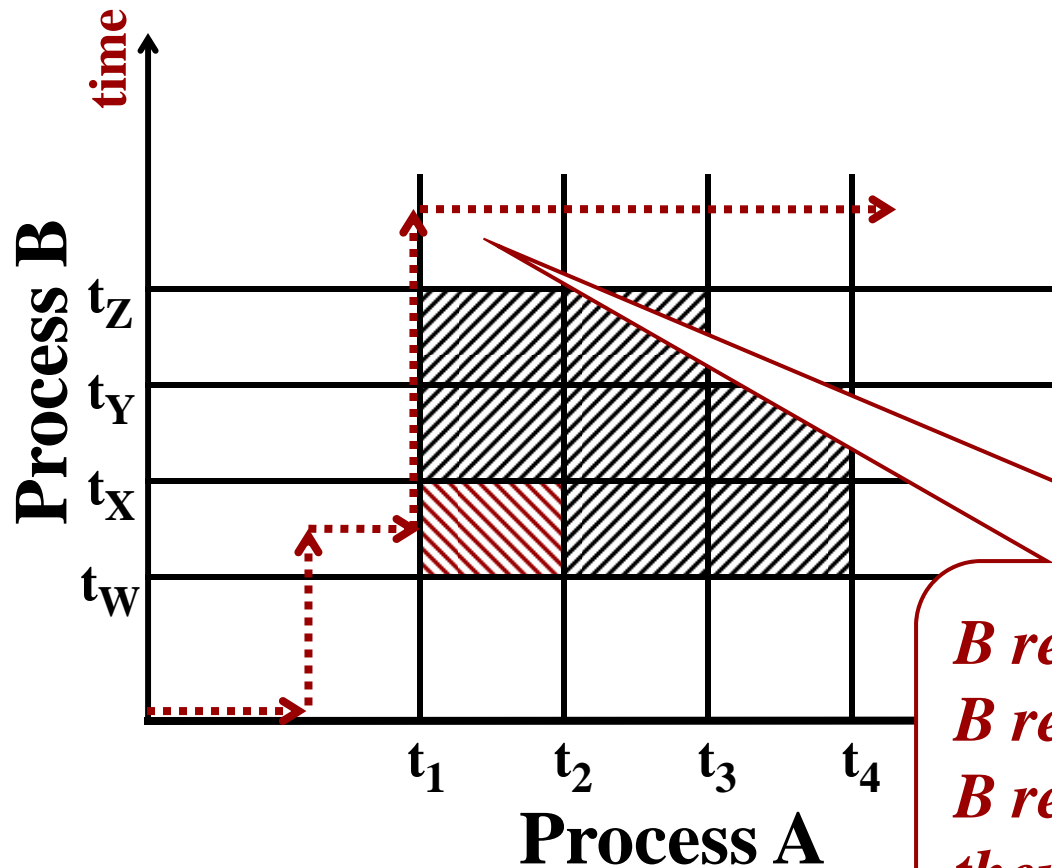


This area is “unsafe”

Avoidance using process-resource trajectories



Avoidance using process-resource trajectories



*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
- ❑ **The Banker's Algorithm:**
 - ❖ *Goal: Avoid unsafe states!!!*
 - ❖ *When a process requests more units, should the system grant the request or make it wait?*

Avoidance with multiple resources

Total resource vector

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

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
($A_1, A_2, A_3, \dots, A_m$)

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 for multiple resources

- ❑ 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 with multiple resources

Total resource vector

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

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
($A_1, A_2, A_3, \dots, A_m$)

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 with multiple resources

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

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

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 with multiple resources

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

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

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 with multiple resources

Tape drives
Plotters
Scanners
CD Roms

$E = (4 \quad 2 \quad 3 \quad 1)$

Tape drives
Plotters
Scanners
CD Roms

$A = (2 \quad 1 \quad 0 \quad 0)$

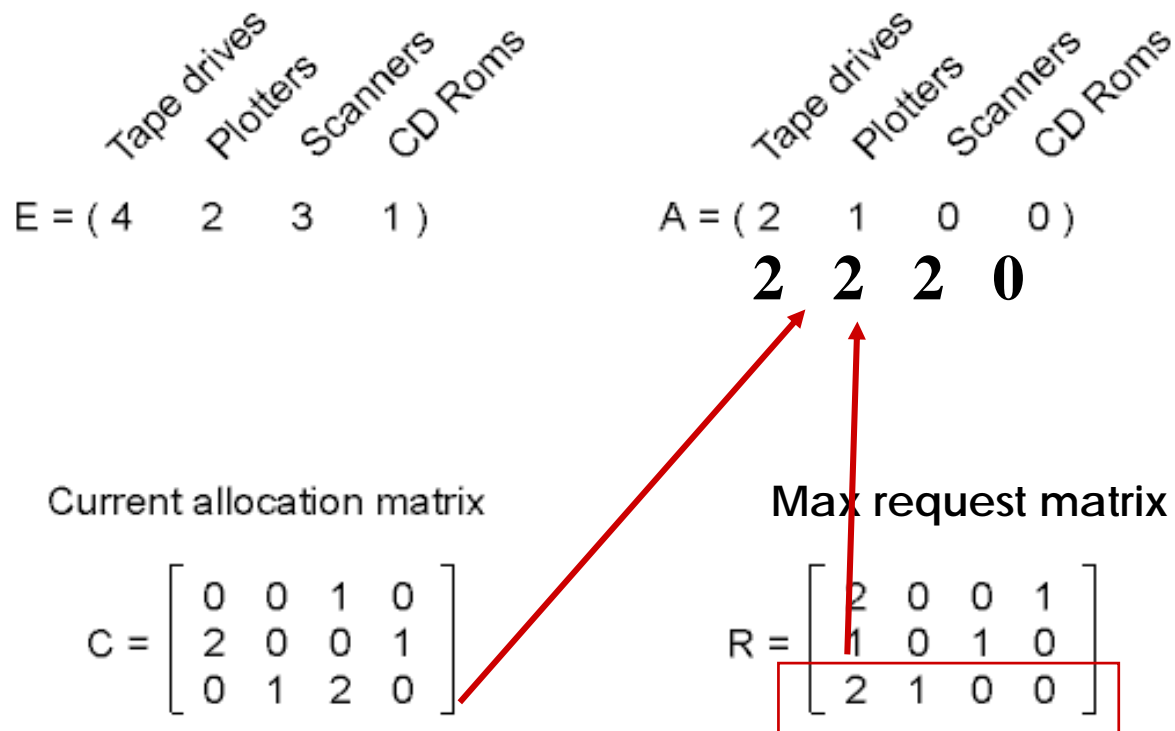
Current allocation matrix

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

Max request matrix

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

Avoidance with multiple resources



Avoidance with multiple resources

$$E = \begin{pmatrix} & \begin{matrix} \text{Tape drives} \\ \text{Plotters} \\ \text{Scanners} \\ \text{CD Roms} \end{matrix} \\ \begin{matrix} 4 & 2 & 3 & 1 \end{matrix} & \end{pmatrix}$$

$$A = \begin{pmatrix} & \begin{matrix} \text{Tape drives} \\ \text{Plotters} \\ \text{Scanners} \\ \text{CD Roms} \end{matrix} \\ \begin{matrix} 2 & 1 & 0 & 0 \end{matrix} \\ \begin{matrix} 2 & 2 & 2 & 0 \end{matrix} & \end{pmatrix}$$

Current allocation matrix

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

Max request matrix

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

Avoidance with multiple resources

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Tape drives Plotters Scanners CD Roms

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 \\ 4 & 2 & 2 & 1 \end{pmatrix}$$

Tape drives Plotters Scanners CD Roms

Current allocation matrix

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

Max request matrix

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

Problems with deadlock avoidance

- ❑ **Deadlock avoidance is often impossible**
 - ❖ because you don't know in advance what resources a process will need!
- ❑ **Alternative approach “deadlock prevention”**
 - ❖ Make deadlock *impossible*!
 - ❖ Attack one of the four conditions that are necessary for deadlock to be possible

Deadlock prevention

- **Conditions necessary for deadlock:**

Mutual exclusion condition

Hold and wait condition

No preemption condition

Circular wait condition

Deadlock prevention

- ❑ **Attacking mutual exclusion?**
 - ❖ a bad idea for some resource types
 - resource could be corrupted
 - ❖ works for some kinds of resources in certain situations
 - eg., when a resource can be partitioned
- ❑ **Attacking no preemption?**
 - ❖ a bad idea for some resource types
 - resource may be left in an inconsistent state
 - ❖ may work in some situations
 - checkpointing and rollback of idempotent operations

Deadlock prevention

- ❑ **Attacking hold and wait?**
 - ❖ Require processes to request all resources before they begin!
 - ❖ Process must know ahead of time
 - ❖ Process must tell system its "max potential needs"
 - eg., like in the bankers algorithm
 - When problems occur a process must release all its resources and start again

Attacking the conditions

- **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"*

Recall this example of deadlock

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...

Recall this example of deadlock

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!**

Why Does Resource Ordering Work?

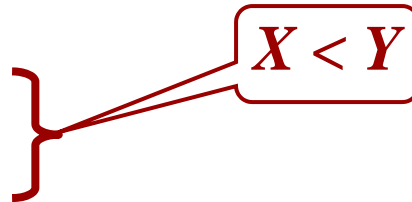
- ❑ Assume deadlock has occurred.
- ❑ **Process A**
 - ❖ holds X
 - ❖ requests Y
- ❑ **Process B**
 - ❖ holds Y
 - ❖ requests Z
- ❑ **Process C**
 - ❖ holds Z
 - ❖ requests X

Why Does Resource Ordering Work?

- Assume deadlock has occurred.

- **Process A**

- ❖ holds X
- ❖ requests Y



- **Process B**

- ❖ holds Y
- ❖ requests Z

- **Process C**

- ❖ holds Z
- ❖ requests X

Why Does Resource Ordering Work?

- 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

Why Does Resource Ordering Work?

- 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

$Z < X$

Why Does Resource Ordering Work?

- Assume deadlock has occurred.

- Process A**

- ❖ holds X
- ❖ requests Y

$X < Y$

This is impossible!

- Process B**

- ❖ holds Y
- ❖ requests Z

$Y < Z$

- Process C**

- ❖ holds Z
- ❖ requests X

$Z < X$

Why Does Resource Ordering Work?

- Assume deadlock has occurred.

- Process A**

- ❖ holds X
- ❖ requests Y

$X < Y$

This is impossible!

- Process B**

- ❖ holds Y
- ❖ requests Z

$Y < Z$

Therefore the assumption must be false!

- Process C**

- ❖ holds Z
- ❖ requests X

$Z < X$

Resource Ordering

- The chief problem:
 - ❖ *It may be hard to come up with an acceptable ordering of resources!*
- Still, this is the most useful approach in an OS
 1. ProcessControlBlock
 2. FileControlBlock
 3. Page Frames
- Also, the problem of resources with multiple units is not addressed.

A word on starvation

- **Starvation and deadlock are two different things**
 - ❖ 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

Quiz

- ❑ What is deadlock?
- ❑ What conditions must hold for deadlock to be possible?
- ❑ What are the main approaches for dealing with deadlock?
- ❑ Why does resource ordering help?