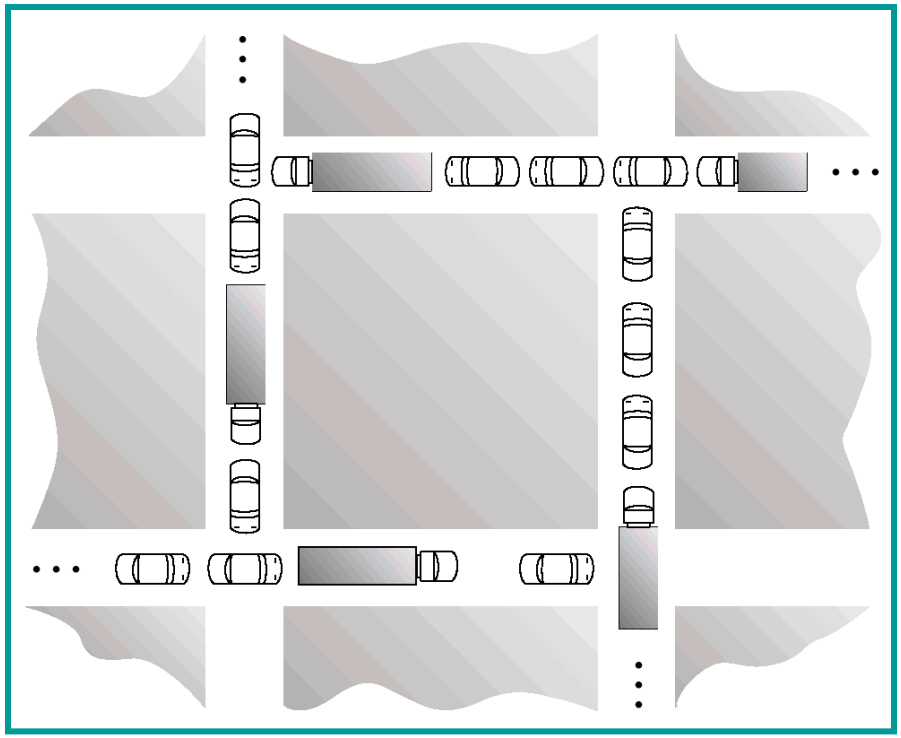
Module 6 - Deadlock

Chapter 7 (Silberchatz)



Deadlocks: important concepts

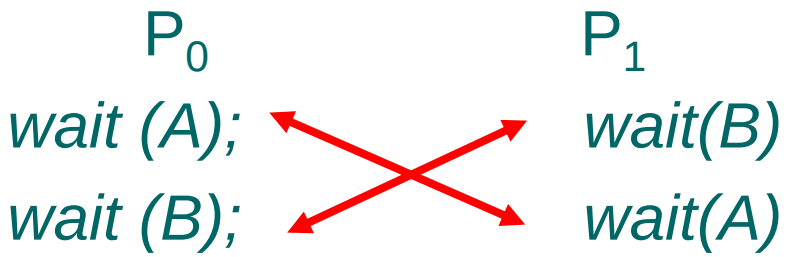
- **The deadlock problem**
- **Characterization: the 4 conditions**
  - Resource allocation graphs
  - Safe sequences
- **Methods of dealing with deadlocks**
  - Prevent deadlocks
  - Avoid deadlocks
    - Safe and unsafe states
  - Detect deadlocks
  - Recover from deadlock

Example 1

- **Two processes coexist in a system, which has 2 telephone lines. only**
- **Proc 1 needs**
  - a phone line to get started
  - the previous line, and an additional one, to finish
- **Proc 2 is the same**
- **Deadlock scenario:**
  - proc 1 request 1 line
  - proc 2 requests 1 line: both lines are engaged
  - deadlock! no proc can complete
    - unless one of the proc can be suspended
    - or can go back
- **Observe that deadlock is not inevitable, e.g. if 1 completes before the start of 2**
- **When does it become inevitable?**

Example 2

- **Semaphores**



- **Deadlock Scenario:**
  - initialization of A and B to 1
  - P0 executes wait (A), A = 0
  - P1 executes wait (B), B = 0
  - P0 and P1 cannot go further
    - What happens instead if P0 fully executes before P1?

Definition (Tanenbaum)

- **A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.**

- **A resource can be a signal, a message, a semaphore, etc.**
  - Interesting example: deadlock between readers or writers on a database ???

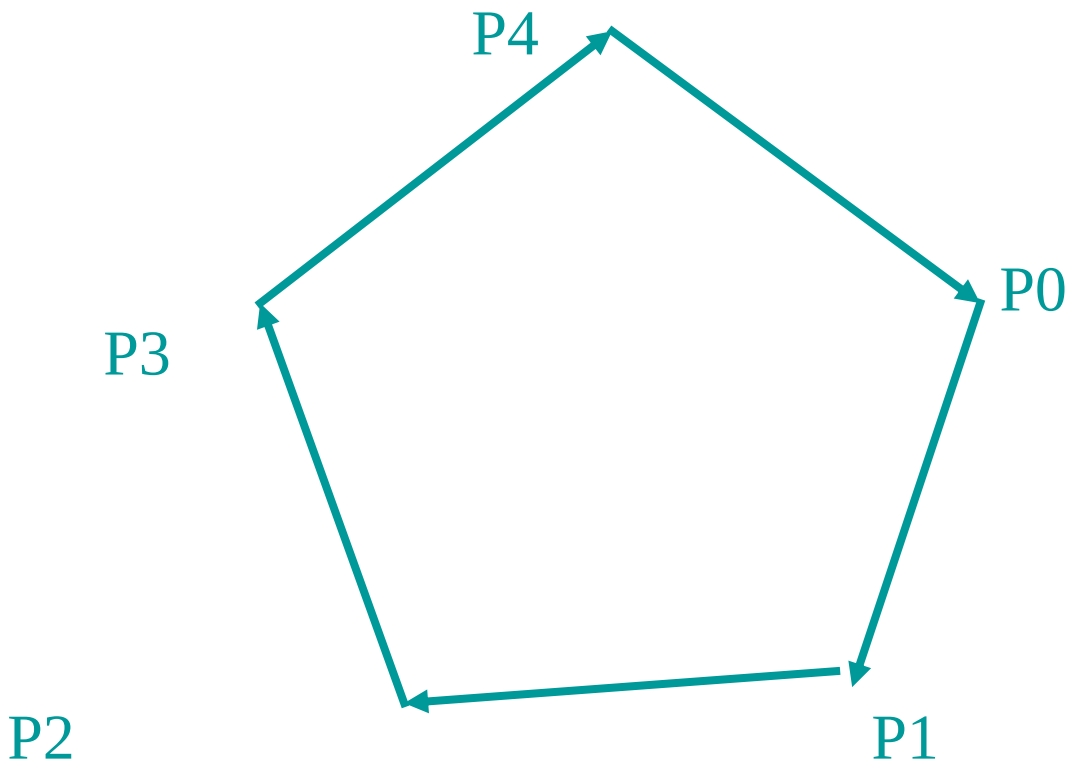
System Model

- **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.**
  - 2 printers, three hard drives, etc.
- **Each process utilizes a resource as follows (using system calls):**
  - request
  - use
  - release
- **Deadlock example**
  - Three processes each hold a CD drive.
  - Each process requires a 2nd drive.
- **Multithreaded programs are good candidates for deadlock**
  - Threads share many resources.

Deadlock characterization

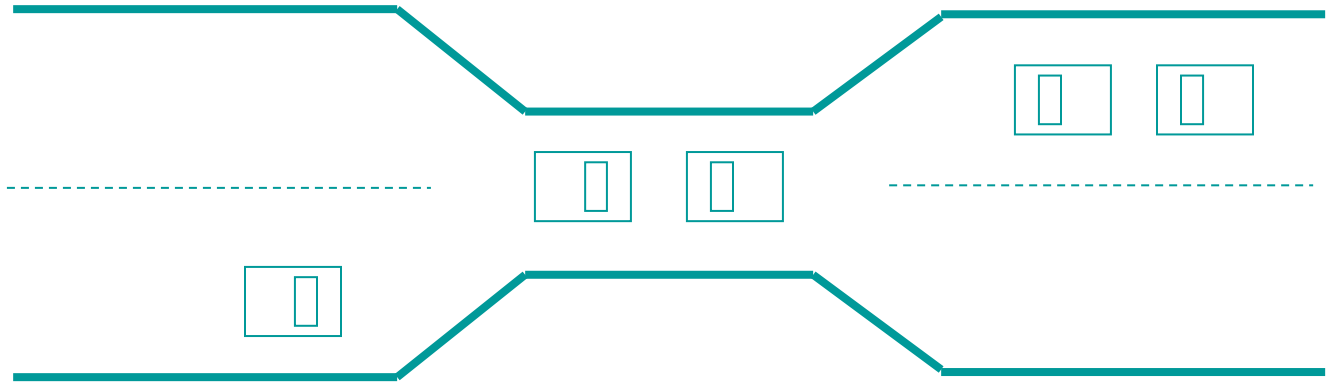
- **Deadlock requires the simultaneous presence of 4 conditions (necessary conditions)**
  - **Mutual exclusion:** the system has non-shareable resources (only 1 proc at a time can use it)
    - E.g .: CPU, memory area, peripheral, but also semaphores, monitors, critical sections
  - **Hold and wait**: a process holding a non shareable resource and is waiting to acquire additional resources held by other processes.
  - **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  - **Circular wait:** there is a process cycle such that each process to complete must use an unshareable resource which is used by the next, and which the next will keep until terminated
    - When the first 3 conditions exist, the circular wait is an indication of a deadlock.
    - The first 3 conditions do not imply necessarily a deadlock, since the circular wait may not occur.

Circular wait - none let go - no process can complete so deadlock



Finally, each process must hold a resource that the next will not let go → deadlock

Exercise



Think about this example where cars are in a deadlock situation on a bridge and see how the different conditions are met.

See also the example on page 1.

Exercise

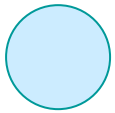
- Consider a system in which each process needs only **one** resource throughout its existence
- Deadlock, is it possible?

Resource-Allocation Graphs

- **A set of vertices V and a set of edges E**
- **V is partitioned into:**
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set which consists of all the procs in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set which consists of all types of resources in the system
- **Request edge - directed edge $P_i \rightarrow R_k$**
- **Assignment edge - directed edge $R_i \rightarrow P_k$**

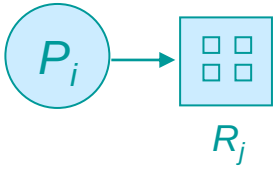
Resource-Allocation Graph (Cont.)

- **Process**

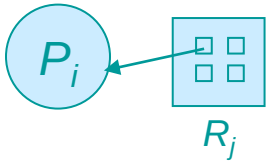


- **Resource Type with 4 instances**

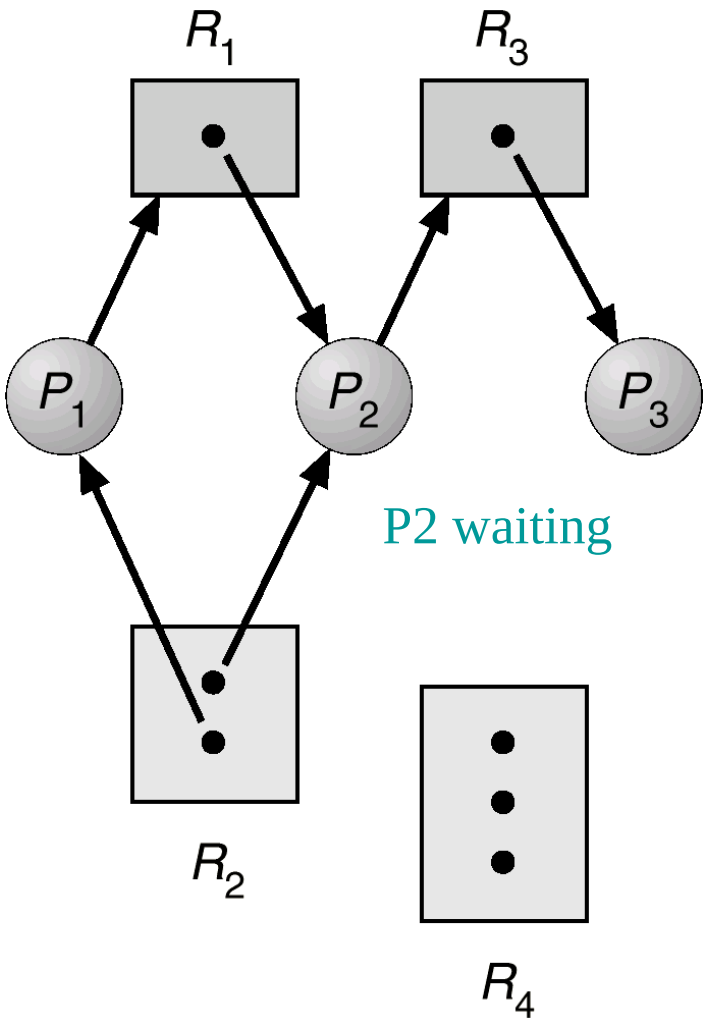
- ***P_i* requests instance of *R_j***



- ***P_i* is holding an instance of *R_j***



Example of a Resource Allocation Graph



P1 waiting

P2 waiting

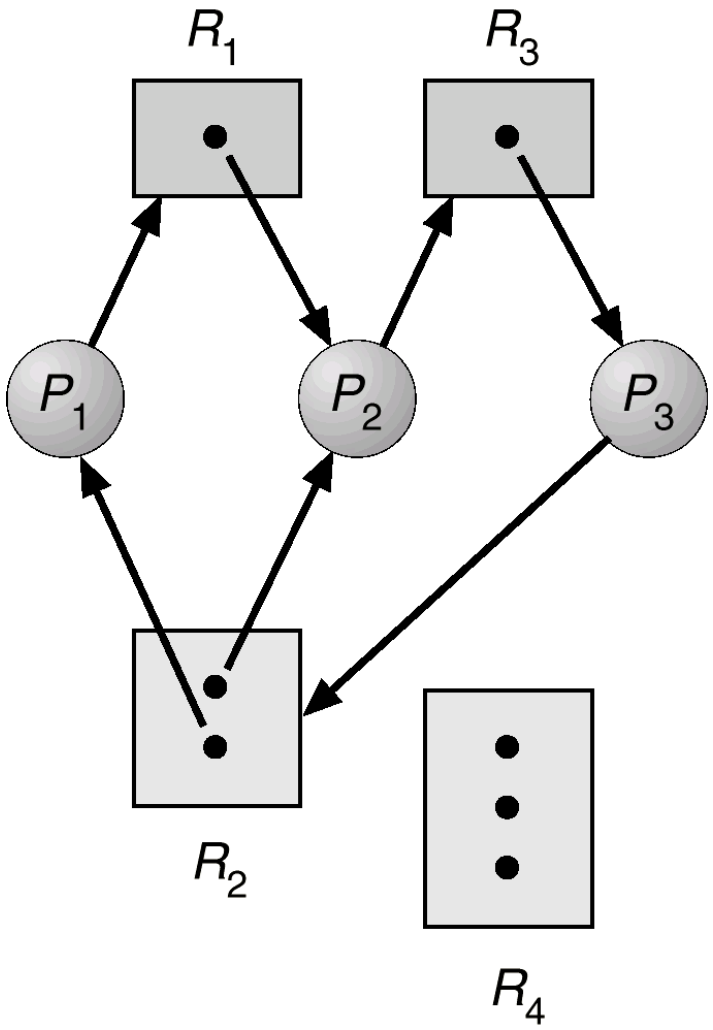
P3 not waiting

Is there a deadlock?

Using these graphs

- **We assume the existence of the first 3 conditions**
  - Mutual Excl., hold and wait, no preemption
- **To show that there is no deadlock, we must show that there is no cycle, because there is a process which can terminate without waiting for any other, and then the others can terminate immediately**
- **<P_3, P_2, P_1> is a process termination sequence: all can finish in this order**

Resource allocation graph with deadlock



Cycles:

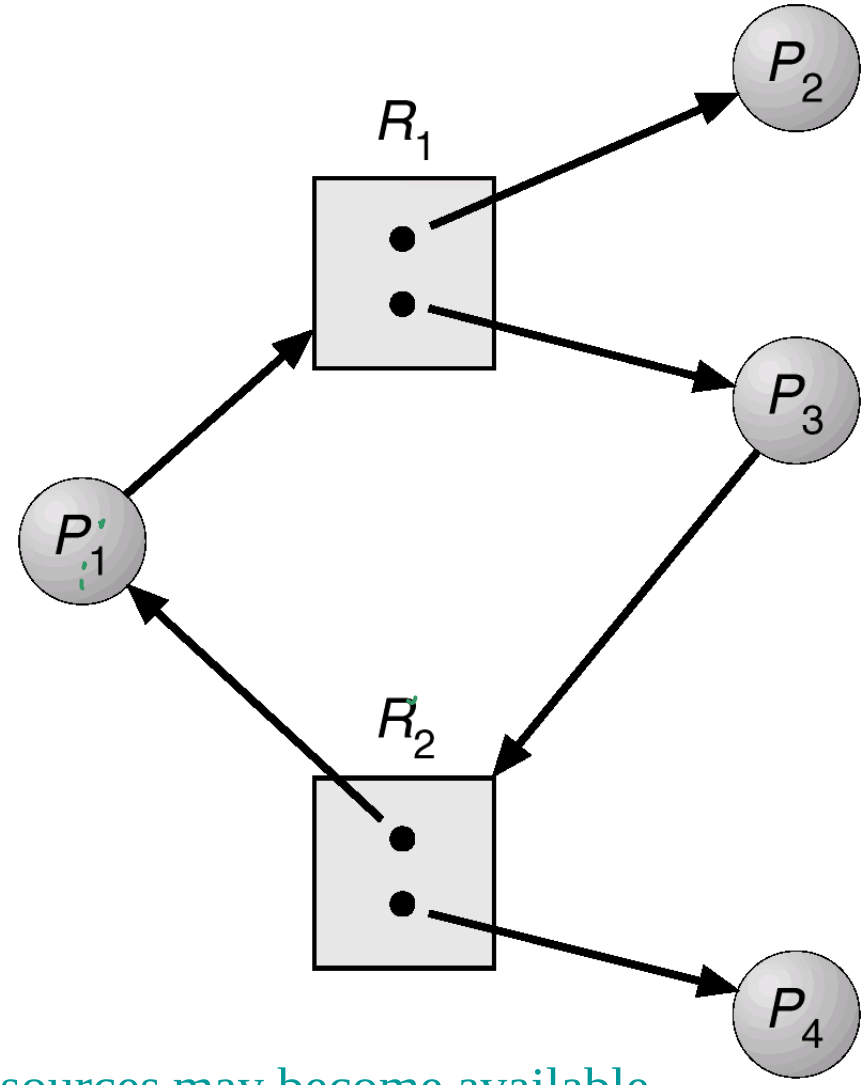
P1 → R1 → P2 → R3 → P3 → R2 → P1

P2 → R3 → P3 → R2 → P2

no proc can terminate

no possible way out

Resource allocation graph with cycle, but no deadlock (Why?)



Circular wait, but resources may become available

Basic Facts

- **The cycles in the resource allocation graph do not necessarily signal a circular wait**
- **If there are no cycles in the graph, no deadlock**
- **If there are cycles:**
  - If only one resource per type, deadlock
    - (Why?!)
  - If several resources per type, possible deadlock
    - The question must be asked: is there a process that can terminate and if so, what other processes can terminate as a result?

Termination hypothesis

- A proc which has all the resources it needs, it uses them for a finite time, then it frees them
- We say the process ends, but it could also continue, no matter what.

Difference between request and wait

- In reality, a process that requests a resource does not necessarily have to stop immediately while waiting for it to be given to it ...
- However if the resource is not given to it, it will have to stop at some point.
- So in the deadlock analysis we make the assumption that a process stops when it requests a resource, if this request is not satisfied immediately

Methods for handling deadlock

- **Prevent: design the system in such a way that deadlock is not possible**
  - difficult, very restrictive
  - suitable for critical systems
- **Avoid: deadlocks are possible, but are avoided (avoidance)**
- **Detect and recover: Allow deadlocks, recover after**
- **Ignore the problem and pretend that deadlocks never occur in the system.**
  - Used by most operating systems, including UNIX/Windows

Deadlock Prevention: prevent at least one of the four conditions that lead to deadlock

- Mutual exclusion: **reduce as much as possible the use of shared resources and critical sections (almost impossible).**
- Hold and wait**: a process that requests new resources cannot hold up other processes (ask for all resources at once).**
- No pre-emption: **If a process asks for resources and cannot obtain them, it is suspended and its resources already held are released.**
- Circular wait: **define an request ordering for resources, a process must ask for resources in this order (e.g. always ask for the printer before the tape drive) (see section 7.4.4 in the textbook for more details).**

Deadlock Avoidance

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

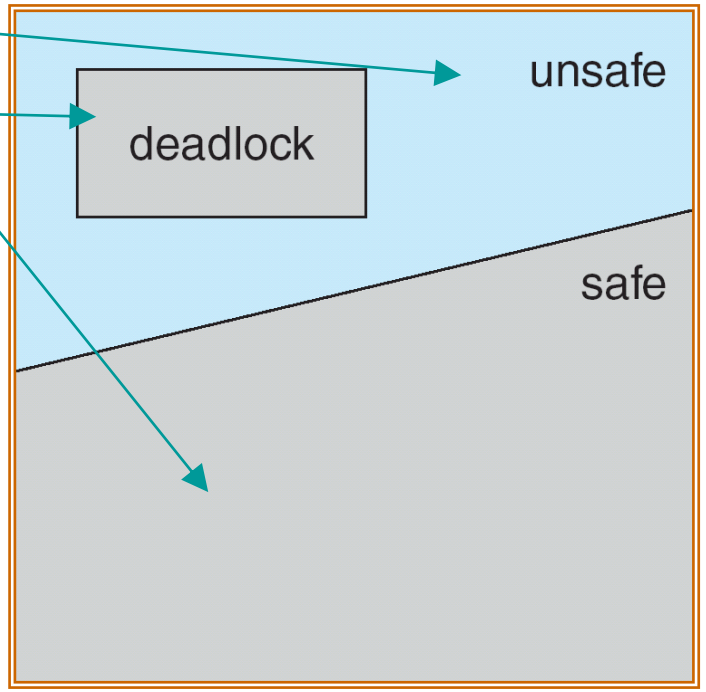
- **Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.**
- **The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.**
- **Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.**

Safe State

- **When a process requests an available resource, the system must decide if immediate allocation leaves the system in a *safe* state.**
- **What is a *safe* state?!**
- **System is in *safe* state if there exists a *safe sequence* of all processes.**

Basic Facts

- If a system is in safe state ⇒ no deadlock possible (i.e. the system can exit without deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state (i.e Do not allocate a resource to a process if the resulting state is not safe
- .



Safe Sequence

- **Safe sequence is a sequence witnessing that we can run all processes to completion**
- **P_1 is the process that can be run to completion using only the available resources**
- **P_2 is the process that can be completed when P1 completes and releases its resources…**
- **Sequence <P_1, P_2, …, P_n> is safe if for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with j<i.**
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Safe Sequence

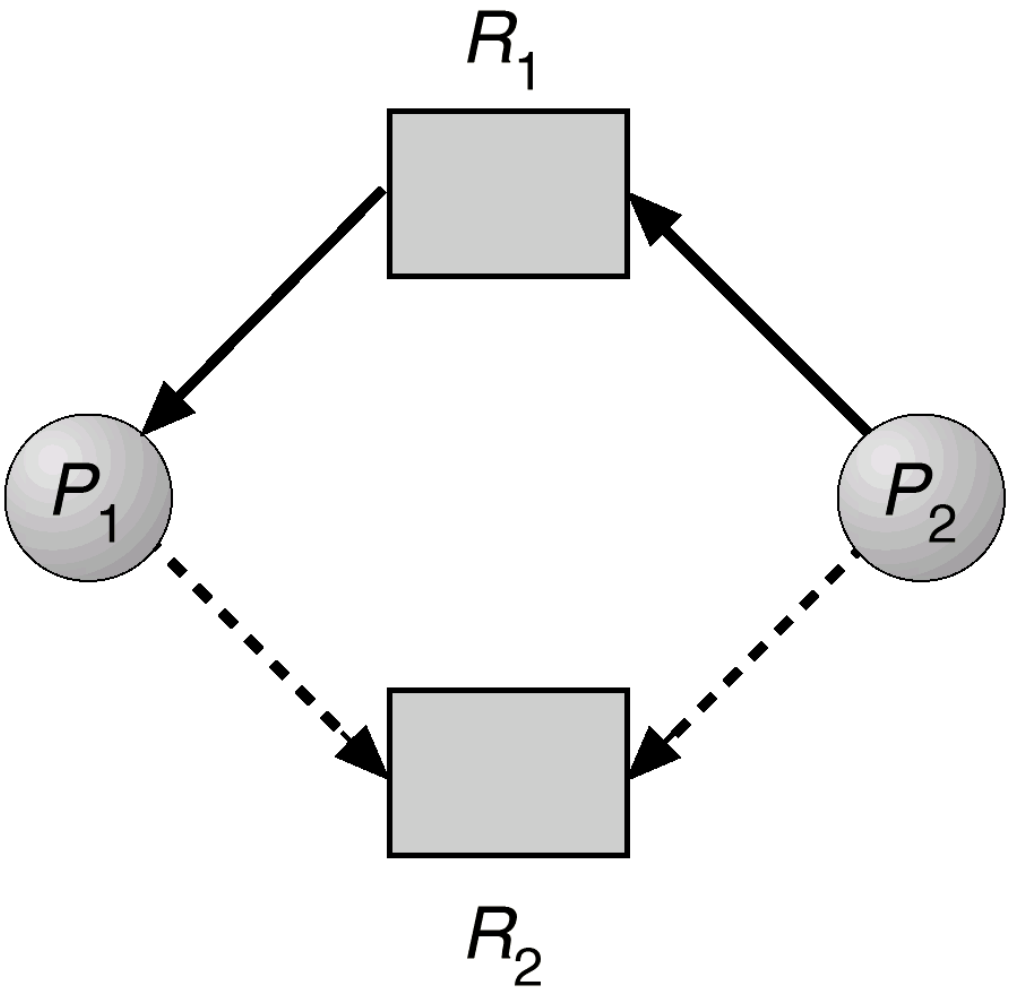
- **A sequence of proc <P_1, P_2,…, P_n> is safe if for each Pi, the resources that P_i can still request can be met by currently available resources + resources used by *the* P_j *that precede them (i.e* P_j, with j<i).**
  - When P_i succeeds, P_i + 1 can get the resources it needs, and terminate, so:
- **<P_1, P_2,…, P_n> is a process termination sequence or a safe sequence: all can end in this order**

Resource-Allocation Graph Algorithm

First consider the simpler case of one-instance resources

- **Note: in such case, a cycle in the Resource-Allocation graph implies deadlock**
- **Lets introduce *claim edges*:**
  - *Claim edge* $P_i \rightarrow R_j$ indicates that process P_j may request resource R_j; represented by a dashed line.
  - Claim edge is converted to request edge when a process requests a resource.
  - When a resource is released by a process, assignment edge reconverts to a claim edge.
- **Resources must be claimed *a priori* in the system.**
- **The algorithm: if satisfying request creates a cycle in the modified R-A graph (including claim edges), reject the request**

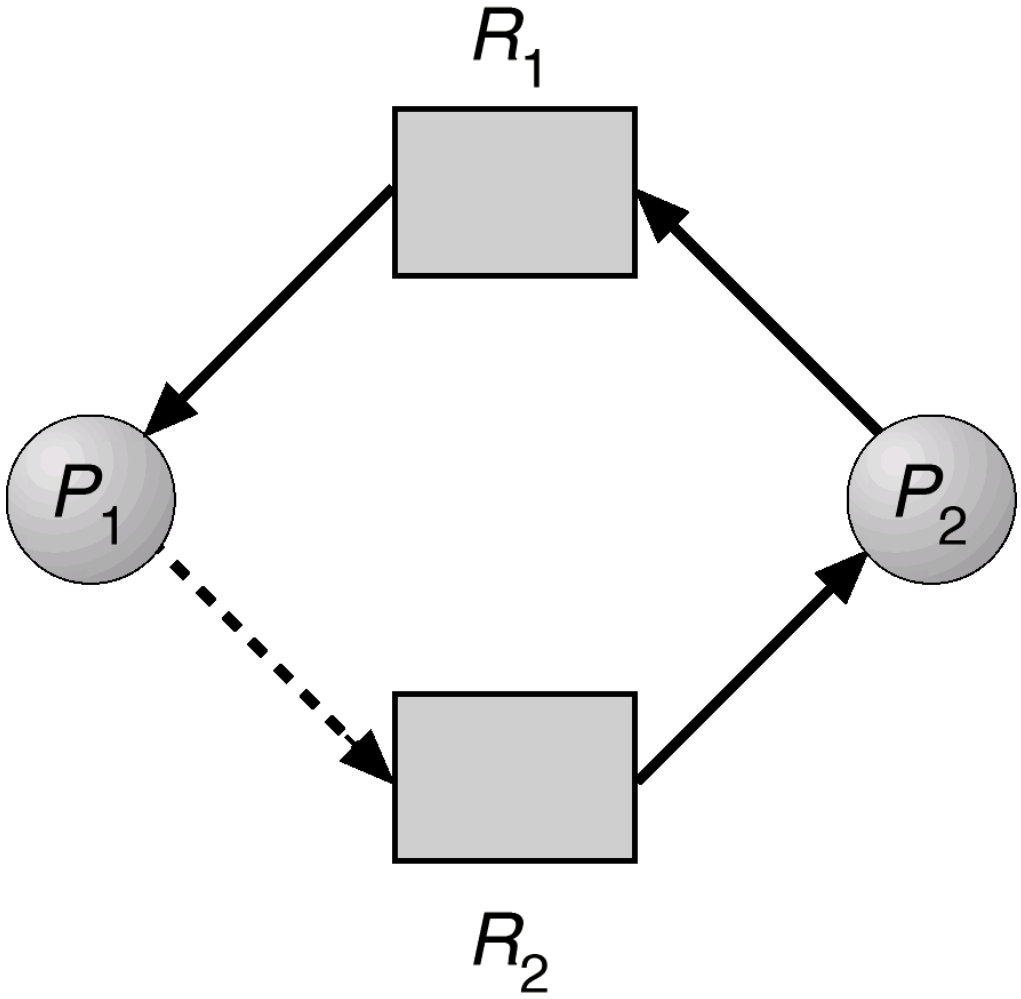
Resource allocation graph with a claim edge



Continuous line: current request;

dashes: possible request in the future

Unsafe State In Resource-Allocation Graph



If P2 requests R2, the latter cannot be given to it, because this can cause a cycle in the graph: P1 req R2,

Refusal to allocate a resource: the banker's algorithm

- Processes are like customers who want to borrow money (resources) from the bank ...
- A banker shouldn't lend money if he can't meet the needs of all his clients
- At all times the state of the system is defined by the values of R(i), C(j, i) for any type i and process j, and by other values of vectors and matrices.

The banker's algorithm

- We also need to know the quantity allocated A(j, i) from resources of type i to process j
- The total amount of type i resource available is given by: $V(i) = R(i) - \Sigma_k A(k, i)$
- N (j, i) is the quantity of resources i needed by process j to complete its task: N (j, i) = C (j, i) - A (j, i)
- To decide whether the request should be granted, the banker's algorithm tests whether this allocation will lead the system into a safe state:
  - grant the request if the state is safe
  - otherwise refuse the request

The banker's algorithm

- **A state is safe iff there is a sequence {P1..Pn} where each Pi is allocated all the resources it needs to complete**
  - ie: we can still run all processes until terminated from a safe state

The banker's algorithm

- **Q (j, i) is the resource quantity i requested by process j during a request.**
- **To determine whether this request should be granted, we use the banker's algorithm:**
  - If Q (j, i) <= N (j, i) for all i then continue. Otherwise report the error (amount claimed exceeded).
  - If Q (j, i) <= V (i) for all i then continue. Otherwise wait (resource not yet available)
  - Pretend that the request is granted and determine the new state:

The banker's algorithm

- V (i) = V (i) - Q (j, i) for all i
- A (j, i) = A (j, i) + Q (j, i) for all i
- N (j, i) = N (j, i) - Q (j, i) for all i

- If the resulting state is conservative, then grant the request. Otherwise the process j must wait for its request Q (j, i); restore the previous state.

- **The safety algorithm is the part that determines if a state is safe**
- **Initialization:**
  - all processes are “unfinished”
  - the work vector first contains the available resources:
  - W (i) = V (i); for all i;

The banker's algorithm

- **REPEAT: Choose an “unfinished” process j such that N (j, i) <= W (i) for all i.**
  - If such j does not exist, goto EXIT
  - Otherwise: “terminate” this process and recover all its resources: W (i) = W (i) + A (j, i) for all i. Then goto REPEAT
- **EXIT: If all processes have “terminated” then this state is safe. Otherwise it is unsafe.**

Banker's algorithm: example

- **We have 3 types of resources with the following quantities:**
  - R(1) = 9, R(2) = 3, R(3) = 6
- **and 4 processes with the initial state:**

	Claimed			Allocated			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	1	1	2
P2	6	1	3	5	1	1			
P3	3	1	4	2	1	1			
P4	4	2	2	0	0	2			

- **Suppose the request of P2 is Q = (1,0,1). Should it be granted?**

Banker's algorithm: example

- resulting state would be:

	Claimed			Allocated			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	0	1	1
P2	6	1	3	6	1	2			
P3	3	1	4	2	1	1			
P4	4	2	2	0	0	2			

- This state is safe with the sequence {P2, P1, P3, P4}. After P2 we have W = (6,2,3) which allows all other processes to terminate. The request is therefore granted

Banker's algorithm: example

- However if, from the initial state, P1 requires Q = (1,0,1). The resulting state would be:

	Claimed			Allocated			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	2	0	1	0	1	1
P2	6	1	3	5	1	1			
P3	3	1	4	2	1	1			
P4	4	2	2	0	0	2			

- This state is not safe because, to complete, each process requires a unit of R1. The request is refused: P1 is blocked.

Banker's algorithm: comments

- **A safe state is without a deadlock. But a unsafe state does not necessarily contain a deadlock.**
  - Ex: P1 from the previous unsafe state could temporarily release a unit from R1 and R3 and thus return to a safe state
- **so it is possible that some processes have to wait without it being necessary**
  - sub-optimal use of resources
- **All the algorithms used to avoid deadlock assume that each process is independent: without synchronization constraint**

Deadlock detection

- The system is allowed to enter a deadlock state
- Deadlock is detected
- We recover from deadlock

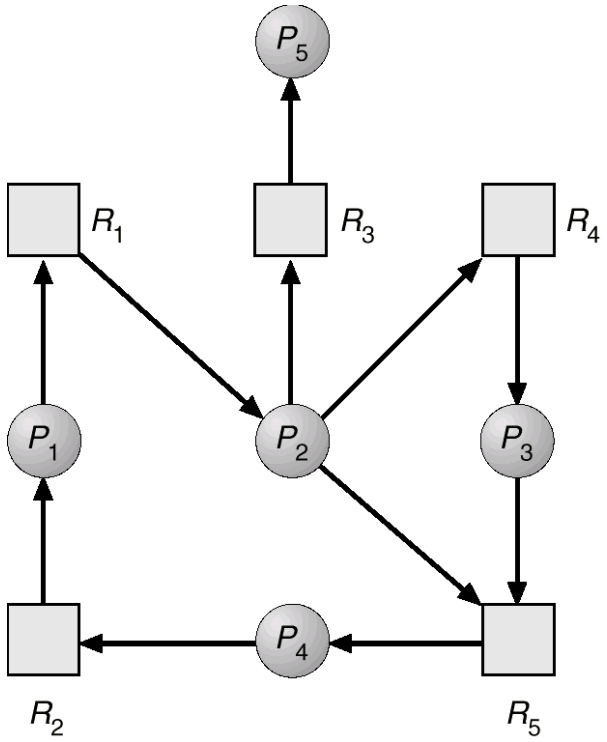
Difference between wait and deadlock

- **It is difficult to detect if there is indeed a deadlock in the system**
- **We could see that a number of processes are waiting for resources**
  - this is normal!
- **To know that there is a deadlock, you must know that no process in a group has a chance to receive the resource**
  - because there is a circular wait!
- **This implies an additional analysis, which few OS bother to do ...**

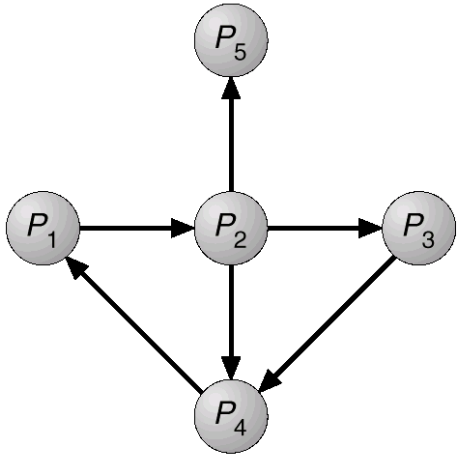
Deadlock detection method for one resource by type

- **Essentially, the method already described**
  - Build a resource allocation graph and see if there is a way all procs can terminate
- **In the case of one resource by type, the algorithm looks for cycles in the graph (order algorithm n^2, if n = number of vertices)**
- **More difficult in the case of several resources by type**

Resource allocation graph and waiting for graph
(case of 1 resource per type)



(a)



(b)

A deadlock detection algorithm

- **Use the previous matrices and vectors for resource allocation**
- **Mark each process not involved in a deadlock. Initially all processes are unmarked (possibly involved). Then perform:**
  - Mark each process j for which: A (j, i) = 0 for all i. (since they cannot be involved)
  - Initialize the working vector: W (i) = V (i) for all i
  - REPEAT: Choose an unmarked process j such that

    Q (j, i) <= W (i) for all i. Stop if such j does not exist.
  - If such a j exists: mark the process j and do

    W (i) = W (i) + A (j, i) for all i. Goto REPEAT
  - At the end: every unmarked process is involved in a dealock

Deadlock detection: comments

- Process j is not involved in a deadlock when

  Q (j, i) <= W (i) for all i.
- We are then optimistic and assume that process j will not require more resources to complete its task
- It will then free all its resources. Then:

  W (i) = W (i) + A (j, i) for all i
- If this assumption is incorrect, deadlock could arise later.
- This deadlock will be detected the next time the detection algorithm is invoked.

Deadlock detection: example

	Request					Allocated					Available				
	R1	R2	R3	R4	R5	R1	R2	R3	R4	R5	R1	R2	R3	R4	R5
P1	0	1	0	0	1	1	0	1	1	0	0	0	0	0	1
P2	0	0	1	0	1	1	1	0	0	0					
P3	0	0	0	0	1	0	0	0	1	0					
P4	1	0	1	0	1	0	0	0	0	0					

- Mark P4 because it has no allocated resources
- Make W = (0,0,0,0,1)
- The request of P3 <= W. Then mark P3 and do W = W + (0,0,0,1,0) = (0,0,0,1,1)
- The algorithm ends. P1 and P2 are involved in a deadlock

Recover deadlocks

- **Terminate all processes in deadlock**
- **Terminate one process at a time, hoping to eliminate the deadlock cycle**
- **In what order: different criteria:**
  - priority
  - need for resources: past, future
  - how long has i trun, how long does it still need
  - etc.

Recovery: resource preemption

- **Minimize the cost of selecting the victim**
- **Rollback: return to a safe state**
  - need to regularly establish and keep 'checkpoints' kinds of snapshots of the current state of the process
- **Starvation possible if a process is always selected**

Combination of approaches

- **Combine the different approaches, if possible, taking into account practical constraints**
  - prevent
  - avoid
  - detect
- **use the most appropriate techniques for each resource class**

Importance of deadlock pb

- **Deadlock is almost ignored in the design of today's systems**
  - With the exception of *critical* systems
- **If this is true, the user will see a system failure or a process failure**
- **In the high concurrent systems of the future, it will become more and more important to prevent it and avoid it.**

Deadlocks: important concepts

- Characterization: the 4 conditions
- Resource allocation graphs
- Safe sequences
- Safe and unsafe states
- Prevent deadlocks
- Avoid deadlocks
- Detect deadlocks
- Recover from deadlock


Thank You!
Ευχαριστώ
ขอบคุณ
Vielen Dank
Merci
Teşekkürler
Dmnvwd
شكراً
متشكرم
Gracias
Grazie
Bedankt
Dankie
THANK YOU
Köszönettel
Obrigado!
شکریا
Díky
謝謝
GADDA GUEY
WAD MAHAD
SAN TAHAY
감사합니다
Urakoze