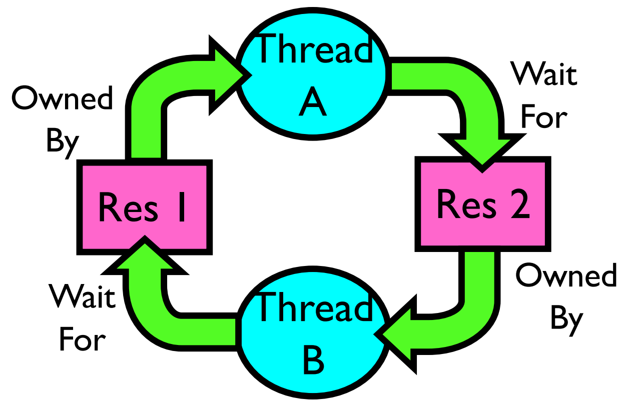
Lecture 6
Deadlock

Prof. Yinqian Zhang

Fall 2025

Starvation vs. Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - Low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - Thread A owns Res 1 and is waiting for Res 2 Thread B owns Res 2 and is waiting for Res 1
  - Deadlock ⇒ Starvation but not vice versa
    - Starvation can end (but does not have to)
    - Deadlock cannot end without external intervention



Conditions for Deadlock

- Deadlock will not always happen
    - Need the exactly right timing
    - Bugs may not exhibit during testing
- Deadlocks occur with multiple resources
    - Cannot solve deadlock for each resource independently
    - System with 2 disk drives and two threads
    - Each thread needs 2 disk drives to function
    - Each thread gets one disk and waits for another one

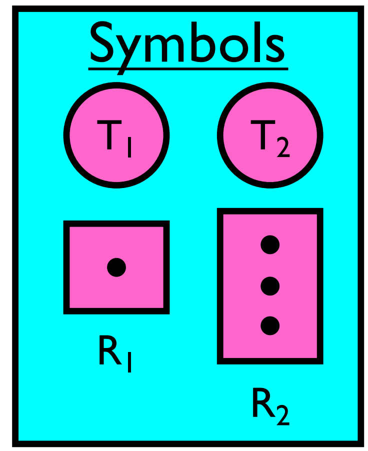
Process A	**Process B**
sem_wait(x)	sem_wait(y)
sem_wait(y)	sem_wait(x)
sem_post(y)	sem_post(x)
sem_post(x)	sem_post(y)

Four Requirements for Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption 非抢占
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - T_1 is waiting for a resource that is held by T_2
    - T_2 is waiting for a resource that is held by T_3
    - …
    - T_n is waiting for a resource that is held by T_1

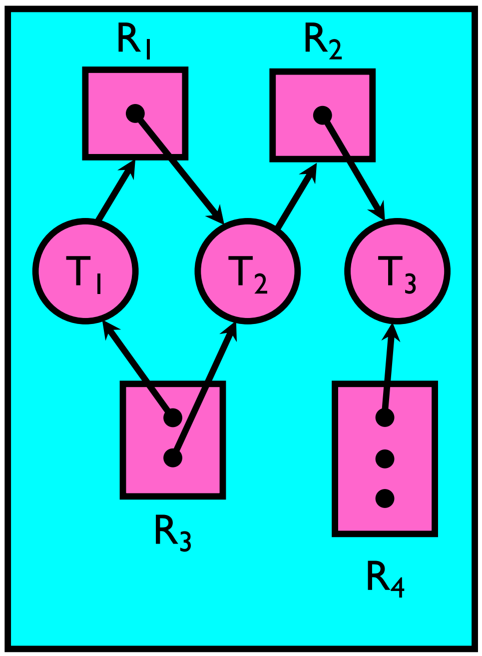
Resource-Allocation Graph

- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - 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 thread utilizes a resource as follows:
    - Request() / Use() / Release()
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$

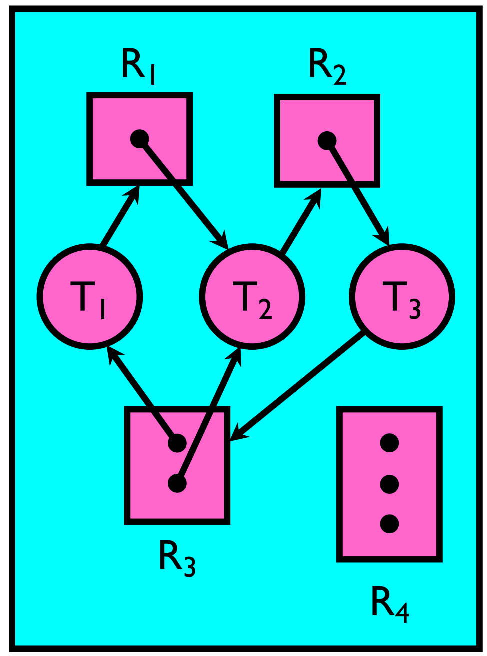


Resource Allocation Graph Examples

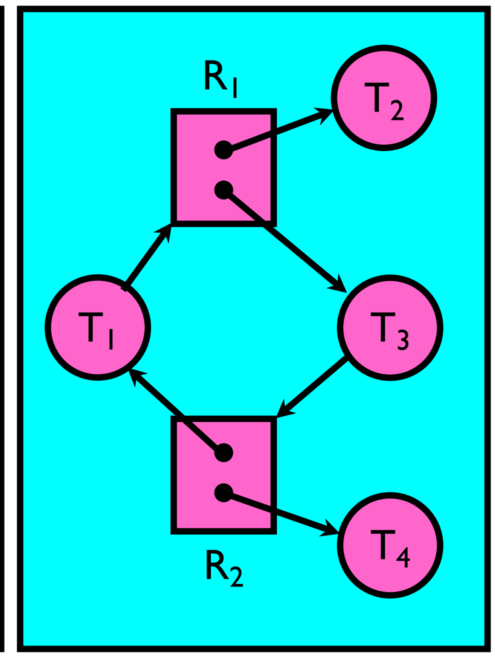
- Recall:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



Allocation Graph With Deadlock



Allocation Graph with Cycle, but No Deadlock

Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - **Deadlock detection**
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks

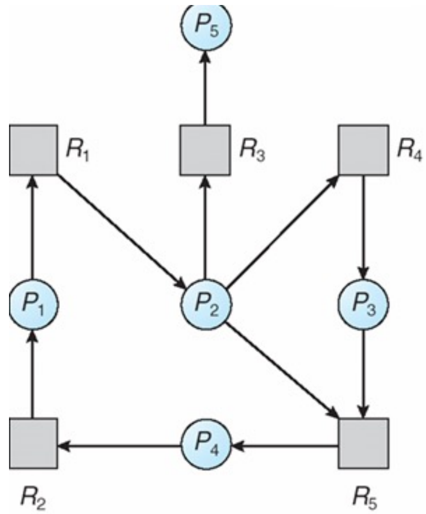
- Ensure that system will never enter a deadlock
  - **Deadlock prevention**
  - Need to monitor all resource acquisitions
  - Selectively deny those that might lead to deadlock

- Ignore the problem and pretend that deadlocks never occur in the system 忽略死锁问题，假装死锁在系统中永远不会发生（Deadlock Ignorance）
  - Used by most operating systems, including UNIX

Deadlock Detection with Resource Allocation Graphs

- Only one of each type of resource $\Rightarrow$ look for cycles

- More than one resource of each type
  - More complex deadlock detection algorithm
  - Next page



Several Instances per 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 process.
- **Request**: An *n* x *m* matrix indicates the current request of each process. If Request $[i_j]$ = k, then process P_i is requesting k more instances of resource type R_j.

Detection Algorithm

可用资源数量 process i是否完成

1. Let **Work** and **Finish** be vectors of length m and n, respectively Initialize:
   (a) **Work** = **Available**
   (b) For i = 1,2, …, n, if $\textbf{Allocation}_i \neq 0$, then **Finish**[i] = false; otherwise, **Finish**[i] = true
2. Find an index i such that both:
   (a) **Finish**[i] == false
   (b) $\textbf{Request}_i \leq$ Work

   If no such i exists, go to step 4

process i已经结束，释放资源

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

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

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in Finish[i] = true for all i

Example (Cont.)

- P_2 requests an additional instance of type C

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0 (not deadlocked), but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

What if Deadlock Detected?

- Terminate process, force it to give up resources
  - Shoot a dining philosopher !?
  - But, not always possible
- Preempt resources without killing off process
  - Take away resources from process temporarily
  - Does not always fit with semantics of computation
- Roll back actions of deadlocked process
  - Common technique in databases (transactions)
  - Of course, deadlock may happen once again

当系统检测到死锁时，回退某些操作，使进程能够从先前的状态重新开始。

Deadlock Prevention

- Try to ensure at least one of the conditions cannot hold to prevent deadlock
  - Remove "Mutual Exclusion": not possible for non-sharable resources
  - Remove "Hold and Wait" – must guarantee that whenever a process requests a resource, it does not hold any other resources
    - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
    - Low resource utilization; starvation possible

解决方法是，确保每个进程在请求资源时，要么请求所有资源并一次性分配，或者只有在没有资源时才请求资源。

问题：这种做法可能会导致资源利用率低，且可能造成饥饿现象，即有些进程无法获得所需资源。

Deadlock Prevention

- Try to ensure at least one of the conditions cannot hold to prevent deadlock
  - Remove "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 process is waiting
    - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

抢占是指：如果一个进程已经持有一些资源，但它请求的其他资源无法立即分配给它，那么该进程必须释放已经持有的所有资源，直到它能够获取所有它需要的资源。

实现方式：
如果一个进程在等待某些资源（这些资源当前正在被其他进程占用），那么该进程会将自己持有的资源释放出去，放到等待队列中，直到能够重新获得它之前的资源以及它正在请求的新资源。

流程：
将被抢占的资源添加到等待列表中。
当进程能够重新获取到其旧资源以及新的请求资源时，才重新开始执行。

问题：
死锁预防：通过这种方式，能够避免因"持有和等待"条件造成死锁，但有时可能会导致进程频繁的资源释放与重新获取，从而降低效率。

Deadlock Prevention

- Try to ensure at least one of the conditions cannot hold to prevent deadlock
  - Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration 强制对所有资源类型进行全局排序，并要求每个进程按升序请求资源。通过这种方式，保证了进程请求资源的顺序，防止了形成循环等待
  - $R=\{R_1,R_2,\ldots,R_m\}$
  - One to one function F:R->N
  - If a process request a resource R_i, it can request another resource R_j if and only if $F(R_i) < F(R_j)$
  - Or, it must first release all resource R_i such that $F(R_i) \geq F(R_j)$

资源排序规则：

将所有资源按总排序排列为 R1, R2, ..., Rm，其中 R1, R2, ..., Rm 是资源类型的顺序。
每个进程的请求顺序必须遵循这个资源排序规则。具体来说，进程在请求资源时，若请求资源 Ri，它只能请求比 Ri 排序靠后的资源（即，如果进程请求资源 Ri，它只能请求 Rj，当且仅当 F(Ri) < F(Rj)，其中 F 是排序函数）。
或者，进程必须首先释放所有已持有的资源 Ri，以确保 F(Ri) ≥ F(Rj)，避免循环等待的出现。

因为进程永远不会请求比自己已持有的资源更前面的资源，从而避免了死锁的循环等待条件。

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**

    这个先验信息对于死锁避免至关重要，因为它允许系统根据资源的最大需求来进行决策，避免资源分配导致的死锁。
  - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

如果存在一个进程执行顺序 <P1, P2, ..., Pn>，使得每个进程 Pi 的资源请求都可以通过当前可用资源加上其他进程已持有的资源来满足，且所有进程都能最终完成，则系统处于安全状态。

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

当一个进程请求一个可用资源时，系统必须判断分配该资源是否会导致系统进入安全状态。
如果立即分配资源后，系统依然能够保证安全，即所有进程都可以顺利完成并释放资源，则该分配是安全的。

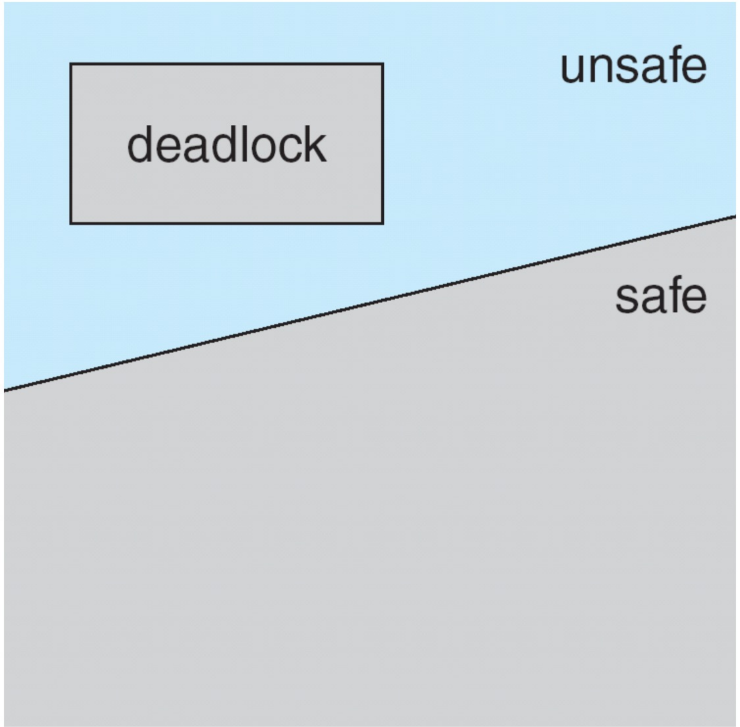
- System is in **safe state** if there exists a sequence $\langle P_1, P_2, \ldots, P_n \rangle$ of ALL the processes in the systems such that **for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$**
  - If what 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, Unsafe, Deadlock State

- If a system is in safe state => no circular wait => no deadlocks

- If a system is in unsafe state => possibility of deadlock

- Deadlock avoidance => ensure that a system will never enter an unsafe state.



Banker's Algorithm

- Multiple instances of each resource type
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Banker's Algorithm (Cont'd)

- Let n = number of processes, and m = number of resources types.
- **Available**: Vector of length m. If **available** [j] = k, there are k instances of resource type R_j available
- **Max**: n x m matrix. If **Max** [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If **Allocation**[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If **Need**[i,j] = k, then P_i may need k more instances of R_j to complete its task

Need [i,j] = **Max**[i,j] – **Allocation** [i,j]

Banker's Algorithm: 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 index i such that both:

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

   (b) $\mathbf{Need}_i \leq \mathbf{Work}$ (i.e., for all k, $\mathbf{Need}_i[k] \leq \mathbf{Work}[k]$)

   If no such i exists, go to step 4

3. $\mathbf{Work} = \mathbf{Work} + \mathbf{Allocation}_i$

   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

Request = request vector for process P_i. If **Request**$_i$ [j] = k then process P_i wants k instances of resource type R_j

1. If **Request**$_i \leq$ **Need**$_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If **Request**$_i \leq$ **Available**, go to step 3. Otherwise, P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

    Available = **Available** – **Request**;

    Allocation$_i$ = **Allocation**$_i$ + **Request**$_i$;

    Need$_i$ = **Need**$_i$ – **Request**$_i$;

- If safe $\Rightarrow$ the resources are allocated to P_i
- If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time T_0:

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

Example (Cont'd)

- The content of the matrix Need is defined to be **Max – Allocation**

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

- The system is in a safe state since the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that **Request** ≤ **Available,** that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

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

- Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Thank you!