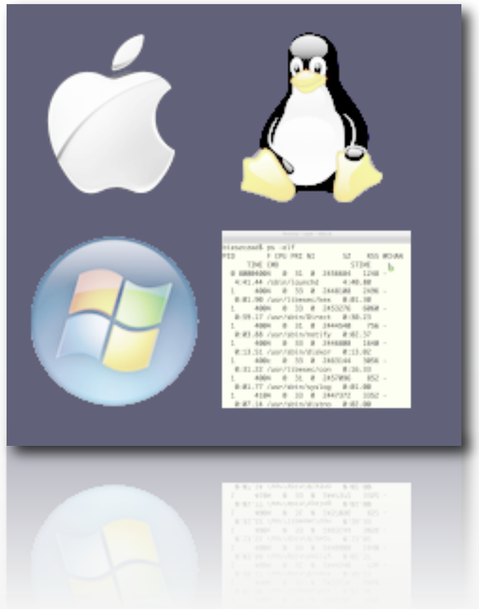


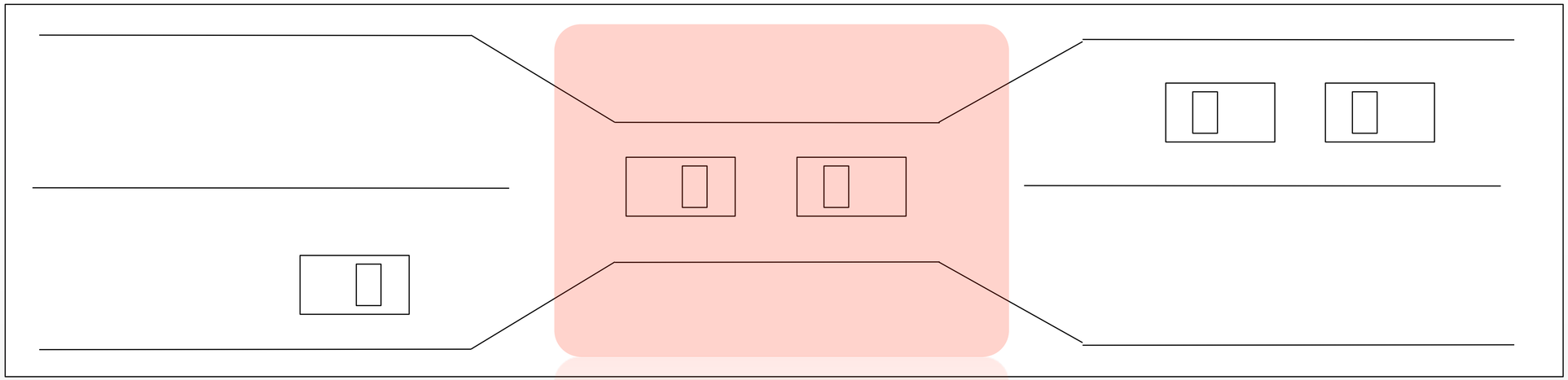
Lecture 8: Deadlocks

COMP362 Operating Systems
Prof. AJ Bieszczad

Outline

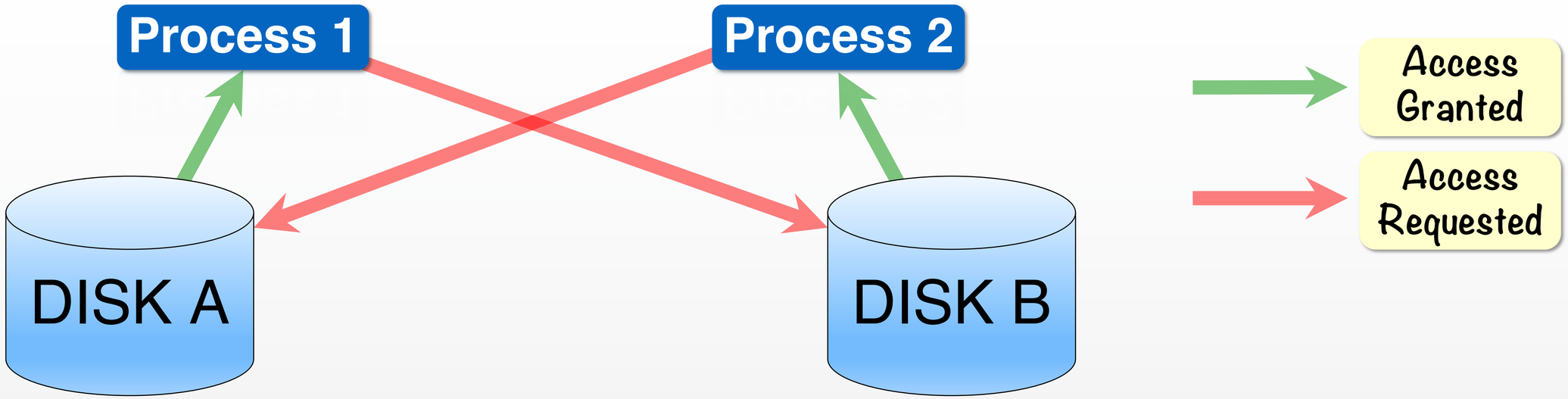
- The Deadlock Problem
- System Modeling
- Conditions for a Deadlock
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
  - Recovery from Deadlock

Problems with Sharing a Narrow Bridge



- Imagine that a section of a bridge is too narrow for two cars
  - i.e., the section is a resource shared by cars coming from both sides.
- If two cars start entering it from opposite directions, the bridge is deadlocked
  - i.e., no traffic in any direction is possible.
- The deadlock can be resolved if one or more cars back up.
- If only one side yields all the time, then the cars on the other side of the river will never cross.

Problems with Sharing Computer Resources



- Say, we have a system with two disk drives, A and B.
- Imagine that Process 1 needs to copy a file from disk A to disk B, and Process 2 needs to copy a file from disk B to disk A.
- In this (somewhat primitive) system, there are no buffers, so a process needs to secure access to both disks to perform the operation.
- One possible scenario is that the Process 1 acquires access to disk A, and Process 2 acquires access to disk B.
- Each process needs two disks to carry out the copy operation, so to then they will both wait.
- Processes are deadlocked waiting for each other to release the other disk.

Problem with Pthreads Locks

Thread 1

```
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
     * Do some work
     */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}
```

Thread 2

```
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
     * Do some work
     */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

- Threads are deadlocked waiting for each other to release the mutex that they do not hold.

Deadlock

- Deadlock can arise if the four following conditions hold simultaneously.

- **Mutual exclusion**
  - only one process at a time can use a resource.

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

- **No preemption**
  - a resource released only voluntarily by the process holding it after that process has completed its task

- **Circular wait**:
  - there exists a set $\{p_0, p_1, ..., p_n\}$ of waiting processes such that p_0 is waiting for a resource that is held by p_1, p_1 is waiting for a resource that is held by $p_2, ..., p_{n-1}$ is waiting for a resource that is held by p_n, and p_n is waiting for a resource that is held by p_0.

Livelock

Thread 1

```
void *do_work_one(void *param){
  bool done = false;
  while (!done){
    pthread_mutex_lock(&first_mutex);
    if (pthread_mutex_trylock(&second_mutex)){
      /**
       * Do some work
      **/
      pthread_mutex_unlock(&second_mutex);
      pthread_mutex_unlock(&first_mutex);
      done = true;
    }
    else
      pthread_mutex_unlock(&first_mutex);
  }
  pthread_exit(0);
}
```

Thread 2

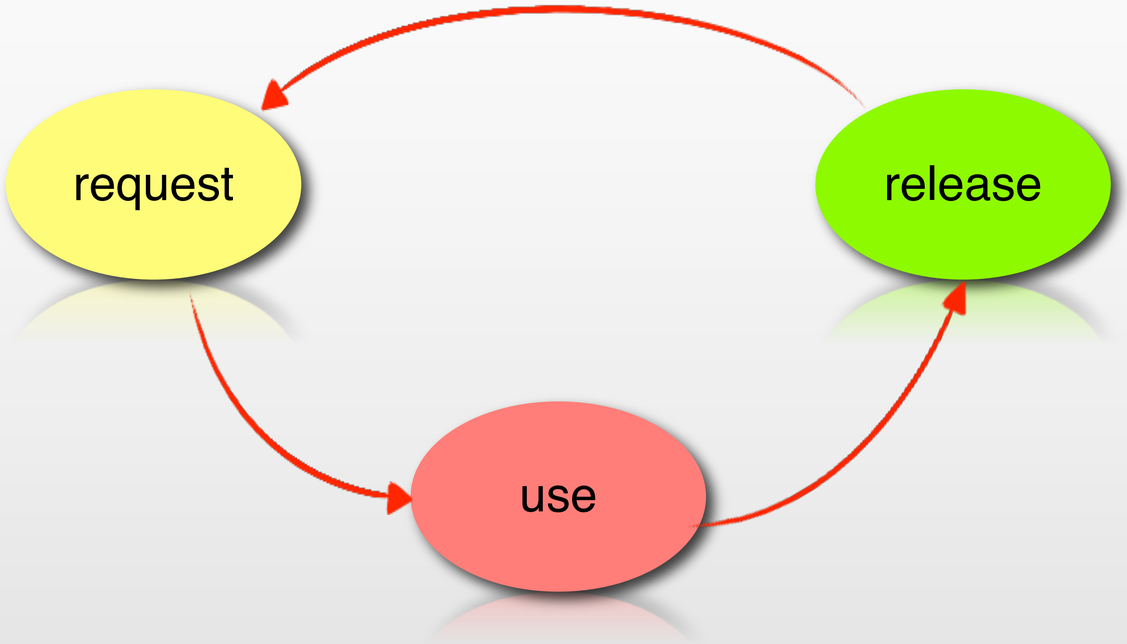
```
void *do_work_two(void *param){
  bool done = false;
  while (!done){
    pthread_mutex_lock(&second_mutex);
    if (pthread_mutex_trylock(&first_mutex)){
      /**
       * Do some work
      **/
      pthread_mutex_unlock(&first_mutex);
      pthread_mutex_unlock(&second_mutex);
      done = true;
    }
    else
      pthread_mutex_unlock(&second_mutex);
  }
  pthread_exit(0);
}
```

- Threads are not blocked, but they may no be able to do any work
- **Backoff** is a technique that randomizes access to resources to prevent contention
  - important concept used in other contexts

System Modeling

- Let us institute the following formal model of a resource allocation system:
- **n** processes $P_1, P_2, ..., P_n$
- **m** resource types $R_1, R_2, ..., R_m$
  - such as CPU cycles, memory space, I/O devices, etc.
- Each resource type R_i has W_i **instances**
  - e.g., five drives, two Ethernet ports, etc.

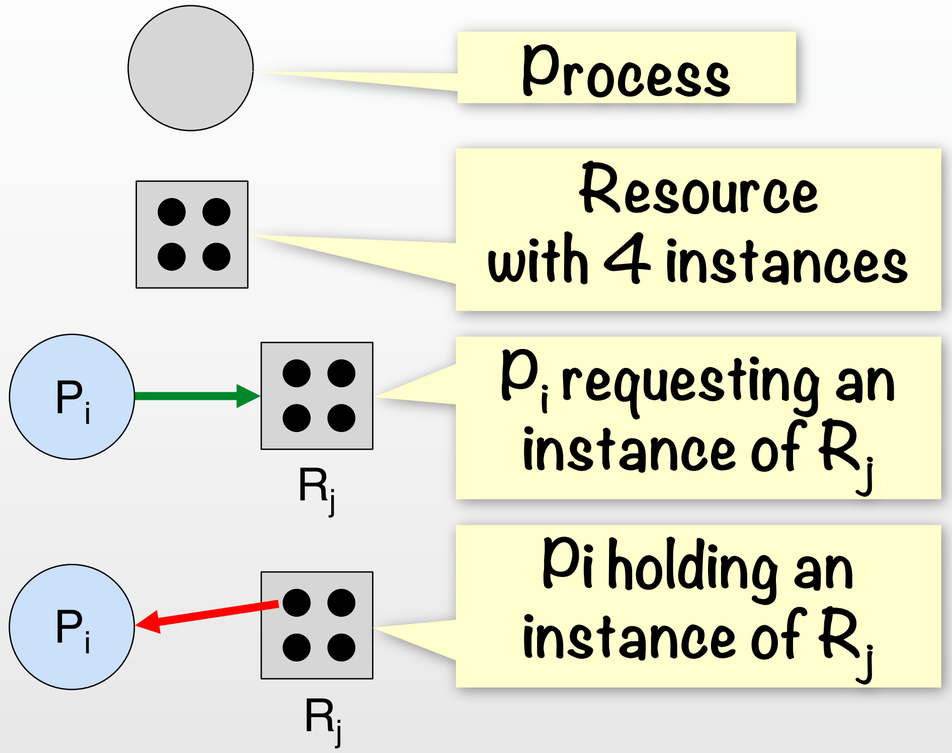
- Each process utilizes a resource as follows:



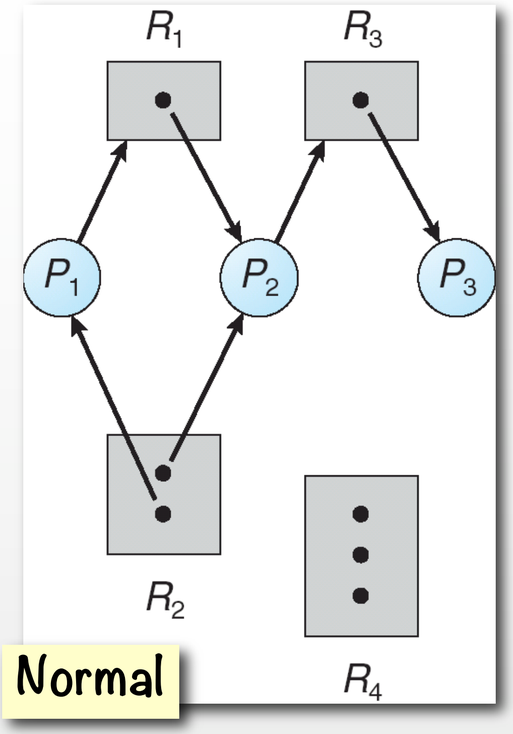
Resource-Allocation Graph

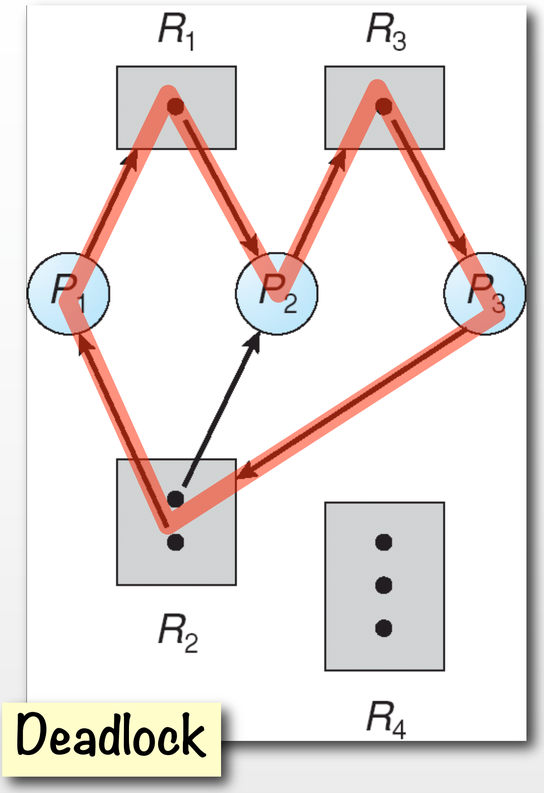
- A set of vertices $V = V_P \cup V_R$ and a set of edges $E = E_R \cup E_A$ such that
  - V is partitioned into two types:
    - $V_P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system.
    - $V_R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.
  - **request edge** $P_i \rightarrow R_j$
    - $E_R = \{..., (P_i, R_j), ...\}$
  - **assignment edge** $R_j \rightarrow P_i$
    - $E_A = \{..., (R_j, P_i), ...\}$

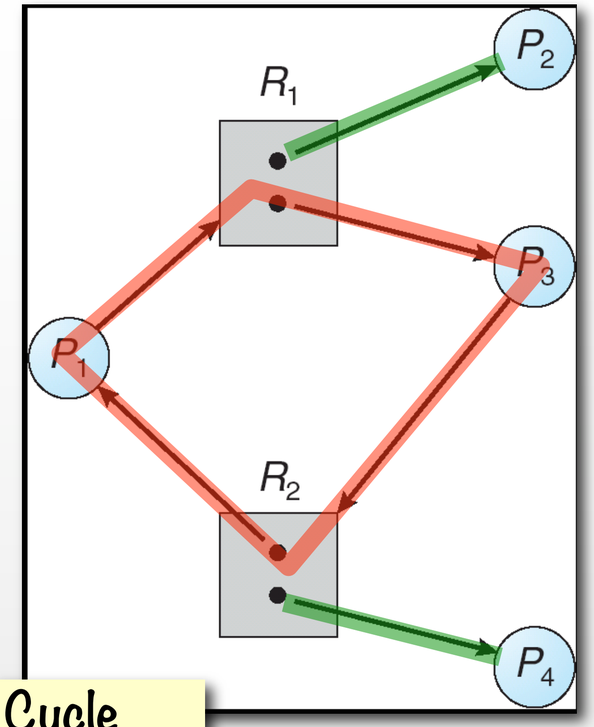
- Graphically:



Examples of a Resource Allocation Graph



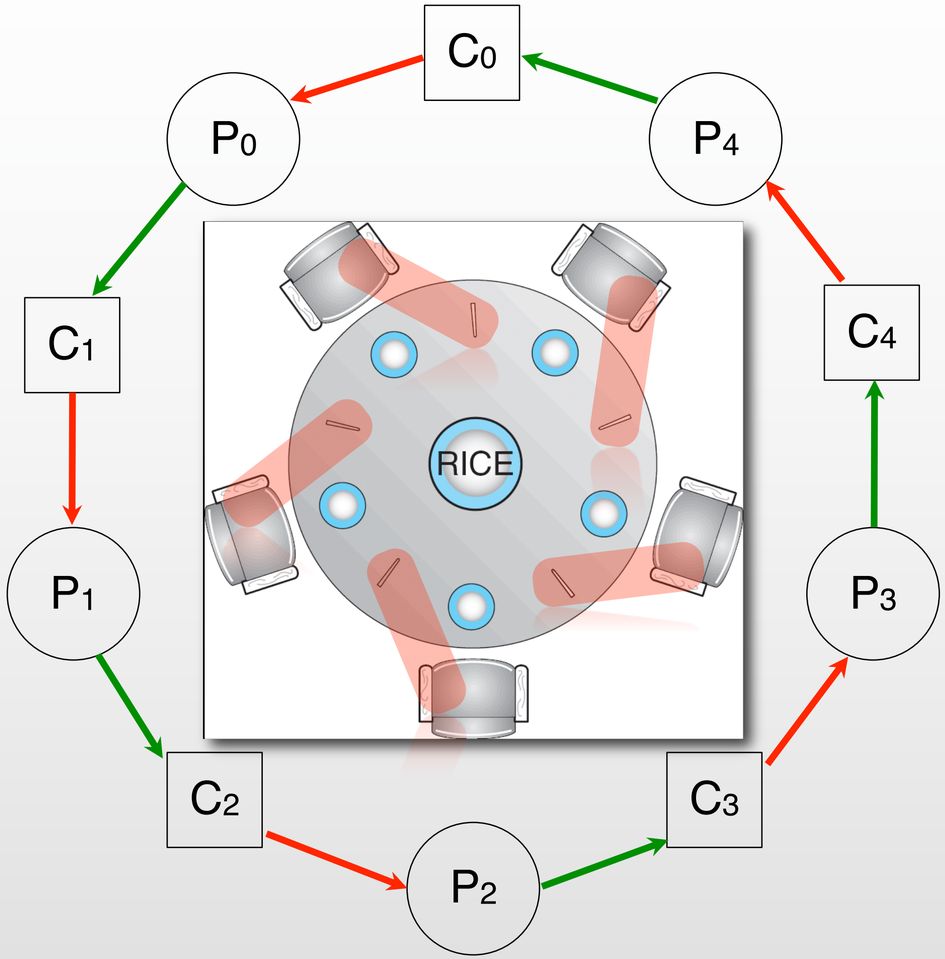




- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock,
  - if several instances per resource type, possibility of deadlock.

Deadlock with Dining Philosophers

- It was possible for a circle to form in the resource allocation graph.




```
do  {
   wait ( chopstick[i] );

   // possible context switch

   wait ( chopstick[ (i + 1) % 5] );

   //  eat (CRITICAL SECTION)

   signal ( chopstick[i] );
   signal ( chopstick[ (i + 1) % 5] );
} while (TRUE);
```

- Note that we are using $p_0, p_1, ..., p_{n-1}$ to simplify the coding

Methods for Handling Deadlocks

- **Deadlock prevention**
  - Engineer the system in a way that eliminates deadlocks completely.
- **Deadlock avoidance**
  - Manage system so it avoids “dangerous” states (i.e., ones that may lead to a deadlock)
- **Deadlock recovery**
  - Allow the system to enter a deadlock state but provide a method for deadlock detection and recovery.

- Most common approach:

Ignore the problem and pretend that deadlocks never occur

  - **used by most operating systems**, including UNIX
  - leaves it up to the programmers to take care of the problem
  - it’s **unacceptable in certain situations**
    - e.g., real-time mission-critical systems
      - imagine a melting nuclear plant whose control system is in a deadlock!

Deadlock Prevention

- Recall that all of the following conditions are **necessary** for a deadlock to occur :

  (1) mutual exclusion,

  (2) hold and wait,

  (3) no preemption, and

  (4) circular wait

To prevent a deadlock, restrain the ways request can be made to remove at least one of the conditions necessary for a deadlock.

- Let's try then to design a system with one of the conditions removed!

Deadlock Prevention

- 1. **Breaking the mutual exclusion condition**
  - cannot control this, because some resources are intrinsically non-sharable and require exclusive access
    - examples?

- 2. **Breaking the hold and wait condition**
  - the system may be designed to demand that whenever a process requests a resource, it does not hold any other resources
    - EITHER require a process to request and be allocated all its resources before it begins execution,
    - OR, allow a process to request resources only when it holds none
      - the process must release held resources and then re-acquire them together with any desired additional resources
  - low resource utilization
  - starvation possible

recall the DP solution with `try_lock()`

Deadlock Prevention (Cont.)

- 3. **Breaking the no preemption condition**
  - the system is allowed to take away resources from a process forcefully
    - unlike (2), this is involuntary system action; a process may run without some resources that it does not immediately needs
    - if a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held by that process are forcefully released
      - preempted resources are added to the list of resources for which the process is waiting
    - process will be restarted only when both the old resources as well as the new ones that it requested are available
  - low resource utilization
  - starvation possible

Deadlock Prevention (Cont.)

4. Breaking the circular wait condition

- impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
  - we create tiers of resources

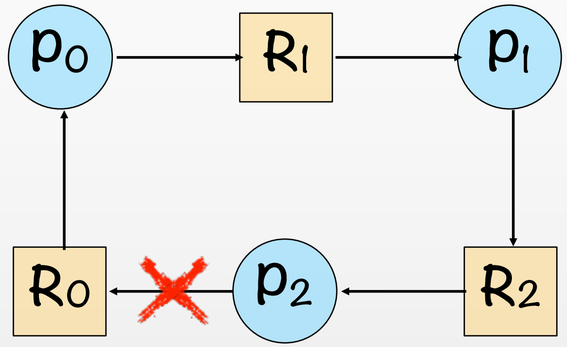
$$F: R \rightarrow N$$

  - where F is a **mapping function** from R to N (**enumeration**)
    - $R = \{R_1, R_2, ..., R_m\}$ is a domain of resources
    - N is the set of natural numbers

- if a process holds R_i then it can request R_j only if $F(R_i) < F(R_j)$
  - alternatively,
    we can insist that to gain access to R_j process must release all R_k such that $F(R_k) >= F(R_j)$

Resource Ordering: *Prevention Proof*

- To prove by contradiction, assume that we followed the rules but got a deadlock
  - therefore, we have a cycle, so for any **i**, p_i waits for some R_{i+1} that is allocated to p_{i+1},



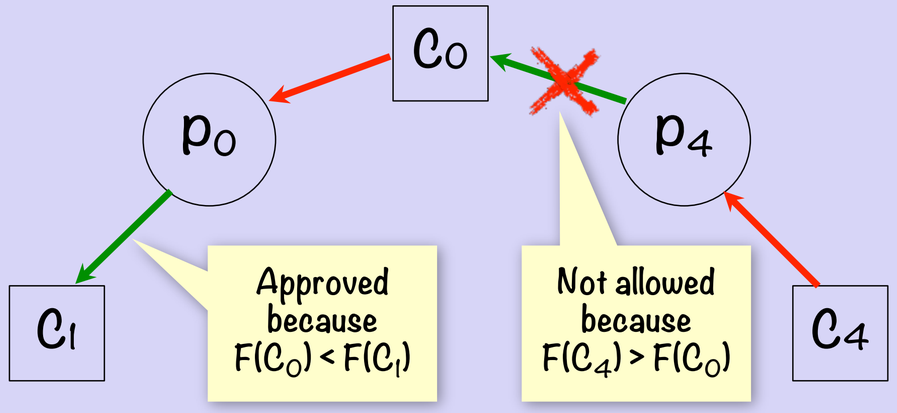
  - however, for p_{i+1} to be allowed to request R_{i+1} it must have held that $F(R_i) < F(R_{i+1})$, so by Mathematical induction and using modulo-*n* increment it holds that $F(R_0) < F(R_1) < \ldots < F(R_{n-1}) < F(R_0)$
  - and that's not possible; hence, the proof

Example

If we imposed the following order in our Dining Philosophers example

$F(C_0) = 1$
$F(C_1) = 2$
$F(C_2) = 3$
$F(C_3) = 4$
$F(C_4) = 5$

the the deadlock would not happen because p_4 has to give up C_4 if it wants to request C_0



Deadlock Avoidance

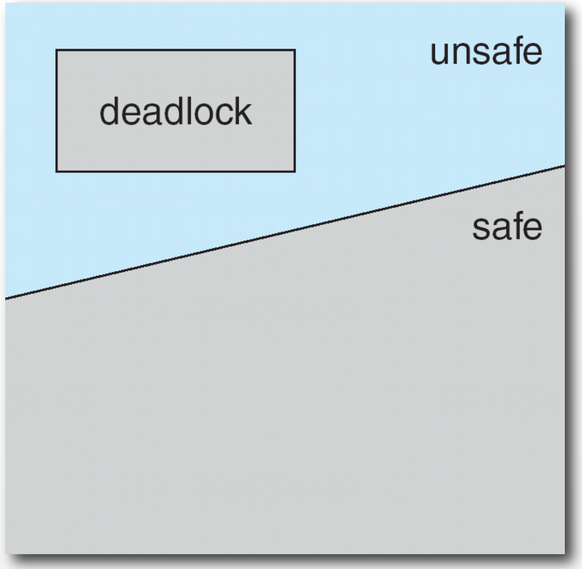
- Deadlock prevention scheme is very expensive
  - the OS would be very restrictive, and therefore inefficient
    - however, it may be necessary in certain mission critical systems

- **Deadlock avoidance** takes a bit less restrictive approach
  - it requires that the system is provided with some additional a priori information about a process available
  - the system uses this information to make decision about resource allocation each time any process requests a resource

- Simplest and most useful model requires that each process declares 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, OS must decide if immediate allocation leaves the system in a safe state



  - If a system is in safe state ⇒ no deadlocks.
  - If a system is in unsafe state ⇒ possibility of deadlock.
  - Avoidance ⇒ ensure that a system will never enter an unsafe state.

- System is in **safe state** if there exists a scheduling sequence $<P_1, P_2, ..., P_n>$ of all the processes in the system such that for each P_i, the resources that P_i may still request can be satisfied by currently available resources and resources held by all the P_j for $j < i$ (i.e., the processes that precede P_i in the scheduling sequence; i.e., the ones that will terminate earlier releasing the resources that they hold).
  - When the system is in a safe state:
    - If P_i resource needs are not immediately available, then P_i can wait until sufficient number of P_j (for $j < i$) have finished. The safe state definition ensures that at some point P_i will be able to get all that it needs to continue
    - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
    - In turn, when P_i terminates, P_k for some $k > i$ can obtain its needed resources, and so on.

Example of Safe State and Unsafe State

- Let the system with 13 devices have p_0, p_1, and p_2 processes with the shown needs

Process	Claims	Current allocation
P_0	10	5
P_1	4	2
P_2	9	2

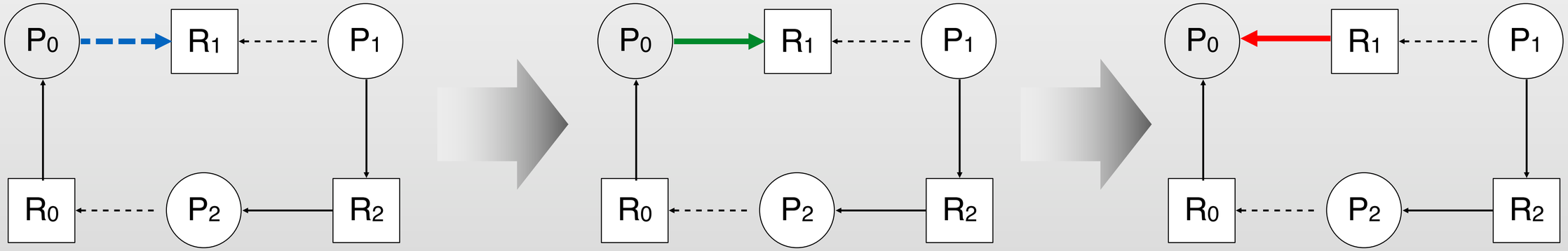
note that there are 4 devices free

- $<p_1, p_0, p_2>$ sequence satisfies the safe state requirements
  - p_1 gets extra 2 devices (while 2 are left free) and then terminates releasing 4 devices; after that, we have 6 devices free in the pool
  - p_0 can now execute with 10 devices (getting 5 from the free pool leaving 1 still available), then it terminates and releases the resources; so we have 11 free
  - p_2 is now able to acquire 7 additional devices (up to the needed 9), execute and terminate
- Now, let's consider that p_2 is allocated two more devices first, the system **<u>may not be able</u>** to fulfill future demands, so this is unsafe state; here is one bad case:
  - with p_2 having 2 more resources (so it has 4 now), only 2 are still free
  - only p_1 can be allocated all claimed resources (2+2=4); none is left free
  - when p_1 terminates and returns the resources, there are 4 free
  - now neither p_0 (10, 5) nor p_2 (9, 4) can continue if they need the maximum number of claimed resources; both p_0 and p_2 may need 5 extra

Note that unsafe state may or may not lead to a deadlock. Nevertheless, to prevent the worst case, we need to avoid such states.

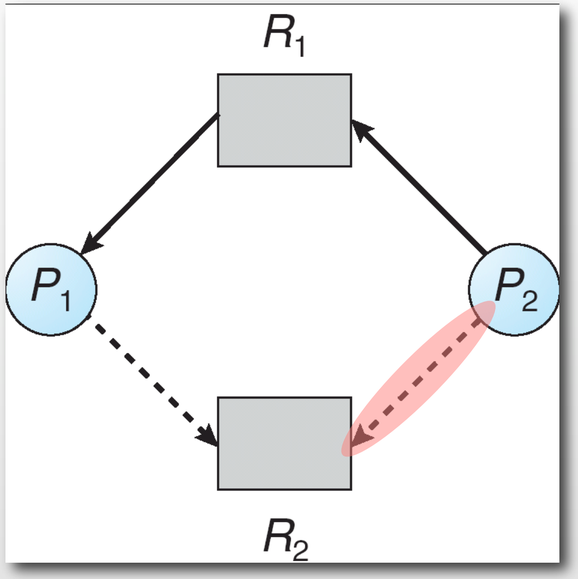
Avoidance with Resource-Allocation Graph

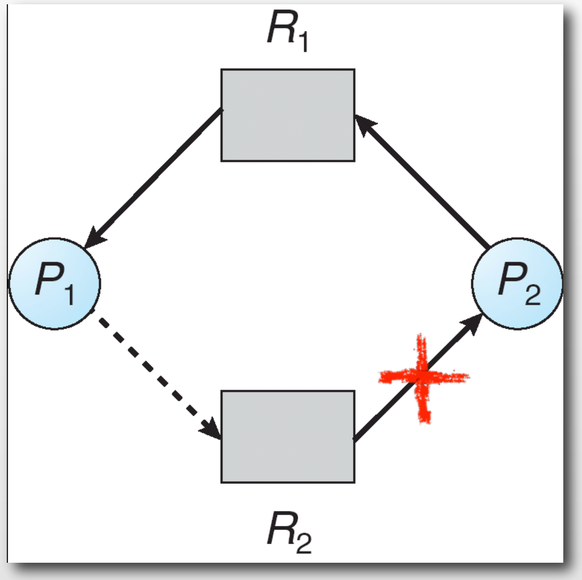
- First, we augment resource allocation graph by adding a **claim edge** $P_i \rightarrow R_j$ indicating that process P_i **may request** resource R_j represented by a dashed line.
- **Claim edge** converts to a **request edge** when a process indeed requests a resource.
- **Request edge** is converted to an **assignment edge** when the resource is allocated to the process.
- When a resource is released by a process, **assignment edge** reconverts to a **claim edge**.
- Resources must be claimed a priori in the system.



Avoidance Algorithm Example

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph
- In the following example, allocating R_2 to P_2 is therefore not allowed!



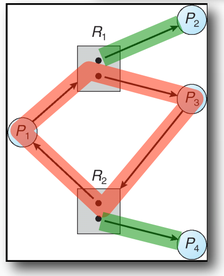


Note that there is no deadlock yet; just the possibility!

- Detecting a cycle in a graph is of $O(n^2)$ complexity

Avoidance with Banker's Algorithm

- Resource allocation graph can be used only with single instance resources
  - since – as we saw earlier - detecting a cycle with a multi-instance resource does not necessarily constitute a deadlock danger



- The **banker's algorithm** (invented by Dijkstra) is suitable for cases with multiple instances of resources.
  - the name comes from the fact that banks should use a similar algorithm to allocate funds (e.g., through loans) in such a way that they always can satisfy the current needs of their customers (be solvent)
- Assumptions:
  - each process must **claim maximum need for resources a priori**,
    - including the resources that may not be used at all in some cases
  - 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**.

- The banker's algorithm is a more complex deadlock avoidance scheme than the resource-allocation graph method
  - more overhead for the system

Data Structures for the Banker's Algorithm

- Let
  - n = number of processes P_i with $0 \le i < n$, and
  - m = number of resources R_j with $0 \le j < m$.

Assumption:
processes do not lie!

- Define the following:
  - **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.
    - Note that Need [i,j] = Max[i,j] – Allocation [i,j].
- We can also use a shorthand notation (using **vector arithmetic**):
  - e.g., Need[i] = Max[i] - Allocation[i]
    - where i is a process number, so these vectors are rows in the matrices corresponding to P_i
- Define a relation ≤ as: $X \le Y$ if for every i: $X[i] \le Y[i]$
  - e.g., $(3,2,4) < (4,3,9)$, and $(3,2,4) \; ? \; (2,5,7)$
  - furthermore, $X < Y$ if $X \le Y$ and $X \ne Y$

Undetermined
"≤" imposes only a partial order

Resource Request Algorithm for Process P_i

- Determines if a resource request from process P_i is acceptable
- Let **`Request[i]`** be a request vector for process P_i.
  - If **`Request[i, j] = k`** then process P_i wants k instances of resource type R_j.
    - as before, we drop **`j`** and work with vectors
- Step 1
  - If **`Request[i] ≤ Need[i]`** go to step 2. Otherwise, raise error condition, since process P_i has exceeded its maximum claim.

this contradicts the assumption that processes don't lie

- Step 2

If **`Request[i] ≤ Available`**, go to step 3. Otherwise process P_i must wait, since resources are not available.

- Step 3
  - **Pretend to allocate** requested resources to P_i by modifying the state as follows:
    - save **`Available`**, **`Allocation[i]`**, and **`Need[i]`**
    - **`Available = Available – Request[i]`**
    - **`Allocation[i] = Allocation[i] + Request[i]`**
    - **`Need[i] = Need[i] – Request[i]`**
  - **Now run the safety algorithm** *(see next slide)*
    If safe state ⇒ the requested resources can be allocated to process P_i.
    If state unsafe ⇒ P_i must wait, and the old resource-allocation state is restored from the saved values

Safety Algorithm in Banker's Algorithm

- Recall: to determine if a certain state is safe we need to find a sequence in which processes can execute; the following algorithm will find a sequence if one exysts
- Step 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`**.
- Step 2.
  - Find an **`i`** such that both:
    - (a) **`Finish[i] = false`**
    - (b) **`Need[i] ≤ Work`**
  - If no such **`i`** exists, go to step 4.
- Step 3.
  - we pretend that the process i is done
  - **`Work = Work + Allocation[i]`**
  - **`Finish[i] = true`**
  - go to step 2.
- Step 4.
  - If ∀**`i Finish [i] == true`** → the system is in a safe state.
- The complexity is $O(m*n^2)$

number of processes

number of resources

a collection of n processes searched (possibly) n times for Finish[i]=false, and then through m resources to see if Need[i] ≤ Work

This is not good...

NOTES

1. This is just detection of a safe state. See the next slide for a complete algorithm for resource allocation.
2. All operations are pretended; neither actual allocation nor termination is carried out.
3. The order is not important! The amount of available resources is increased by any process that "terminates", so we always improve the situation.

State Safety Example

- n = 5 processes P_0 through P_4
- m = 3 resource types:
  - A (10 instances)
  - B (5 instances), and
  - C (7 instances).

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

Available		
A	B	C
3	3	2

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

	Need		
	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Work		
A	B	C
3	3	2

- The system is in a safe state since the sequences such as $<P_1, P_3, P_4, P_2, P_0>$ and $<P_3, P_4, P_1, P_2, P_0>$ satisfy safety criteria.

Resource Request Example

- Given the state shown on the previous slide, can P_1 request $(1,0,2)$ be granted?
  - Check the **Request ≤ Need**
    - that is, $(1,0,2) \leq (1,2,2) \Rightarrow$ true
  - Check that **Request ≤ Available**
    - that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true
      - so if the **Request** was granted, we would have:

	Allocation			Need		
	A	B	C	A	B	C
P0	0	1	0	7	4	3
P1	3	0	2	0	2	0
P2	3	0	2	6	0	0
P3	2	1	1	0	1	1
P4	0	0	2	4	3	1

Work		
A	B	C
2	3	0

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement ⇒ **Request** can be granted.

More Examples

- Can request for (3,3,0) by P_4 be granted given the current state as:

	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

Work		
A	B	C
3	3	2

- ?
- Can request for (0,2,0) by P_0 be granted assuming the same current state?

Deadlock Detection

- Prevention and avoidance algorithms are time consuming, so they are used only in most critical systems
- Therefore, usually it is assumed that deadlocks may happen
- If a deadlock occurs, the operating system may be engineered to detect and resolve it
- OS that implements a deadlock detection scheme does not prevent the system from entering a deadlock state
- Instead, it tries to detect potential deadlocks, and if discovered, the OS tries to recover from the problem

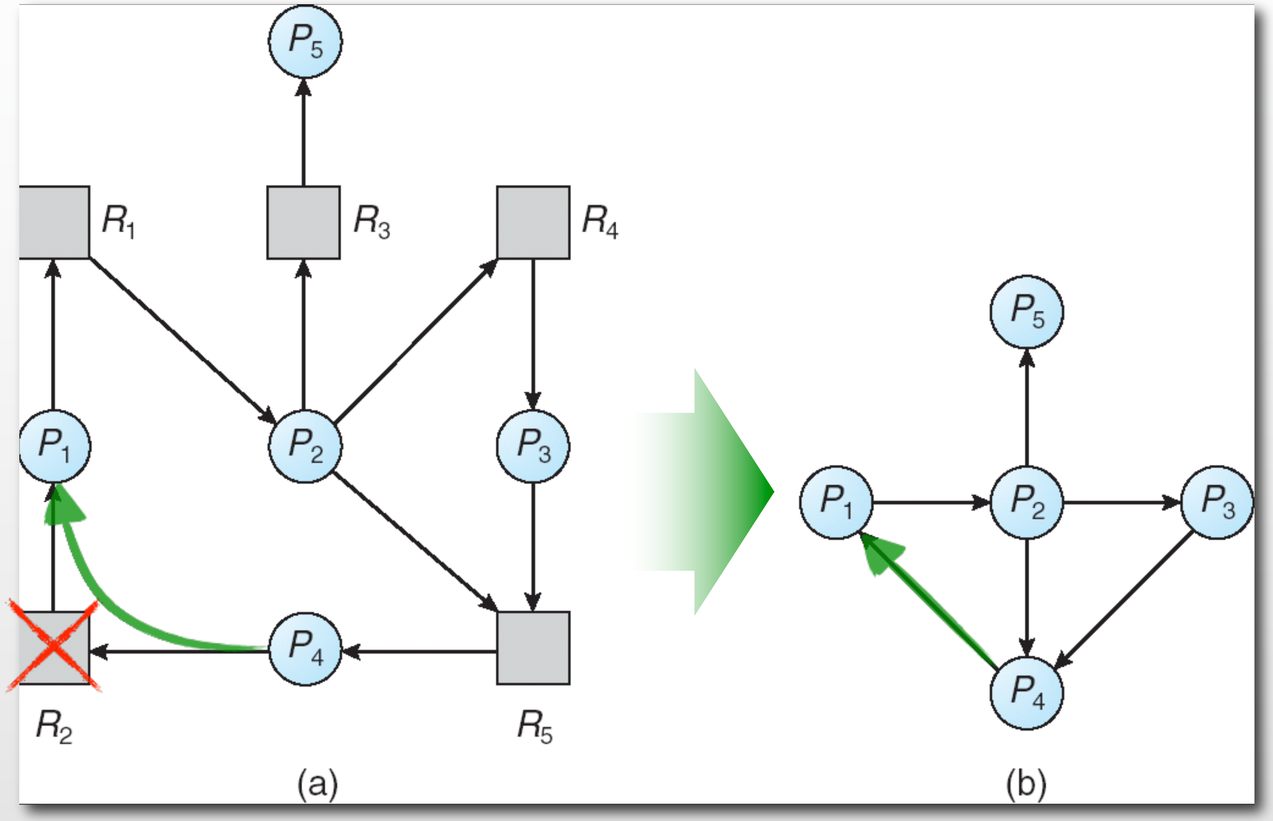
- For that, the OS needs:
  - **Detection algorithm**
  - **Recovery scheme**

Single Instance of Each Resource Type

- Maintain a **wait-for graph**
  - Nodes are processes.
  - There is an edge $P_i \rightarrow P_j$ if P_i is waiting for P_j
    - i.e., if there was R_k such that
      - P_i requested R_k
      - R_k was allocated to P_j
- Periodically invoke an algorithm that searches for a cycle in the graph.
  - If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph

Corresponding wait-for graph



This is not good either... but at least we control the frequency

Several Instances of a Resource Type

- We cannot use the wait-for graph to manage multiple instances of resources, so we need another algorithm (*again: because a cycle is not necessarily a deadlock*)
- Based on the banker's algorithm (*see earlier slides*)
- Like earlier, let
  - n = number of processes P_i with $0 \leq i < n$, and
  - m = number of resources types.
- Similarly to the deadlock avoidance algorithm, we use a number of data structures:
  - 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

- **Step 1**
  - Let **`Work`** and **`Finish`** be vectors of length **`m`** and **`n`**, respectively
  - Initialize:
    - **`Work = Available`**
    - for **`i`** = 0, 1, 2, ..., **`n`**, **if `Allocation[i] ≠ 0`**, **then `Finish[i] = false`**; otherwise, **`Finish[i] = true`**.
- **Step 2**
  - Find an index **`i`** such that both:
    - **`Finish[i] == false`**
    - **`Request[i] ≤ Work`**
    - if no such **`i`** exists, go to **step 4**.
- **Step 3**
  - **`Work = Work + Allocation[i]`**
  - **`Finish[i] = true`**
  - go to **step 2**.
- **Step 4**
  - If ∃**`i`**, 0 ≤ **`i`** < **`n`**, **`Finish[i] == false`** ⇒ then the system is in deadlock state.
  - Moreover, if **`Finish[i] == false`**, then p_i is deadlocked.
    - So, we not only know that there is a deadlock, but also which processes are deadlocked
- Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state.

only processes with some resources allocated may be deadlocked

simulate process termination

As earlier, this is not good... but... at least we control the frequency here

Deadlock Detection Example

- Five processes P_0 through P_4
  - three resource types
    - A (7 instances)
    - B (2 instances)
    - C (6 instances).

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

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

- However, if - for example - P_2 is allocated an additional instance of type C then there is a problem:
  - we can reclaim resources held by process P_0, but the processes P_1, P_2, P_3, and P_4 are stuck with **Finish[i] = false**, so there is a deadlock, and all of these processes are deadlocked.

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

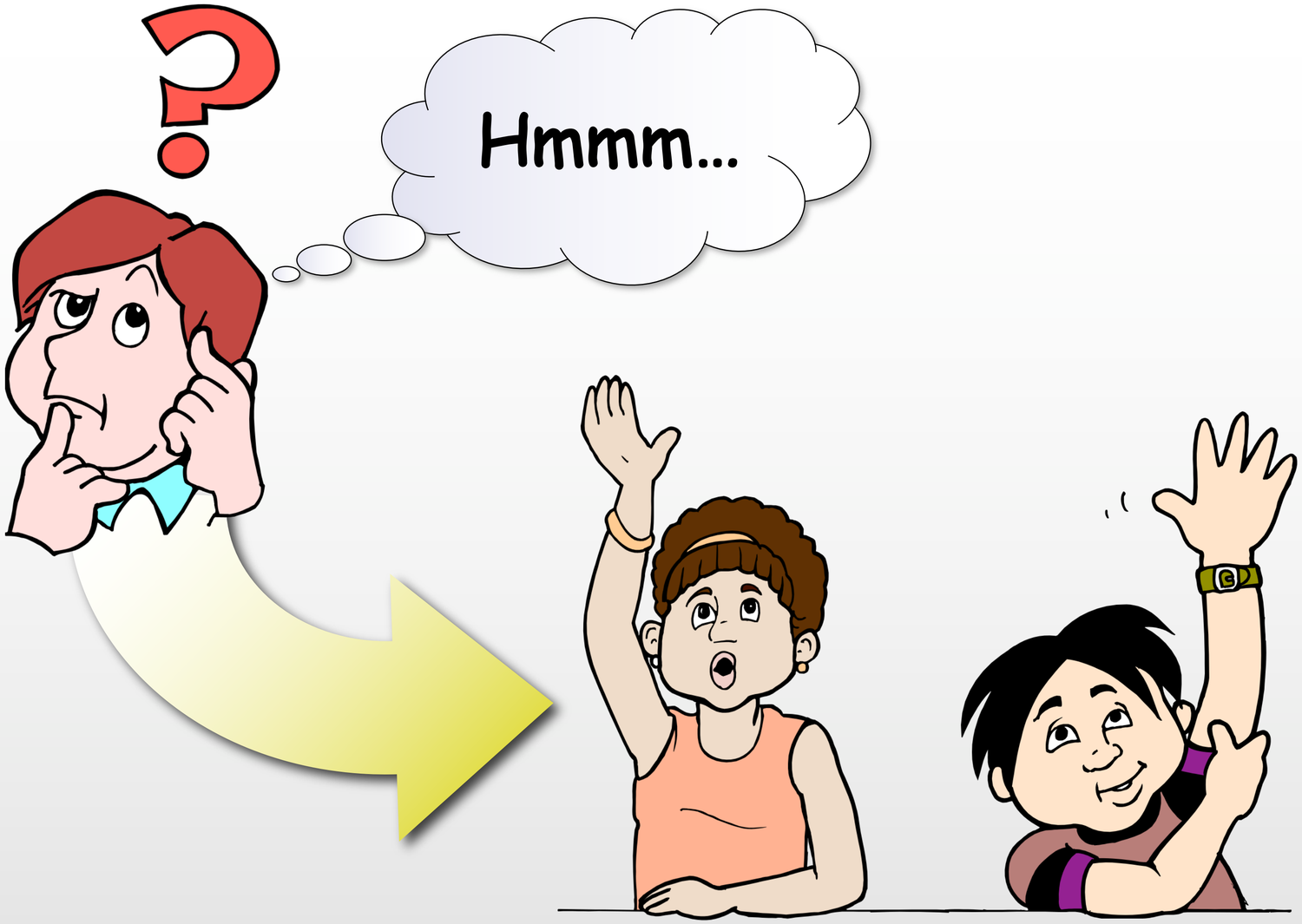
Detection-Algorithm Usage

- Many ”engineering” questions to answer
- When, and how often to invoke?
  - depends on:
    - how often a deadlock is likely to occur?
    - how many processes will need to be rolled back/terminated?
      - e.g., one for each disjoint cycle
- There may be many cycles in the wait-for graph, which one is “more important” to break?
- Which of the many deadlocked processes “causes” the deadlock?

Recovery from Deadlock: Process Termination

- Recovery from a deadlock can be done by terminating all or some processes, or by reclaiming resources
  - leaving the process alive and just reclaiming is very difficult
- How to select a victim?
  - minimize cost - but what exactly is the cost?
  - Abort all deadlocked processes.
    - simple but dirty
  - Abort one process at a time until the deadlock cycle is eliminated.
    - Question: In which order should we choose to abort?
      - Priority of the process.
      - How long process has computed, and how much longer to completion.
      - Resources the process has used.
      - Resources process needs to complete.
      - How many processes will need to be terminated.
      - Is process interactive or batch?
- Problem: Starvation
  - same process may always be picked as victim

- May consider **rollbacks**
  - i.e., return a deadlocked process to some safe state, and restart the process from that state
  - extremely expensive, since the system must save safe states



COMP362 Operating Systems

Prof. AJ Bieszczad