

Operating Systems
(INFR09047)
2019/2020 Semester 2

Deadlocks

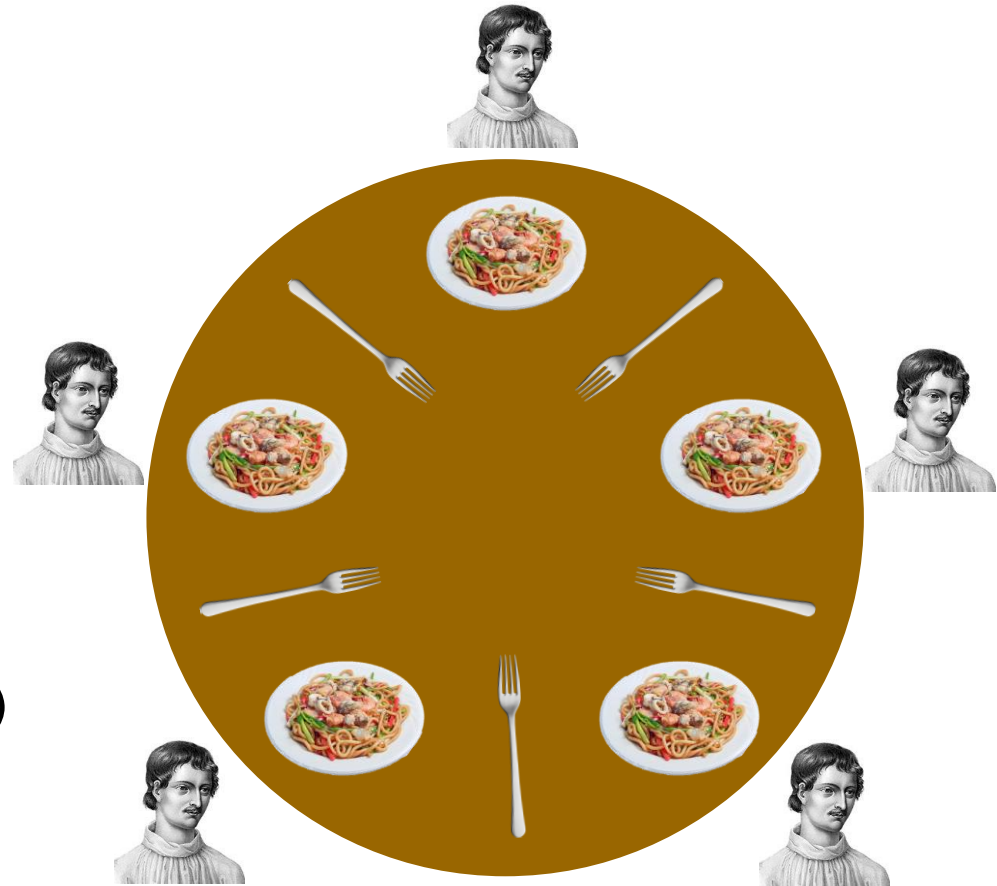
abarbala@inf.ed.ac.uk



No one can proceed (by following its own *official* direction)

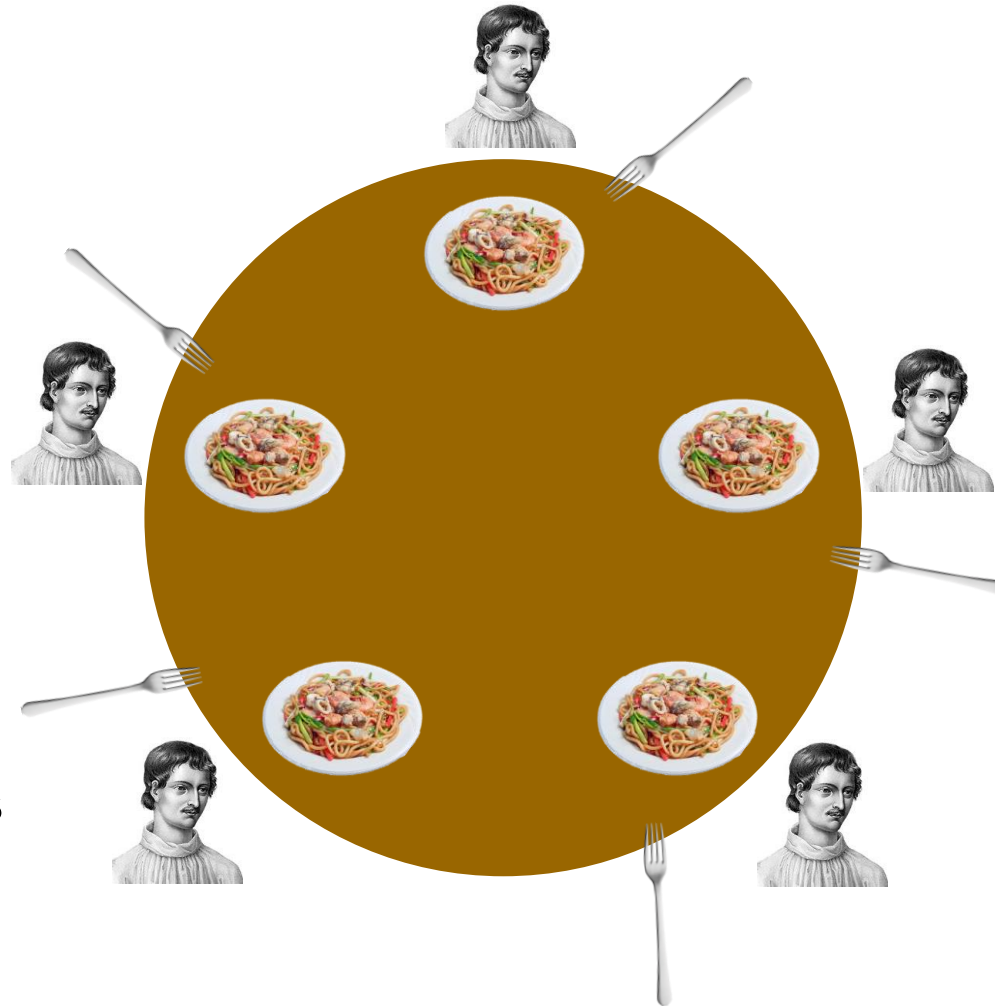
Dining Philosophers Problem

- Dijkstra, 1965
- 5 philosophers, each with
 - A spaghetti dish
 - A fork
- To eat, a philosopher needs **two** forks
 - Spaghetti are too slippery
 - (Yes, we do that in Italy too)
- A philosopher alternates periods of
 - Eating
 - Thinking



Deadlock Problem

- When
- all **grab** the fork at their **left** at the same time
- and **wait** for the one at their **right**
 - They will all **indefinitely wait** without no one eating
- **Cause**
 - Each philosopher needing what another philosopher has
 - Result of sharing resources



Deadlock Example #1

From real-life example to code example

Semaphore `mutexA = 1` /* protects resource A */

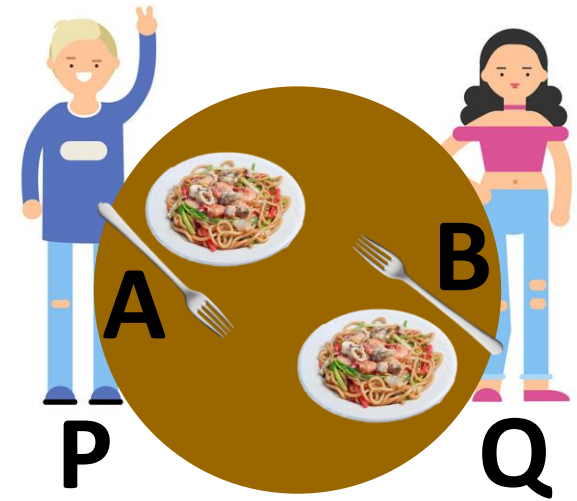
Semaphore `mutexB = 1` /* protects resource B */

Process P:

```
{  
  /* initial compute */  
  down(mutexA)  
  down(mutexB)  
  /* use both resources */  
  up(mutexA)  
  up(mutexB)  
}
```

Process Q:

```
{  
  /* initial compute */  
  down(mutexB)  
  down(mutexA)  
  /* use both resources */  
  up(mutexB)  
  up(mutexA)  
}
```



Possible Deadlock

Deadlock Example #2

From real-life example to code example

Semaphore `mutexA = 1` /* protects resource A */

Semaphore `mutexB = 1` /* protects resource B */

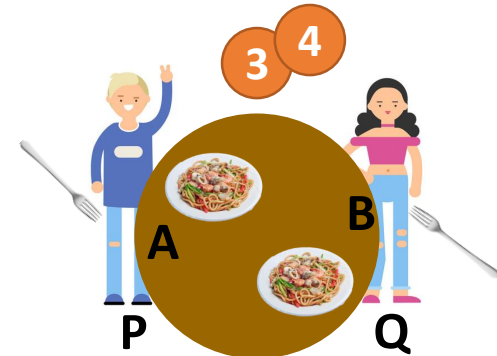
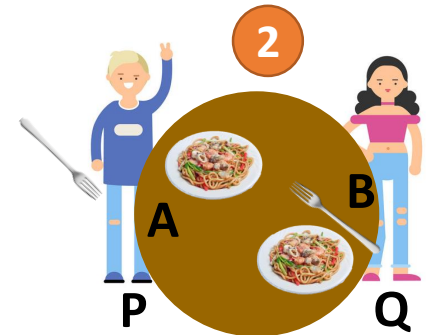
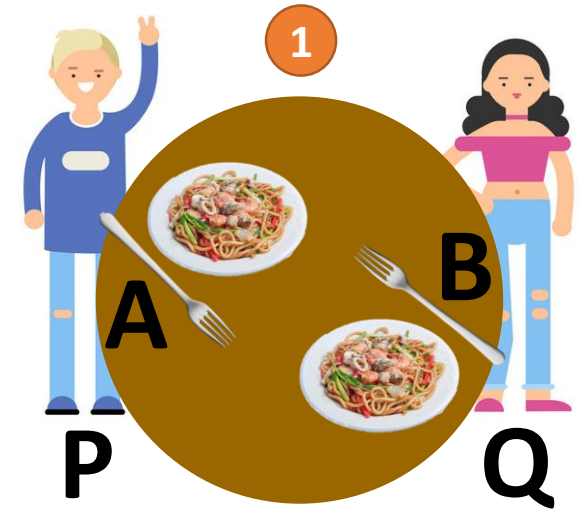
Process P:

```
{  
  /* initial compute */  
  down(mutexA)  
  down(mutexB)  
  /* use both resources */  
  up(mutexA)  
  up(mutexB)  
}
```

Process Q:

```
{  
  /* initial compute */  
  down(mutexB)  
  down(mutexA)  
  /* use both resources */  
  up(mutexB)  
  up(mutexA)  
}
```

Actual Deadlock



Deadlock Example – Try it out!

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;
int main() {
    pthread_mutex_init(&first_mutex, NULL);
    pthread_mutex_init(&second_mutex, NULL);
    /** * create thread one and thread two */
    /** * wait threads to finish */
    return 0;
}
/* thread one runs in this function */
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 two runs in this function */
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);
}
```

- Complete the program
 - Add `pthread_create`, and `pthread_join`
 - Add `printf` to track program progresses
- Compile the program
 - Don't forget `-pthread`
- Run the program
 - How many times should run it before a deadlock?

Deadlock

- In a **multiprogramming** environment
 - Threads or processes compete for a finite number of resources
 - Each thread or process requests resources
 - If resources are not available, the thread or process waits
 - Waiting thread or process **never runs again** because the requested resource(s) is(are) held by other waiting threads or processes
- **Liveness failure**
- **Deadlock Definition**
 - A situation in which every process/thread in a set of processes/threads
 - **is waiting for an event** that can be **caused only by another process/thread in the set**

Livelock

- Example
 - **Two people** attempt to pass in a hallway
 - One moves to his right, the other to her left, still obstructing each other's progress
 - Then he moves to his left, and she moves to her right, and so forth
 - They aren't blocked, but they aren't making any progress
- **Liveness failure**, similar to deadlock
 - Both prevent two or more threads/processes from proceeding
 - Threads/processes are unable to proceed **for different reasons**
- **Deadlock**
 - Occurs when every thread/process in a set **is blocked waiting** for an event that can be caused only by another thread/process in the set
- **Livelock**
 - Occurs when a thread/process **continuously attempts** an action that fails

The next slides focus on deadlock

Necessary Conditions for **Deadlock**

A. Mutual exclusion

- Each resource is either currently assigned to exactly **one** process/thread or is available

B. Hold and wait

- Processes/threads currently holding resources that were granted earlier can request new resources

C. No preemption

- Resources previously granted cannot be forcibly taken away from a process/thread
 - They must be explicitly released by the process/thread holding them

D. Circular wait

- There must be a circular list of two or more processes/thread
 - Each waiting for a resource held by the next member of the chain

All conditions must hold for a deadlock to occur

System Model

(Resources and Processes)

*A system consists of a **finite number of resources**, partitioned into several types/classes, to be **distributed among a number of competing threads/processes***

- Resource types/classes (**m**)
 - **R_1, R_2, \dots, R_m**
 - E.g., printers, disks, etc.
- Each resource type **R_i** has **E_i** instances
 - E.g., 3 printers, 5 disks, etc.
- Assume serially reusable resources
 - request -> use -> release
- Processes (**n**)
 - **P_1, P_2, \dots, P_n**
 - Instead of processes a model with **threads** can be developed

Resource-allocation Graph #1

- A way to visualize **CURRENT STATE** of threads/processes
 - Like a snapshot
- A set of vertices V and a set of edges E
- V is partitioned into two types
 - **Process** vertices $P = \{P_1, P_2, \dots, P_n\}$, the set of processes
 - **Resource** vertices $R = \{R_1, R_2, \dots, R_m\}$, the set of resource types
 - **Not resources!**
- E is partitioned into two types
 - **Request** edge – directed edge $P_i \rightarrow R_j$
 - **The process is blocked waiting for that resource**
 - **Assignment** edge – directed edge $R_j \rightarrow P_i$
 - **The process owns the resource**

Resource-allocation Graph #2

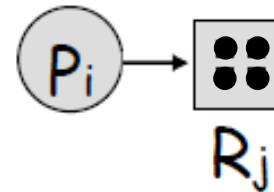
- Process node



- Resource node with 4 instances

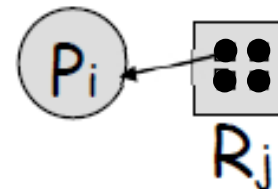


- P_i requests instance of R_j



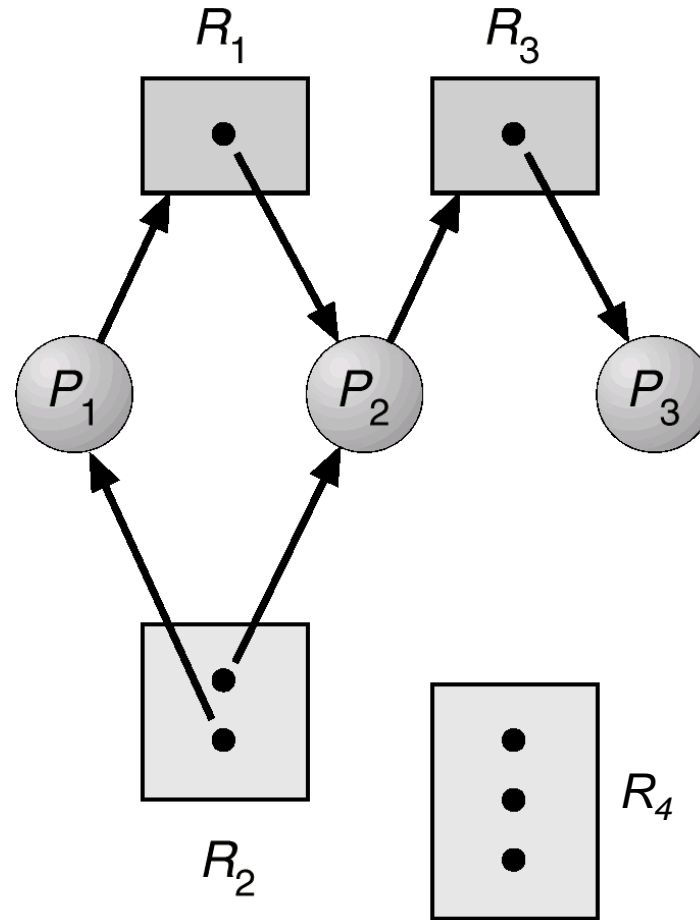
Request Edge

- P_i is holding an instance of R_j



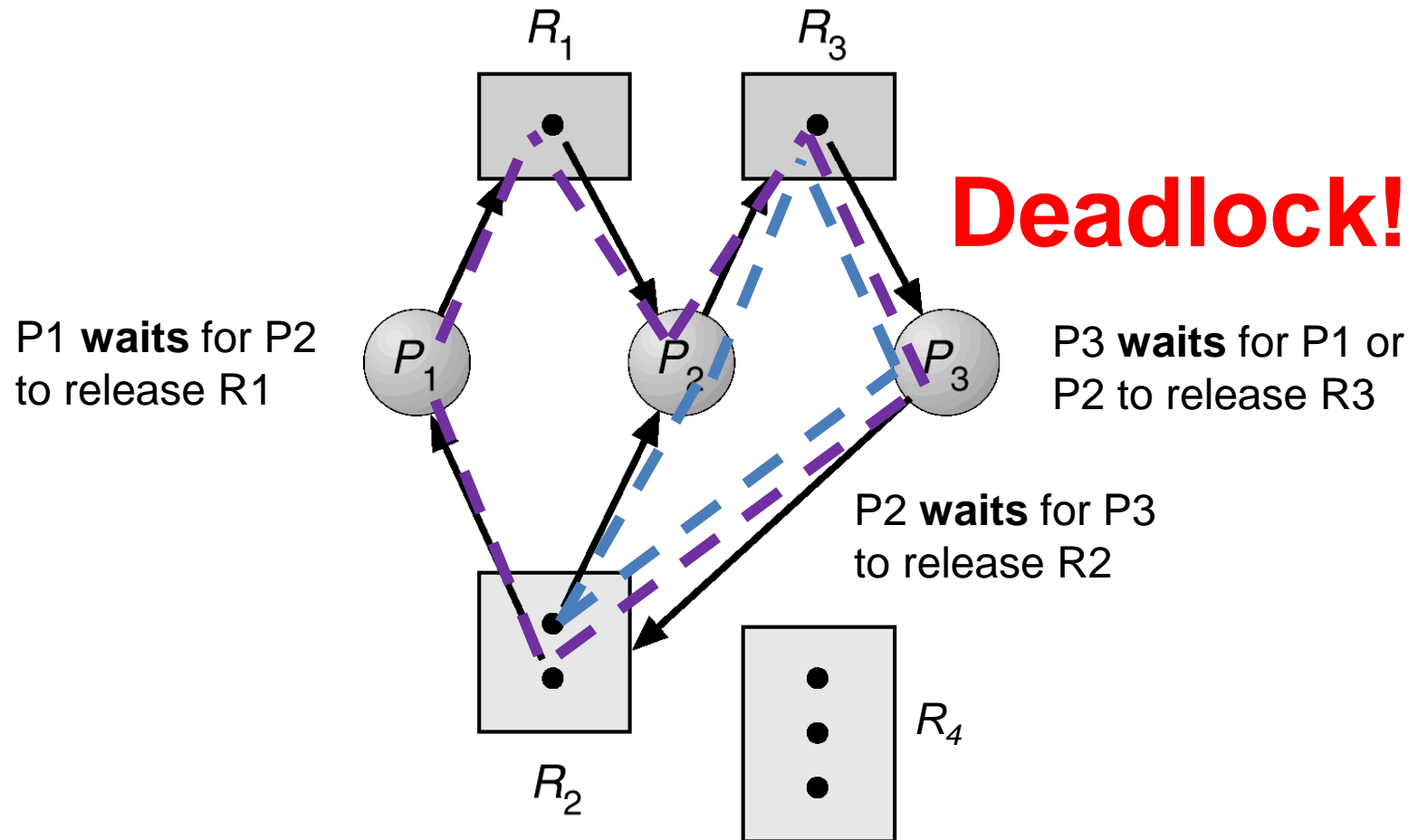
Assignment Edge

Resource Allocation Graph with No Cycle and No Deadlock



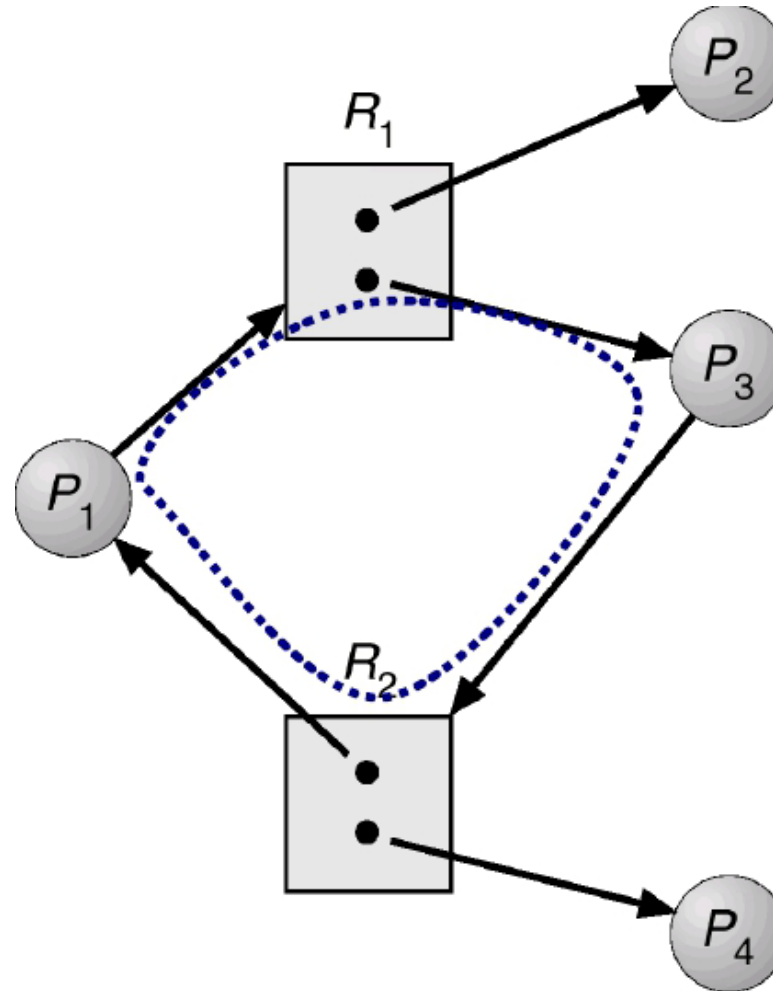
Graph contains **no cycles** → No thread/process is **deadlocked**

Resource Allocation Graph with Cycles and Deadlock



If the graph **contains a cycle**, then a deadlock may exist

Resource Allocation Graph with Cycle and No Deadlock



A cycle **is not sufficient** to imply a deadlock

Notes

- If each resource type has **exactly one instance**
 - Then a cycle implies that a **deadlock has occurred**
- If the cycle involves only **a set of resource types**, each of which has **only a single instance**
 - Then a **deadlock has occurred**
- Each thread involved in the cycle is deadlocked
 - A cycle in the graph is both **a necessary and a sufficient condition**
- If each resource type has **several instances**
 - then a cycle doesn't necessarily imply that a deadlock has occurred
 - a cycle in the graph **is a necessary but not a sufficient condition** for the existence of deadlock

HOW TO DEAL WITH DEADLOCKS?

Handling Deadlocks

- **Ensure deadlock **never** occurs**
 - **Prevention** (development time)
 - Negate one of the four necessary conditions
 - **Avoidance** (runtime)
 - Each resource request is analyzed and denied if may deadlock
- **Allow deadlock to **happen****
 - **Detection and recovery**
 - System enters a deadlocked state, detect it, and recover
 - **Example:** databases
 - **Do nothing** (Ostrich algorithm)^{***}
 - Ignore the problem and pretend deadlocks never occur
 - **Example:** most operating systems, including Linux and Windows

Deadlock Prevention

- Principle (Havender, 1968)
 - Ensure that at least one of the **necessary conditions** cannot hold
 - Mutual exclusion
 - Hold and wait
 - No preemption
 - Circular wait
 - Hence, deadlocks will be structurally impossible
- **Requires to change the way software is written**

Attacking the **Mutual Exclusion** Condition

- **Idea**
 - Avoid assigning a resource unless absolutely necessary
 - Try to make sure that as few processes as possible may actually claim the resource
- **Practically**
 - For read, don't assign resources to a single process
 - Make data read only
 - For write, use a mediator which serializes the write
 - Spooler example, the mediator is the spooler daemon
- **Problem**
 - Processes may deadlock filling up the mediator

Attacking the **Hold and Wait** Condition

- **Idea**

- Prevent processes that hold resources from waiting for more resources

- **Practically**

- Require all processes to request all their resources before “starting execution”
- If everything is available, the process will be allocated whatever it needs and can run to completion
- If one or more resources are busy, nothing will be allocated and the process will just wait
 - And then reallocate everything again

- **Problem**

- Processes don't know all their resources

Attacking the **No Preemption** Condition

- **Idea**

- If a process is holding some resources
- and request another resource that is not available,
- then all resources the process is currently holding are preempted (released)

- **Practically**

- Applied to resources whose state can be easily saved and restored later
 - CPU registers, memory space, etc.

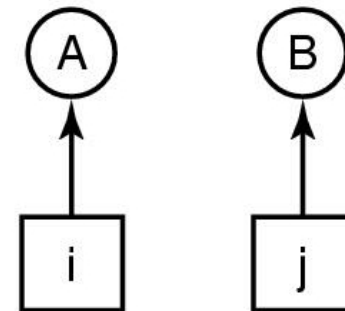
- **Problem**

- Not all resources can be virtualized

Attacking the **Circular Wait** Condition

- **Idea**
 - All requests of resources must be made in numerical order by processes
- **Practically**
 - See Figure
- **Problem**
 - May be impossible to find an ordering to satisfy everyone

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive



Deadlock Example

How to solve this with **Deadlock Prevention**

Semaphore muxA = 1 /* protects resource A */

Semaphore muxB = 1 /* protects resource B */

Process P:

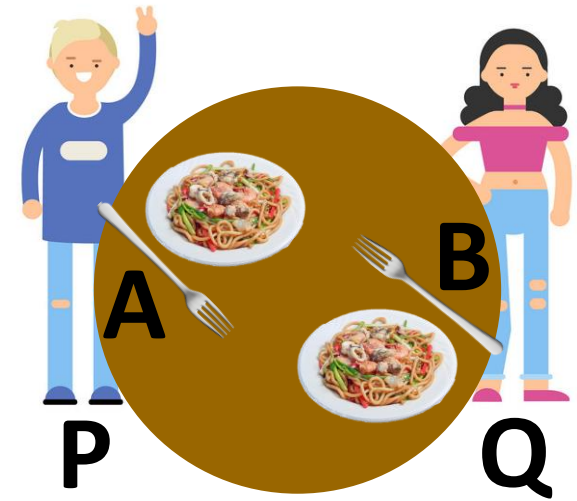
```
{  
  /* initial compute */  
  down(muxA)  
  down(muxB)  
  /* use both resources */  
  up(muxB)  
  up(muxA)  
}
```

Process Q:

```
{  
  /* initial compute */  
  down(muxB)  
  down(muxA)  
  /* use both resources */  
  up(muxB)  
  up(muxA)  
}
```

Process Q:

```
{  
  /* initial compute */  
  down(muxA)  
  down(muxB)  
  /* use both resources */  
  up(muxB)  
  up(muxA)  
}
```



Deadlock-free code!

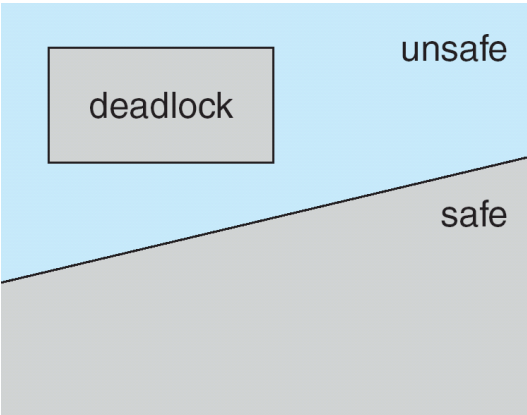
Deadlock Avoidance

- **The OS/runtime**
- Be given **additional information** in advance
 - What resources a thread/process will **request and use** during lifetime
 - Complete **sequence of requests and releases** of each thread/process
- **Decides for each request** if a thread/process waits, considering the **resource-allocation state**
 - **Currently available** resources
 - **Currently allocated** resources to each thread/process
 - **Future requests** and releases of each thread/process
- Algorithm dynamically examines resource-allocation state
 - For **circular-wait condition** existence

Safe State

- If the system **can allocate resources** to each thread/process (up to its maximum) **in some order** and still **avoid deadlocks**
- A system is in a safe state only if there exists a **safe sequence**
 - A sequence $\langle P_1, P_2, \dots, P_n \rangle$ of **ALL the processes in the systems**
 - For each P_i , the resources that P_i can still request can be satisfied by
 - Currently **available resources** plus
 - **Resources held** by all the P_j , with $j < i$
 - In this situation, if P_i resource needs **are not immediately available**
 - P_i can wait until all P_j have 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 States

- A **safe state** is not a **deadlocked state**
 - A **deadlocked state** is an **unsafe state**
 - Not all **unsafe states** are **deadlocks**
 - An **unsafe state** may lead to a **deadlock**
- 
- In a **safe state**
 - OS/runtime **can avoid unsafe (and deadlocked) states**
 - In an **unsafe state**
 - OS/runtime **cannot prevent** threads from requesting resources in such away that a **deadlock occurs**
 - Deadlock avoidance **algorithms**
 - **One instance** of each resource type: variation of the graph algorithm
 - **Multiple instances** of each resource type: Banker algorithm

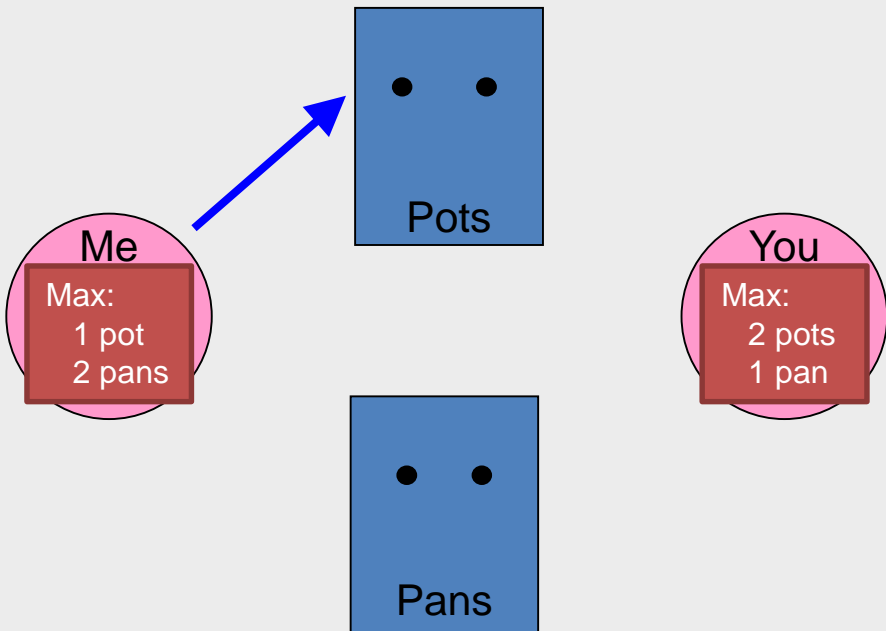
Banker's Algorithm

- Algorithm could be used in a **banking system**
 - To ensure the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers
- Background
 - Set of **controlled resources** is known to the system
 - **Number of units** of each resource is known to the system
 - Each application declares **maximum possible requirement** of each resource type
- When an **application requests** a set of resource
 - System determine whether the allocation of these resources will **leave the system in a safe state**
 - YES, the resources are allocated
 - NO, must wait until some other releases enough resources

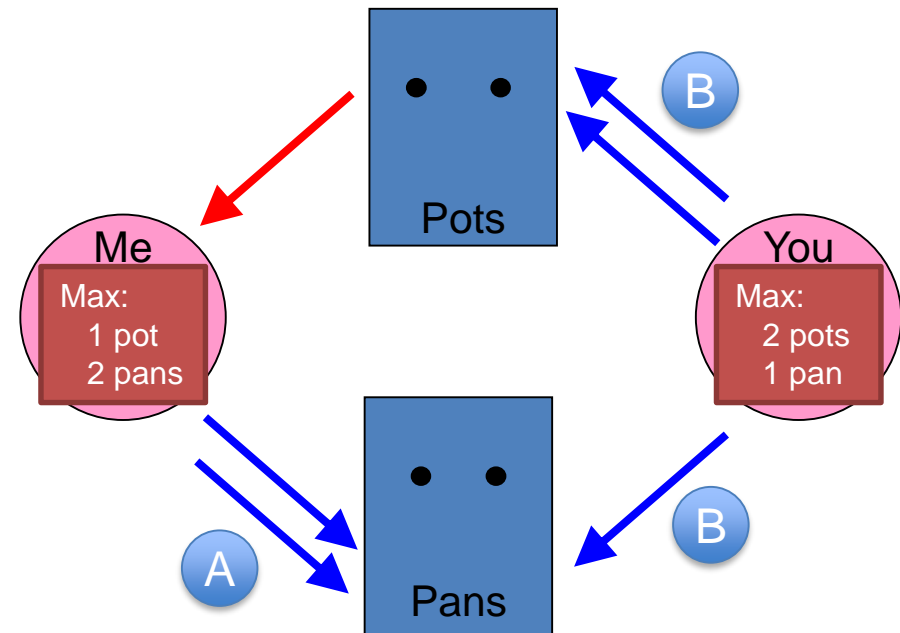
Banker's Example

Request 1

- I request a pot



- Suppose we allocate, then everyone requests its max



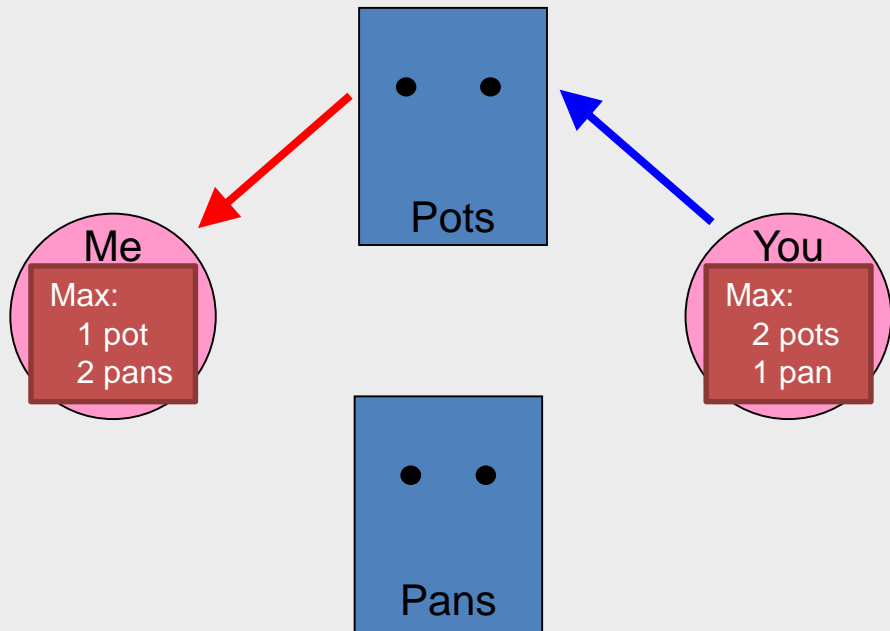
First I acquire all my resources and then you do the same

Safe state

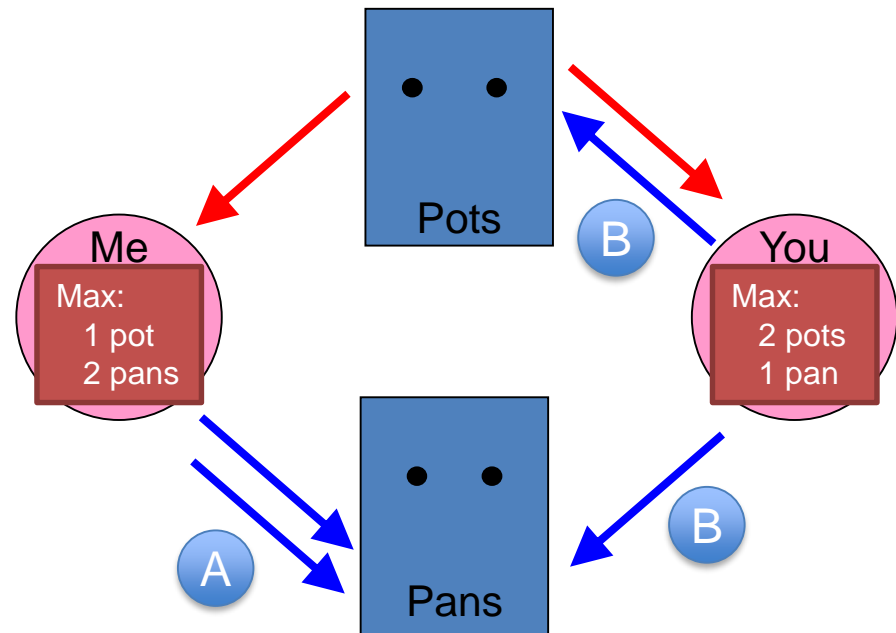
Banker's Example

Request 2

- You request a pot



- Suppose we allocate, then everyone requests its max



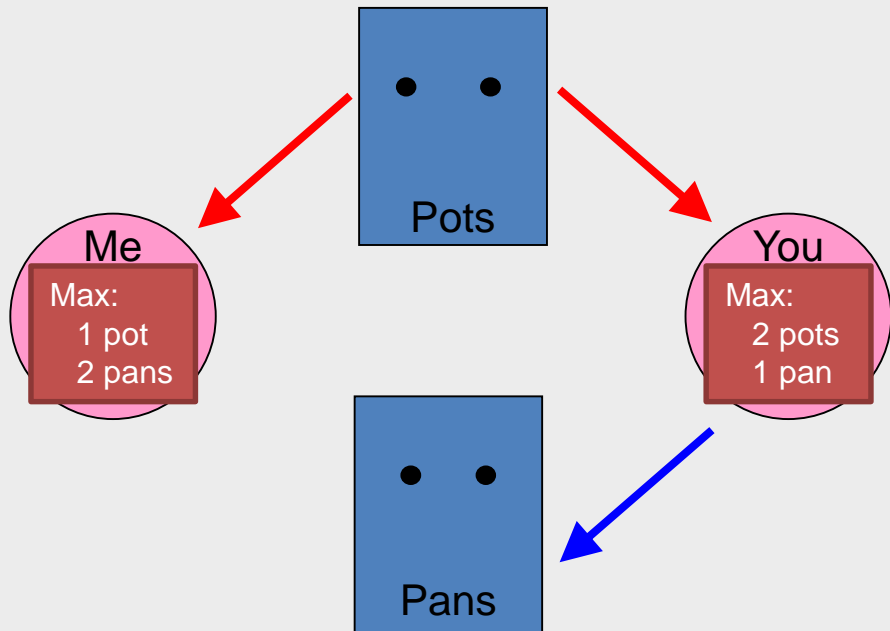
First I acquire all my resources and then you do the same

Safe state

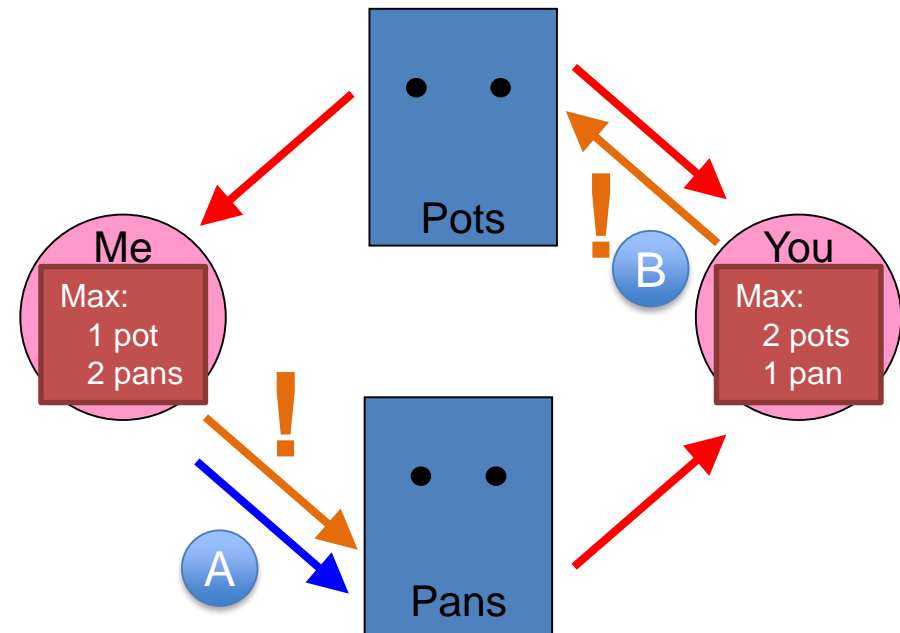
Banker's Example

Request 3a

- You request a pan



- Suppose we allocate, then everyone requests its max



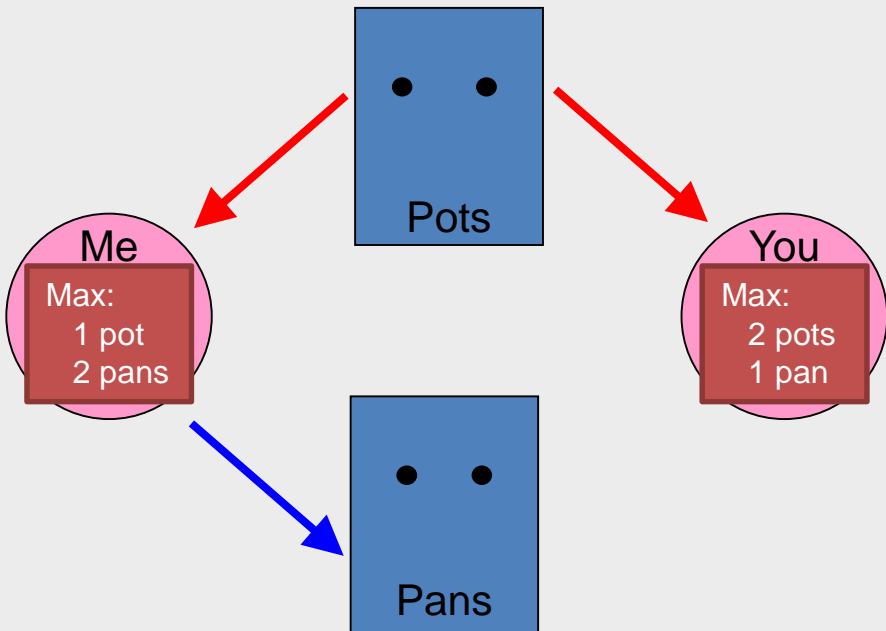
First I acquire all my resources and then you do the same (A) – not possible!
First you acquire all your resources and then I do the same (B) – not possible!

UNSAFE state

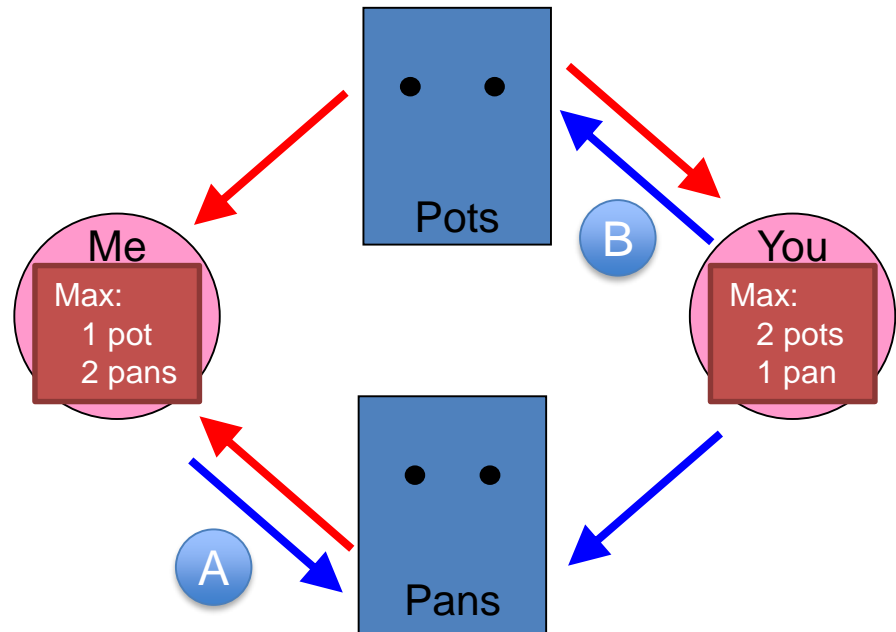
Banker's Example

Request 3b

- I request a pan



- Suppose we allocate, then everyone requests its max



First I acquire all my resources and then you do the same

Safe state

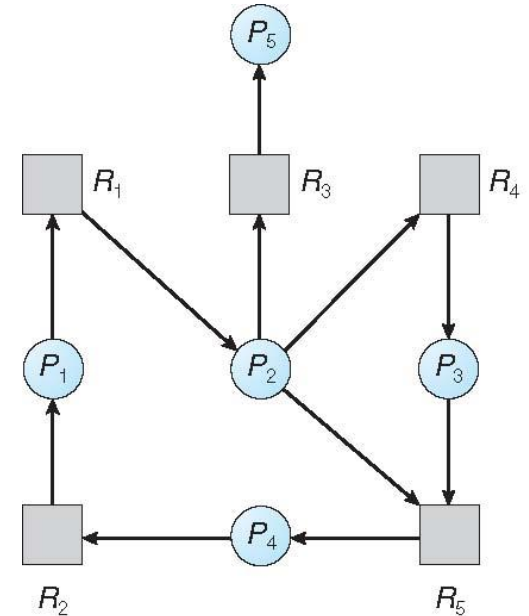
Deadlock Detection and Recovery

- Allow system to **enter a deadlock state**
- **Detection**
 - Algorithm that examines the state of the system to determine whether a deadlock has occurred
- **Recovery scheme**
 - Algorithm to recover from the deadlock

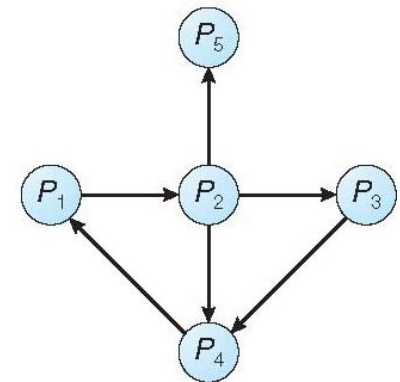
Detection

Single Instance of Each Resource Type

- Maintain **wait-for** graph
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- **Periodically searches** for cycles in the graph
 - A cycle implies a deadlock
- An algorithm to **perform search**
 - Requires an order of n^2 operations
 - n is the number of vertices in the graph
- **Doesn't apply** for multiple instances of each resource type
 - Variation of the banker algorithm



Resource allocation graph



Wait-for graph

Recovery from Deadlock

- **Process/Thread Termination**

- Abort **all** deadlocked processes
- Abort **one** process at a time until the deadlock cycle is eliminated
 - What process to select?

- May leave resources in an **incorrect state**

- **Resource Preemption**

- Successively preempt resources from thread/processes and give them to others until deadlock cycle is broken
 1. **Selecting a victim** – resource and process to preempt
 2. **Rollback** – return to some safe state, restart process for that state
 3. **Starvation** – avoid same process/thread as victim
- Instead of recovering the **operator can be informed**