Synchronization

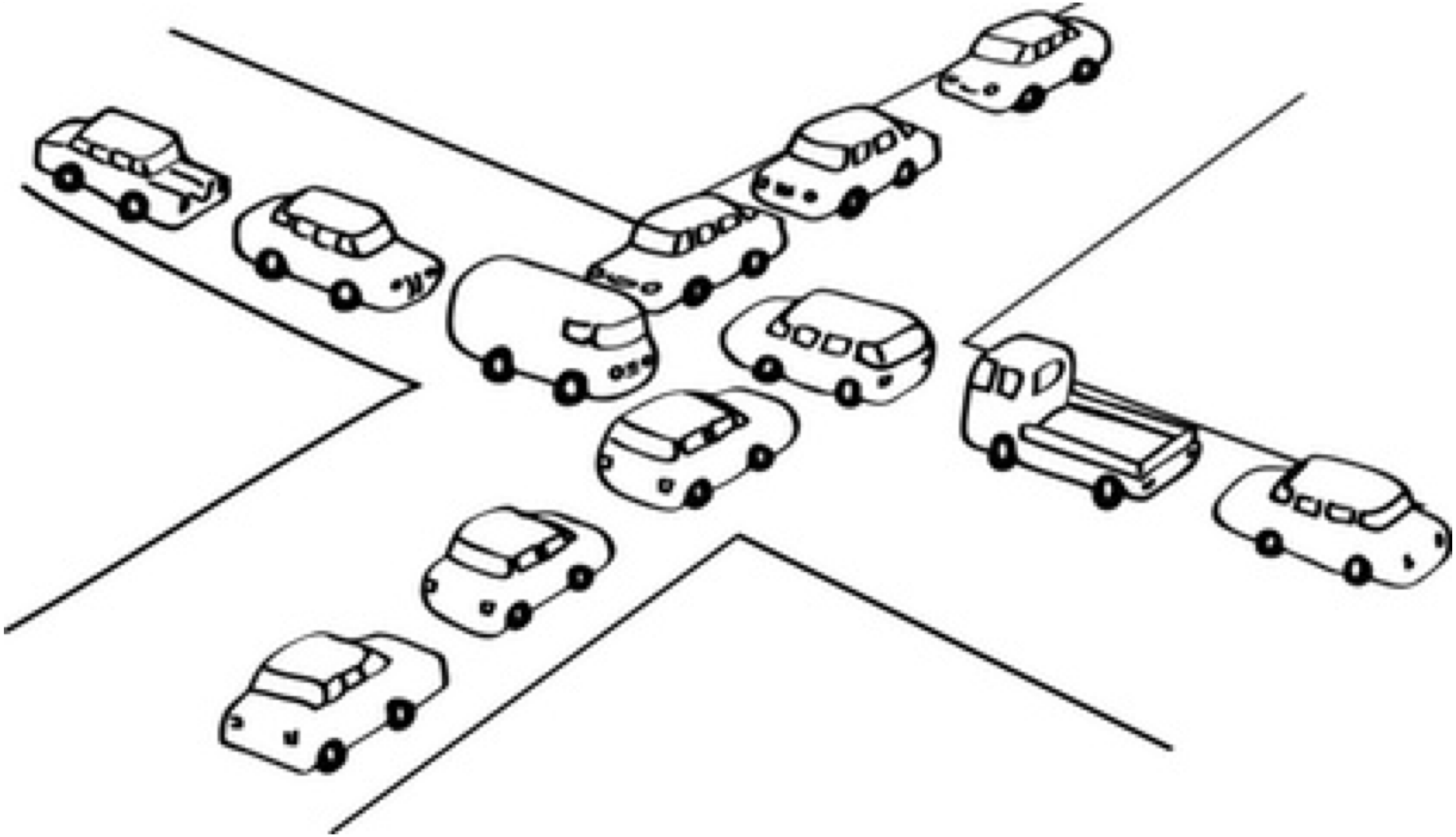
Deadlocks and prevention

Some of the slides are adapted from from Operating System Concepts (Silberschatz, Galvin, Gagne)**.**

Preemption - recall

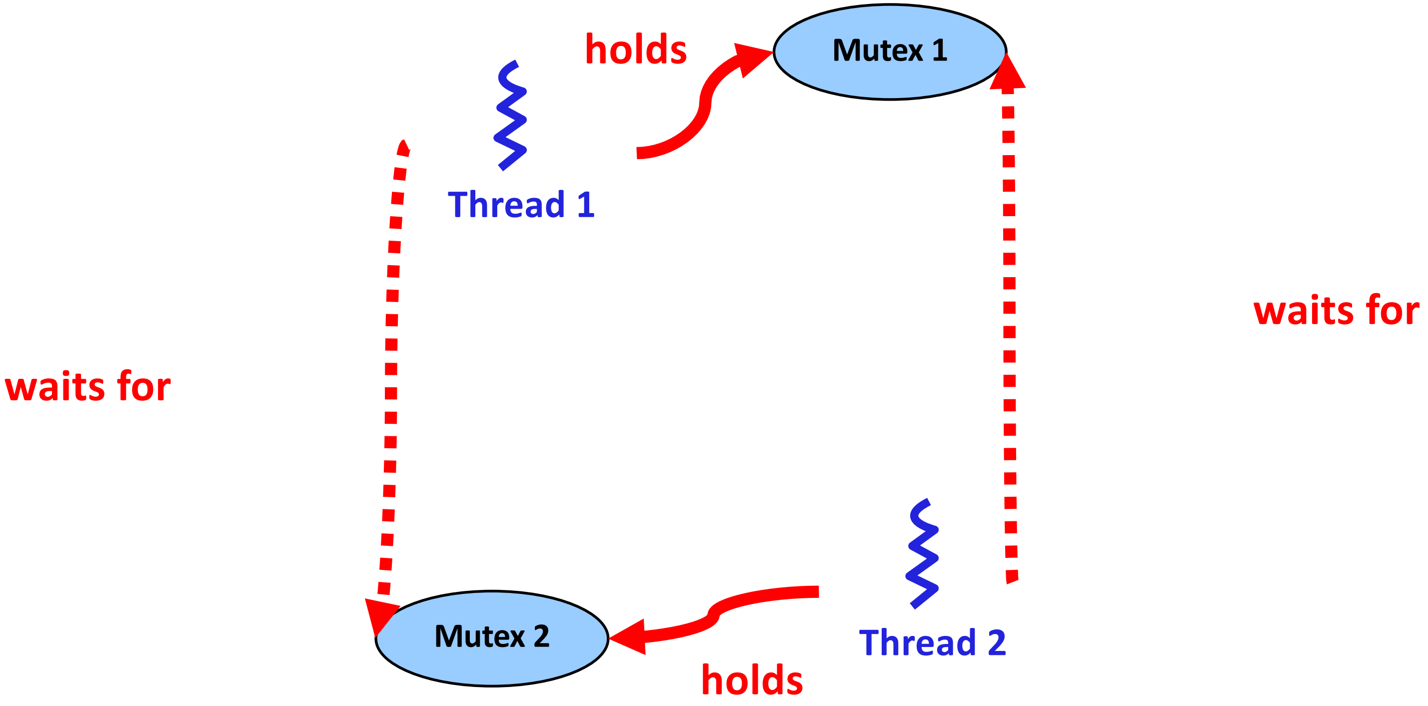
- **Preemption is to forcefully take a resource from a thread/process**
  - Resources can be CPU/lock/disk/network etc.
  - Resources can be
    - **Preemptible** (e.g. CPU)
    - **Non-preemptible** (e.g. mutex, lock, virtual memory region)
- **e.g. CPU is a preemptible resource**
  - A **preemptive OS can stop** a thread/process at any time
    - i.e. forcefully take the CPU from the current thread/process and give it to another.
  - A **non-preemptive OS can’t stop** a thread/process at any time
    - The OS has to wait for the current thread/process to yield (give away the CPU) voluntarily.
- **e.g. a lock is not a preemptible resource. The OS;**
    - cannot forcefully take away the lock and give it to another,
    - has to wait for the current thread/process to voluntarily release it.
  - **Why isn't it safe to forcibly take a lock away from a thread?**

What’s a deadlock?



Deadlock

- **A set of blocked threads/processes each holding a resource and waiting to acquire a resource held by another process in the set.**
- **A deadlock happens when**
  - Two (or more) threads waiting for each other
  - None of the deadlocked threads ever make progress



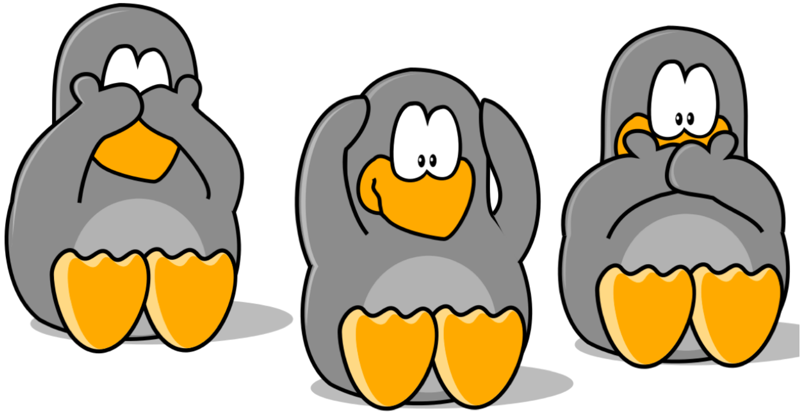
Starvation

- **A thread/process not making any progress since other threads/processes are using the resources that it needs.**
  - CPU as a resource: A thread/process not getting the CPU, since other the scheduler is giving the CPU to other “higher priority” thread/processes.
    - More on this in the upcoming lecture on scheduling.
  - Lock as a resource: : A thread/process not getting a lock that it has requested, since others have it.

- **Starvation ≠ Deadlock**
  - Deadlock => Starvation
  - Starvation ≠> Deadlock

Pedestrians who wants to cross Eskişehir Yolu are likely to “starve” due to traffic!

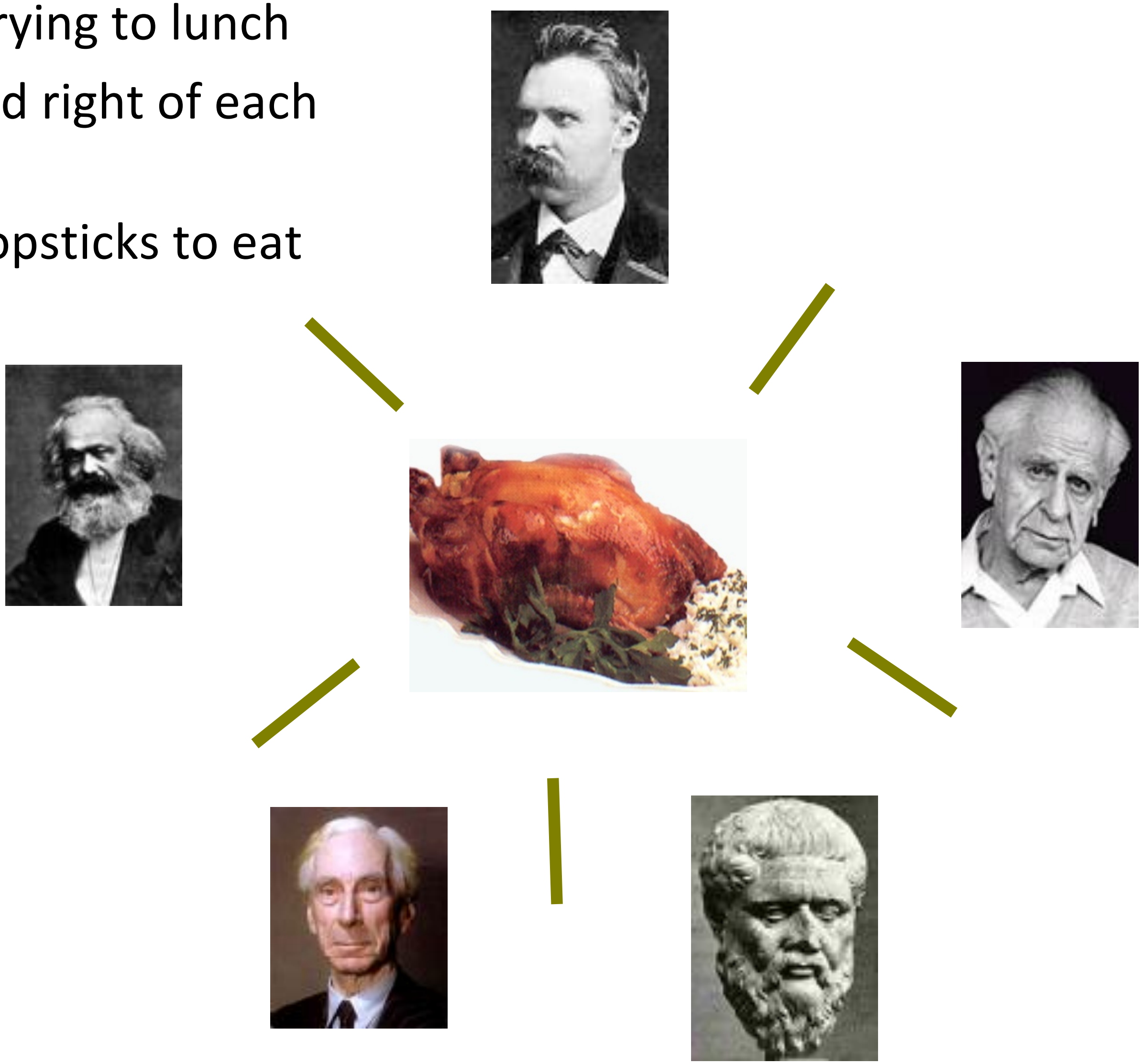
Methods for Handling Deadlocks



- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

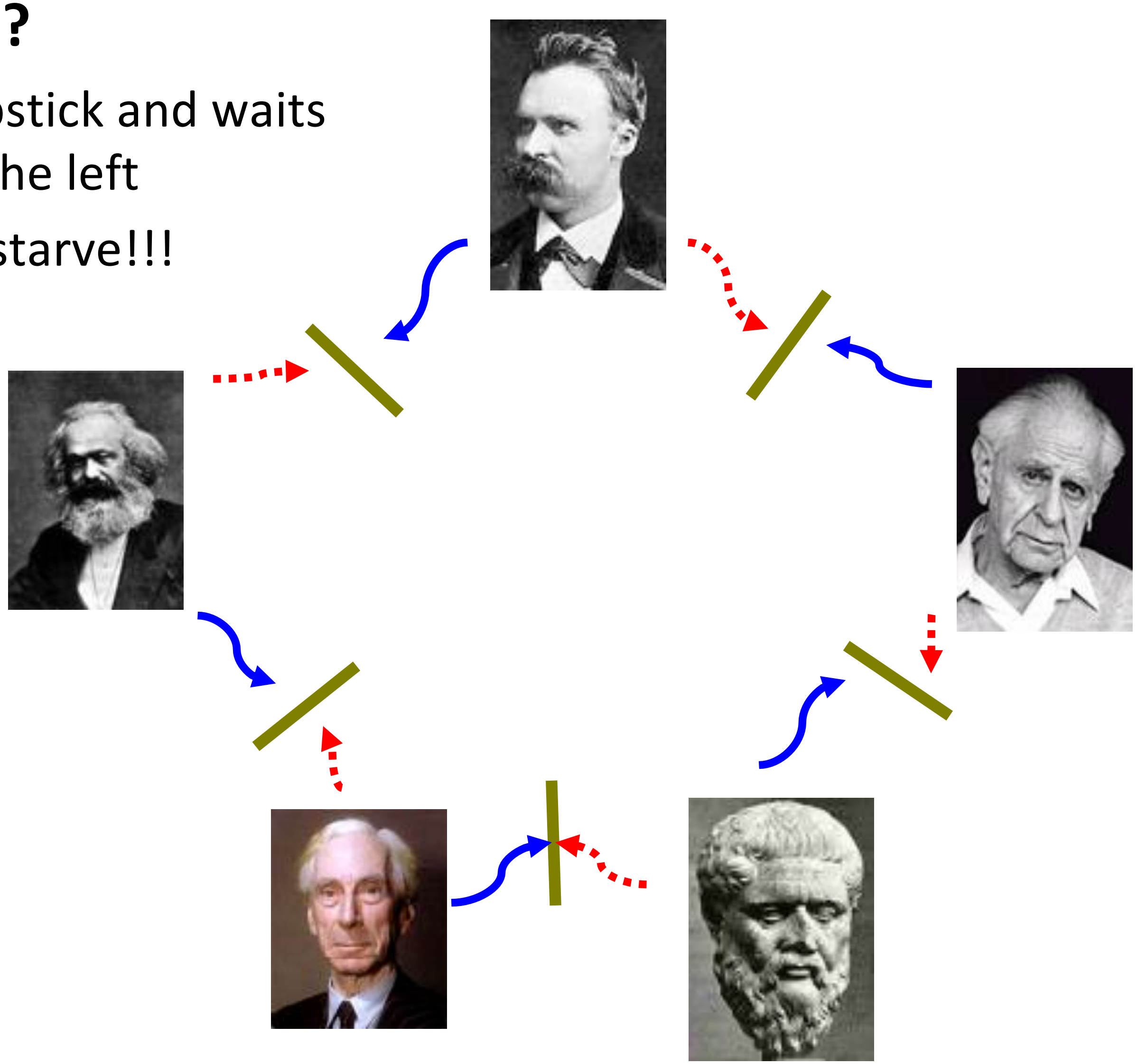
Dining Philosophers

- **Classic deadlock problem**
  - Multiple philosophers trying to lunch
  - One chopstick to left and right of each philosopher
  - Each one needs two chopsticks to eat



Dining Philosophers

- **What happens if everyone grabs the chopstick to their right?**
  - Everyone gets one chopstick and waits forever for the one on the left
  - All of the philosophers starve!!!



Deadlock Characterization

- **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 can be 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_0} 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.

Deadlock can arise if all four conditions hold simultaneously!

Deadlock Prevention

- **Ensure that at least one of the four conditions do not hold!**
- **Mutual Exclusion**
    - not required for sharable resources;
    - must hold for non-sharable resources (e.g. a printer).
- **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,
    - Allow process to request resources only when the process has none.
        - low resource utilization;
        - starvation possible.

Deadlock Prevention (Cont.)

- **No 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.
  - Can be applied to resources whose state can be saved such as CPU, and memory. Not applicable to resources such as printer and tape drives.

- **Circular Wait**
  - impose a total ordering of all resource types, and
  - require that each process requests resources in an increasing order of enumeration.

Circular Wait - 1

- **Each resource is given an ordering:**
  - F(tape drive) = 1
  - F(disk drive) = 2
  - F(printer) = 3
  - F(mutex1) = 4
  - F(mutex2) = 5
  - …….
- **Each process can request resources only in increasing order of enumeration.**
- **A process which decides to request an instance of Rj should first release all of its resources that are F(Ri) >= F(Rj).**

Circular Wait - 2

- **For instance an application program may use ordering among all of its synchronization primitives:**
  - F(semaphore1) = 1
  - F(semaphore2) = 2
  - F(semaphore3) = 3
  - …….
- **After this, all requests to synchronization primitives should be made only in the increasing order:**
  - Correct use:
    - down(semaphore1);
    - down(semaphore2);
  - Incorrect use:
    - down(semaphore3);
    - down(semaphore2);
- **Keep in mind that it’s the application programmer’s responsibility to obey this order.**

Dining Philosophers

- **How do we solve this problem??**
  - (Apart from letting them eat with forks.)

How to solve this problem?

- **Solution 1: Don't wait for chopsticks**
  - Grab the chopstick on your right
  - Try to grab chopstick on your left
  - If you can't grab it, put the other one back down
  - Breaks "no preemption" condition – no waiting!
- **Solution 2: Grab both chopsticks at once**
  - Requires some kind of extra synchronization to make it atomic
  - Breaks "multiple independent requests" condition!
- **Solution 3: Grab chopsticks in a globally defined order**
  - Number chopsticks 0, 1, 2, 3, 4
  - Grab lower-numbered chopstick first
    - Means one person grabs left hand rather than right hand first!
  - Breaks "circular dependency" condition
- **Solution 4: Detect the deadlock condition and break out of it**
  - Scan the waiting graph and look for cycles
  - Shoot one of the threads to break the cycle

Deadlock Avoidance

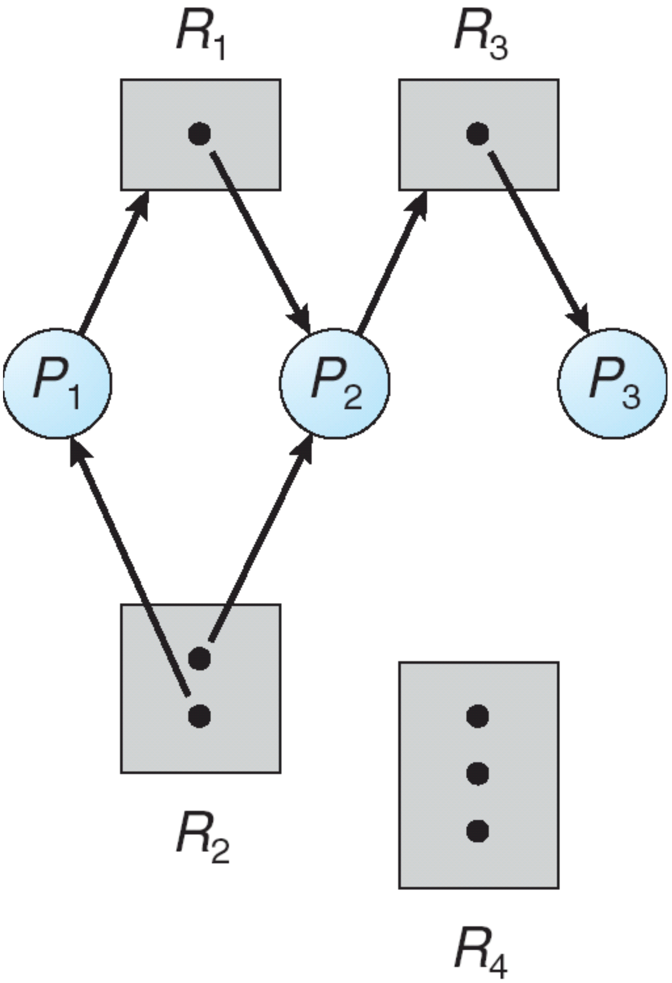
- **Requires that the system has some additional a priori information available.**
  - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
    - Is this possible at all?
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
    - When should the algorithm be called?
  - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

System Model

- **Resource types R_1, R_2, . . ., R_m**
  - CPU,
  - memory,
  - I/O devices
    - disk
    - network
- **Each resource type R_i has W_i instances.**
  - For instance a quad-core processor has
    - 4 CPUs
- **Each process utilizes a resource as follows:**
  - request
  - use
  - release

Resource-Allocation Graph

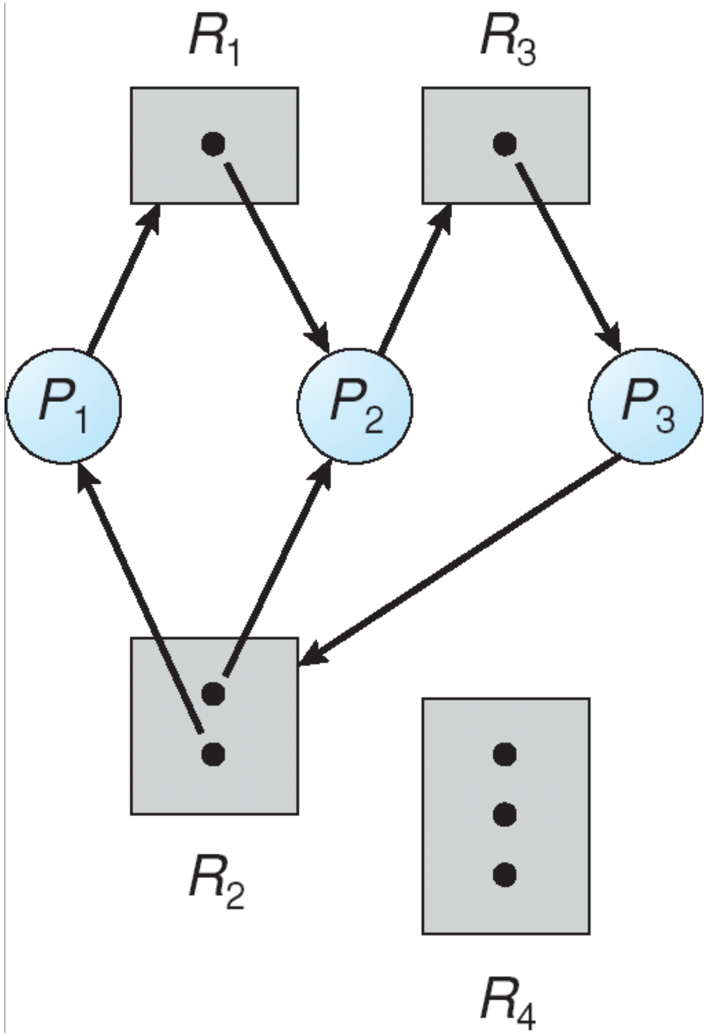
- **A set of vertices *V* and a set of edges *E*.**
- **V is partitioned into two types:**
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- **request edge – directed edge $P_1 \text{–>} R_j$**
- **assignment edge – directed edge $R_j \text{<–} P_i$**



Resource Allocation Graph With A Deadlock

- **If there is a deadlock**
  - => there is a cycle in the graph.

- **However the reverse is not true!**
- **If there is a cycle in the graph**
  - =/> there is a deadlock

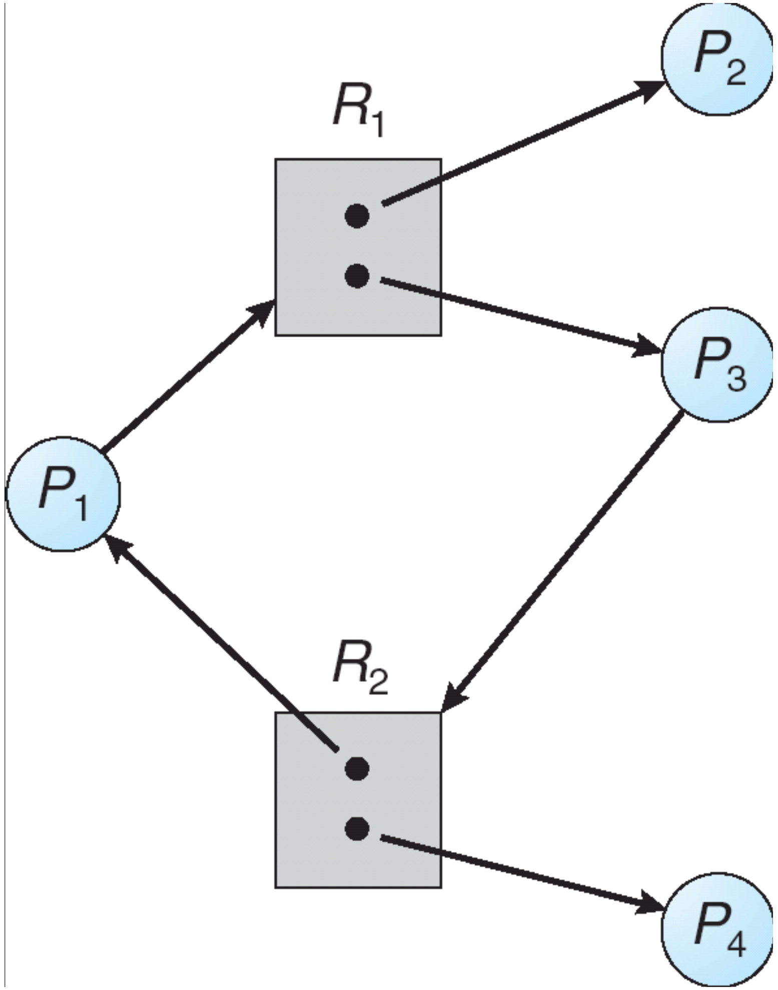


Resource Allocation Graph With A Cycle But No Deadlock

- **However the existence of a cycle in the graph does not necessarily imply a deadlock.**

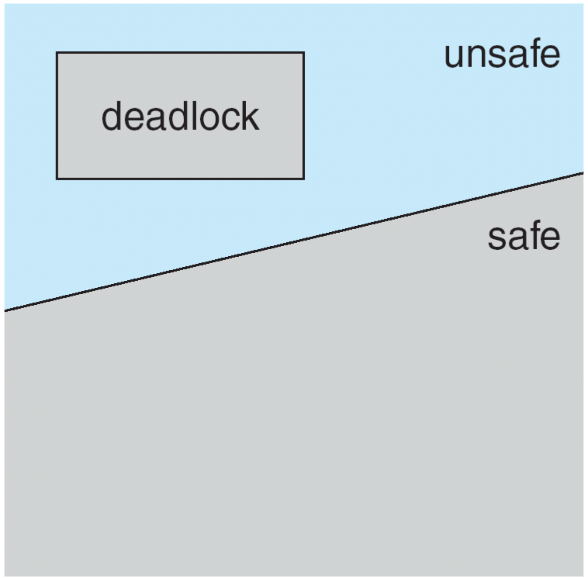
Overall message:

- **If graph contains no cycles =>**
  - no deadlock.
- **If graph contains a cycle =>**
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.



Safe, unsafe and deadlock states

- 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.



Safe State

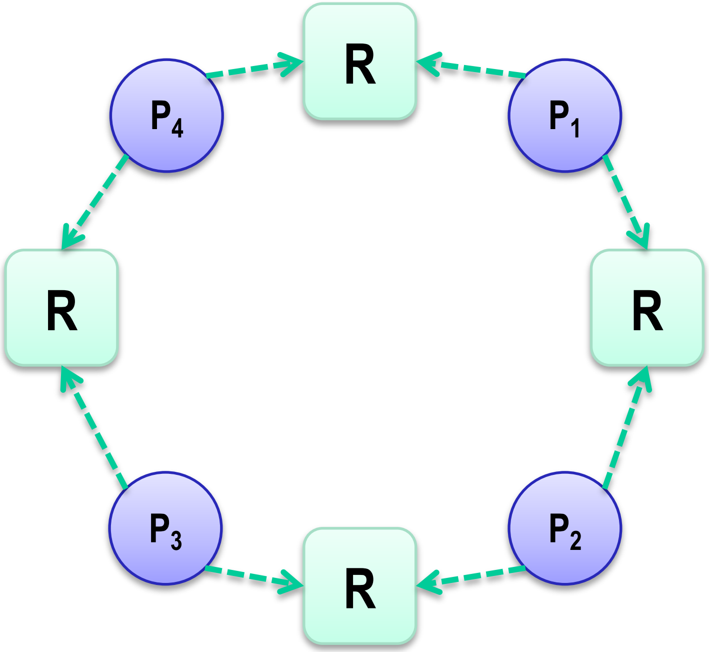
- **When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.**

- **System is in safe state if there exists a safe sequence of all processes.**

- **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.

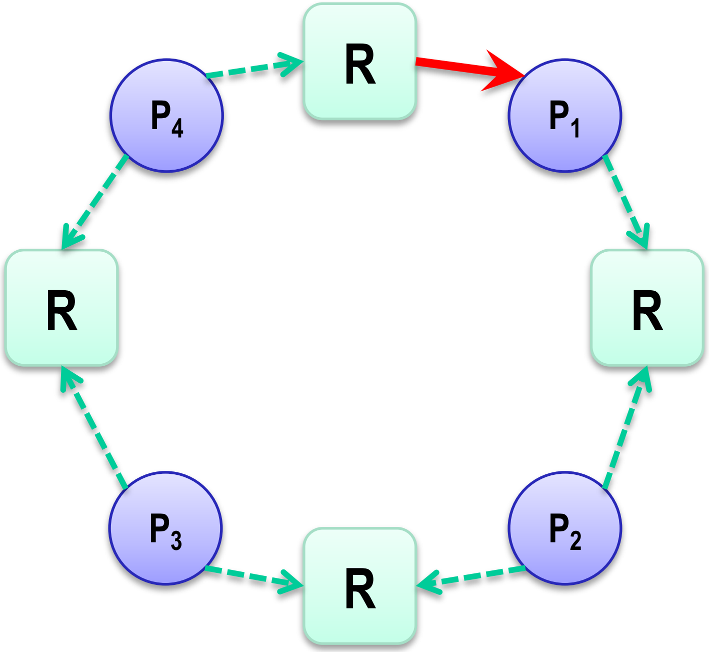
Resource Allocation Graph: Dining Philosopher's example - 1

- **Initial configuration:**
  - 4 philosophers
  - 4 sticks.



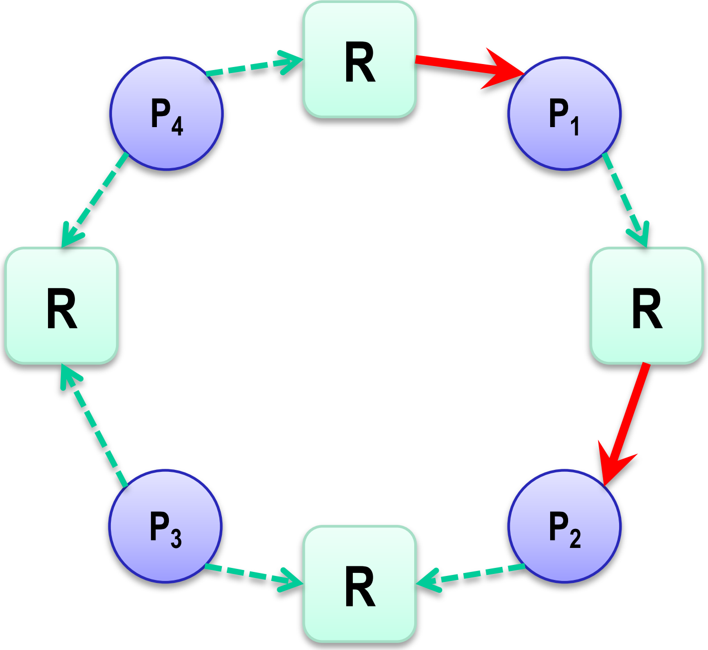
Resource Allocation Graph: Dining Philosopher's example - 2

- P_1 gets right stick



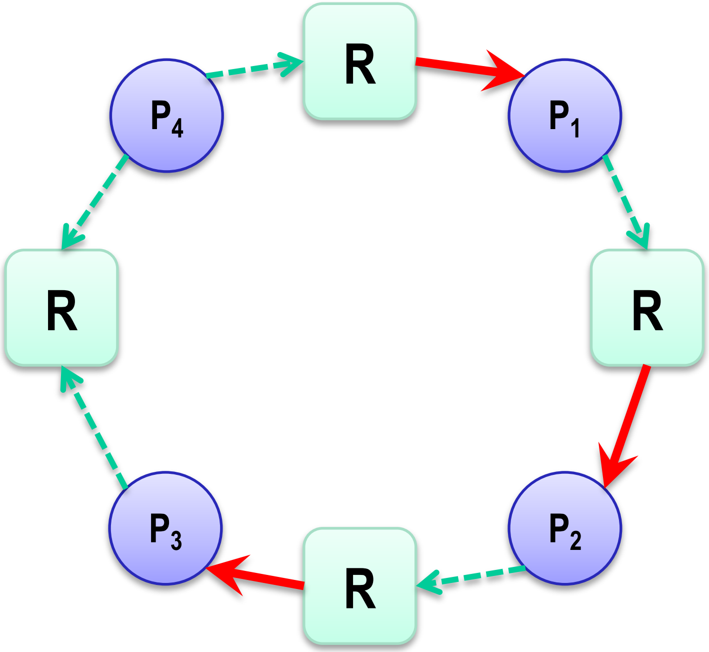
Resource Allocation Graph: Dining Philosopher's example - 3

- P_2 gets right stick



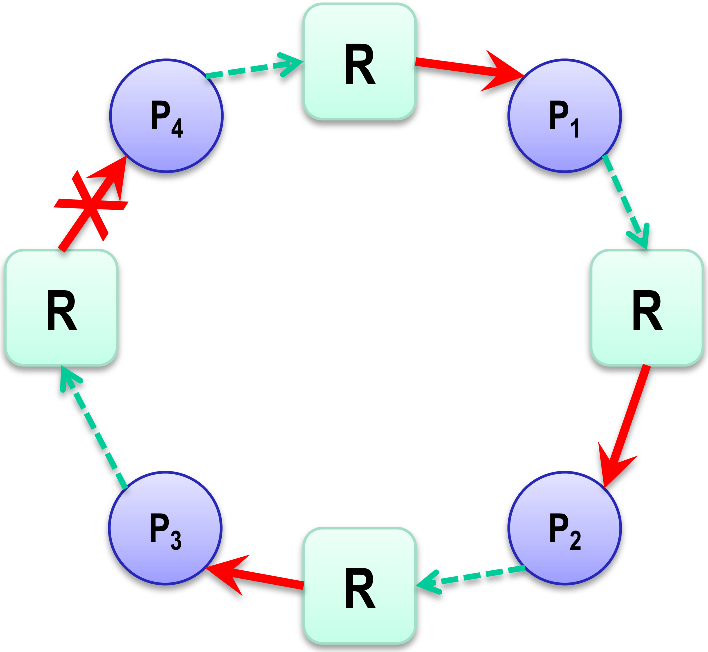
Resource Allocation Graph: Dining Philosopher's example - 4

- P_3 gets right stick



Resource Allocation Graph: Dining Philosopher's example – 5

- P_4 requests right stick.
  - Cycle!!
  - Rejected.



Monitors: Finite resource problem

- 5 instances of a resource
- N processes.
- Only 5 processes can use the resources simultaneously.

Process code

```
Allocate MA; //resource allocation monitor.
….
MA.acquire();
// use the resource
MA.release();
....
```

Monitor code

```
Monitor Allocate
{
     int count=5;
     condition c;

     void acquire(){
          if (count == 0)
               c.wait();
          count--;
     }
     void release(){
          count++;
          c.signal(); //i.e. notify()
     }
}
```

Monitors: Dining Philosophers

```
#define LEFT  (i+4)%5
#define RIGHT (i+1)%5
```

```
Monitor DiningPhilosophers
{
     enum{THINKING,
          HUNGRY,
          EATING
     }state[5];
     condition cond[5];

     void pickup(int i){
          state[i] =  HUNGRY;
          test(i);
          if (state[i] != EATING)
               cond[i].wait();
     }

     void putdown(int i){
          state[i]=THINKING;
          // test left and right neighbor
          test(LEFT);
          test(RIGHT);
     }
```

```
     void test(int i){
          if( (state[LEFT] != EATING) &&
          (state[RIGHT] != EATING) &&
          (state[i] == HUNGRY))
          {
               state[i] =  EATING;
               cond[i].signal();
          }
     }

     void initialize(){
          for (int i=0l i<5; i++)
               state[i] = THINKING;
     }

} // end Monitor
```

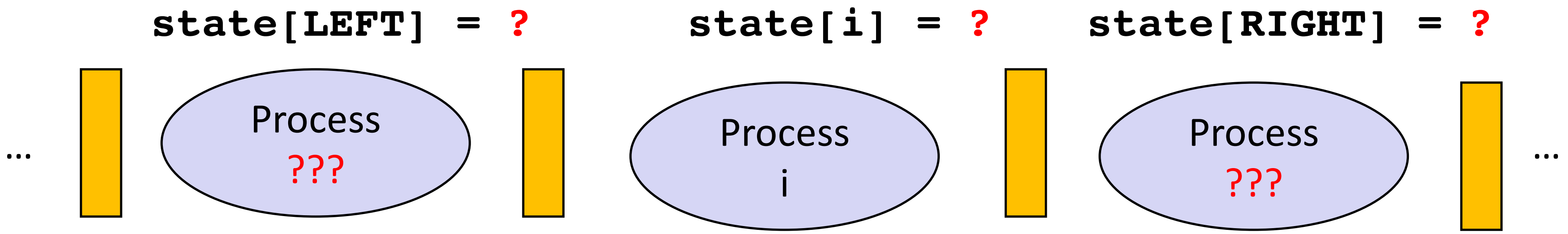
```
DiningPhilosopher DP;
…
while(1){
     // THINK..
     DP.pickup(i);
     // EAT (use resources)
     DP.putdown(i);
     // THINK..
}
```

Monitors: Dining Philosophers

- **What are the ID’s to access neighbor philosophers?**

```
#define LEFT ???
#define RIGHT ???
```

```
state=
      THINKING?
      HUNGRY?
      EATING?
```



`state[LEFT] = ?` `state[i] = ?` `state[RIGHT] = ?`

… Process ??? Process i Process ??? …

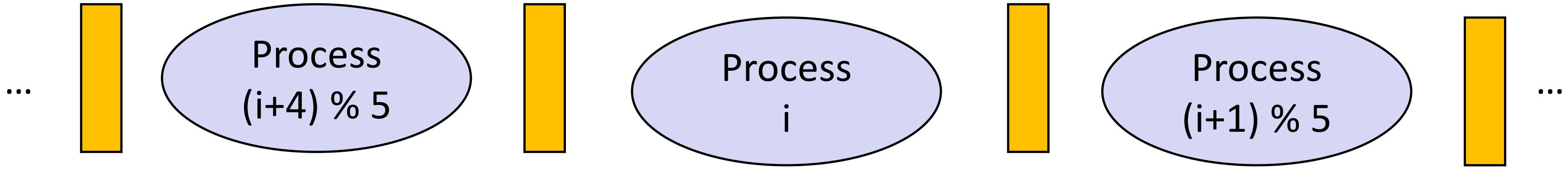
Monitors: Dining Philosophers

- **What are the ID’s to access neighbor philosophers?**

```
#define LEFT (i+4)%5
#define RIGHT (i+1)%5
```

```
state=
     THINKING?
     HUNGRY?
     EATING?
```

`state[LEFT] = ?` `state[i] = ?` `state[RIGHT] = ?`



`test((i+4)%5)` `test(i)` `test((i+1)%5)`

Banker’s Algorithm

- **Suppose “worst case/maximum” resource needs of each process is known in advance**
  - E.g. limit on your credit card
- **Observation: If we give a process the maximum of its resources**
  - Then it will execute to complete
  - After that it will give back all the resources
- **When a process request a new resource during its execution**
  - The OS decides whether to give it the resource at that time or not
- **A request is delayed if there does not exist a sequence of processes that would ensure the successful completion of all the processes, even if they need the “maximum” of their resources.**

Why Banker’s Algorithm? While giving credits, a banker should ensure that it never allocates all of its cash in such a way that none of its creditors can finish their work and pay back the loan.

Banker's algorithm - example – 1

- **System:**
  - 5 processes P1-P5
  - 3 resource types: A (10), B (5), C(7)

Maximum			
	A	B	C
P1	7	5	3
P2	3	2	2
P3	9	0	2
P4	2	2	2
P5	4	3	3

Banker's algorithm - example - 2

- **System:**
  - 5 processes P1-P5
  - 3 resource types: A (10), B (5), C(7)
- **System state at t0**

	Allocated			**Max**			**Needs**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P1	0	1	0	7	5	3	7	4	3
P2	2	0	0	3	2	2	1	2	2
P3	3	0	2	9	0	2	6	0	0
P4	2	1	1	2	2	2	0	1	1
P5	0	0	2	4	3	3	4	3	1

Available		
A	**B**	**C**
3	3	2

The system is in a safe state since the sequence <P2,P4,P5,P3,P1> would guarantee the completion of all processes.

Banker's algorithm - example - 3

- **P2 request (1,0,2)**
- **Check that request <= Available**
  - (1,0,2) <= (3,3,2)
- **Look for a safe sequence:**
  - <P2,P4,P5,P1,P3> is possible!

	Allocated			**Max**			**Needs**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P1	0	1	0	7	5	3	7	4	3
P2	3	0	2	3	2	2	1	2	2
P3	3	0	2	9	0	2	6	0	0
P4	2	1	1	2	2	2	0	1	1
P5	0	0	2	4	3	3	4	3	1

Available		
A	**B**	**C**
2	3	0

Banker's algorithm - example - 4

- **P5 requests (3,3,0)**
- **Check that request <= Available**
  - ???
- **Look for a safe sequence:**
  - ???

	Allocated			**Max**			**Needs**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P1	0	1	0	7	5	3	7	4	3
P2	3	0	2	3	2	2	1	2	2
P3	3	0	2	9	0	2	6	0	0
P4	2	1	1	2	2	2	0	1	1
P5	0	0	2	4	3	3	4	3	1

Available		
A	**B**	**C**
2	3	0

Banker's algorithm - example - 5

- **P1 requests (0,2,0)**
- **Check that request <= Available**
  - ???
- **Look for a safe sequence:**
  - ???

	Allocated			**Max**			**Needs**		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
P1	0	1	0	7	5	3	7	4	3
P2	3	0	2	3	2	2	1	2	2
P3	3	0	2	9	0	2	6	0	0
P4	2	1	1	2	2	2	0	1	1
P5	0	0	2	4	3	3	4	3	1

Available		
A	**B**	**C**
2	3	0

Banker’s Algorithm – Dining Philosophers - 1

- **System resources:**
  - 1 chopsticks in 5 positions
  - Total resources: (1, 1, 1, 1, 1)
- **5 “philosopher” processes**
- **Maximum resources table is:**

Maximum					
	C1	C2	C3	C4	C5
P1	1	1	0	0	0
P2	0	1	1	0	0
P3	0	0	1	1	0
P4	0	0	0	1	1
P5	1	0	0	0	1

Banker’s Algorithm – Dining Philosophers - 2

- **Safe state:**

	Allocated					Needs				
	C1	**C2**	**C3**	**C4**	**C5**	**C1**	**C2**	**C3**	**C4**	**C5**
P1	1	0	0	0	0	0	1	0	0	0
P2	0	1	0	0	0	0	0	1	0	0
P3	0	0	1	0	0	0	0	0	1	0
P4	0	0	0	1	0	0	0	0	0	1
P5	0	0	0	0	0	1	0	0	0	1

Available				
C1	**C2**	**C3**	**C4**	**C5**
0	0	0	0	1

- **<P4, P3, P2, P1, P5> is feasible.**

Banker’s Algorithm – Dining Philosophers - 3

- P5 is given the C5

	Allocated					Needs				
	C1	**C2**	**C3**	**C4**	**C5**	**C1**	**C2**	**C3**	**C4**	**C5**
P1	1	0	0	0	0	0	1	0	0	0
P2	0	1	0	0	0	0	0	1	0	0
P3	0	0	1	0	0	0	0	0	1	0
P4	0	0	0	1	0	0	0	0	0	1
P5	0	0	0	0	1	1	0	0	0	0

Available				
C1	**C2**	**C3**	**C4**	**C5**
0	0	0	0	0

- None of the processes get their need.
- Unsafe. Rejected.