Operating Systems
BCA –IV th Sem

Deadlocks

By
Abhilasha Pandey

Text @ OS notes book

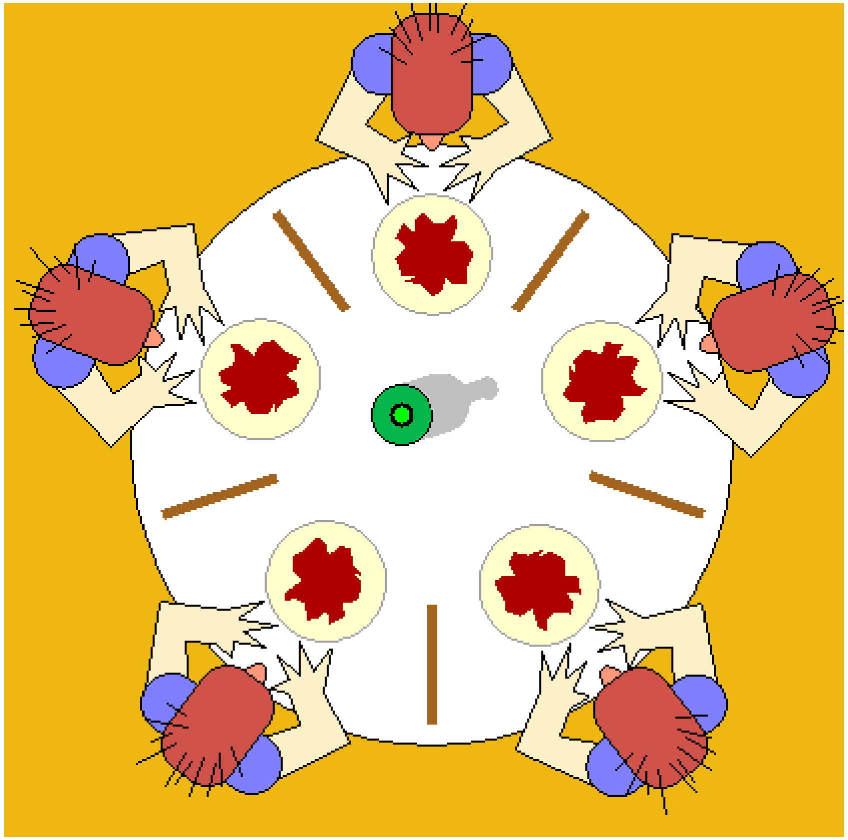
- **Much of the material appears in Section 3.2 in Feitelson's OS notes book**
  - Literature section in course homepage
- **In case this presentation and book conflicts**
  - As always, this presentation wins

Intro

- **In the previous lecture**
  - We've talked about how to synchronize access to shared resources
- **When synchronizing, if we're not careful**
  - Our system might enter a deadlock state
- **The popular formal CS definition of a dealock**
  - "A set of processes is deadlocked if each process in the set is waiting for an event that only a process in the set can cause"
- **Typically associated with synch-ing the use of resources**
  - Let's revise the definition accordingly
  - A set of processes is deadlocked if each process in the set is waiting for a resource held by another process in the set
- **"The dining philosophers problem"**
  - The canonical example in introductory OS lectures to demonstrate deadlocks

Dining philosophers – rules

- Five philosophers are sitting around a large round table, each with a bowl of Chinese food in front of him
- Between periods of meditation, they may start eating whenever they want to, with their bowls being filled frequently
- But there are only five chopsticks available, one between every pair of bowls -- and for eating Chinese food, one needs two chopsticks...
- When a philosopher wants to start eating, he must pick up the chopstick to the left of his bowl and the chopstick to the right of his bowl

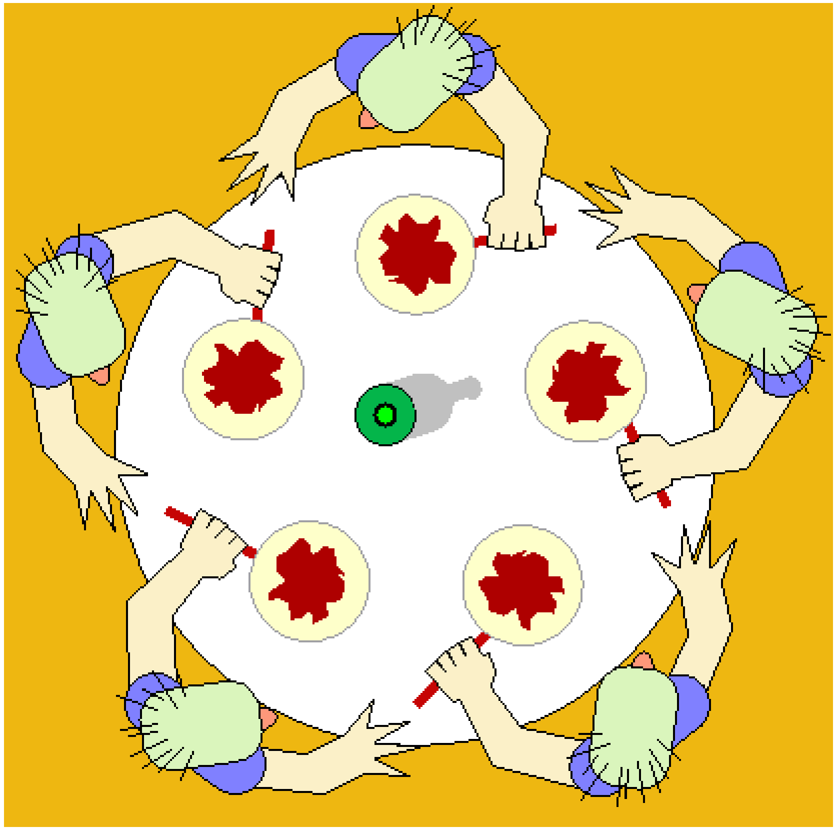


Dining philosophers – naive solution

- **Semaphore for each fork**
  - semaphore_t fork[5]
  - (What it forks were place in the middle of the table and any philosopher would be able to grab any fork? Would we still need 5 semaphores?)
- **Naive (faulty) algorithm**
  - philosopher(i):
    while(1) do…
    - thinking for a while
    - wait(fork[i])
    - wait(fork[(i+1) % 5])
    - eat
    - signal(fork[(i+1) % 5])
    - signal(fork[i])

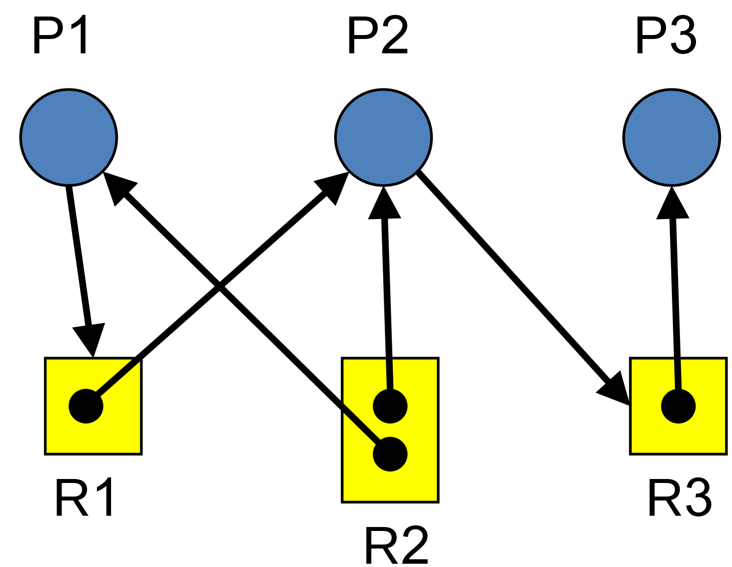
Dining philosophers – problem

- All the philosophers become hungry at the exact same time
- They simultaneously pick up the chopstick to their left
- They then all try to pick up the chopstick to their right
- Only to find that those chopsticks have already been picked up (by the philosopher on their right)
- The philosophers then continue to sit there indefinitely, each holding onto one fork, glaring at his neighbor angrily
- They are deadlocked



Resource allocation graph

- **When considering resource management**
  - Convenient to represent system state with a directed graph
- **2 types of nodes**
  - Process = round node
  - Resource type = square node
    - Within resource, each instance = a dot
- **2 types of edges**
  - Request = edge from process to resource type
  - Allocation = edge from resource instance to a process



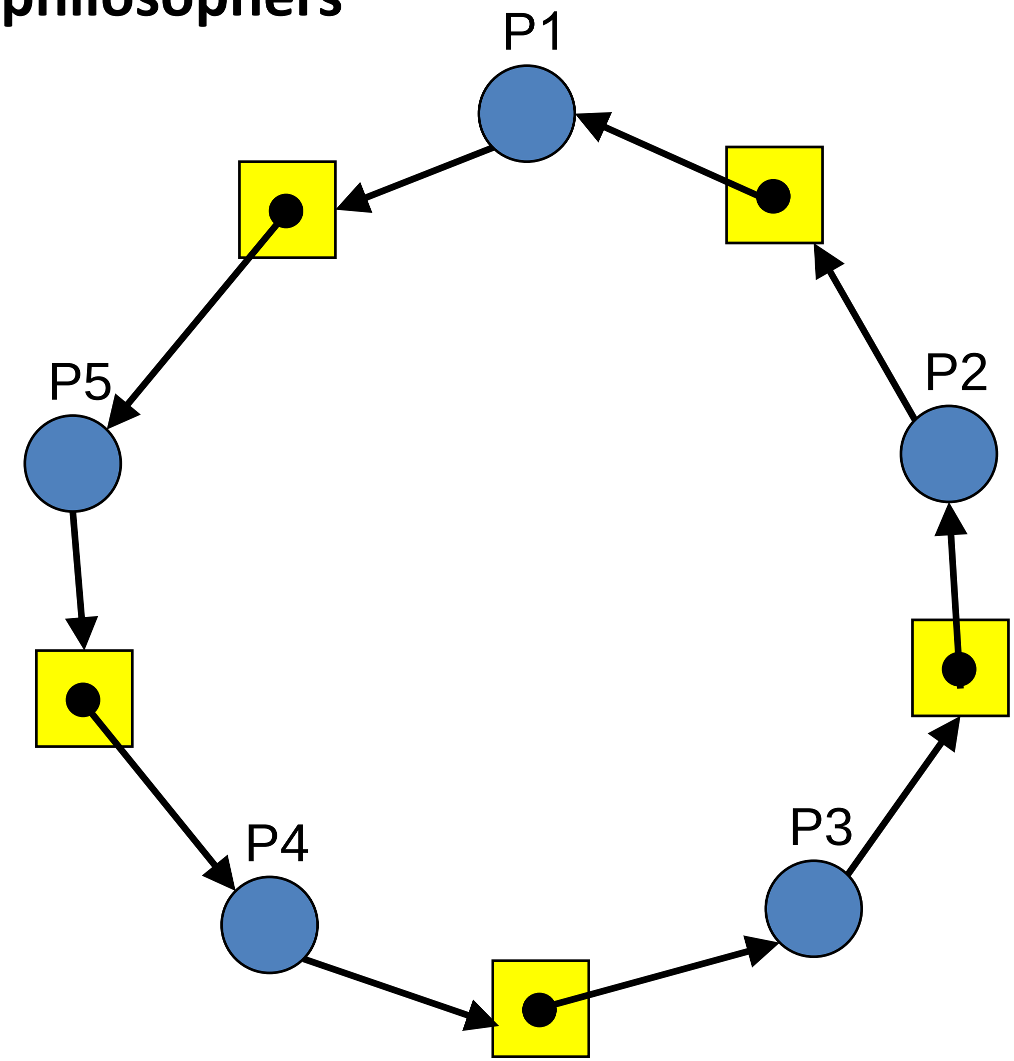
P1	Holds instance of R2. Waits for R1.
P2	Holds instances of R1 & R2. Waits for R3.
P3	Holds instance of R3.

Resource allocation graph

- **Examples of resources of which there's a**
  - Single instance?
  - Multiple instances?
- **Assume we have n printers attached to a computer**
  - Do we need n instances of the same generic printer type?
  - Or n separate printer types?
  - Or something in between?

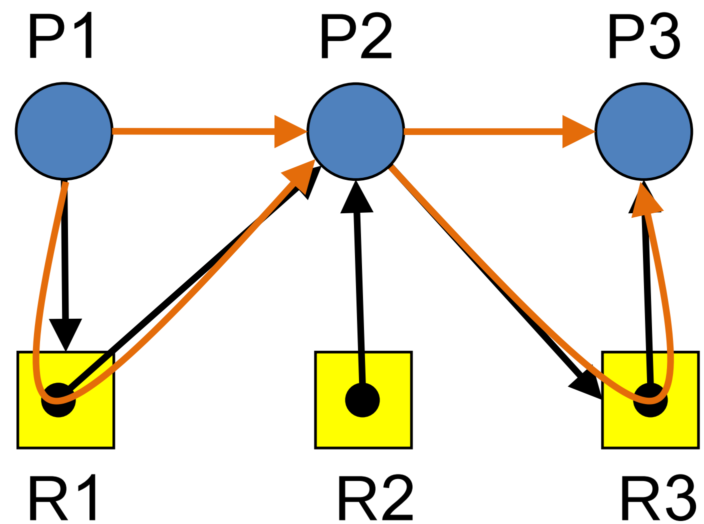
Resource allocation graph

- **Dining philosophers**



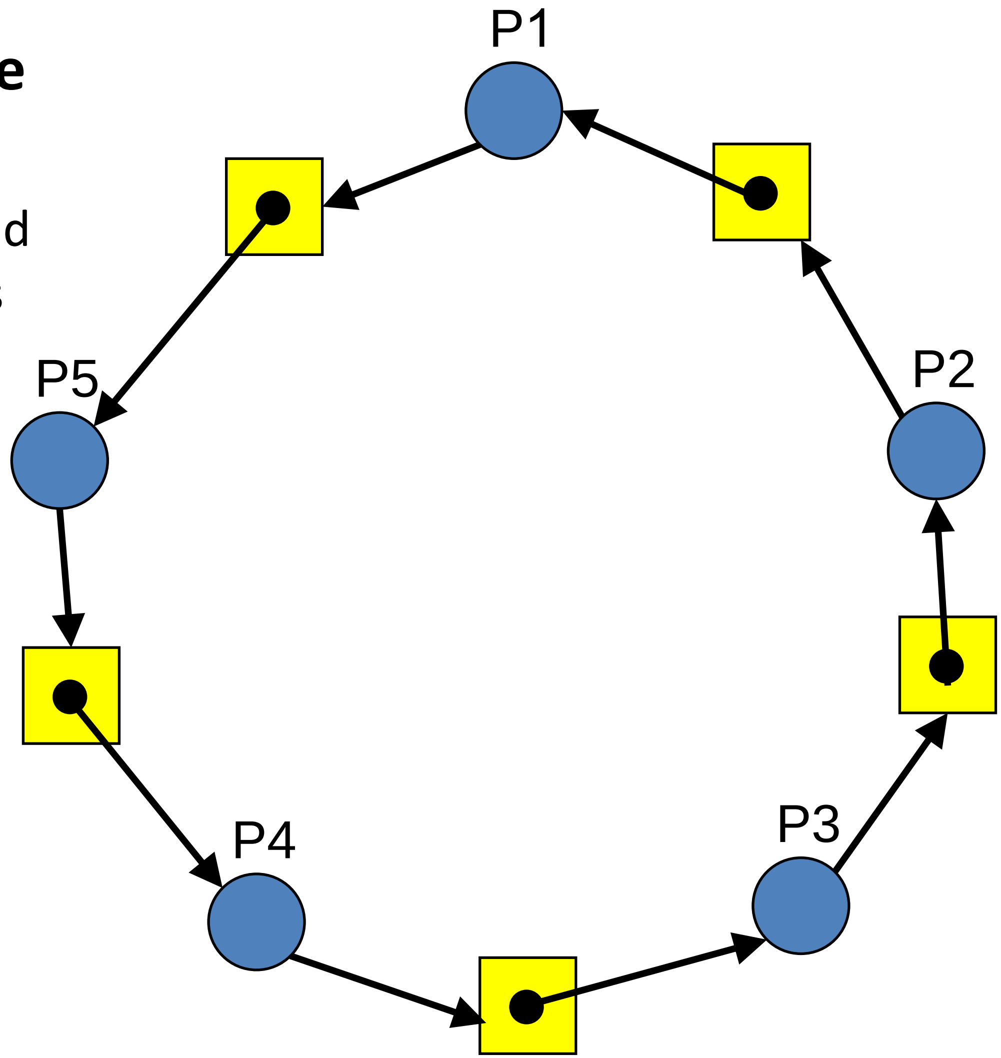
Resource allocation graph

- **When there's only one instance per resource type**
  - Can simplify graph
  - By eliminating resources and only marking dependencies between processes



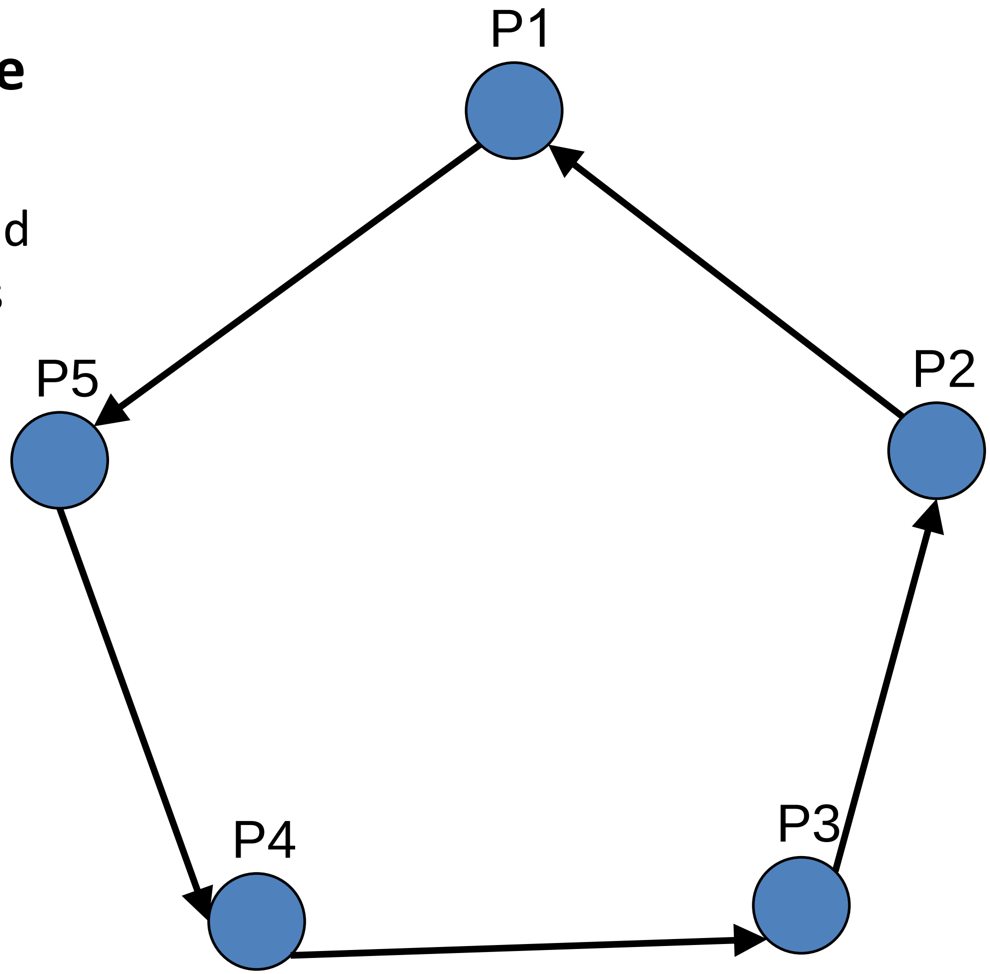
Resource allocation graph

- **When there's only one instance per resource type**
  - Can simplify graph
  - By eliminating resources and only marking dependencies between processes



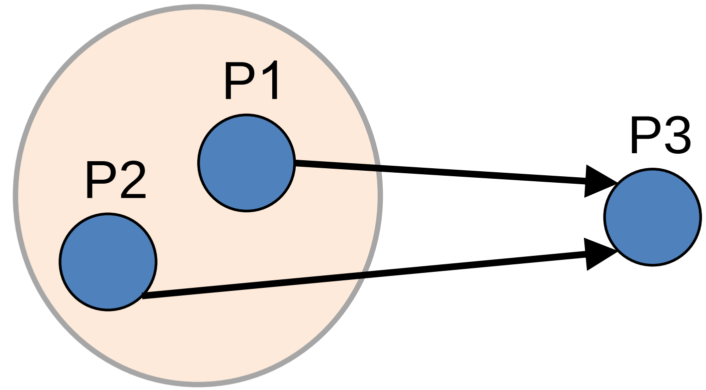
Resource allocation graph

- **When there's only one instance per resource type**
  - Can simplify graph
  - By eliminating resources and only marking dependencies between processes

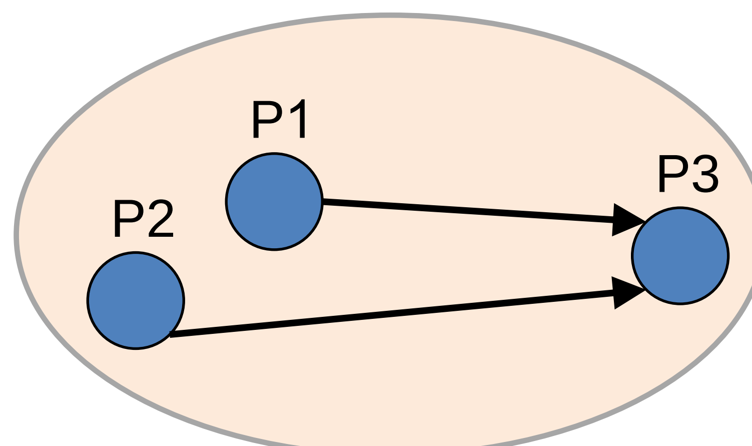


Recall the formal definition of deadlock

- **Definition**
  - A set of processes is deadlocked if each process in the set is waiting for a resource held by another process <u>in the set</u>
- **Why "in the set"?**
  - No deadlock, even though every process in the set is waiting for a resource held by another process:

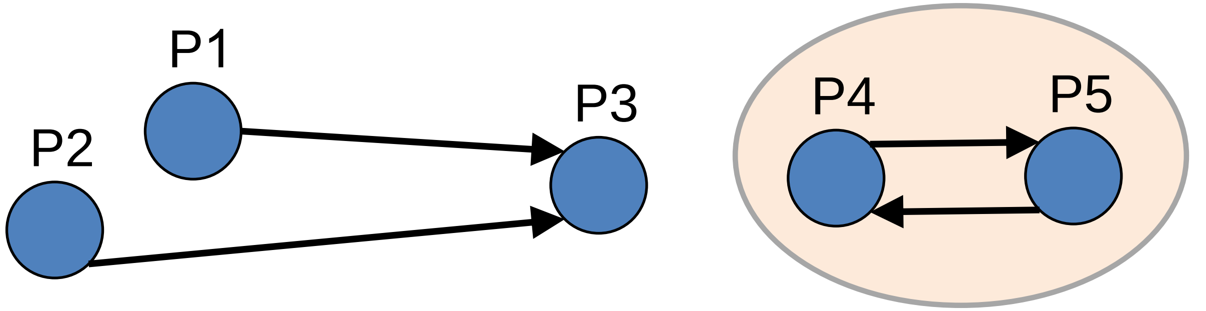


  - Indeed, if including P3, then since P3 isn't waiting for a resource held by another process => no deadlock:



Recall the formal definition of deadlock

- **Definition**
  - A set of processes is deadlocked if each process in the set is waiting for a resource held by another process in the set
- **Can the set be a subset?**
  - Of course



Necessary conditions for deadlock

- ***All*** **of these** ***must*** **hold in order for a deadlock to occur**
  1. Mutual exclusion
     - Some resource is (i) used by more than one process, but is (ii) exclusively allocated to one process at a time, not shared
     - If used by only one process, or can be shared => can't deadlock
  2. Hold & wait
     - Processes may hold one resource and wait for another
     - If resources allocated atomically altogether => can't deadlock
  3. Circular wait
     - P(i) waits for resource held by P((i+1) % n)
     - Otherwise, recursively, there exists one process that need not wait
  4. No resource preemption
     - If resources held can be released (e.g., after some period of time), then can break circular wait

DEALING WITH DEADLOCKS

Who’s responsible?

- **Who is responsible for dealing with deadlocks?**
  - Typically you (the programmer)
  - The OS doesn’t do it for you
  - You need to know how to do it and implement it yourself

Can divide ways into 2

1. **Design the system such that it is never allowed to enter into a deadlock situation**
   – Usable
2. **Allow the system to experience deadlock, but put in mechanisms to detect & recover**
   – Less usable in practice

Violate 1 of the 4 conditions

- **We’ve enumerated 4 conditions that must hold for deadlock to occur**
  - So violating any one of them with eliminate the possibility of deadlocking

Violate "hold and wait"

- **Instead of acquiring resources one by one**
  - Each process requests all resources it'll need at the outset
  - System can then either provide all resources immediately
  - Or block process until all requested resources are available
- **Con**
  - Processes will hold on to their resources for more time than they actually need them
  - Limits concurrency and hence performance
- **Refinement**
  - Before a process issues a new (atomic) request for resources
  - It must release all resources it currently holds
    - (And of course, before that, bring system to consistent state)
  - Risking the resources will be allocated to other processes

Violate “no resource preemption”

- **Under some circumstances, for some resources**
  - Can choose a victim process and release all its resources
  - For example, if there isn’t enough memory, can write the victim’s state to disk and release all its memory

Violate “mutual exclusion”

- **It is possible to implement many canonical data structures (such as a linked list)**
  - Without using any form of explicit synchronization
    - No spinlocks, no semaphores, etc.
  - But while still allowing multiple threads to concurrently use of the data structure
- **How?**
  - Using HW-supported atomic operations only (such as test-and-set)
  - Such algorithms are (also) called “lock free”
    - Not to be confused with the “lock free” algorithm definition from a previous lecture (= “some thread always makes progress”)
- **Mature field**
  - Books on how to do it (formally proving implementations are correct)
  - Existing libraries to use without being exposed to the complexities

Violate "circular wait"

- **Probably the most usable / practical / flexible way to prevent deadlocks**
  - (When lock-free data structures, that are getting popular, are unavailable)
- **How it's done**
  - All resources are numbered in one sequence
    - *Ord(printer)=1, Ord(scanner)=2, Ord(lock_x)=3, Ord(lock_y)=4, …*
  - Processes must request resources in increasing Ord() order
  - Namely, a process holding some resources can only request additional resources that have strictly higher numbers
  - A process that wishes to acquire a resource that has a lower order
    - Must first release all the resources it currently holds

Violate "circular wait"

- **Proof that it works**
  - Assume by contradiction that there exists a cycle
  - Without loss of generality, further assume that
    - P(i) waits for P((i+1) % n)
  - Let M(i) be
    - The maximal Ord() amongst the resources that P(i) holds
  - Thus, since
    - Each P(i) acquires resources in order, and
    - P(i) waits for a resource held by P((i+1) % n)
  - Then
    - $M(i) < M((i+1) \% n)$
      $\Rightarrow M(0) < M(1) < M(2) < \ldots < M(n) < M(0)$
      $\Rightarrow M(0) < M(0)$
      => contradiction

Violate “circular wait”

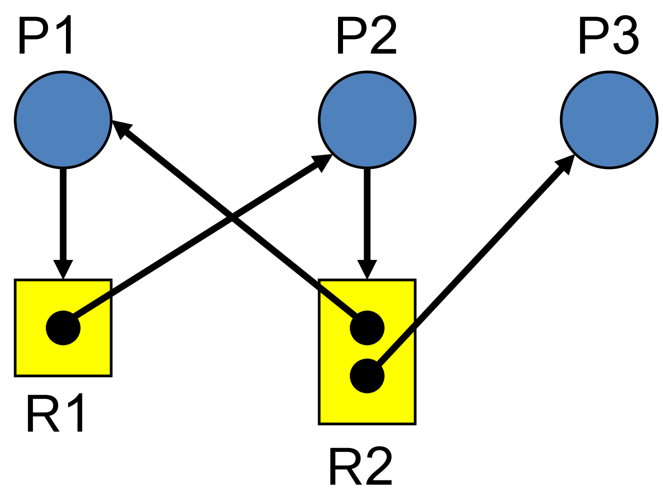
- **We number the chopsticks 0…4 and lock in order**

  - ```
    if ( i < 4)    // i can be 0…4
                   wait( fork[i] )
                   wait( fork[i+1] )
    else // i==4
                   wait( fork[0] )      // smaller
                   wait( fork[4] )      // bigger
    eat
    signal( fork[i] )
    signal( fork[(i+1) % 5] )
    ```
    ```

Deadlock detection

- **If there's only one instance of each resource type**
  - Search for a cycle in the (simplified) resource allocation graph
    - Found ⇔ deadlock

- **In the general case, which allows multiple instances per type**
  - Necessary conditions for deadlock != sufficient conditions for deadlock
  - Indeed, a graph can have a cycle while the system is *not* deadlocked
  - Example……………………………………



- **Can nevertheless detect deadlocks in general case**
  - But algorithm outside the scope of this course

Recovery from a deadlock

- **After a deadlock has been detected (previous slide)**
  - Need to somehow recover
- **If possible, this is done by terminating some of the processes**
  - Until deadlock is resolved
  - Sometimes make sense, sometimes doesn’t
- **Or, if possible, by preempting resources**
  - Of deadlocked processes
- **Finding a minimal (“optimal”) set of processes to terminate or resources to preempt is a hard problem**

Deadlock avoidance

- **Rules**
  - n processes
  - k resource types (each type may have 1 or more instances)
  - Upon initialization, each processes declares maximal number of resource-instances it'll need for each resource type
  - While running, OS maintains how many resources are currently used by each process
  - And how many resource instances per type are currently free
- **Upon process resource allocation request**
  - OS will allocate only iff allocation isn't dangerous, namely
  - It knows for a fact that it'll be able to avoid deadlock in the future
  - Otherwise, the process will be blocked until a better time
  - Algorithm is thus said to be conservative, as there's a possibility for no deadlock even if allocation is made, but OS doesn't take the chance
- **Upon process termination**
  - Process releases all its resources

Deadlock avoidance

- **Example**
  - Banker's algorithm (by Dijkstra)
  - Uses the notation of "safe state"
    - A state whereby we're sure that all processes can be executed, in a certain order, one after the other, such that each will obtain all the resources it needs to complete its execution
  - By ensuring such a sequence exists after each allocation => avoid deadlock
- **Banker's data structure**
  - max[p] = (m_1,m_2, …, m_k) = max resource requirements for process p
  - cur[p] = (c_1,c_2, …, c_k) = current resource allocation for process p
  - avail = (a_1, a_2, …., a_k) = currently free resources
  - R = (r_1, r_2, …, r_k) = the current resource request for process p
- **Example**
  - max[p] = (3,0,1), cur[p] = (3,0,0)
  - Note that max[p] >= cur[p] always holds // compare by coordinates

Banker's algorithm

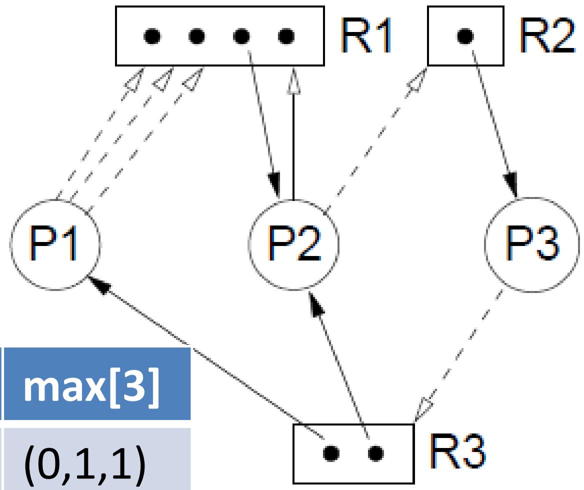
- **Tentatively assume that request R was granted**
  - cur[p] += R // vector addition
  - avail -= R // vector subtraction
- **Check if "safe state" (can satisfy all processes in some order)**
  - initialize P to hold all process
  - while(P isn't empty) {

```
    found = false
    for each p in P {   // find one p that can be satisfied
        if( max[p] – cur[p] <= avail ) // worst case for p
            avail += cur[p]     // "release" p's resources
            P -= {p}
            found = true
    }
    if( ! found ) return FAILURE
}
return SUCCESS
```

Banker's algorithm – runtime complexity

- **O(n^2)**
  - Even though number of possible orders is n!
  - Because resources increase monotonically as processes terminate,
  - As long as it's possible to execute any set of processes
    - Execution order not important
    - (There is never any need to backtrack and try another order)

Banker's algorithm – example



- **Initial system state**

cur[1]	cur[2]	cur[3]	avail	max[1]	max[2]	max[3]
(0,0,1)	(1,0,1)	(0,1,0)	(3,0,0)	(3,0,1)	(2,1,1)	(0,1,1)

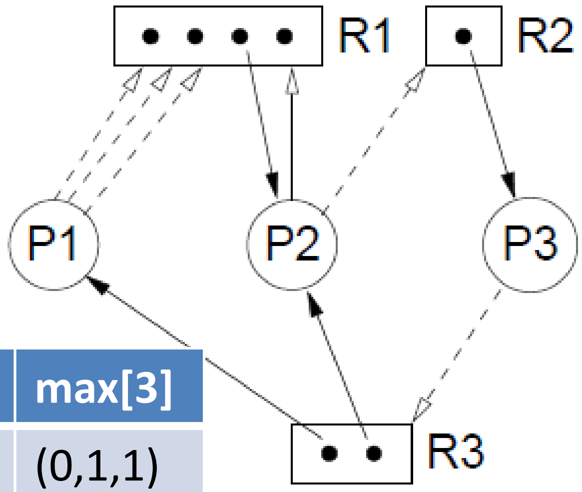
- **P1 requires instance of R1 [R = (1,0,0)]**
  - Granting the request yields

cur[1]	cur[2]	cur[3]	avail	max[1]	max[2]	max[3]
(**1**,0,1)	(1,0,1)	(0,1,0)	(**2**,0,0)	(3,0,1)	(2,1,1)	(0,1,1)

  - Safe, because there are enough R1 instance so that P1's max additional request can be satisfied: max[1]-cur[1]=(2,0,0); so after P1's termination

	cur[2]	cur[3]	avail		max[2]	max[3]
	(1,0,1)	(0,1,0)	(**3**,0,**1**)		(2,1,1)	(0,1,1)

Banker's algorithm – example



- **Copied from previous slide**

	cur[2]	cur[3]	avail		max[2]	max[3]
	(1,0,1)	(0,1,0)	(**3**,0,**1**)		(2,1,1)	(0,1,1)

- **Not enough to satisfy P2 (why?), but can satisfy P3**
  - R3 = (0,1,1) – (0,1,0) = (0,0,1) (<= avail = (3,0,1))

	cur[2]	cur[3]	avail		max[2]	
	(1,0,1)		(**3**,**1**,**1**)		(2,1,1)	

Ways to deal with deadlocks

1. **Deadlock "prevention"**
   - Design system in which deadlock cannot happen
   - Violate 1 of the 4
2. **Deadlock "avoidance"**
   - System manages to stay away from deadlock situations by being careful on a per resource-allocation decision basis
   - Banker's
3. **Deadlock detection & recovery**
   - Allow system to enter deadlock state, but put in place mechanisms that can detect, and recover from, this situation

THANKS