Dinning Philosopher Problem

```
Repeat
        wait(chopstick[I]);
        wait(chopstick[I+1 mod 5]);
        ….
        Eat
        ….
        Signal(chopstick[I]);
        Signal(chodstick[I+1 mod 5]);
        ….
        Think
        ….
Until false;
```

Is there a problem?
How can we avoid the problem?

Atmost 4 can eat together
wait till both are available
odd picks right/even picks left

Starvation?

Necessary Conditions for Deadlock

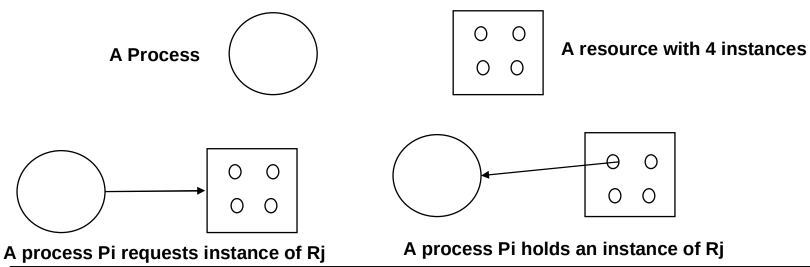
Mutual Exclusion
Hold and Wait
No Preemption
Circular Wait

Methods for Dealing

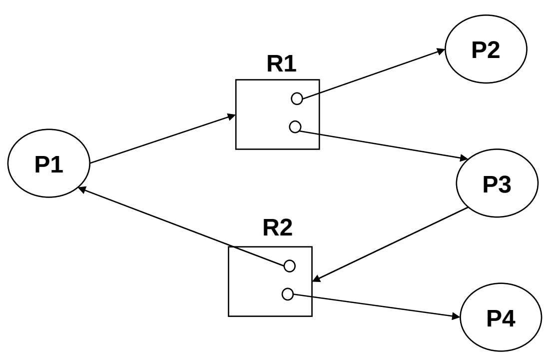
Avoidance
Prevention
Recover
Do nothing

Resource Allocation Graph

- A Set of All Resources R= {R1, R2, R3,… Rm)
- A Set of All Processes V= { V1, V2, V3 … Vn}
- A Set of Request Edges (Pi ➡ Rj)
- A Set of Assignment Edges (Rj ➡ Pi)



A RAG with Cycles



- A RAG with no cycle => no deadlock
- A RAG with cycle =>
  - if only one instance per resource type, then deadlock
  - othgerwise, possiblity of deadlock

Deadlock Prevention

- **Prevent Mutual Exclusion?**
  - ♦ **Some resources are intrinsically nonsharable.**
- **Prevent Hold and Wait**
  - ♦ **Protocol-1: request everything at once and complete**
  - ♦ **Protocol-2: before every new request release all**
- **Prevent No preemption**
  - ♦ **If a new request cannot be allocated then take away all.**
  - ♦ **If a process is waiting by holding a resource which is requested by another new process, then take it away from the first process.**
- **Prevent Circular wait**
  - ♦ **Impose a total ordering on all resources. All request must be in order. Example:**
    - » **tape 1, disk 5, printer 12**
    - » **proof?**

Deadlock Avoidance

- **Deadlock Prevention schemes can reduce resource utilization and system throughput.**
- **Deadlock avoidance schemes utilizes some additional info (for example how the processes will request resources, what will be their maximum requests, etc.) to avoid deadlock.**
- **Safe State:**
  - ♦ **A state is safe, if the system can allocate resources to each process (up to its maximum) in atleast one order and still avoid deadlock.**

Bankers Algorithm

Available[M]:
Max[NxM]:
Allocation[NxM];
Need[NxM] : Need[i,j]=Max[i,j]-Allocation[i,j] Tavailable[M], and Tfinish[N]
(we will refer to entire vector by Need[i], or allocation[I])

1. Tavailable[j]=Available[j] for all j
Tfinish[i]=false for all i
2. Find an i such that
Tfinish[i]=false and
Need[i] < Tavailable;
If No such i exists go to step 4.
3. Tavail=Tavail+Allocation[i] /*consider process i done*/
Tfinish[i]=true;
go to step 2.
4. If Tfinish[i]=true for all i then it is in safe state

Examples:

Total 12

	Max.	Allocation1
P0	10	5
P1	4	2
P2	9	2

Is it Safe?

	Max.	Allocation2
P0	10	5
P1	4	2
P2	9	3

Is it Safe?

Total A=10 B=5 C=7

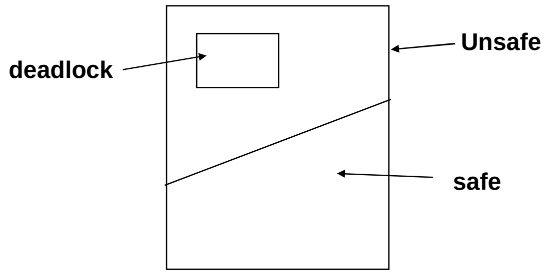
	Max. ABC	Allocation ABC
P0	753	010
P1	322	200
P2	902	302
P3	222	211
P4	433	002
Available:		332

Is it safe? (yes)

What if P1 requests additional 102?
What if now P4 requests additional 330?
What if P0 requests additional 020?

Questions?

- Is a deadlock state is also unsafe state?
- Does an unsafe state always lead to a deadlock state?



Recovery

- When should OS check?
  - How often deadlock is likely to occur?
  - How many processes are affected?
  - When often recovery should be initiated?
- Should we abort all processes involved?
- Should we abort one at a time?
- Should we preempt one resource at a time?
- Can there be starvation?
- Combined approach:
  - Internal resources (PCB etc.): use ordering.
  - Central memory: use preemption.
  - Job resources (printers etc.): Use avoidance.
  - Swappable space (backup scratch pad): use preallocation.