Deadlocks

What is deadlock & how it occurs

Detecting potential deadlocks

– resource allocation graphs

Recovery techniques

Prevention techniques

Livelock and starvation

Deadlocks

Example: two processes want to scan a document, and then save it on a CD

P0

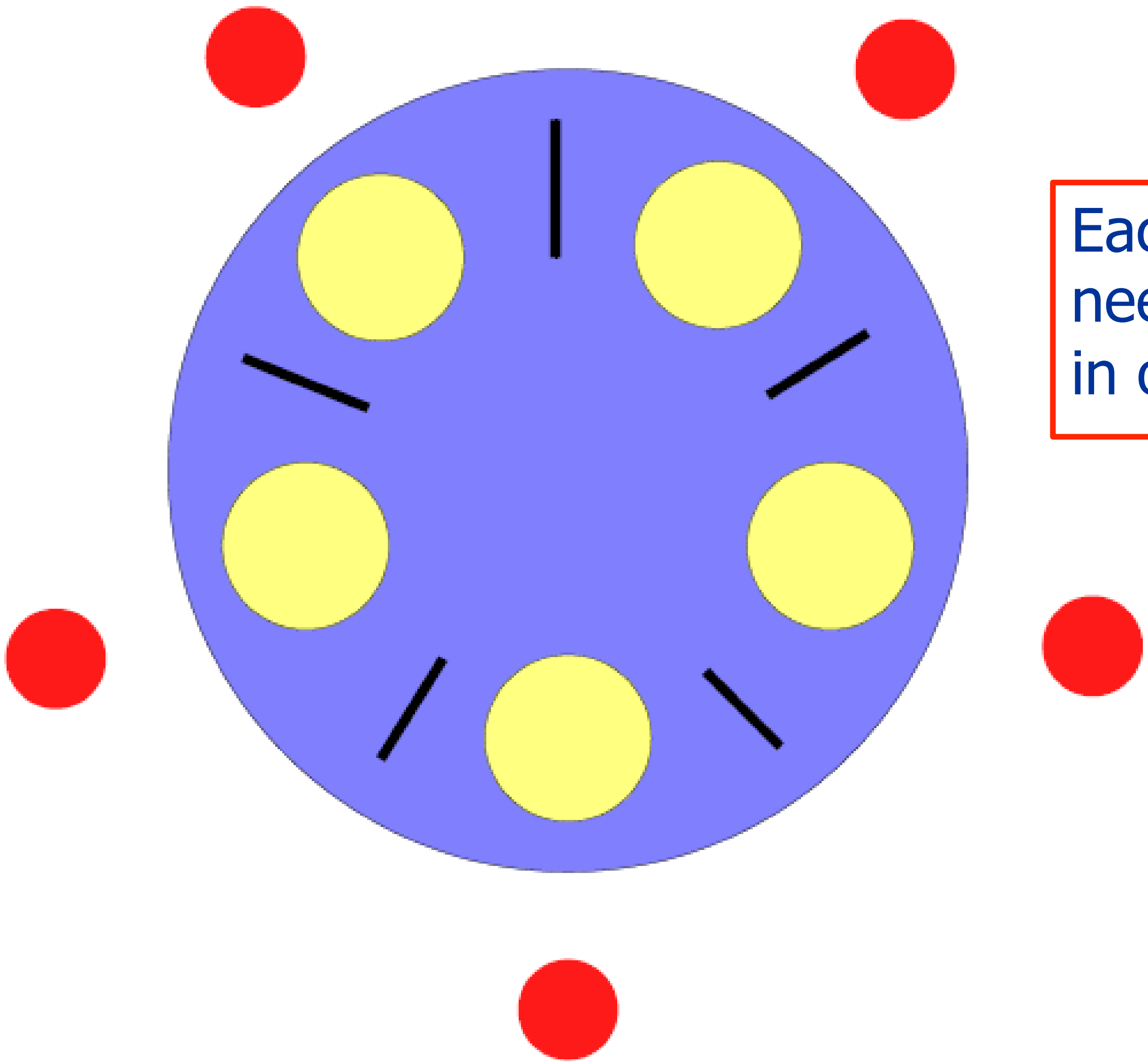
```
down(scanner);
down(cd_writer);
scan_and_record();
up(cd_writer);
up(scanner);
```

P1

```
down(cd_writer);
down(scanner);
scan_and_record();
up(scanner);
up(cd_writer);
```

Deadlock?

Dining Philosophers



Each philosopher needs 2 chopsticks in order to eat

Dining Philosophers

```
var chopstick:  array [0..4] of Semaphore

procedure philosopher(i:int)
  loop
    down(chopstick[i])
    down(chopstick[i+1 mod 5])
    eat
    up(chopstick[i])
    up(chopstick[i+1 mod 5])
    think
    end loop
  end philosopher
```

Does this work?

What if everybody takes chopstick[i] at same time?

Deadlock

Set of processes is deadlocked if each process is **waiting for an event** that only **another process** can cause

Resource deadlock is most common, 4 conditions must hold:

1. **Mutual exclusion:** each resource is either available or assigned to exactly one process
2. **Hold and wait :** process can request resources while it holds other resources, requested earlier
3. **No preemption:** resources given to a process cannot be forcibly revoked
4. **Circular wait:** two or more processes in a circular chain, each waiting for a resource held by the next process

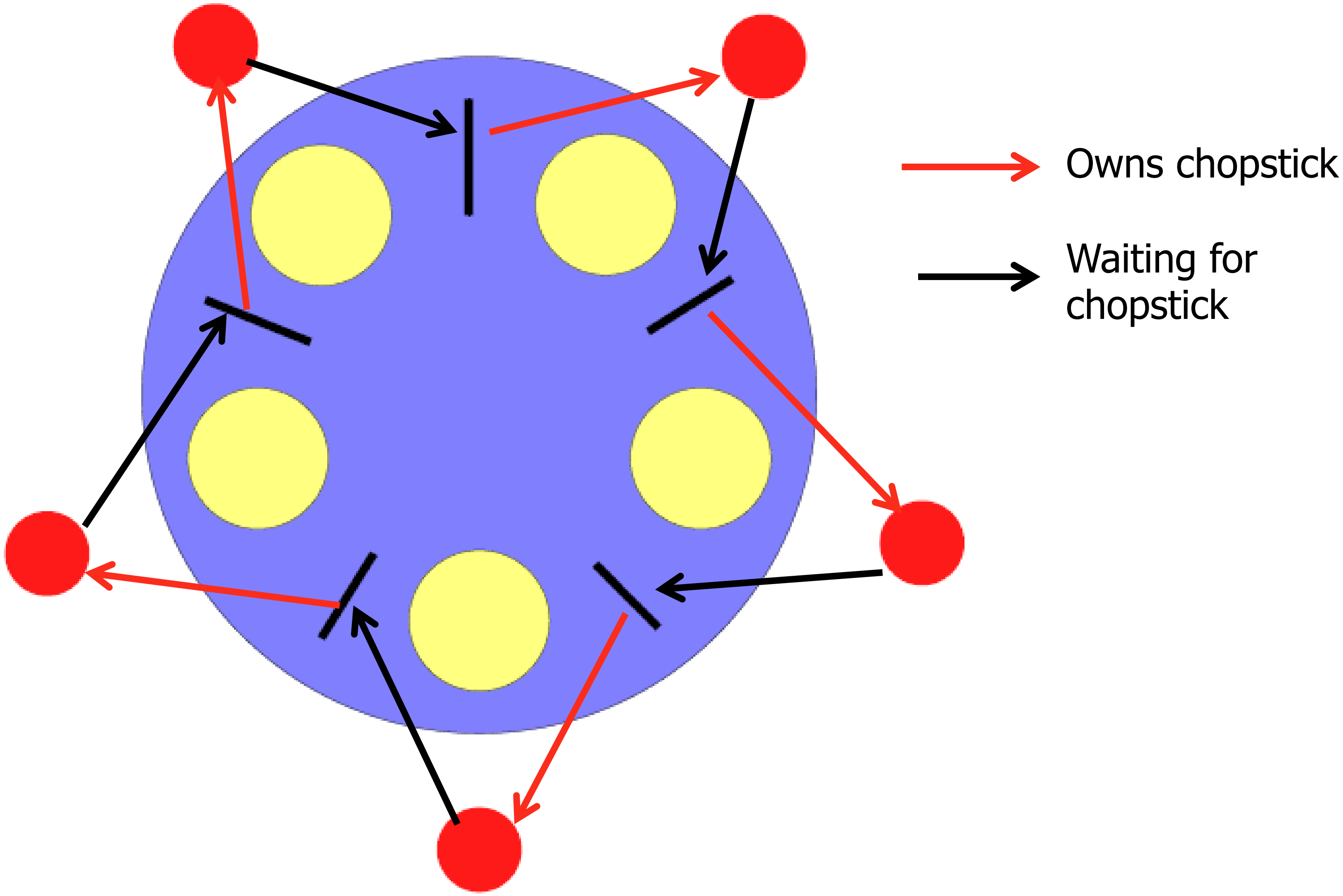
Resource Allocation Graphs

Directed graph models resource allocation

- Directed arc from resource to process means that the process is currently owning that resource
- Directed arc from process to resource means that the process is currently blocked waiting for that resource

Cycle = deadlock

Dining Philosophers – Deadlock Cycle



Strategies For Dealing With Deadlock

- Ignore it
  - "The Ostrich Algorithm"
  - Contention for resources is low → deadlocks infrequent
- Detection and recovery
- Dynamic avoidance by careful resource allocation
- Prevention by negating 1 of the 4 conditions

Detection and Recovery

Detects deadlock and recovers ***after the fact***
Dynamically builds resource ownership graph and looks for cycles
When an ***arc*** has been inspected it is marked and not visited again

1. For each node do:
2. Initialise **L** to the empty list
3. Add the current node to **L** and check if it appears in **L** two times. Yes: cycle!
4. From current node check if any unmarked outgoing arc

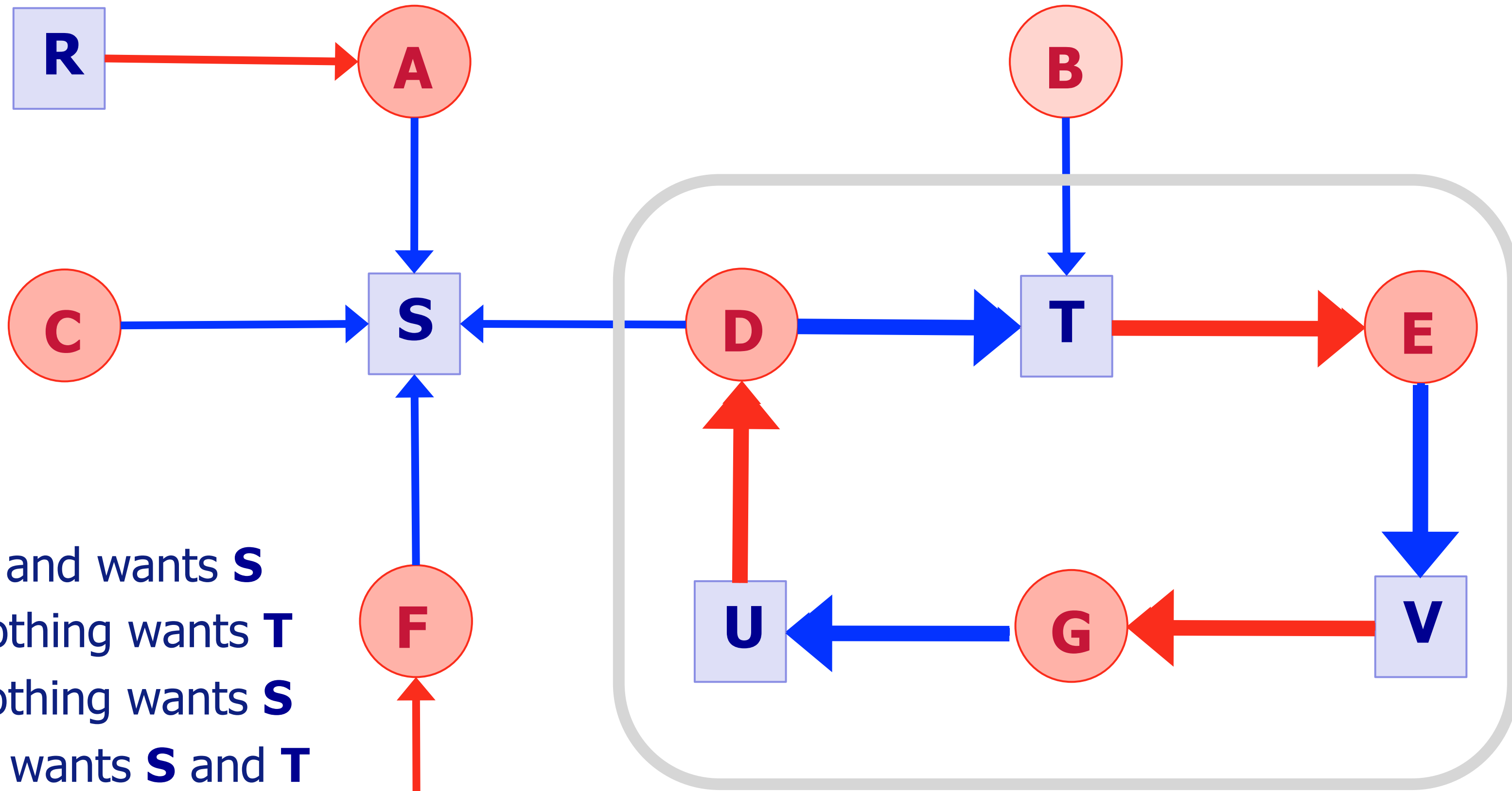
   Yes: goto **5**, No: goto **6**
5. Pick unmarked outgoing arc, mark it, follow it to new current node and goto **3**
6. If this is initial node then no cycles detected, terminate

   else reached dead end, remove it, go back to previous node and make it current and goto **3**

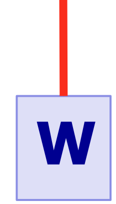
We are doing a depth-first search from each node in the graph, checking for cycles.

[Tanenbaum, MOS 6.4.1]

Detection – Example



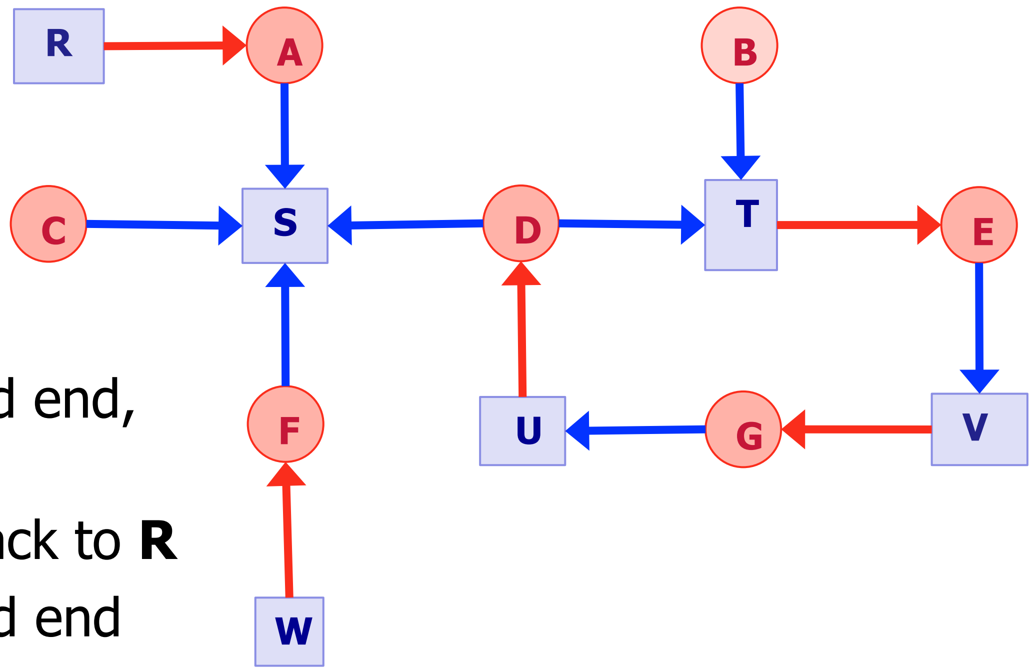
- **A** holds **R** and wants **S**
- **B** holds nothing wants **T**
- **C** holds nothing wants **S**
- **D** holds **U** wants **S** and **T**
- **E** holds **T** and wants **V**
- **F** holds **W** and wants **S**
- **G** holds **V** and wants **U**



Cycle?

Detection – Example (2)

- Starting at **R**, initialise **L** = [ ]
- Add **R** to list and move to **A** (only possibility)
- Add A giving **L = [R,A]**
- Go to S so **L = [R,A,S]**
- **S** has not outgoing arcs → dead end, backtrack to **A**
- **A** has no outgoing arcs, backtrack to **R**
- Restart at **A**, add A to ***L*** → dead end
- Restart at **B**, follow outgoing arcs until **D**, now **L = [B,T,E,V,G,U,D]**
- Make random choice:
  - **S** → dead end and backtrack to **D**
  - Pick T update **L = [B,T,E,V,G,U,D,T]**
- Cycle: Deadlock found, STOP



Recovery

Pre-emption:

- Temporarily take resource from owner and give to another

Rollback:

- Processes are periodically **checkpointed** (memory image, state)
- On a deadlock, **roll back** to previous state

Killing processes:

- Select random process in cycle and kill it!
  - OK for compile jobs, not so good for database, why?

Strategies For Dealing With Deadlock

- Ignore it
- Detection and recovery
- Dynamic avoidance
  - System grants resources when it knows that it is safe to do so
- Prevention

Banker's Algorithm (Dijkstra 1965)

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

- Four customers A, B, C and D
  - Credit unit = £1K
- Banker knows that all customers don't need max credit
  - So reserves only 10 (instead of 22) units
- Each customer randomly asks for credit
- For each process A-D,
  - Has = number of resource items allocated
  - Max = number of items required.

Banker's Algorithm – Save vs. Unsafe States

SAFE

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

UNSAFE

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Safe state:

- Are there enough resources to satisfy ***any*** (maximum) request from some customer?
- Assume that customer repays loan, and then check next customer closest to the limit, etc.

A state is **safe** iff there exists a sequence of allocations that *guarantees* that all customers can be satisfied

Banker’s Algorithm – Save vs. Unsafe States

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

SAFE

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

UNSAFE

A state is **safe** iff there exists a sequence of allocations that *guarantees* all customers can be satisfied

Banker’s Algorithm – Save vs. Unsafe States

SAFE

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

UNSAFE

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

Request granted only if it leads to a safe state

Unsafe state does not have to lead to deadlock, but banker cannot rely on this behaviour

Algorithm can be generalized to handle multiple resource types

Strategies For Dealing With Deadlock

Ignore it

Detection and recovery

Dynamic avoidance

Prevention

- *Attack one of the four deadlock conditions:*
  - *Mutual exclusion,*
  - *Hold and wait*
  - *No preemption*
  - *Circular wait*

Deadlock Prevention

Attacking the Mutual Exclusion Condition

- E.g., share the resource

Attacking the Hold and Wait Condition

- Require all processes to request resources before start
  - If not all available then wait
- Issue: need to know what you need in advance

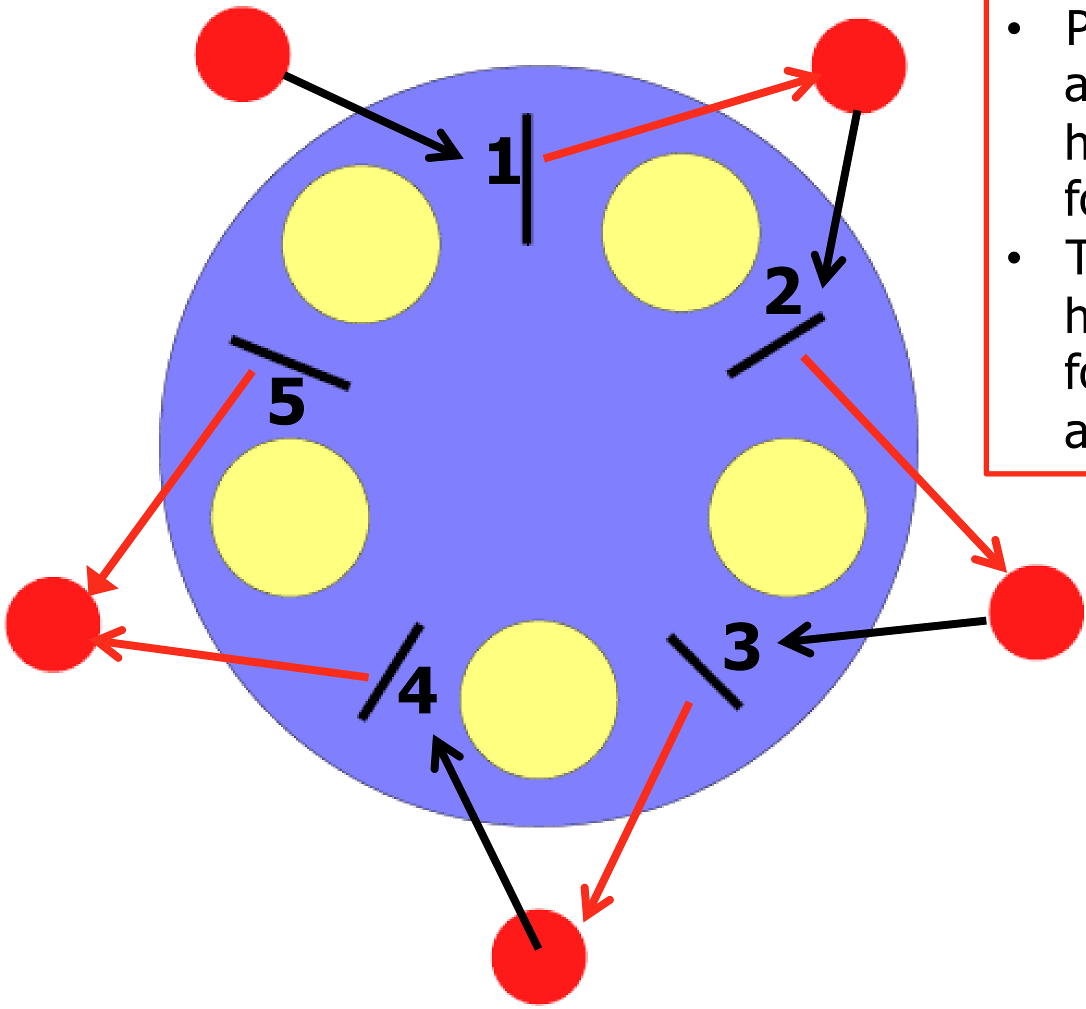
Attacking the No-Preemption condition

- E.g., forcing a process to give up printer half way through. Usually not good

Attacking Circular Wait Problem

- Force single resource per process, if needs second, must release first.
  - Optimality issues
- Number resources, processes must ask for resources in this order
  - Issue: large number of resources…can be difficult to organise

Dining Philosophers – Ordering Resources



- Request resources in specific order
- Philosopher gets chopstick 4 and can then ask for 5, but if he gets 5 first he cannot ask for chopstick 4.
- The process holding the highest resource will never ask for a resource already assigned.

Communication Deadlock

E.g., process **A** sends message to **B** and blocks waiting on **B's** reply

B didn't get **A's** message then **A** is blocked and **B** is blocked waiting on message ➔ ***deadlock***!

Ordering resources, careful scheduling not useful here

What should we use?

- Communication protocol based on timeouts

Livelock

- **Livelock**: Processes/threads are not blocked, but they or the system as a whole does not make progress

- Example 1: `Enter_region()` tests mutex then either grabs resource or reports failure. If attempt fails, it tries again. Processes loop after gaining first resource but failing second.

```
process_A
{
   Enter_region (resource1)
   Enter_region (resource2)
   Use(resource1, resource2)
   Leave_region (resource2)
   Leave_region (resource1)
}
```

```
process_B
{
   Enter_region (resource2)
   Enter_region (resource1)
   Use(resource1, resource2)
   Leave_region (resource1)
   Leave_region (resource2)
}
```

- Example 2: System receiving and processing incoming messages. Processing thread has lower priority and never gets a chance to run under high load (**receive livelock**)

Starvation

Concerns policy

Who gets what resource when

Many jobs want printer, who gets it?

- Smallest file? Suits majority, fast turnaround, but what about occasional large job?
- FCFS is more fair in this case

Deadlock Summary

Deadlocks occur from:

- Accessing limited resources – not enough to go round
- Incorrect programming of synchronisation

Resource allocation graphs can detect potential cyclic deadlock

Recovery: pre-emption, rollback, kill process

Prevention

- Use safe resource allocation strategy
- Avoid unnecessary mutual exclusion – share instead
- Ordered resource allocation

Livelock: no progress – incorrect programming?

Starvation: often due to priority