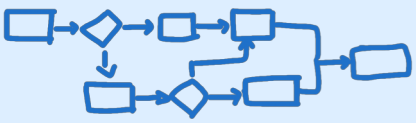
Imperial College London



Operating Systems
Deadlocks

Course 211
Spring Term 2018-2019

http://www.imperial.ac.uk/computing/current-students/courses/211/calendar/

(Slides courtesy of Cristian Cadar)

Peter Pietzuch

prp@doc.ic.ac.uk
http://www.doc.ic.ac.uk/~prp

Deadlocks

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

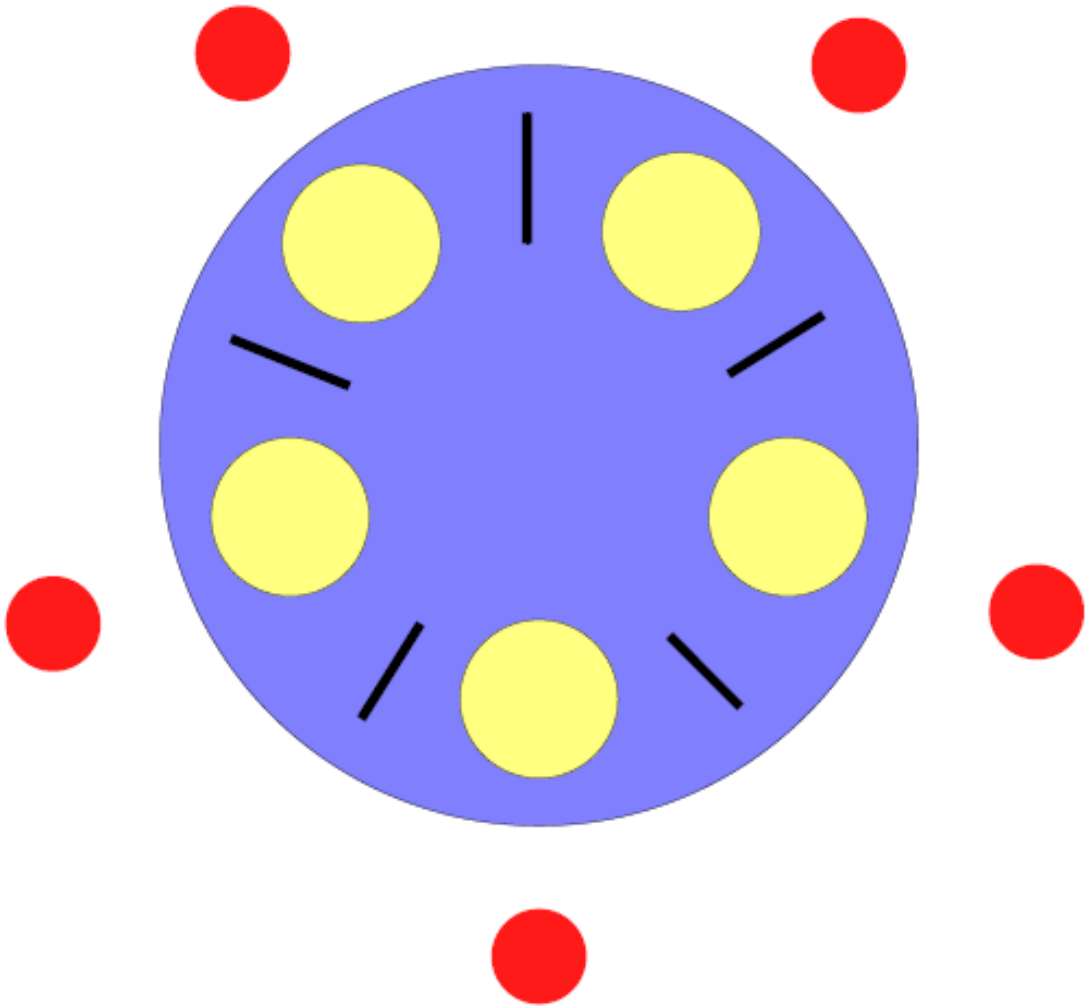
P0

```
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);
```

Dining Philosophers



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:

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

Tutorial Questions

Can the set of processes deadlocked include processes that are not in the circular chain in the corresponding resource allocation graph?

Can a single-processor system have no processes ready and no process running? Is this a deadlocked system?

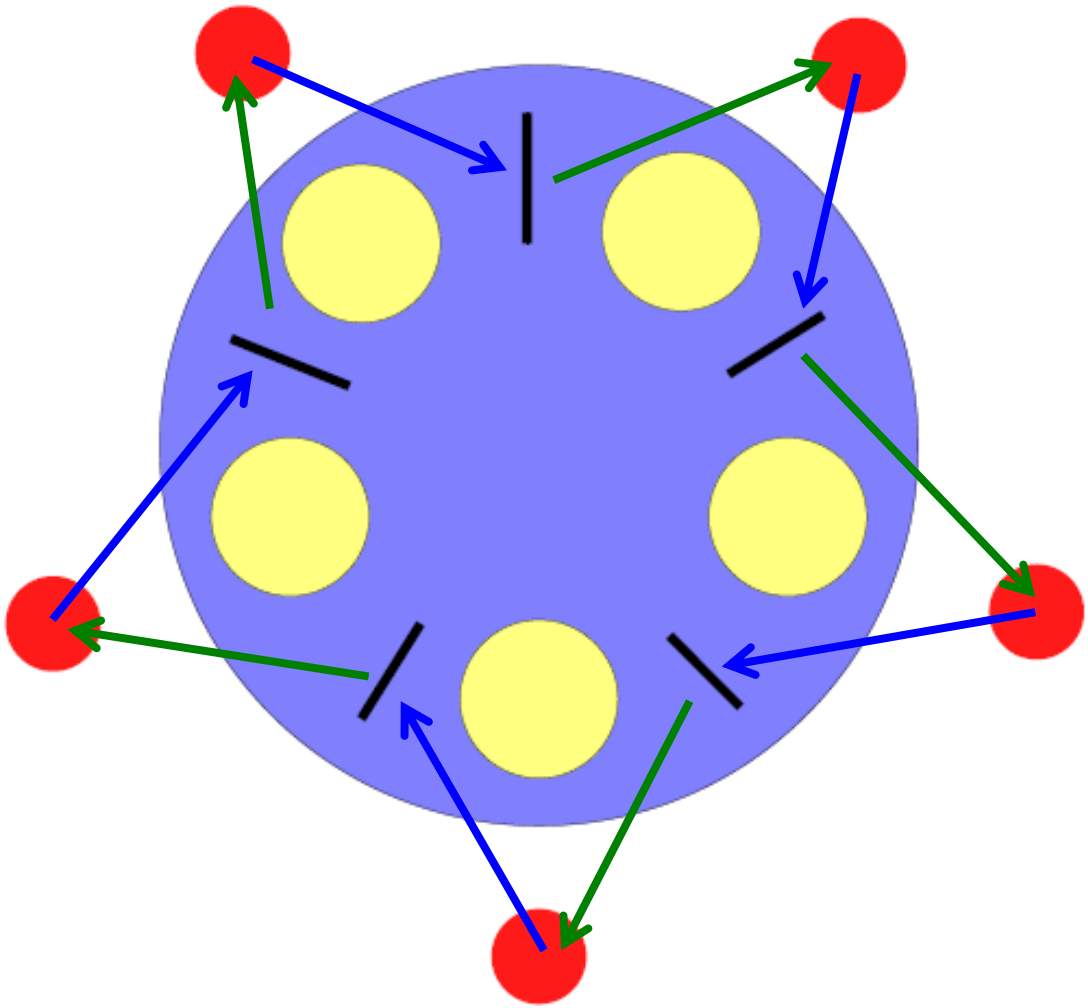
Resource Allocation Graphs

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

1. Ignore it

- "The Ostrich Algorithm"
- Contention for resources is low → deadlocks infrequent

2. Detection & recovery

- After system is deadlocked: (1) detect the deadlock and (2) recover from it

3. Dynamic avoidance

- Dynamically consider every request and decide whether it is safe to grant it; needs some information regarding potential resource use

4. Prevention

- Prevent deadlocks by ensuring at least one of the four deadlock conditions can never hold

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

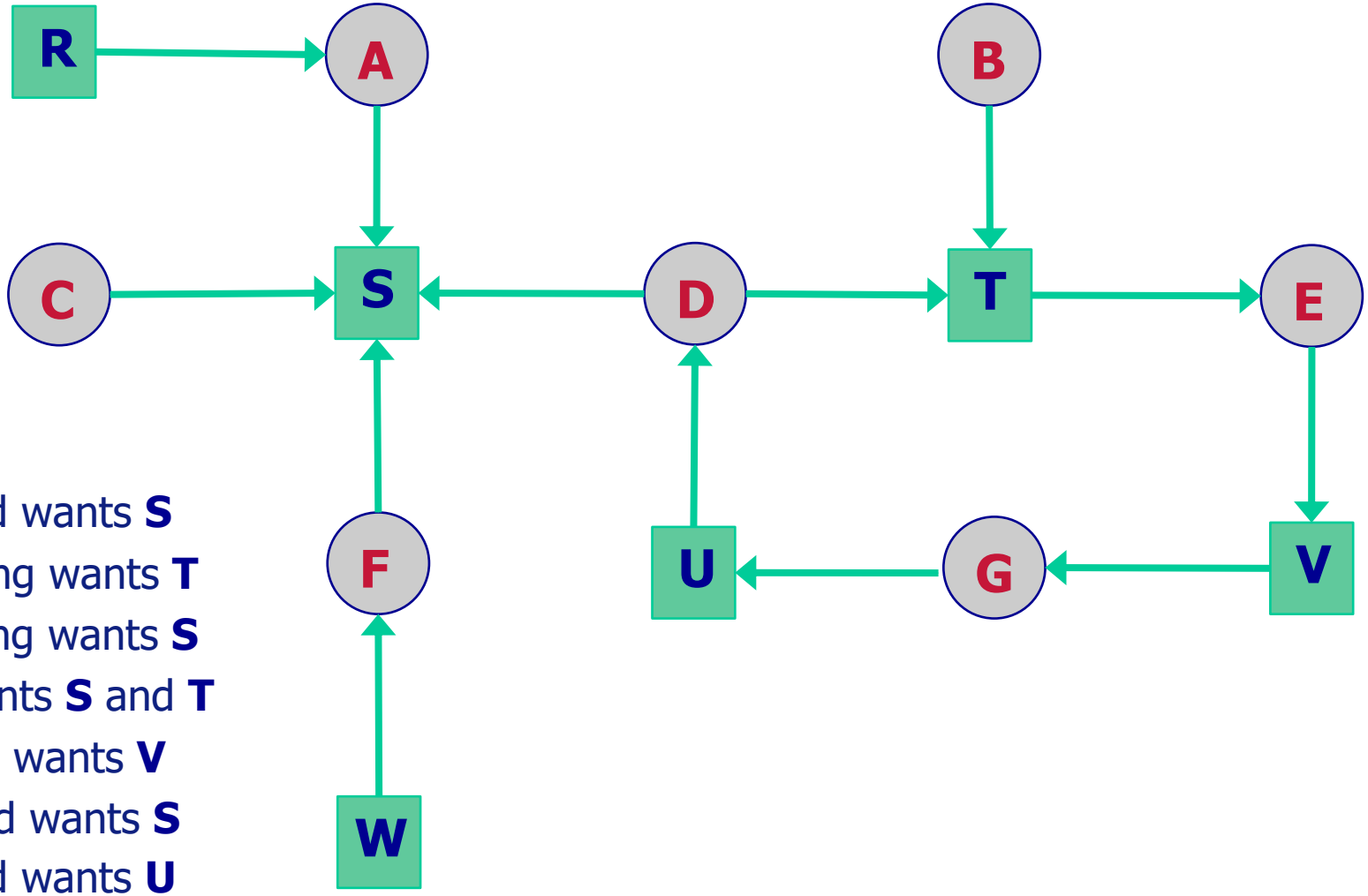
1. For each node do:
2. Initialise **L** to the empty list
3. Add the current node to **L** and check if it appears in **L** two times. Yes: cycle!
4. From current node check if any unmarked outgoing arc

   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**

Depth-first search from each node in the graph, checking for cycles.

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**

Detection Example

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**

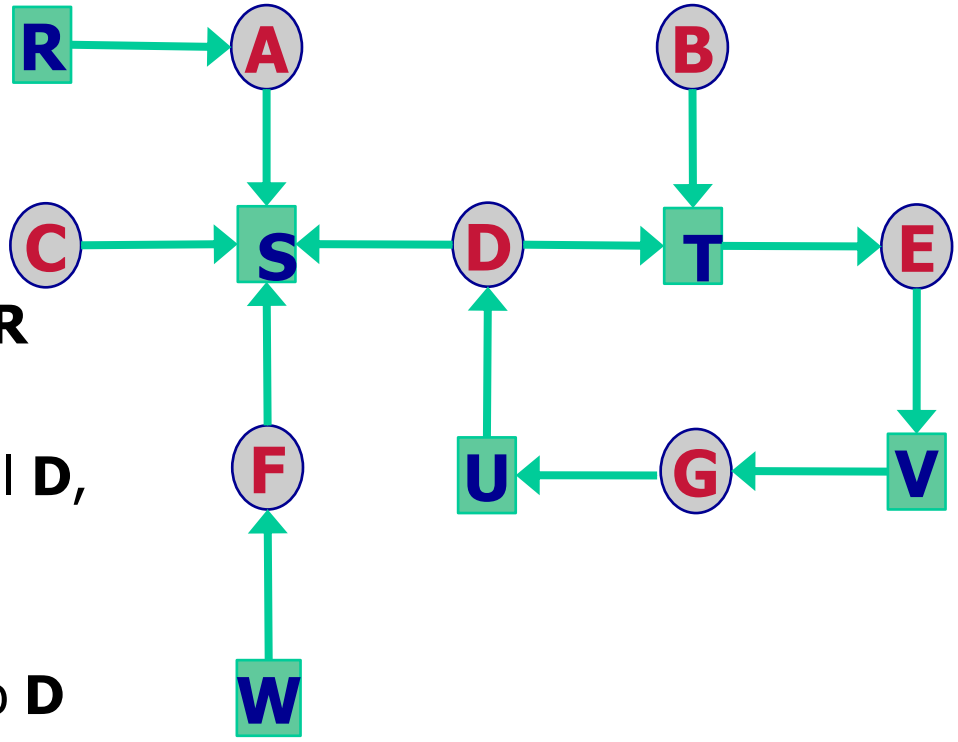
Restart at **A**, add A to **L** → dead end

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
  - When is it going to resolve the deadlock?

Killing processes:

- Select random process in cycle and kill it!
  - E.g. may work for compile jobs, but not for DB system

Strategies For Dealing With Deadlock

1. Ignore it

2. Detection & recovery

3. Dynamic avoidance

- System grants resources when it knows that it is safe to do so

4. Prevention

Banker's Algorithm (Dijkstra 1965)

Set-up with a single type of resource

- N customers have a maximum credit limit expressed in number of credit units
  - E.g. 1 credit unit = £1K

Each customer may ask for its maximum credit at some point, use it, and then repay it

Banker knows that all customers don't need max credit at the same time

- So it reserves less than the sum of all credit limits

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 reserves only 10 (instead of 22) units

Banker's Algorithm: Safe 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, ...

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

Banker's Algorithm: Safe 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 generalised to handle multiple resource types

Strategies For Dealing With Deadlock

1. Ignore it

2. Detection & recovery

3. Dynamic avoidance

4. Prevention

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

Deadlock: 4 conditions

Mutual exclusion: each resource is either available or assigned to exactly one process

Hold and wait: process can request resources while it holds other resources earlier

No preemption: resources given to a process cannot be forcibly revoked

Circular wait: two or more processes in a circular chain, each waiting for a resource held by the next process

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

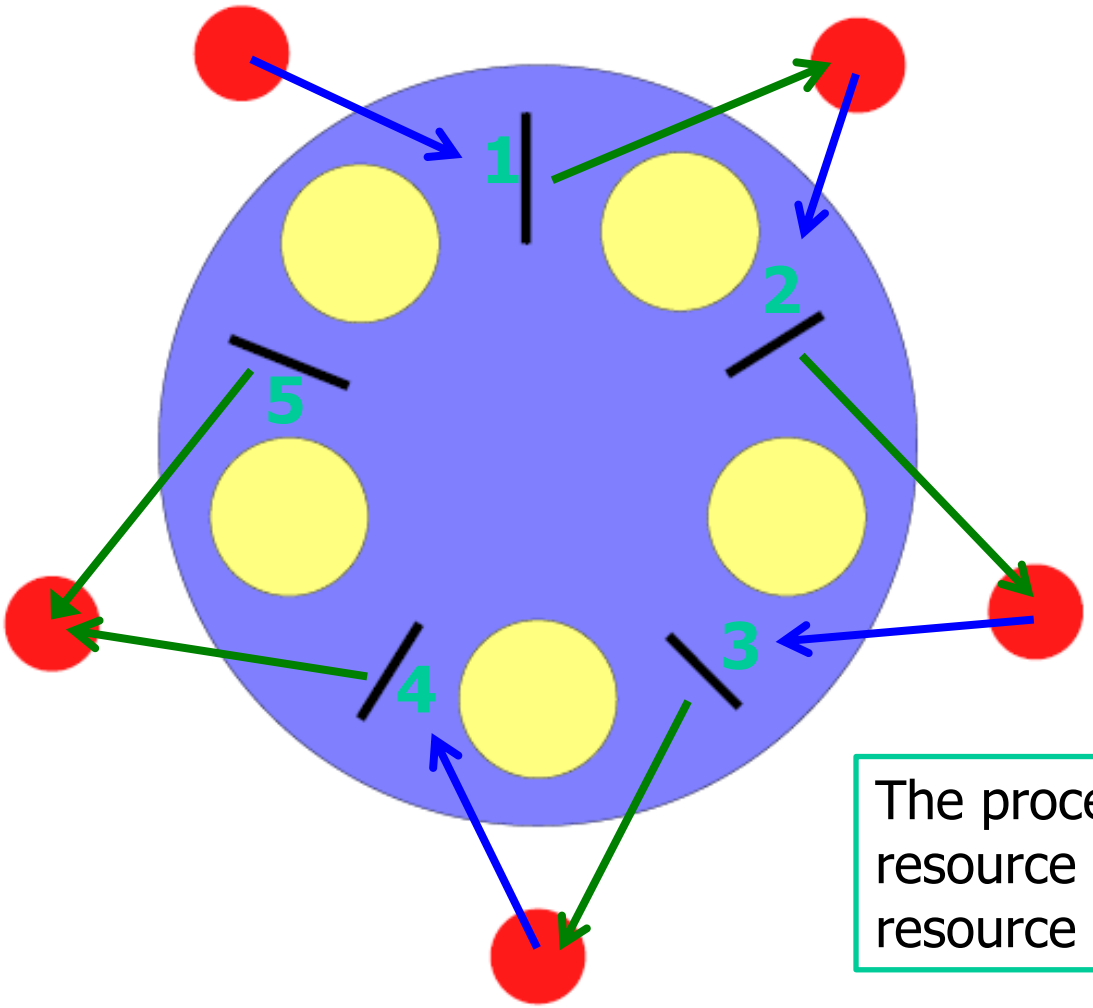
Attacking the No-Preemption condition

- E.g. forcing process to give up printer half way not good

Attacking Circular Wait condition

- Force single resource per process?
  - Optimality issues
- Number resources, processes must ask for resources in order
  - Issue: large number of resources…can be difficult to organise

Dining Philosophers – Ordering Resources



The process holding the highest resource will never ask for a resource already assigned.

Communication Deadlock

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 not making progress

- Example 1: **`Enter_region()`** tests mutex, then either grabs resource or reports failure. If attempt fails, it tries again.

```
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

Deadlocks: Summary

Resource deadlock: set of processes each waiting for a resource another one holds

- Four conditions required for deadlock
- Resource allocation graphs can be used to reason about deadlocks

Strategies for avoiding deadlock

- Ignoring
- Detection and recovery
- Dynamic avoidance
- Prevention

Communication deadlock: concerns policy

Livelock: overall system makes no progress

Tutorial Question: Banker's Algorithm

Consider the following system with five processes, P1–P5, and three resources, A-C. There are 10 instances of A, 5 of B and 7 of C.

Process	Allocation			Need		
	A	B	C	A	B	C
P1	0	1	0	7	4	3
P2	3	0	2	0	2	0
P3	3	0	2	6	0	0
P4	2	1	1	0	1	1
P5	0	0	2	4	3	1

Available: A: 2 B: 3 C: 0

(1) Is this system in a **safe state**?

(2) Can we accept P1's request for 2 instances of B?

Tutorial Answer: Banker's Algorithm

Tutorial Question: Deadlock

Two processes, A and B, each need three records, 1, 2, and 3, in a database. If A asks for them in the order 1, 2, 3, and B asks for them in the same order, deadlock is not possible. However, if B asks for them in the order 3, 2, 1, then deadlock is possible. With three resources, there are 3! = 6 possible combinations each process can request resources.

What fraction of all combinations is guaranteed to be deadlock free?

Tutorial Answer: Deadlock

What fraction of all combinations is guaranteed to be deadlock free?