Deadlocks

Loïc Guégan

Adapted from J. Kubiatowicz @ 2010 UCB, O. J. Anshus and T. Larsen and P. Ha @ UiT,
K. Li @ Princeton, A. S. Tanenbaum @ 2008, A. Silberschatz @ 2009

UiT The Arctic University of Norway

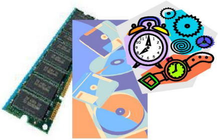
Spring - 2024

Outline

- What is deadlock?
- How can deadlock occur?
- How to deal with deadlocks?
- Examples

Resources

- Resources - passive entities needed by threads to do their work
  - CPU time, disk space, memory



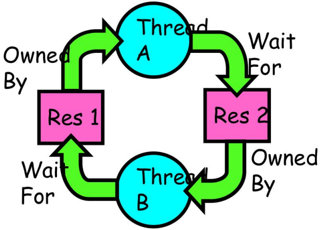
- Two types of resources:
  1. **Premptable** - can take away
     - ⋆ e.g. CPU
  2. **Non-premptable** - must leave it with the thread
     - ⋆ Disk space, chunk of virtual address space
     - ⋆ Mutual exclusion - the right to enter a critical section

- Resources may require exclusive access or may be sharable
  - Read-only files are typically sharable
  - Printers are not sharable during time of printing

- One of the major tasks of an operating system is to **manage resources**

Definition: Starvation vs Deadlock

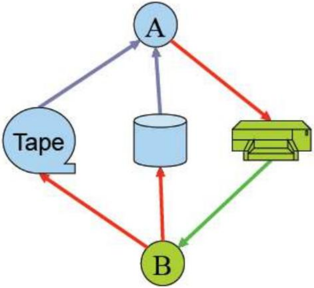
- Starvation ⇒ thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock ⇒ circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
  - Thread B owns Res 2 and is waiting for Res 1



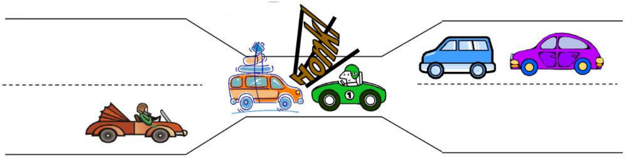
- Deadlock vs Starvation
  - Starvation can end (but do not have to)
  - Deadlock <u>cannot end</u> without external intervention

An example

- A utility program
  - Copy a file from tape to disk
  - Print the file to printer
- Resources
  - Tape
  - Disk
  - Printer
- A deadlock
  - **A** holds tape and disk, then requests printer
  - **B** holds printer, then requests tape and disk



Bridge crossing example



- Each segment of road can be viewed as a resource
  - Cars must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (release resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast ⇒ no one goes west

Deadlock properties

- Deadlock not always deterministic – Example 2 mutexes:

Thread A	Thread B
`x.P();`	`y.P();`
`y.P();`	`x.P();`
`y.V();`	`x.V();`
`x.V();`	`y.V();`

  - Deadlock won't always happen with this code
    - Have to have exactly the right timing ("wrong" timing?)

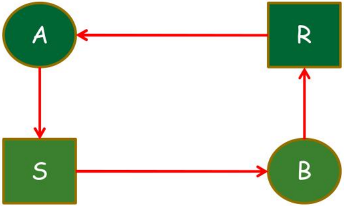
- Deadlocks occur with multiple resources
  - Can't decompose the problem
  - Can't solve deadlock for each resource independently
- Example: System with 2 disk drives and two threads
  - Each thread needs 2 disk drives to function
  - Each thread gets one disk and waits for another one

Resource Allocation Graph

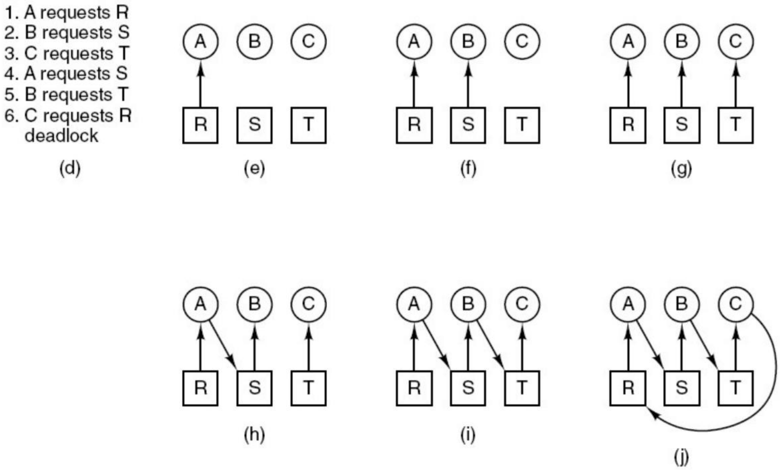
- Process A is holding resource R

- Process B is wait for resource S

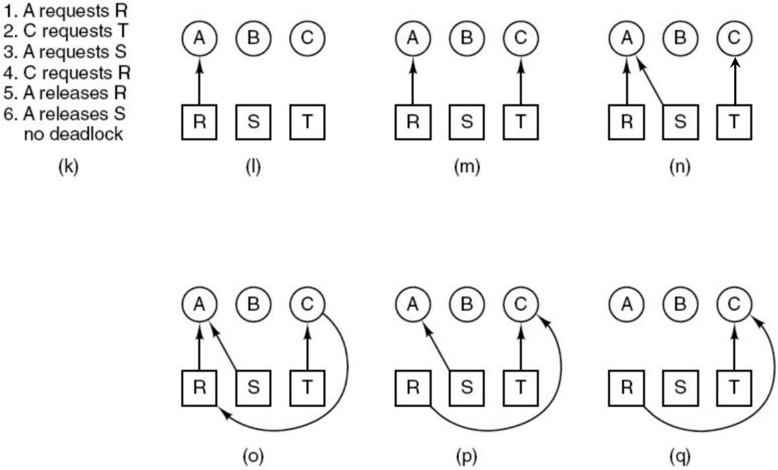
- A cycle in a resource allocation graph ➔ deadlock
- If A waits for S while holding R, and B waits for R while holding S, then



Deadlock Modeling

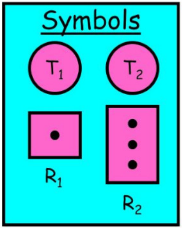


Deadlock Modeling



Multiple instance of a resource

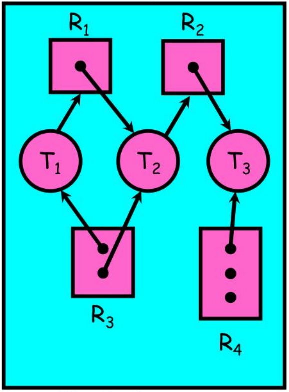
- System Model
  - A set of Threads $T_1, T_2, \ldots, T_n$
  - Resource types $R_1, R_2, \ldots, R_m$
    CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances.
  - Each thread utilizes a resource as follows:
    - **`Request() / Use() / Release()`**



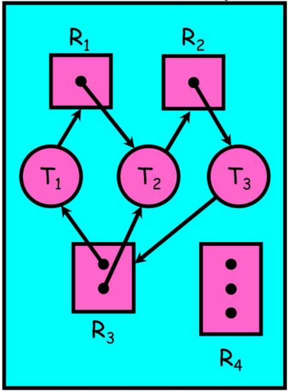
- Resource-Allocation Graph:
  - V is partitioned into two types:
    - $T = \{T_1, T_2, \ldots, T_n\}$, the set threads in the system.
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set of resource types in system
  - Request edge: directed edge $T_i \rightarrow R_j$
  - Assignment edge: directed edge $R_j \rightarrow T_i$

Resource Allocation Graph examples

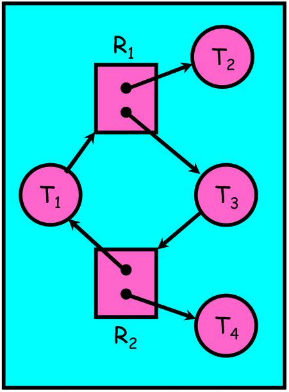
- Recall:
  - request edge – directed edge $T_1 \rightarrow R_j$
  - assignment edge – directed edge $R_j \rightarrow T_i$



Simple Resource Allocation Graph



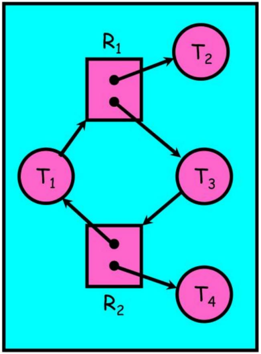
Allocation Graph With Deadlock



Allocation Graph With Cycle, but No Deadlock

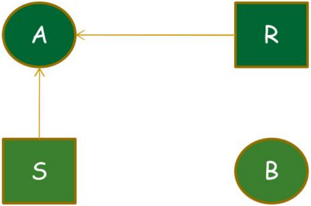
Necessary conditions for deadlock

- Mutual exclusion condition
- Hold and wait
- No resource preemption
- Circular waiting

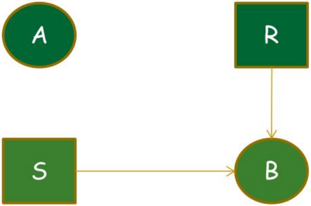


Eliminate competition for resources?

- If running A to completion and then running B, there will be no deadlock.
- Should CPU scheduling algorithm eliminate competition for resources?



First iteration



Second iteration

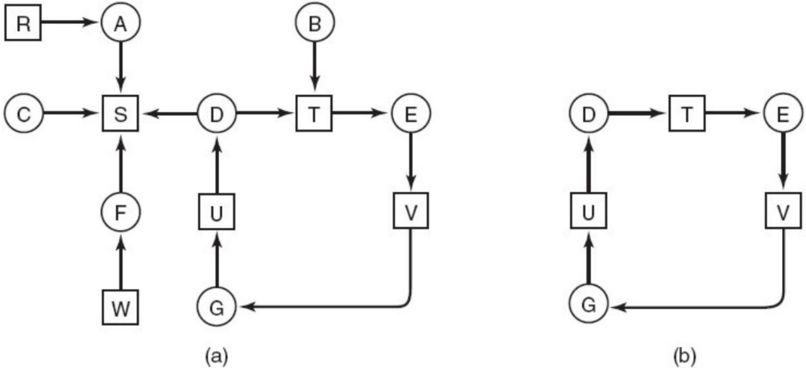
Outline

- What is deadlock?
- How can deadlock occur?
- **Strategies to deal with deadlocks?**
  - Ignore the problem
  - Detection and recovery
  - Dynamic avoidance
  - Prevention
- Examples

Strategies

- Ignore the problem
  - It is user's fault
- Detection and recovery
  - Fix the problem afterwards
- Dynamic avoidance
  - Careful allocation
- Prevention
  - Remove one of the four conditions

Deadlock detection with ONE resource of each type (1)



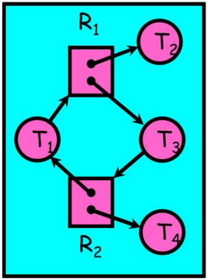
(a) A resource graph. (b) A cycle extracted from (a).

Deadlock detection algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let [X] represent an **m**-ary vector of non-negative integers (quantities of resources of each type), e.g. [2,2]:

    `[FreeResources]:` Current free resources each type, e.g. [0,0]
    `[Request`$_X$`]:` Current requests from thread X, e.g. [1,0] for T_1
    `[Alloc`$_X$`]:` Current resources held by thread X, [0,1] for T_1
  - See if tasks can eventually terminate on their own

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
   done = true
   Foreach node in UNFINISHED {
      if ([Request_node] <= [Avail]) {
         remove node from UNFINISHED
         [Avail] = [Avail] + [Alloc_node]
         done = false
      }
   }
} until(done)
```



  - Nodes left in UNFINISHED ⇒ deadlocked

Deadlock detection with MULTIPLE resource of each type

$$\begin{array}{cccc} \textit{Tape drives} & \textit{Plotters} & \textit{Scanners} & \textit{CD Roms} \end{array}$$

$$\mathbf{E} = (4 \quad 2 \quad 3 \quad 1)$$

Resources in existence

$$\begin{array}{cccc} \textit{Tape drives} & \textit{Plotters} & \textit{Scanners} & \textit{CD Roms} \end{array}$$

$$\mathbf{A} = (2 \quad 1 \quad 0 \quad 0)$$

Resources available

$$\mathbf{C} = \begin{array}{c} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix} \end{array} \begin{array}{l} \text{Thread 1} \\ \text{Thread 2} \\ \text{Thread 3} \end{array}$$

Current allocation matrix

$$\mathbf{R} = \begin{array}{c} \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} \text{Thread 1} \\ \text{Thread 2} \\ \text{Thread 3} \end{array}$$

Request matrix

Deadlock detection with MULTIPLE resource of each type

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (4 & 2 & 3 & 1) \end{array}$$

Resources in existence

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (2 & 2 & 2 & 0) \end{array}$$

Resources available

$$C = \begin{array}{c|l} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} & \begin{array}{l} \text{Thread 1} \\ \text{Thread 2} \\ \text{Thread 3} \end{array} \end{array}$$

Current allocation matrix

$$R = \begin{array}{c|l} \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} & \begin{array}{l} \text{Thread 1} \\ \text{Thread 2} \\ \text{Thread 3} \end{array} \end{array}$$

Request matrix

Deadlock detection with MULTIPLE resource of each type

$$E = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (4 & 2 & 3 & 1) \end{array}$$

Resources in existence

$$A = \begin{array}{cccc} \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ (4 & 2 & 2 & 1) \end{array}$$

Resources available

$$C = \begin{array}{cl} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} & \begin{array}{l} \text{Thread 1} \\ \text{Thread 2} \\ \text{Thread 3} \end{array} \end{array}$$

Current allocation matrix

$$R = \begin{array}{cl} \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} & \begin{array}{l} \text{Thread 1} \\ \text{Thread 2} \\ \text{Thread 3} \end{array} \end{array}$$

Request matrix

Deadlock detection with MULTIPLE resource of each type

$$\begin{array}{ccccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ E = (& 4 & 2 & 3 & 1\ ) \end{array}$$

Resources in existence

$$\begin{array}{ccccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ A = (& 4 & 2 & 3 & 1\ ) \end{array}$$

Resources available

$$C = \begin{array}{c} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} \text{Thread 1} \\ \text{Thread 2} \\ \text{Thread 3} \end{array}$$

Current allocation matrix

$$R = \begin{array}{c} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{array}{l} \text{Thread 1} \\ \text{Thread 2} \\ \text{Thread 3} \end{array}$$

Request matrix

Recovery

- Kill process/thread
  - Can you always do this?
- Preempt resources without killing threads
  - What if the resources are in a critical section?
- Roll back actions of deadlocked threads
  - Like transactions in databases

Outline

- What is deadlock?
- How can deadlock occur?
- **Strategies to deal with deadlocks?**
  - Ignore the problem
  - Detection and recovery
  - **Dynamic avoidance**
  - Prevention
- Examples

Avoidance

- Model
  - Each thread requests resources **one at a time**
- Safety conditions:
  - It is not deadlocked
    - Found scheduling order in which every thread can run to completion (even if all request their max resources)

Examples (single resource)

Total: 8

	Has	Max
P_1	2	6
P_2	2	3
P_3	3	5

Free: 1

	Has	Max
P_1	2	6
P_2	3	3
P_3	3	5

Free: 0

	Has	Max
P_1	2	6
P_2	0	0
P_3	3	5

Free: 3

	Has	Max
P_1	2	6
P_2	0	0
P_3	5	5

Free: 1

	Has	Max
P_1	2	6
P_2	0	0
P_3	0	0

Free: 6

	Has	Max
P_1	4	6
P_2	1	3
P_3	2	5

Free: 1

?

Banker's algorithm (multiple resources)

- Toward right idea:
  - State maximum resource needs in advance
  - Allow particular thread to proceed if:

    (available resources - #requested) $\geq$ max remaining that might be needed by any thread
- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - Evaluate each request and grant if threads are still deadlock free afterward
    - Technique: pretend each request is granted, then run deadlock detection algorithm, substituting ($[Max_{node}]-[Alloc_{node}] \leq [Avail]$) for ($[Request_{node}] \leq [Avail]$)
      Grant request if result is deadlock free (conservative!)
    - Keeps system in a "SAFE" state, i.e. there exists a sequence $\{T_1, T_2, \ldots T_n\}$ with T_1 requesting all remaining resources, finishing, then T_2 requesting all remaining resources, etc..

Safe state in the Banker's algorithm (single resources)

- Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

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

Free: 10

(a)

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

Free: 2

(b)

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

Free: 1

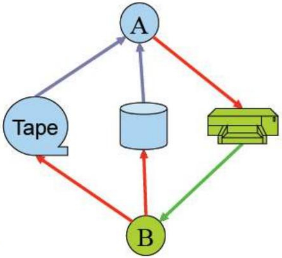
(c)

Outline

- What is deadlock?
- How can deadlock occur?
- **Strategies to deal with deadlocks?**
  - Ignore the problem
  - Detection and recovery
  - Dynamic avoidance
  - **Prevention**
- Examples

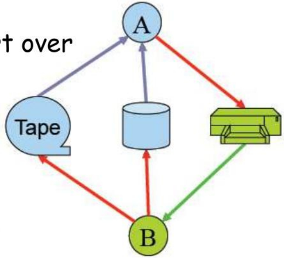
Prevention: avoid mutual exclusion

- Some can be made sharable
  - Read-only files, memory, etc
- Some resources are not physically sharable
  - Printer, tape, etc
  - Some can be virtualized by spooling
- What about the tape-disk-printer example?



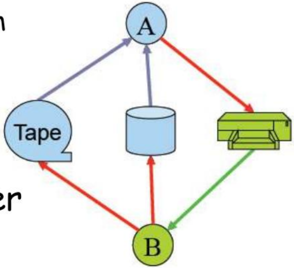
Prevention: avoid hold and wait

- Two-phase locking
  - Assumption
    - Processes know all resources they will need at the beginning
  - Phase I:
    - Try to lock all resources at the beginning
  - Phase II:
    - If successful, use the resources and release them
  - Otherwise, release all resources and start over
- Application
  - Telephone company's circuit switching
- What about the tape-disk-printer example?



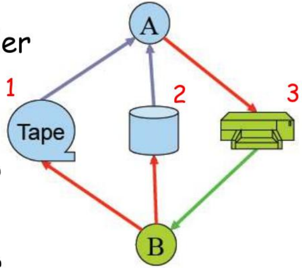
Prevention: allow preemption

- Make the scheduler be aware of resource allocation
  - Method
    - If the system cannot satisfy a request from a thread holding resources, preempt the thread and release its resources
    - Schedule a thread only if the system satisfies all resources
- What about the tape-disk-printer example?



Prevention: avoid circular wait

- Impose an order of requests for all resources
- Method
  - *Assign a unique id to each resource*
  - *All requests must be in ascending order of the ids*



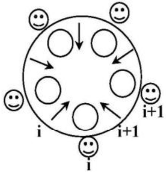
- What about the tape-disk-printer example?
- Does this method have no circular wait?

Outline

- What is deadlock?
- How can deadlock occur?
- Strategies to deal with deadlocks?
  - Ignore the problem
  - Detection and recovery
  - Dynamic avoidance
  - Prevention
- **Examples**
  - **Dining philosophers**

Dining philosophers 1

"Dining Philosophers"



•Each: need 2 forks to eat

•5 philosophers: 10 forks

•5 forks: 2 can eat concurrently

Things to observe:

•A fork can only be used by one at a time, please

•No deadlock, please

•No starving, please

•Concurrent eating, please

s(i): One semaphore per fork to be used in **mutex** style P-V

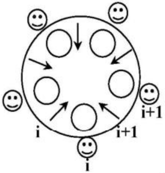
$\mathbf{T_i}$

$\mathbf{T_i}$

$\mathbf{T_i}$

Dining philosophers 1

"Dining Philosophers"



- Each: need 2 forks to eat
- 5 philosophers: 10 forks
- 5 forks: 2 can eat concurrently

Things to observe:

- A fork can only be used by one at a time, please
- No deadlock, please
- No starving, please
- Concurrent eating, please

s(i): One semaphore per fork to be used in **mutex** style P-V

Mutex on whole table:
- *1 can eat at a time*

```
P(mutex);
eat;
V(mutex);
```

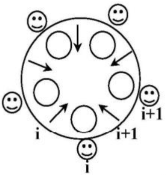
T_i

T_i

T_i

Dining philosophers 1

"Dining Philosophers"

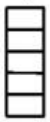


- Each: need 2 forks to eat
- 5 philosophers: 10 forks
- 5 forks: 2 can eat concurrently

Things to observe:

- A fork can only be used by one at a time, please
- No deadlock, please
- No starving, please
- Concurrent eating, please

s



s(i): One semaphore per fork to be used in **mutex** style P-V

Mutex on whole table:	P(mutex);
•*1 can eat at a time*	eat; V(mutex);

T_i

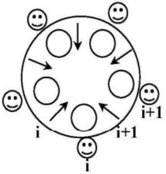
Get L; Get R;	P(s(i)); P(s(i+1)); eat; V(s(i+1)); V(s(i));
•*Deadlock possible*	
S(i) = 1 initially	

T_i

T_i

Dining philosophers 1

“Dining Philosophers”



- Each: need 2 forks to eat
- 5 philosophers: 10 forks
- 5 forks: 2 can eat concurrently

Things to observe:

- A fork can only be used by one at a time, please
- No deadlock, please
- No starving, please
- Concurrent eating, please

s(i): One semaphore per fork to be used in **mutex** style P-V

Mutex on whole table: P(mutex); eat; V(mutex); T_i

- *1 can eat at a time*

Get L; Get R; T_i

- *Deadlock possible*

S(i) = 1 initially

```
P(s(i));
  P(s(i+1));
    eat;
  V(s(i+1));
V(s(i));
```

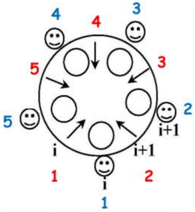
Avoid hold-and-wait

Get L; Get R if free else Put L; T_i

- *Starvation possible*

Dining philosophers 1

Dining Philosophers



Can we in a simple way do better than this one?

Get L; Get R;

•Deadlock possible

```
P(s(i));
   P(s(i+1));
      eat;
   V(s(i+1));
V(s(i));
```

S(i) = 1 initially

Avoid circular wait

•Remove the danger of circular waiting (deadlock)

•T1-T4: Get L; Get R;

•T5: Get R; Get L;

T_1, T_2, T_3, T_4:

```
P(s(i)):
   P(s(i+1));
      <eat>
   V(s(i+1));
V(s(i));
```

T_5

```
P(s(1));
   P(s(5));
      <eat>
   V(s(5));
V(s((1));
```

•**Non-symmetric solution. Still quite elegant**

Summary: strategies to deal with deadlocks

- Ignore the problem
  - It is user's fault
- Detection and recovery
  - Fix the problem afterwards
- Dynamic avoidance
  - Careful allocation
- Prevention (Negate one of the four conditions)
  - Avoid the mutual exclusion
  - Avoid the hold and wait
  - Allow preemption
  - Avoid the circular wait

References

- A. S. Tanenbaum, Modern Operating Systems.
- A. Silberschatz et. al., Operating System Concepts.

Thanks for your attention!

Questions?