Lesson 5

Deadlocks

Ch 6 [Stall 05]

Problem
Dining Philosophers
Deadlock occurrence
Deadlock detection
Deadlock prevention
Deadlock avoidance

Motivational Example

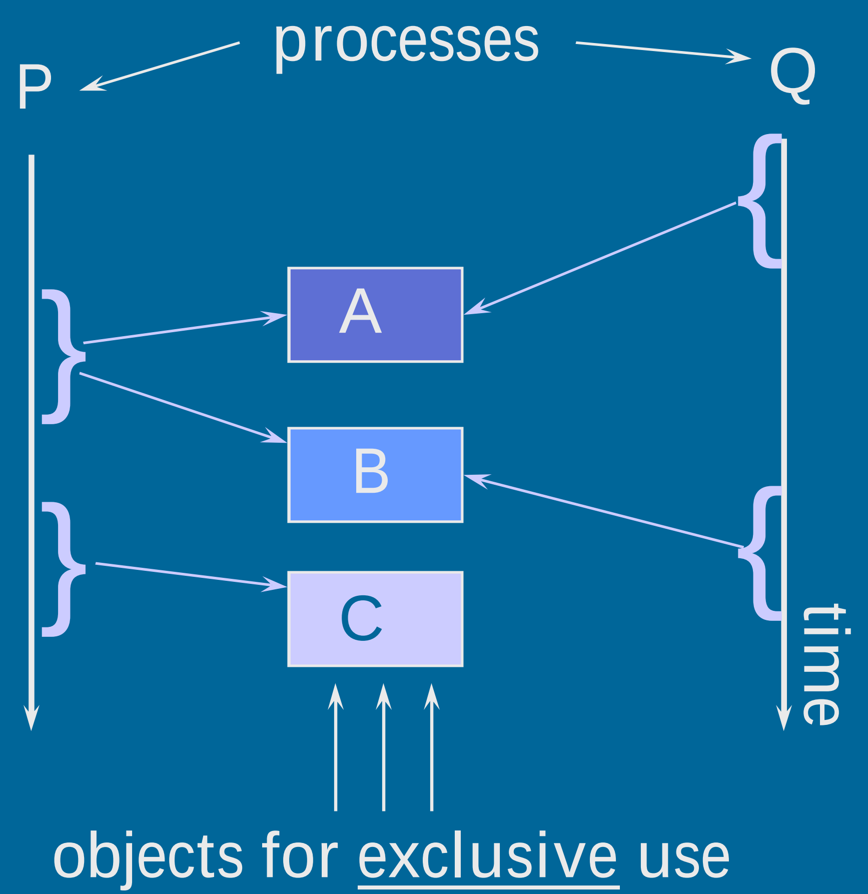
- New possible laptop for CS dept use
  - Lenovo 400, dual-core, Intel Centrino 2 technology
  - Ubuntu Linux 8.10
- Wakeup from suspend/hibernation, freezes often

  http://ubuntuforums.org/showthread.php?t=959712
- Read, study, experiment – some 15 hours?
  - No network?, at home/work?, various units?, …., ???
  - Problem with Gnome desktop, not with KDE, …, ???
- Could two processors cause it?
  - Shut down one processor during hibernation/wakeup
  - Wakeup works fine now
- Same problem with many new laptops running Linux
  - All new laptops with Intel Centrino 2 with same Linux driver?
- Concurrency problem in display driver startup?
  - Bug not found yet, use 1-cpu work-around

http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commitdiff;h=70740d6c93030b339b4ad17fd58ee135dfc13913

(search "i915_enable_vblank" …)

Deadlock: Background



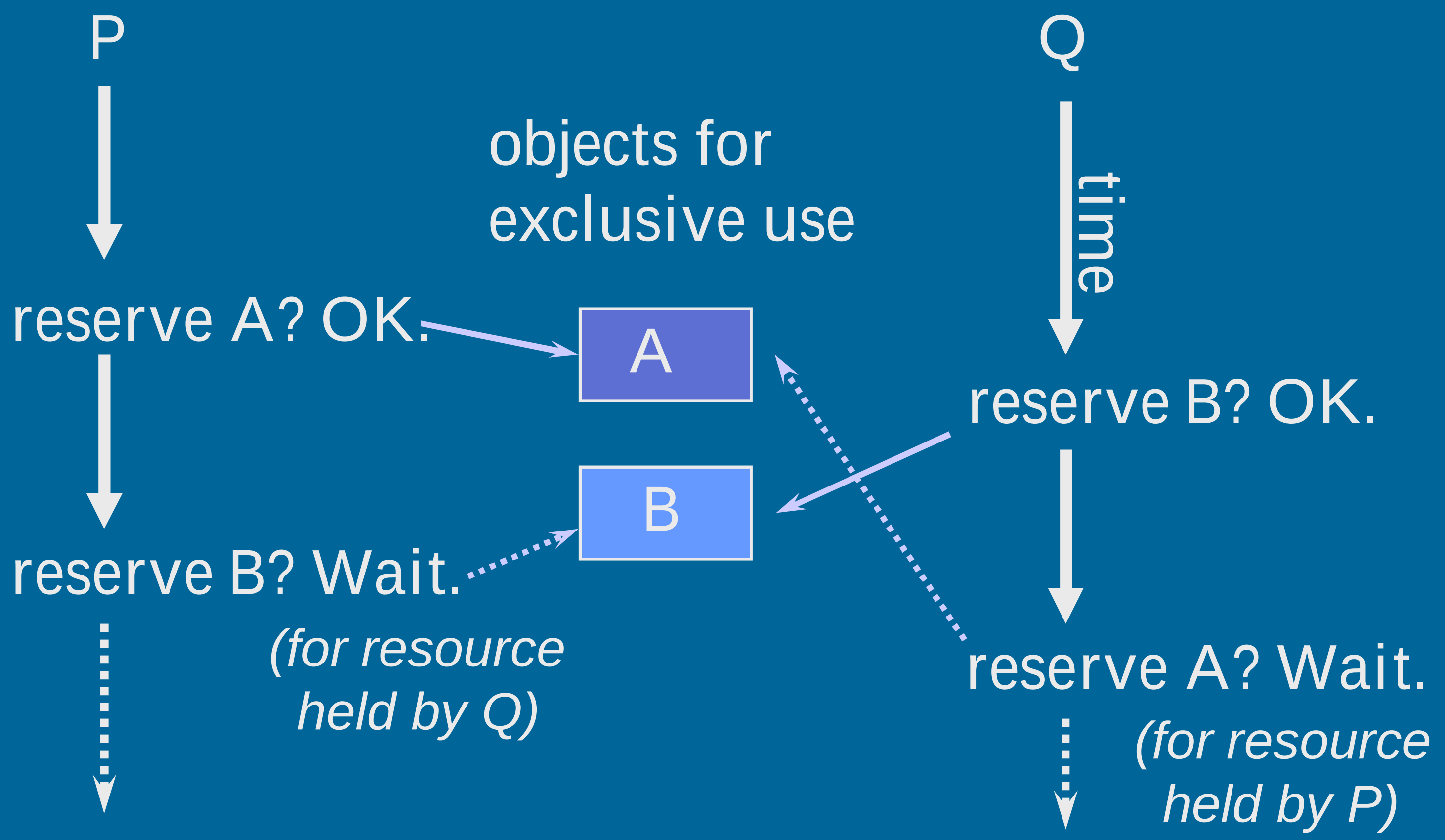
object?

buffer,
page,
user input,
critic. section
disk driver,
scanner,
message,
...

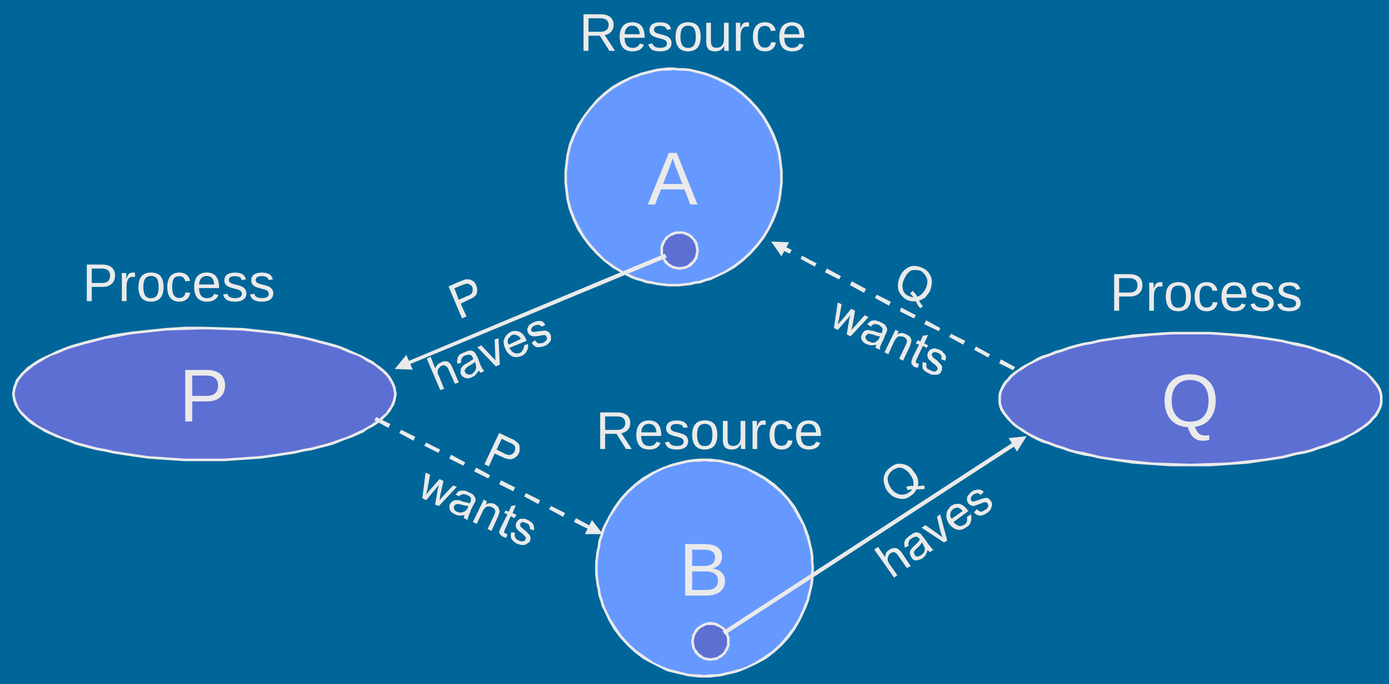
Basic problem: a process needs multiple objects at the same time

Mutex: competition for one object (critical section)

Deadlock: an Example (10)

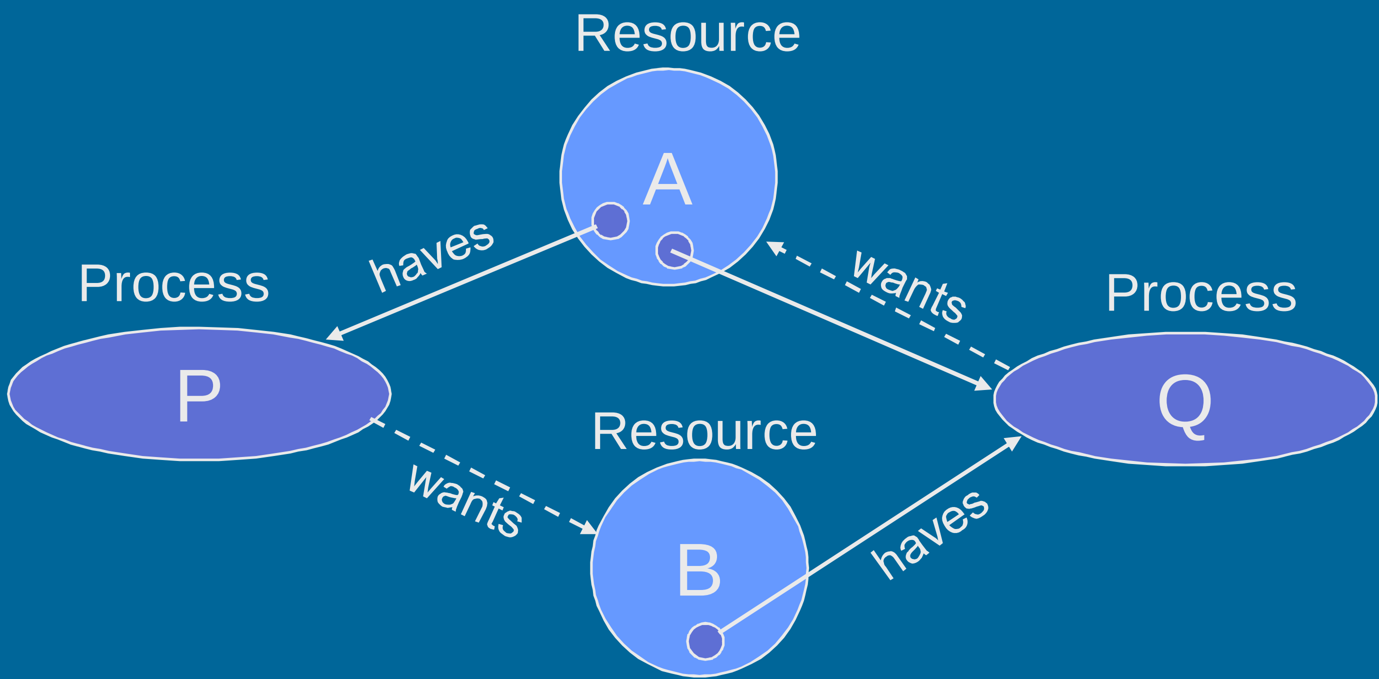


Resource Reservation Graph



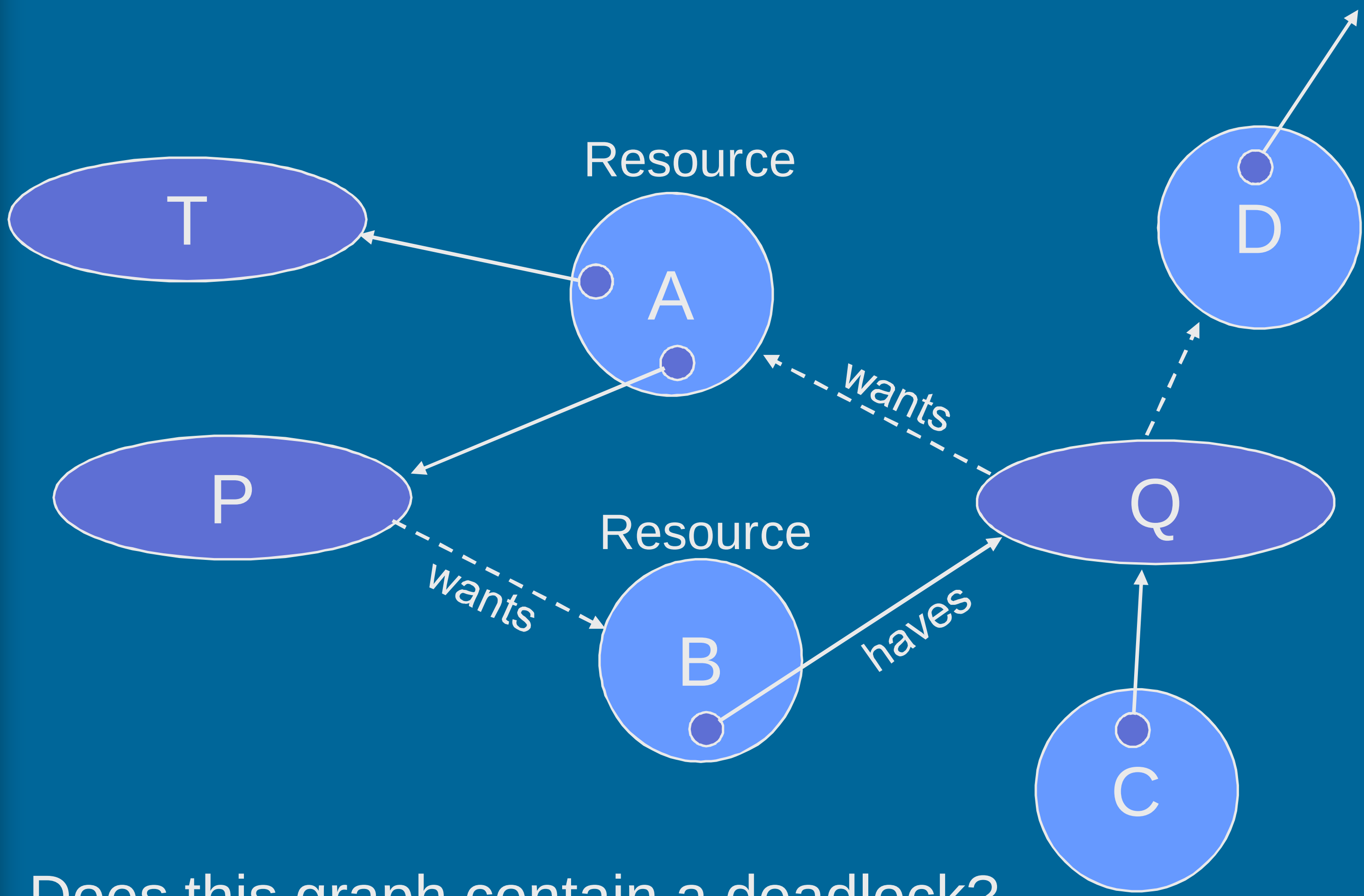
Deadlock cycle in resource reservation graph

Resource Reservation Graph



Does this graph contain a deadlock?

Resource Reservation Graph

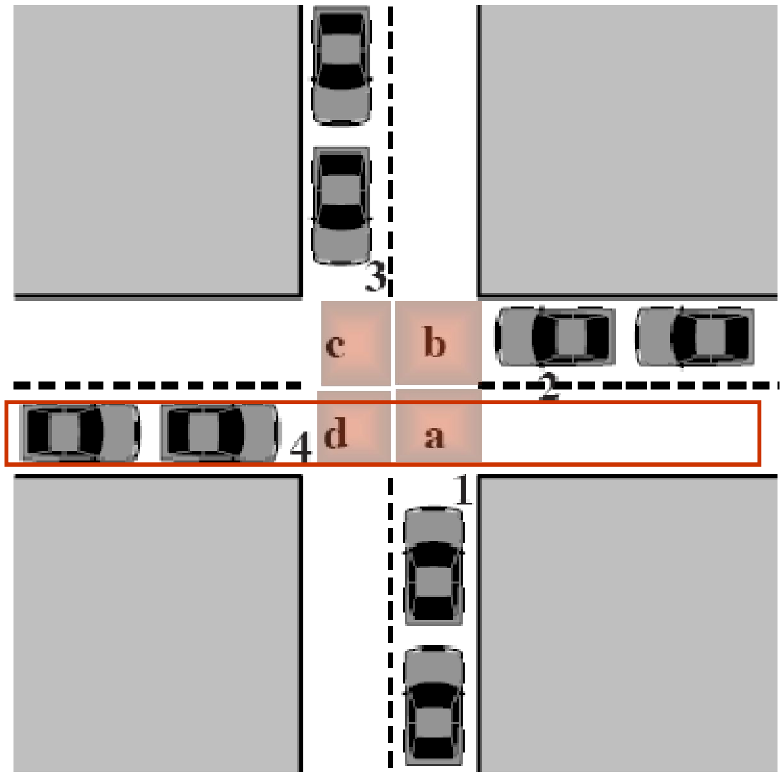


Does this graph contain a deadlock?

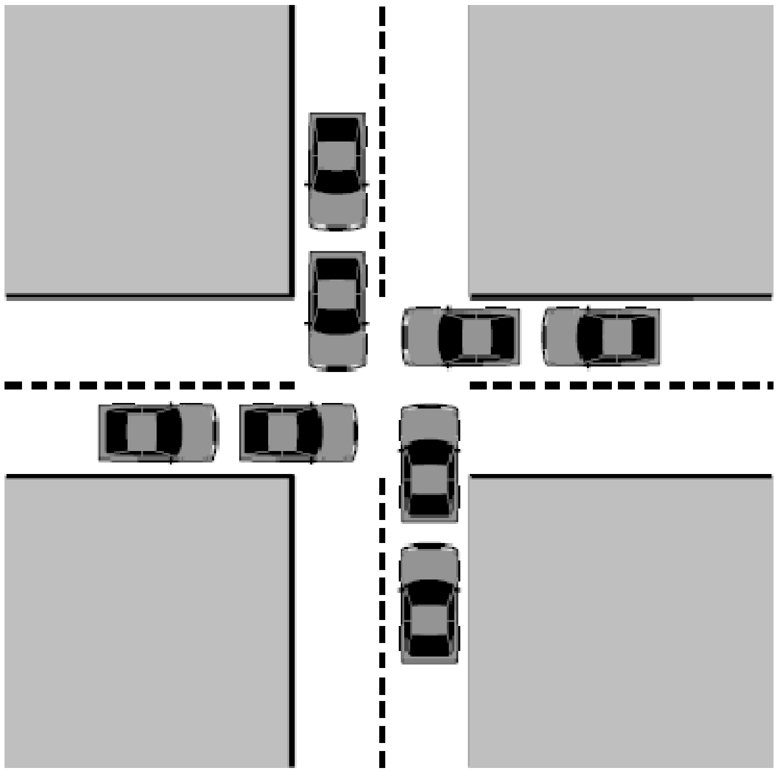
Gridlock

(Fig. 6.1 [Stal06])

Real life gridlock: http://img209.imageshack.us/img209/5781/deadlocknajkcomafarialibh3.jpg



(a) Deadlock possible



(b) Deadlock

- Processes: cars 1, 2, 3 and 4
- Resources: quadrants a, b, c, d
  - Car 4 needs quadrants d and a (exclusive use for each)

Consequences

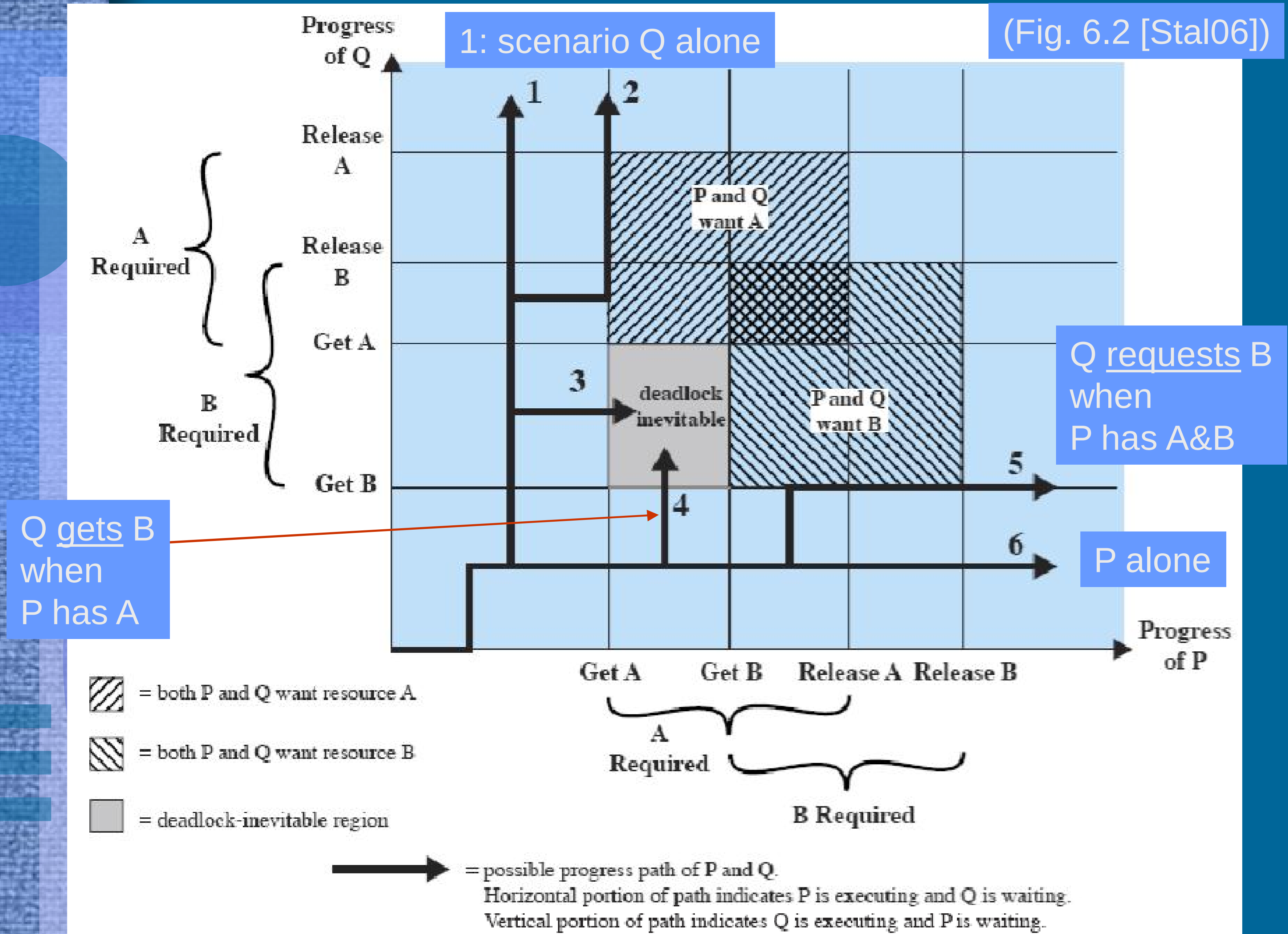
- The processes do not advance
  - Cars do not move
- Resources remain reserved
  - Cpu? Street quadrant?
  - Memory? I/O-devices?
  - Logical resources (semaphores, critical sections, ...)?
- The computation fails
  - Execution never finishes?
    - One application?
  - The system crashes? Traffic flow becomes zero?

Resources

- Reusable resources
  - Limited number or amount
  - Wait for it, allocate it, deallocate (free) it
  - Memory, buffer space, intersection quadrant
  - Critical section code segment execution
  - …
- Consumable resources
  - Unlimited number or amount
  - Created and consumed
  - Someone may create it, wait for it, destroy it
  - Message, interrupt, turn for critical section
  - …

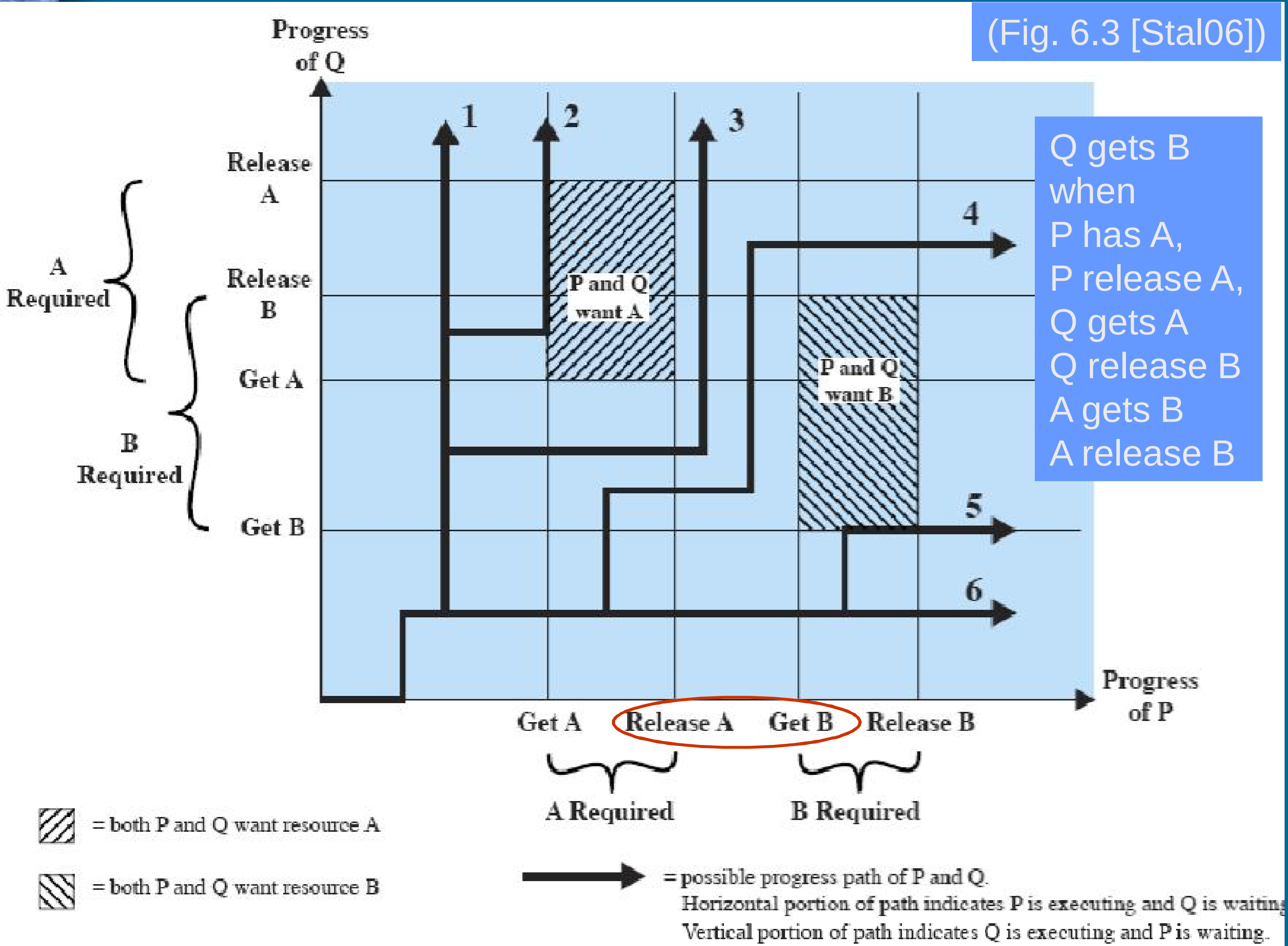
uudelleen-käytettävä resurssi

kulutettava resurssi



1: scenario Q alone
(Fig. 6.2 [Stal06])
Progress of Q
Release A
Release B
Get A
Get B
A Required
B Required
1
2
3
4
5
6
P and Q want A
deadlock inevitable
P and Q want B
Q requests B when P has A&B
Q gets B when P has A
P alone
Progress of P
Get A
Get B
Release A
Release B
A Required
B Required
= both P and Q want resource A
= both P and Q want resource B
= deadlock-inevitable region
= possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

(Fig. 6.3 [Stal06])



Q gets B when
P has A,
P release A,
Q gets A
Q release B
A gets B
A release B

Definitions

- Deadlock

  lukkiintuminen

  – Eternal wait in blocked state
  – Does not block processor (unless one resource is processor)

- Livelock

  ”elolukko”

  – Two or more processes continuously change their state (execute/wait) as response to the other process(es), but never advance to real work
  – E.g., ping-pong ”you first – no, you first - ...”
    - two processes alternate offering the turn to each other - no useful work is started
  – Consumes processor time

- Starvation

  nälkiintyminen

  – the process will never get its turn
  – E.g., in ready-to-run queue, but never scheduled

Deadlock Problems

- How to know if deadlock <u>exists</u>?
  - How to <u>locate</u> deadlocked processes?
- How to <u>prevent</u> deadlocks?
- How to know if deadlock <u>might occur</u>?
- How to <u>break</u> deadlocks?
  - Without too much damage?
  - Automatically?
- How to <u>prove</u> that your solution is free of deadlocks?

Good Deadlock Solution

- Prevents deadlocks in advance, or
  detects them, breaks them, and fixes the system
- Small overhead
- Smallest possible waiting times
- Does not slow down computations when no danger exists
- Does not block unnecessarily any process when the resource wanted is available

Conditions for Deadlock (6)

- Three policy conditions
  - S1. Mutual exclusion
    - one user of any resource at a time (not just code)
  - S2. Hold and wait
    - a process may hold allocated resources while waiting for others
  - S3. No preemption
    - resource can not be forcibly removed from a process holding it
- A dynamic (execution time) condition takes place
  - D1. Circular wait: a closed chain of processes exists, each process holds at least one resource needed by the next process in chain

Coffman, 1971

yksi käyttäjä

pidä ja odota

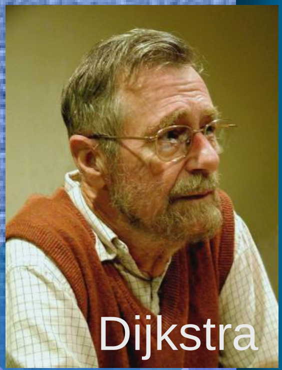
ei keskeytettävissä

kehäodotus

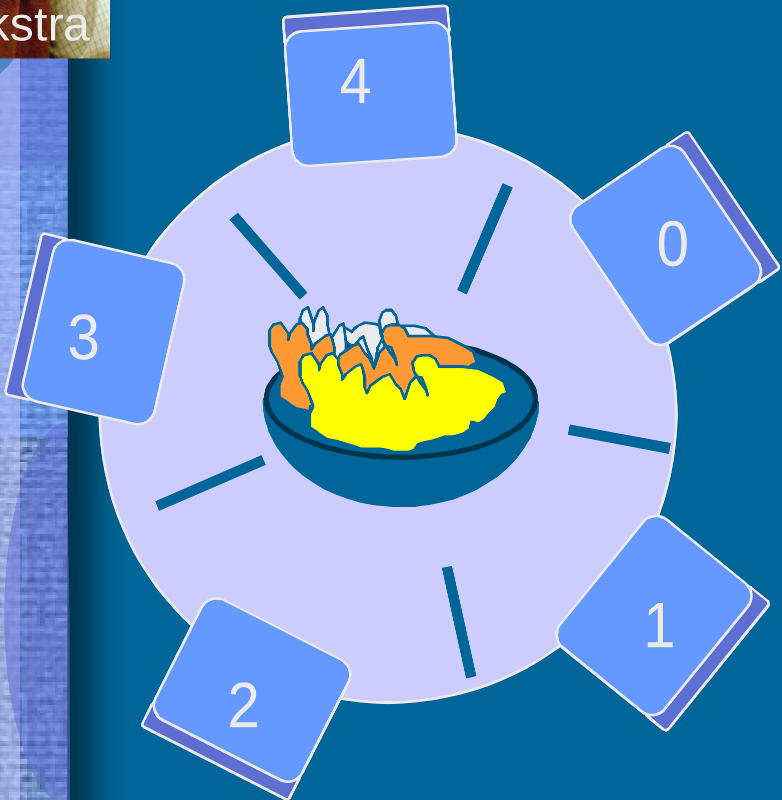
E.g., slide 5

E.G. Coffman

http://portal.acm.org/citation.cfm?id=356588&coll=GUIDE&dl=GUIDE&CFID=4442763&CFTOKEN=75849639&ret=1#Fulltext



Dining Philosophers (Dijkstra)



Philosopher:

- think
- take two forks ...
- … one from each side
- eat rice until satisfied
- return the forks

Problem:

- how to reserve the forks without causing
  - deadlock
  - starvation

and everybody may be present

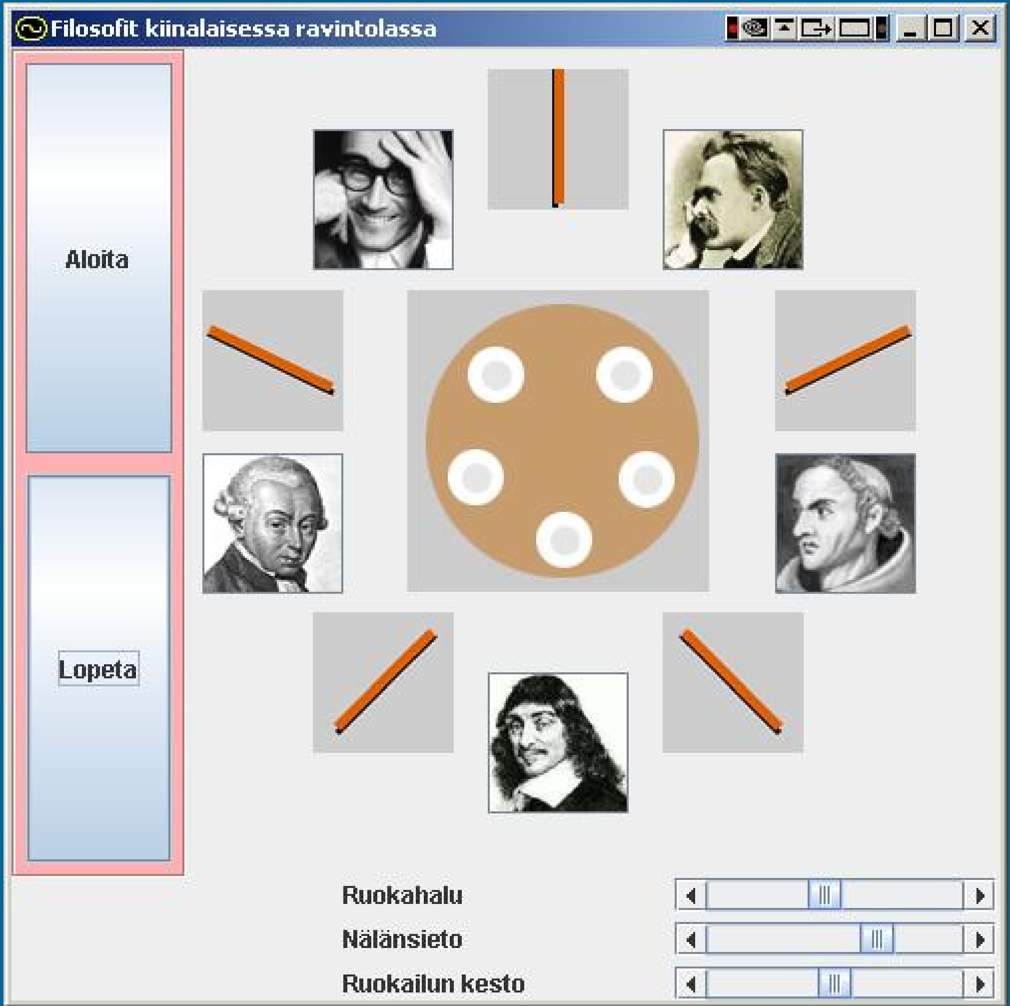
See philosopher art in web

http://images.google.fi/images?q=dinig%20philosophers&ie=UTF-8&oe=utf-8&rls=org.mozilla:en-US:official&client=firefox-a&um=1&sa=N&tab=wi

Dining Philosophers in Java

- Tapio Lehtomäki, MikroBitti
- Load program from course schedule page
- Modify paths in script philosophers.bat and run it
- Modify program for homework?
  - Next year?

Lehtomaki.zip



http://www.cs.helsinki.fi/u/kerola/rio/Lehtomaki/Lehtomaki.zip

Trivial Solution #1

(Fig. 6.12 [Stal06])

```
/* program     diningphilosophers */
semaphore fork [5] = {1}; /* mutex, one at a time */
int i;
void philosopher (int i)
{
     while (true)
     {
          think();
          wait (fork[i]);                   /* left fork */
          wait (fork [(i+1) mod 5]);/* right fork */
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
     }
```

- Possible deadlock – not good
  - All 5 grab left fork "at the same time"

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};  /* only 4 at a time, 5th waits */
int i;
void philosopher (int I)
{
    while (true)
    {
     think();
     wait (room);
     wait (fork[i]);
     wait (fork [(i+1) mod 5]);
     eat();
     signal (fork [(i+1) mod 5]);
     signal (fork[i]);
     signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
         philosopher (3), philosopher (4));
}
```

(Fig. 6.13 [Stal06])

- No deadlock, no starvation, and no company while eating – not good
- Waiting when resources are available – not good

which scenario?

Deadlock Prevention

- How to prevent deadlock occurrence in advance?
- Deadlock possible only when all 4 conditions are met:
  - S1. Mutual exclusion — poissulkemistarve
  - S2. Hold and wait — pidä ja odota
  - S3. No preemption — ei saa ottaa pois kesken kaiken
  - D1. Circular wait — kehäodotus
- Solution: disallow <u>any</u> <u>one</u> of the conditions
  - S1, S2, S3, or D1?
  - Which is possible to disallow?
  - Which is easiest to disallow?

Disallow S1 (mutual exclusion)

- Can not do always
  - There are reasons for mutual exclusion!
    - Can not split philosophers fork into 2 resources
- Can do sometimes
  - Too high granularity blocks too much
    - Resource *room* in trivial solution #2
  - Finer granularity allows parallelism
    - Smaller areas, parallel usage, more locks
    - More administration to manage more locks
    - Too fine granularity may cause too much administration work
  - Normal design approach in data bases, for example
- Get more resources, avoid mutex competition?
  - Buy another fork for each philosopher?

Disallow S2 (hold and wait)

- Request all needed resources at one time
- Wait until all can be granted simultaneously
  - Can lead to starvation
    - Reserve both forks at once (simultaneous wait!)
    - Neighbouring philosophers eat all the time alternating

- Inefficient
  - long wait for resources (to be used much later?)

  - worst case reservation (long wait period for resources which are possibly needed - who knows?)
- Difficult/impossible to implement?
  - advance knowledge: resources of all possible execution paths of all related modules ...

Disallow S3 (no preemption)

- Allow preemption in crisis
- Release of resources => fallback to some earlier state
  - Initial reservation of these resources
  - Fall back to specific checkpoint
  - Checkpoint <u>must have been saved</u> earlier
  - Must know <u>when</u> to fall back!
- OK, if the system has been designed for this
  - Practical, if saving the state is cheap and the chance of deadlock is to be considered
  - Standard procedure for transaction processing
-
```
wait (fork[i]);
if "all forks taken" then
    "remove fork" from philosopher [i⊕1]
wait (fork[i⊕1])
```
  - What will philosopher i⊕1 do now? Think? Eat? Die?

Disallow D1 (circular wait)

- Linear ordering of resources
  - Make reservations in this order only – no loops!
- Pessimistic approach – prevent “loops” in advance
  - Advance knowledge of resource requirements needed
  - Reserve all at once in given order
  - Prepare for ”worst case” behavior

Forks in global ascending order
philosophers 0, 1, 2, 3:

```
    wait (fork[i]);
    wait (fork[i+1]);
```

last philosopher 4:

```
    wait (fork[0]);
    wait (fork[4]);
```

- Optimistic approach – worry only at the last moment
  - Reservation dynamically as needed (but in order)
  - Reservation conflict => restart from some earlier stage
    - Must have earlier state saved somewhere

Deadlock Detection and Recovery (4)

- Let the system run until deadlock problem occurs
  - "Detect deadlock existance"
  - "Locate deadlock and fix the system"
- Detection is not trivial:
  - Blocked group of processes is deadlocked? or
  - Blocked group is just waiting for an external event?
- Recovery
  - Detection is first needed
  - Fallback to a previous state (does it exist?)
  - <u>Killing</u> one or more members of the deadlocked group
    - Must be able to do it <u>without overall system damage</u>
- Needed: information about resource allocation
  - In a form suitable for deadlock detection!

Resource Allocation

- Processes Pi ∈ P1..Pn
- Resources (or objects) Rj ∈ R1..Rm
- Number of resources of type Rj
  - total amount of resources **R** $= (r_1, ..., r_m)$
  - currently free resources **V** $= (v_1, ..., v_m)$
- Allocated resources (allocation matrix)
  - **A** $= [\, a_{ij} \,]$, "process Pi has a_{ij} units of resource Rj"
- Outstanding requests (request matrix)
  - **Q** $= [\, q_{ij} \,]$, "process Pi requests q_{ij} units of resource Rj"

How many R4 resources exists?

R1	R2	R3	R4	R5
?	1	1	?	1

Resource vector **R**

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector **V**

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix **Q**

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix **A**

Which resources are now free?

Who has now R4?

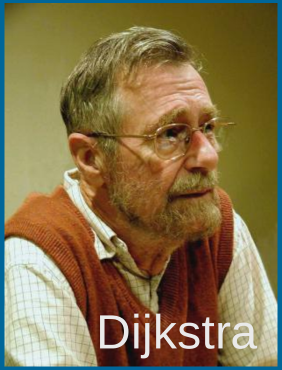
P2 has now R1 and R2,

P2 wants now R3 and R5

Is there now a deadlock or not?

(Fig. 6.10 [Stal06])

Deadlock Detection (Dijkstra) (4)



1. Find a (any) process that could terminate
   - All of its current resource requests can be satisfied
2. Assume now that
   a. This process terminates, and
   b. It releases all of its resources
3. Repeat 1&2 until can not find any more such processes
4. If any processes still exist, they are deadlocked
   a. They all each need something
   b. The process holding that something is waiting for something else
      - That process can not advance and release it

Deadlock Detection Algorithm (DDA)

DL1. [*Remove the processes with no resources*]
Mark all processes with null rows in **A**.

DL2. [*Initialize counters for available objects*]
Initialize a working vector **W** = **V**

DL3. [*Search for a process* Pi *which could get all resources it requires*]
Search for an unmarked row *i* such that

$$q_{ij} \leq w_j \qquad j = 1..n$$

If none is found terminate the algorithm.

DL4. [*Increase* **W** *with the resources of the chosen process*]
Set **W** = **W**+A_{i*} i.e. $w_j = w_j + a_{ij}$ *when* $j = 1..n$
<u>Mark process Pi</u> and return to step DL3.

When the algorithm terminates, unmarked processes correspond to deadlocked processes. Why?

Example: Initial state

	allocation matrix A	request matrix Q
row 1:	1 0 1 1 0	0 1 0 0 1
2:	1 1 0 0 0	0 0 1 0 1
3:	0 0 0 1 0	0 0 0 0 1
4:	0 0 0 0 0	1 0 1 0 1

E.g., ”process 2 has resources 1 & 2, and it wants resources 3 & 5”

all resources R: 2 1 1 2 1

free resources V: 0 0 0 0 1

Who holds resource 4?

Which resources are free?

(Fig. 6.10 [Stal06])

Deadlock or not?

What now?

Example: Deadlock Detection

A

1 0 1 1 0
1 1 0 0 0
~~0 0 0 1 0~~
~~0 0 0 0 0~~

DL4: mark

DL1: mark

Q

0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1

DL3: no request can be satisfied:
$\nexists\, i \;\; \forall j: \;\; q_{ij} \le w_j$
$\rightarrow$ Deadlock

DL3: this request can be satisfied:
$q_{3j} \le w_j \;\; \forall j$

all resources R: 2 1 1 2 1

free resources V: 0 0 0 0 1

DL2: copy

may become free W: 0 0 0 0 1

+

DL4: new W 0 0 0 1 1

Example: Deadlock Detection (phases)

A				
1	0	1	1	0
1	1	0	0	0
0	0	0	1	0
0	0	0	0	0

Q				
0	1	0	0	1
0	0	1	0	1
0	0	0	0	1
1	0	1	0	1

all resources R: 2 1 1 2 1

free resources V: 0 0 0 0 1

may become free W:

Example: Deadlock Detection (phases)

A

1	0	1	1	0
1	1	0	0	0
0	0	0	1	0
~~0~~	~~0~~	~~0~~	~~0~~	~~0~~

DL1: mark

Q

0	1	0	0	1
0	0	1	0	1
0	0	0	0	1
1	0	1	0	1

all resources R: 2 1 1 2 1

free resources V: 0 0 0 0 1

may become free W:

Example: Deadlock Detection (phases)

A

$$\begin{matrix}1 & 0 & 1 & 1 & 0\\ 1 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0\\ \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0}\end{matrix}$$

DL1: mark

Q

$$\begin{matrix}0 & 1 & 0 & 0 & 1\\ 0 & 0 & 1 & 0 & 1\\ 0 & 0 & 0 & 0 & 1\\ 1 & 0 & 1 & 0 & 1\end{matrix}$$

all resources R: 2 1 1 2 1

free resources V: 0 0 0 0 1

DL2: copy

may become free W: 0 0 0 0 1

Example: Deadlock Detection (phases)

A

$$\begin{matrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} \end{matrix}$$

DL1: mark

Q

$$\begin{matrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ \boxed{0 \; 0 \; 0 \; 0 \; 1} \\ 1 & 0 & 1 & 0 & 1 \end{matrix}$$

DL3: this request can be satisfied: $q_{3j} \leq w_j \;\; \forall j$

all resources R: 2 1 1 2 1

free resources V: 0 0 0 0 1

may become free W: $\boxed{0\;0\;0\;0\;1}$

DL2: copy

Example: Deadlock Detection (phases)

A

1 0 1 1 0
1 1 0 0 0
$\boxed{0\ 0\ 0\ 1\ 0}$
~~0 0 0 0 0~~

DL1: mark

Q

0 1 0 0 1
0 0 1 0 1
$\boxed{0\ 0\ 0\ 0\ 1}$
1 0 1 0 1

DL3: this request can be satisfied: $q_{3j} \leq w_j \ \ \forall j$

all resources R: 2 1 1 2 1

free resources V: 0 0 0 0 1

DL2: copy

may become free W: $\boxed{0\ 0\ 0\ 0\ 1}$

DL4: new W + 0 0 0 1 1

Example: Deadlock Detection (phases)

A

1 0 1 1 0
1 1 0 0 0
~~0 0 0 1 0~~ DL4: mark
~~0 0 0 0 0~~ DL1: mark

Q

0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1

DL3: this request can be satisfied:
$q_{3j} \leq w_j \ \forall j$

all resources R: 2 1 1 2 1

free resources V: 0 0 0 0 1

DL2: copy

may become free W: 0 0 0 0 1

DL4: new W + 0 0 0 1 1

Example: Deadlock Detection (phases)

A

1 0 1 1 0
1 1 0 0 0
~~0 0 0 1 0~~
~~0 0 0 0 0~~

DL4: mark

DL1: mark

Q

0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1

DL3: no request can be satisfied:
$\nexists\, i \;\; \forall j: \quad q_{ij} \leq w_j$
$\rightarrow$ Deadlock

DL3: this request can be satisfied:
$q_{3j} \leq w_j \;\; \forall j$

all resources R: 2 1 1 2 1

free resources V: 0 0 0 0 1

DL2: copy

may become free W: 0 0 0 0 1

+

DL4: new W 0 0 0 1 1

Example: Breaking Deadlocks

- Processes P1 and P2 are in deadlock
  - What next?
- Abort P1 and P2
  - Most common solution
- Rollback P1 and P2 to previous safe state, and try again
  - Rollback states <u>must exist</u>
  - May deadlock again (or may not!)
- Abort P1 because it is less important
  - Must have some basis for selection
  - Who makes the decision? Automatic?
- Preempt R3 from P1
  - Must be <u>able to</u> preempt (easy if R3 is CPU?)
  - Must know <u>what</u> to preempt from <u>whom</u>
  - <u>How many</u> resources need preemption?

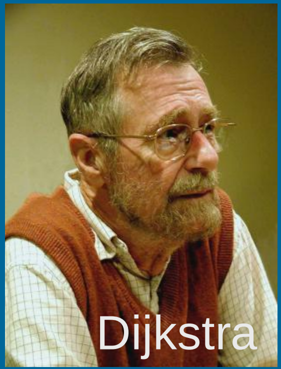
Deadlock Avoidance with DDA

- Use Dijstra's algorithm to avoid deadlocks in advance?
- Banker's Algorithm

Pankkiirin algoritmi

  - Originally for one resource (money)
  - Why "Banker's"?
    - "Ensure that a bank never allocates its available cash so that it can no longer satisfy the needs of all its customers"

Banker's Algorithm (6)



- Keep state information on resources allocated to each process
- Keep state information on number of resources each process might still allocate
- For each resource allocation, first find an ordering which allows processes to terminate, if that allocation is made
  - Assume that allocation is made and then use DDA to find out if the system remains in a safe state even in the worst case
  - If deadlock is possible, reject resource request
  - If deadlock is not possible, grant resource request

Deadlock Avoidance with Banker's Algorithm (6)

Matrices as before, and some more

- For each process: the maximum needs of resources
  - **C** = $[c_{ij}]$, "Pi may request c_{ij} units of Rj"
- The current hypothesis of resources in use
  - **A'** = $[a'_{ij}]$, "if this allocation is made, Pi would have a'_{ij} units of Rj"

Possible allocation

- The current hypothesis of future maximum demands
  - **Q'** = $[q'_{ij}]$, "Pi could still request q'_{ij} units of Rj"
    Q' = **C** - **A'**

Possible request

- Apply DDA to **A'** and **Q'**
  - If no deadlock possible, grant resource request

Banker's Algorithm Example

Allocation A

	R1	R2	R3	R4	R5
P1	0	1	0	0	0
P2	1	1	0	0	0
P3	0	0	1	0	1
P4	0	0	1	1	0

Requests Q

	R1	R2	R3	R4	R5
P1	1	0	0	0	0
P2	0	0	0	0	1
P3	0	0	0	1	0
P4	0	0	0	0	1

Max allocation C

	R1	R2	R3	R4	R5
P1	2	1	0	1	0
P2	1	1	0	0	1
P3	1	0	1	1	1
P4	0	2	1	1	1

	R1	R2	R3	R4	R5
Resources R	2	3	2	1	2
Available V	1	1	0	0	1

(Fig. 16.11, Bacon, Concurrent Systems, 1993)

P1 requests R1. Is request granted?

Could system deadlock, if R1 is granted?

Banker's Algorithm Example (7)

If P1 request for R1 approved, can deadlock occur?

Possible allocation A'

	R1	R2	R3	R4	R5
P1	~~1~~	~~1~~	~~0~~	~~0~~	~~0~~
P2	~~1~~	~~1~~	~~0~~	~~0~~	~~0~~
P3	~~0~~	~~0~~	~~1~~	~~0~~	~~1~~
P4	~~0~~	~~0~~	~~1~~	~~1~~	~~0~~

Q' = C – A'

Possible requests Q'

	R1	R2	R3	R4	R5
P1	1	0	0	1	0
P2	0	0	0	0	1
P3	1	0	0	1	0
P4	0	2	0	0	1

Max allocation C

	R1	R2	R3	R4	R5
P1	2	1	0	1	0
P2	1	1	0	0	1
P3	1	0	1	1	1
P4	0	2	1	1	1

Resources R

R1	R2	R3	R4	R5
2	3	2	1	2

Available V

R1	R2	R3	R4	R5
1	1	0	0	1

Possibly available V'

R1	R2	R3	R4	R5
0	1	0	0	1

	R1	R2	R3	R4	R5	
W	0	1	0	0	1	**DDA-4: mark P2**
W	1	2	0	0	1	**DDA-4: mark P4**
W	1	2	1	1	1	**DDA-4: mark P1**
W	2	3	1	1	1	**DDA-4: mark P3**
W	2	3	2	1	2	

DDA: no deadlock, allocation request OK

Avoidance: Problems

- Each allocation: a considerable overhead
  - Run Banker's algorithm for 20 processes and 100 resources?
- Knowledge of maximum needs
  - In advance?
    - An educated guess? Worst case?
  - Dynamically?
    - Even more overhead
- A safe allocation does not always exist
  - An unsafe state does not always lead to deadlock
  - You may want to take a risk!

Another Banker's Algorithm example: B. Gray, Univ. of Idaho

http://www.if.uidaho.edu/~bgray/classes/cs341/doc/banker.html

Summary

- Difficult real problem
- Can detect deadlocks

  Dijkstra's DDA
  - Need specific data on resource usage
- Difficult to break deadlocks
  - How will killing processes affect the system?
- Can prevent deadlocks

  Bankers
  - Prevent any one of those four conditions
    - E.g., reserve resources always in given order
  - Can analyze system at resource reservation time to see whether deadlock might result
    - Complex and expensive