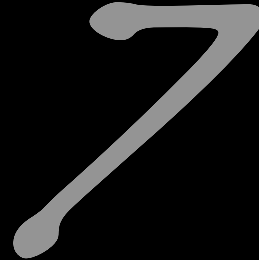
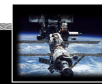
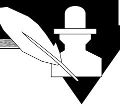


Safety & Liveness

Uwe R. Zimmer
The Australian National University

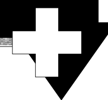




References for this chapter

[Ben-Ari90]
M. Ben-Ari
Principles of Concurrent and Distributed Programming
1990
Prentice-Hall,
ISBN 0-13-711821-X

[Bacon98]
J. Bacon
Concurrent Systems
1998 (2nd Edition)
Addison Wesley Longman Ltd,
ISBN 0-201-17767-6



Models and Terminology

Correctness in concurrent systems

Extended concepts of correctness in concurrent systems:

¬ Termination is often not intended or even considered a failure

- Safety properties:

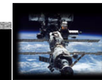
$$(P(I) \wedge Processes(I, S)) \Rightarrow \Box Q(I, S)$$

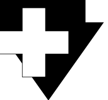
where $\Box Q$ means that Q does *always* hold

- Liveness properties:

$$(P(I) \wedge Processes(I, S)) \Rightarrow \Diamond Q(I, S)$$

where $\Diamond Q$ means that Q does *eventually* hold (and will then stay true)
and S is the current state of the concurrent system





Models and Terminology

Correctness in concurrent systems

- Liveness properties:

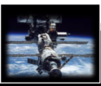
$$(P(I) \wedge Processes(I, S)) \Rightarrow \Diamond Q(I, S)$$

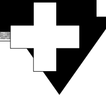
where $\Diamond Q$ means that Q does *eventually* hold (and will then stay true)

Examples:

- Requests need eventually to be completed
- The state of the system needs eventually be displayed to the outside
- No part of the system is to be delayed forever (fairness)

☞ Interesting liveness properties can be extremely hard to be proven





one central liveness property: Fairness

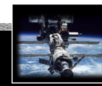
- Liveness properties:

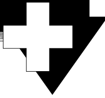
$$(P(I) \wedge Processes(I, S)) \Rightarrow \Diamond Q(I, S)$$

where $\Diamond Q$ means that Q does *eventually* hold (and will then stay true)

Fairness (as a means to avoid starvation):

- **Weak fairness**: $\Diamond \Box R \Rightarrow \Diamond G$
  resource will eventually be granted, if a process requests continually
- **Strong fairness**: $\Box \Diamond R_i \Rightarrow \Diamond G$
  resource will eventually be granted, if a process requests infinitely often
- **Linear waiting**: resource will be granted
  before any other process had the same resource granted more than once.
- **First-in, first-out**: resource will be granted
  before any other process which applied for the same resource at a later point in time.





Correctness in concurrent systems

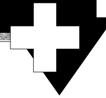
- Safety properties:

$$(P(I) \wedge Processes(I, S)) \Rightarrow \Box Q(I, S)$$

where $\Box Q$ means that Q does *always* hold

Examples:

- Mutual exclusion (no resource collisions)
- Absence of deadlocks
  (and other forms of 'silent death' and 'freeze' conditions)
- Specified responsiveness or free capabilities
  (typical in real-time / embedded systems or server applications)

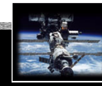


Synchronization may lead to

☞ *DEADLOCKS*

(avoidance / prevention of those is one central safety property)

… a closer look on deadlocks
and what can be done about them …



Reserving resources in reverse order

```
var reserve_1, reserve_2: semaphore := 1;
```

```
process P1;
   statement X;

   wait (reserve_1);
   wait (reserve_2);
      statement Y; - employ resources
   signal (reserve_2);
   signal (reserve_1);

   statement Z;
end P1;
```

```
process P2;
   statement A;

   wait (reserve_2);
   wait (reserve_1);
      statement B; - employ resources
   signal (reserve_1);
   signal (reserve_2);

   statement C;
end P2;
```

Sequence of operations : [A | X] ➟ {[B ➟ Y] xor [Y ➟ B]} ➟ [C | Z]
or : [A | X] ➟ deadlocked!

Circular dependencies

```
var reserve_1, reserve_2, reserve_3: semaphore := 1;

process P1;                process P2;                process P3;
   statement X;               statement A;               statement K;

   wait (reserve_1);          wait (reserve_2);          wait (reserve_3);
   wait (reserve_2);          wait (reserve_3);          wait (reserve_1);
      statement Y;               statement B;               statement L;
   signal (reserve_2);        signal (reserve_3);        signal (reserve_1);
   signal (reserve_1);        signal (reserve_2);        signal (reserve_3);

   statement Z;               statement C;               statement M;
end P1;                    end P2;                    end P3;
```

Sequence of operations : [A | X | K] ➟ {[B ➟ Y ➟ L] xor …} ➟ [C | Z | M]
or : [A | X | K] ➟ deadlocked!

Necessary deadlock conditions:

1. **Mutual exclusion**:
   resources cannot be used simultaneously
2. **Hold and wait**:
   a process applies for a resource, while it is holding another resource (sequential requests)
3. **No pre-emption**:
   resources cannot be pre-empted; only the process itself can release resources
4. **Circular wait**:
   a ring list of processes exists, where every process waits for release of a resource by the next one

☞ system ***may*** be deadlocked, if ***all*** these conditions apply!

Deadlock strategies:

1. Ignorance
   ☞ Kill unresponsive processes
2. Deadlock detection & recovery
   ☞ find deadlocked processes and recover the system in a coordinated way
3. Deadlock avoidance
   ☞ the resulting system state is checked before any resources are actually assigned
4. Deadlock prevention
   ☞ the system prevents deadlocks by its structure

Deadlock prevention

(remove one of the four deadlock conditions)

1. **Mutual exclusion**:
   Applicable to specific cases only; usually this can only be removed by replication of resources.
2. **Hold and wait**:
   Processes are forced to allocate all their required resources at once,
   often at the time of admittance to the main dispatcher – done in many static realtime-systems.
3. **No pre-emption**:
   If the current state of a resource can be stored and restored easily, then they can be pre-empted.
   Usually resources are pre-empted from processes, which are currently not ready to run.
4. **Circular wait**:
   A circular wait can be avoided by a global ordering of all resources, e.g. resources can only be requested in a specific order – hard to maintain in a dynamic system configuration.

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

$RAG = \{V, E\}$; vertices and edges

$V = P \cup R$; vertices are processes or resource types:

$P = \{P_1, P_2, \dots, P_n\}$; processes

$R = \{R_1, R_2, \dots R_k\}$; resource types

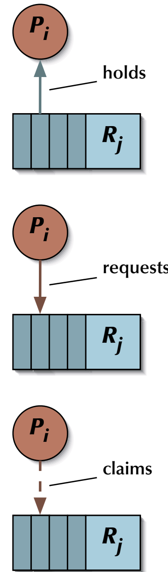
$E = E_r \cup E_a \cup E_c$; claims, requests and assignments

$E_c = \{P_i \rightarrow R_j, \dots\}$; claims

$E_r = \{P_i \rightarrow R_j, \dots\}$; requests

$E_a = \{R_i \rightarrow P_j, \dots\}$; assignments

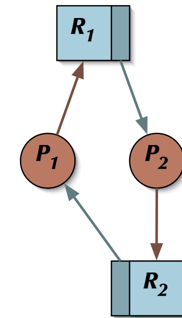
Note: a resource may have more than one instance



Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

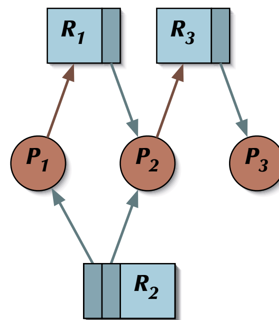
the two process, reverse allocation deadlock:



Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

no, there is no circular dependency



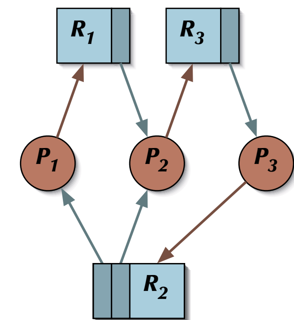
Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

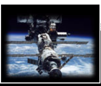
yes, there are circular dependencies:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

as well as: $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

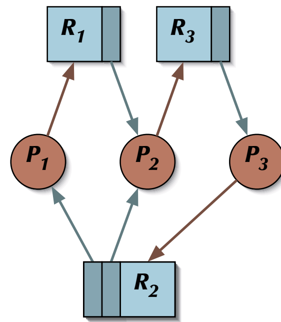


☞ ***IF some processes are deadlocked, THEN there are cycles in the resource allocation graph***



Edge Chasing

(Chandy, Misra & Haas ☞ distributed version)



∀ blocking process:

- send probe containing three process id's:

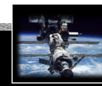
  [the blocked, the sending, the receiving process]

∀ blocked process receiving a probe:

- propagate the probe to the process holding the resource, which this process requests
  (while updating the second and third proc.-id's.)

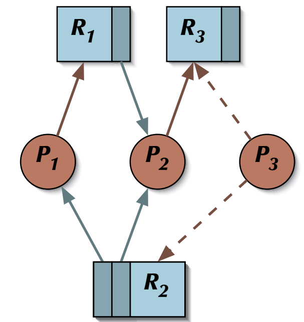
∀ blocking process receiving its own probe:

☞ possible deadlock detected!



Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

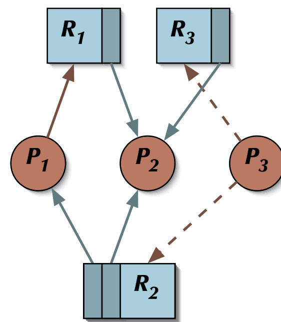


Assuming all claims of P_3 are known in advance,

☞ Could the deadlock situation be avoided?

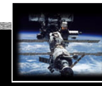
Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



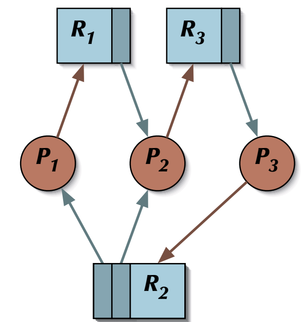
yes, when resources are assigned so that there are no resulting circular dependencies:

☞ in this case: assign R_3 to P_2 (instead of P_3)



Resource Allocation Graphs

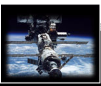
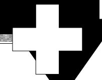
(Silberschatz, Galvin & Gagne)



$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

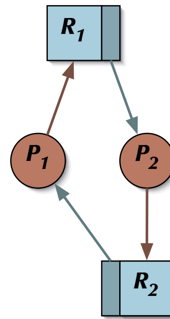
as well as: $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

☞ ***ARE some processes deadlocked, IF there are cycles in the resource allocation graph?***

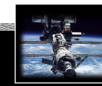
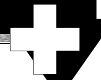
Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



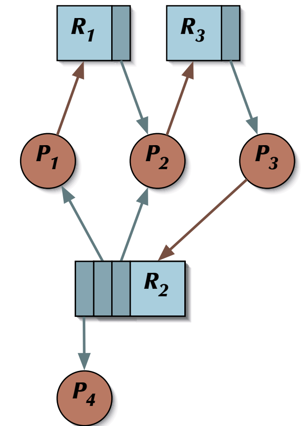
yes,
if there is only one instance per resource type:

☞ ***IF there are cycles in the resource allocation graph AND there is only one instance per resource type, THEN some processes are deadlocked!***

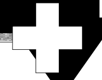
Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)



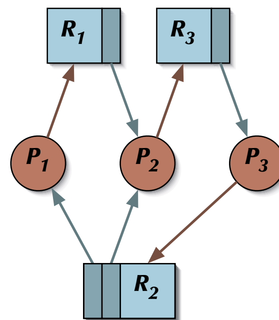
no,
if there is more than one instance per resource type:

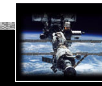
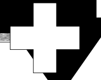
☞ ***IF there are cycles in the resource allocation graph AND there is more than one instance per resource type, THEN some processes may be deadlocked!***



How to detect deadlocks in the general case?

(of multiple instances per resource)



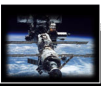
Banker's algorithm

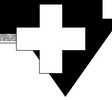
There are n processes and m resource types in the system. Let $i \in 1 \ldots n$ and $j \in 1 \ldots m$:

- *Allocated*$[i, j]$
  ☞ the number of resources of type j allocated by process i.
- *Free*$[j]$
  ☞ the number of available resources of type j.
- *Claimed*$[i, j]$
  ☞ the number of resources of type j required by process i to complete *eventually*.
- *Request*$[i, j]$
  ☞ the number of *currently* requested resources of type j by process i.

Temporary variables:

- *Completed*$[i]$: boolean vector indicating processes, which may complete right now.
- *Simulated_Free*$[j]$: available resources, if some processes complete and de-allocate.





Banker's algorithm

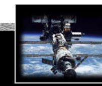
Checking for a deadlock situation

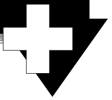
1. $Simulated_Free \Leftarrow Free$; $\forall i$: $Completed[i] \Leftarrow False$

2. **While** $\exists i$: $\neg Completed[i]$
   and $\forall j$: $Requested[i, j] < Simulated_Free[j]$ **do:** {request *i* can be granted}

   $\forall j$: $Simulated_Free[j] \Leftarrow Simulated_Free[j] + Allocated[i, j]$
   $Completed[i] \Leftarrow True$

3. **If** $\forall i$: $Completed[i]$ **then** the system is **deadlock-free**!
   (otherwise all processes *i* with $Completed[i] = False$ are deadlocked)





Banker's algorithm

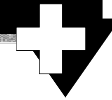
Checking the current system state

1. $Simulated_Free \Leftarrow Free$; $\forall i$: $Completed[i] \Leftarrow False$

2. **While** $\exists i$: $\neg Completed[i]$
   and $\forall j$: $Claimed[i, j] < Simulated_Free[j]$ **do:** {meaning process *i* can complete}

   $\forall j$: $Simulated_Free[j] \Leftarrow Simulated_Free[j] + Allocated[i, j]$
   $Completed[i] \Leftarrow True$

3. **If** $\forall i$: $Completed[i]$ **then** the system is **safe**!
   (e.g. no process is currently deadlocked and no process can be deadlocked in any future state)



Banker's algorithm

Checking the validity of a resource request

```
If (Request < Claimed) and (Request < Free) then

   Free      := Free      - Request;
   Claimed   := Claimed   - Request;
   Allocated := Allocated + Request;
```

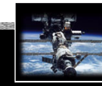
☞ ***Apply system state check (as above)***

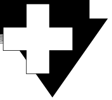
```
   If System_is_safe then
```

☞ ***Actually grant request***

```
   else
      -- restore former system state (Free, Claimed, Allocated)

   end if;
end if;
```



Deadlock detection / prevention

☞ Distributed version?

- Most resources are assigned to a local group of processes.

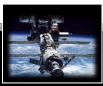
☞ Split the system into nodes

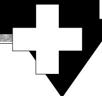
☞ Organize them as hierarchical trees or other topologies

☞ Check for deadlocks locally
☞ **find local deadlocks immediately**

☞ Exchange information about blocked tasks occasionally
☞ **detect global deadlocks eventually**

Menasce & Muntz – Ho & Ramamoorthy





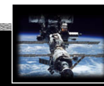
Deadlock recovery

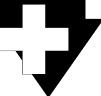
☞ Stop or restart one or multiple of the deadlocked processes and reclaim its resources

☞ Pre-empt one of the involved resources (and restore an earlier state of the victim process)

Deadlock recovery does not deal with the source of the problem!
(the system may deadlock again right away)

☞ use deadlock prevention or deadlock avoidance instead





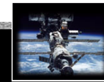
Deadlocks

- **Ignorance & recovery**
  - ☞ 'kill some seemingly persistently blocked processes from time to time' (exasperation)
- **Deadlock detection & recovery**
  - ☞ multiple methods for detection, e.g. resource allocation graphs, Banker's algorithm
  - ☞ recovery is mostly 'ugly'
- **Deadlock avoidance**
  - ☞ check system safety before allocating resources, e.g. Banker's algorithm
- **Deadlock prevention**
  - ☞ eliminate one of the pre-conditions for deadlocks

Terminology

Reliability ::=
measure of success with which a system conforms to its specification
or
low failure rate.

Failure ::= deviation of a system from its specification
Error ::= system state which lead to failures
Fault ::= the reason for an error



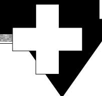
Faults on different levels

- Inconsistent or inadequate specification

  ☞ frequent source for disastrous faults
- Software design errors

  ☞ frequent source for disastrous faults
- Component & communication system failures

  ☞ rare and mostly predictable



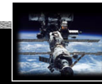
Faults in the logic domain

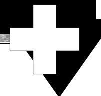
- Non-termination / -completion

  ☞ systems frozen in a deadlock state, blocked for missing input, or in infinite loop
- Value overruns, other inconsistent states

  ☞ sometimes caught by the run-time environment
- Wrong results

  ☞ wrong implementation with respect to the specification





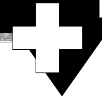
Faults in the time domain

- Transient faults

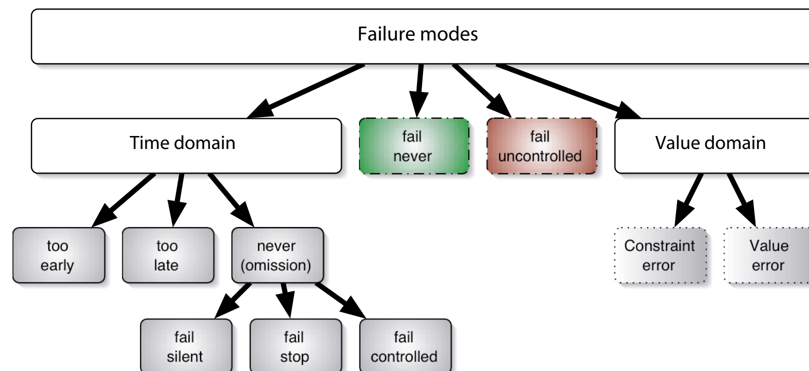
  ☞ many communication system failures, electric interference, etc.
- Intermittent faults

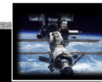
  ☞ transient errors which occur more than once (e.g. overheating effects)
- Permanent faults

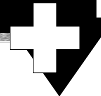
  ☞ stay in the system until they are repaired by some means



Observable failures states



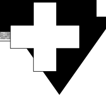




Fault prevention, avoidance, removal, …

and / or

☞ *Fault tolerance*



Fault tolerance

- Full fault tolerance

  the system continues to operate in the presence of 'foreseeable' error conditions without any significant failures — also this might induct a reduced operation period.

- Graceful degradation (fail soft)

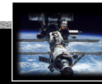
  the system continues to operate in the presence of 'foreseeable' error conditions, accepting a partial loss of functionality or performance.

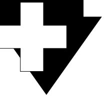
- Fail safe

  the system halts and maintains its integrity

☞ Full fault tolerance is not maintainable for an infinite operation time!

☞ Graceful degradation might have multiple levels of reduced functionality.





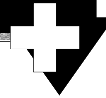
Atomic operations

Definitions given in different scenarios:

An operation is atomic if the processes performing it …

- … *are not aware of the existence of any other active process*, and *no other active process is aware of the activity of the processes* during the time the processes are performing the action.
- … *do not communicate* with other processes while the action is being performed.
- … *cannot detect any outside state change* and *do not reveal their own state changes* until the action is complete.

☞ … can be considered to be *indivisible and instantaneous*.



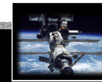
Atomic operations

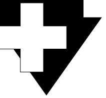
Important implications:

☞ An atomic operation …

- … is either performed fully, or not at all.
- … is declared as failed, if any part of the operation fails

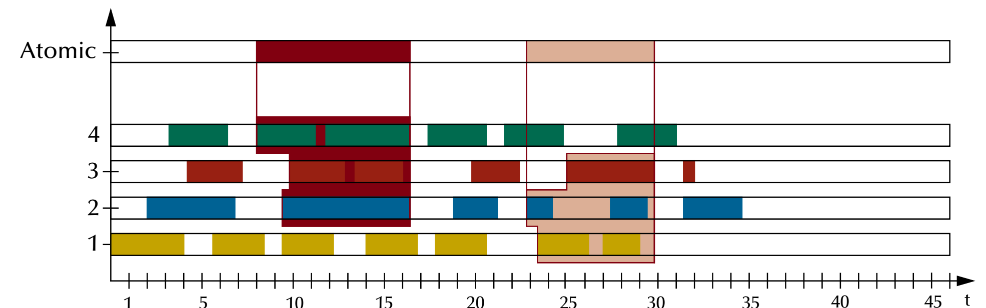
  (and everything is reset to the original state).

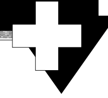




Atomic operations

Time-lines:





Idempotent operations

Definition:

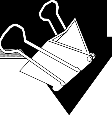
An operation is idempotent if …

- … the observable effects of the operation are *identical* after executing it *once* and after executing it *multiple times*.

Observations:

- Idempotent operations are often atomic, but do not need to be.
- Atomic operations do not need to be idempotent.

© 2008 Uwe R. Zimmer, The Australian National University



Safety & Liveness

- **Liveness**
  - Fairness
- **Safety**
  - Deadlock detection
  - Deadlock avoidance
  - Deadlock prevention
- **Failure modes**
  - Definitions, fault sources and basic fault tolerance
- **Atomic & Idempotent operations**
  - Definitions & implications

© 2008 Uwe R. Zimmer, The Australian National University