



Safety & Liveness

Liveness
Fairness

Liveness properties: $(P() \wedge \text{Processes } (I, S)) \Rightarrow \diamond Q(I, S)$
where $\diamond Q$ means that Q does eventually hold and will then stay true)

- Fairness** (as a means to avoid starvation) Resources will be granted ...
- **Weak fairness:** $\diamond \text{Req} \Rightarrow \diamond G \dots$ eventually, if a process requests continually.
 - **Strong fairness:** $\diamond \text{Req} \Rightarrow \diamond G \dots$ eventually, if a process requests infinitely often.
 - **Linear waiting:** $\diamond \text{Req} \Rightarrow \diamond G \dots$ before any other process had the same resource granted more than once (common fairness in distributed systems).
 - **First-in, first-out:** $\diamond \text{Req} \Rightarrow \diamond G \dots$ before any other process which applied for the same resource at a later point in time (common fairness in single-mode systems).

Safety & Liveness

Towards synchronization
Circular dependencies

```

var reserve_1, reserve_2, reserve_3 : semaphore := 1;
process P1;
statement X;
wait (reserve_1);
wait (reserve_2);
statement Y;
signal (reserve_2);
signal (reserve_1);
end P1;
process P2;
statement A;
wait (reserve_2);
wait (reserve_3);
statement B;
signal (reserve_3);
statement C;
end P2;
process P3;
statement K;
wait (reserve_3);
wait (reserve_1);
statement L;
signal (reserve_1);
signal (reserve_3);
end P3;
Sequence of operations  $A \rightarrow B \rightarrow C; X \rightarrow Y \rightarrow Z; K \rightarrow M; L \rightarrow M;$ 
or  $[A \mid X] \parallel [B \mid Y] \parallel [C \mid M] \parallel [K \mid L] \parallel [Z] \parallel [M]$ 
followed by a deadlock situation.

```

Safety & Liveness

Deadlocks

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.

References for this chapter

[Bert96] Bertsekas, M., Gallager, P., Forouzan, B., Kulkarni, S., Muthukrishnan, S., Panigrahy, A., Ravi, Y., Srikant, R., Walther, J., Whitt, W., Zang, L., and Zou, Y. *Introduction to Probability*. Cambridge University Press, 1996.

[Chandy983] Chandy, K.M., Lynch, N.A., and Lam, S.T. *Parallel Algorithms for Distributed Systems*. Cambridge University Press, 1998.

[Lynch93] Lynch, N.A. *Dist. Alg.*. Morgan Kaufmann, 1993.

[Lynch96] Lynch, N.A. *Dist. Alg.*. Morgan Kaufmann, 1996.

[Lynch98] Lynch, N.A. *Dist. Alg.*. Morgan Kaufmann, 1998.

[Silberschatz2001] Silberschatz, Abraham, Galvin, Peter, & Gagne, Greg. *Operating Systems: Concepts & Design*. John Wiley & Sons, Inc., 2001.

Safety & Liveness

Revisiting

Correctness concepts in concurrent systems

Safety properties: $(P() \wedge \text{Processes } (I, S)) \Rightarrow \square Q(I, S)$
where $\square Q$ means that Q does always hold

- Examples:**
- **Mutual exclusion** (no resource collisions) *is* **has been addressed**
 - **Absence of deadlocks** *is* **to be addressed now** (and other forms of silent death and "freeze" conditions)
 - **Specified responsiveness** or free capabilities *is* **Real-time systems** (typical in real-time / embedded systems or server applications)

Safety & Liveness

Deadlocks

Necessary deadlock conditions:

1. **Mutual exclusion:** resources cannot be used simultaneously.

Repetition

Correctness concepts in concurrent systems

Extended concepts of correctness in concurrent systems:
- Termination is often not intended or even considered a failure

- Safety properties:** $(P() \wedge \text{Processes } (I, S)) \Rightarrow \square \square Q(I, S)$
where $\square \square Q$ means that Q does always hold
- Liveness properties:** $(P() \wedge \text{Processes } (I, S)) \Rightarrow \diamond \square Q(I, S)$
where $\diamond \square Q$ means that Q does eventually hold (and will then stay true) and S is the current state of the concurrent system

Safety & Liveness

Deadlocks

Most forms of synchronization may lead to

Deadlocks

(Avoidance / prevention of deadlocks is one central safety property)

- is* How to predict them?
- is* How to find them?
- is* How to resolve them?
- is* ... or are there structurally dead-lock free forms of synchronization?

Safety & Liveness

Deadlocks

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).

Safety & Liveness

Deadlocks

Deadlock strategies:

- **Ignorance & restart** *is* kill or restart unresponsive processes, power-cycle the computer, ...
- **Deadlock detection & recovery** *is* find deadlocked processes and recover the system in a coordinated way
- **Deadlock avoidance** *is* the resulting system state is checked before any resources are actually assigned
- **Deadlock prevention** *is* the system prevents deadlocks by its structure

Repetition

Correctness concepts in concurrent systems

Liveness properties: $(P() \wedge \text{Processes } (I, S)) \Rightarrow \diamond \square Q(I, S)$
where $\diamond \square Q$ means that Q does eventually hold (and will then stay true)

- Examples:**
- Requests need to complete eventually.
 - The state of the system needs to be displayed eventually.
 - No part of the system is to be delayed forever (fairness).
- is* Interesting liveness properties can become very hard to proof

Safety & Liveness

Towards synchronization

Reserving resources in reverse order

```

var reserve_1, reserve_2 : semaphore := 1;
process P1;
statement X;
wait (reserve_1);
wait (reserve_2);
statement Y;
signal (reserve_2);
signal (reserve_1);
end P1;
process P2;
statement A;
wait (reserve_2);
wait (reserve_1);
statement B;
signal (reserve_1);
signal (reserve_2);
end P2;
Sequence of operations  $A \rightarrow B \rightarrow C; X \rightarrow Y \rightarrow Z; [A, B, C] \parallel [X, Y, Z] \parallel [B \mid Y]$ 
or  $[A \mid X] \parallel [B \mid Y]$  followed by a deadlock situation.

```

Safety & Liveness

Deadlocks

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.

Safety & Liveness

Deadlocks

Deadlock prevention

(Remove one of the four necessary deadlock conditions)

1. **Break Mutual exclusion:**
 - is* Mutual exclusion hold and wait
 - is* No pre-emption
 - is* Circular wait

Safety & Liveness

Deadlocks

Deadlock prevention
(Remove one of the four necessary deadlock conditions)

- 1. Break Mutual exclusion:**
By replicating critical resources, mutual exclusion becomes unnecessary (only applicable in very specific cases).
- 2. Break Hold and wait:**
Allocation of all required resources in one request. Processes can either hold *none* or *all* of their required resources.
- 3. Introduce Pre-emption:**
Provide the additional infrastructure to allow for pre-emption of resources. Mind that resources cannot be pre-empted, if their states cannot be fully stored and recovered.
- 4. Break Circular waits:**

Mutual exclusion
 Hold and wait
 No pre-emption
 Circular wait

© 2003 Peter D. Zinnen, The Australian National University page 476 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Deadlock prevention
(Remove one of the four necessary deadlock conditions)

- 1. Break Mutual exclusion:**
By replicating critical resources, mutual exclusion becomes unnecessary (only applicable in very specific cases).
- 2. Break Hold and wait:**
Allocation of all required resources in one request. Processes can either hold *none* or *all* of their required resources.
- 3. Introduce Pre-emption:**
Provide the additional infrastructure to allow for pre-emption of resources. Mind that resources cannot be pre-empted, if their states cannot be fully stored and recovered.
- 4. Break Circular waits:**

Mutual exclusion
 Hold and wait
 No pre-emption
 Circular wait

© 2003 Peter D. Zinnen, The Australian National University page 477 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Deadlock prevention
(Remove one of the four necessary deadlock conditions)

- 1. Break Mutual exclusion:**
By replicating critical resources, mutual exclusion becomes unnecessary (only applicable in very specific cases).
- 2. Break Hold and wait:**
Allocation of all required resources in one request. Processes can either hold *none* or *all* of their required resources.
- 3. Introduce Pre-emption:**
Provide the additional infrastructure to allow for pre-emption of resources. Mind that resources cannot be pre-empted, if their states cannot be fully stored and recovered.
- 4. Break Circular waits:**

Mutual exclusion
 Hold and wait
 No pre-emption
 Circular wait

© 2003 Peter D. Zinnen, The Australian National University page 478 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Deadlock prevention
(Remove one of the four necessary deadlock conditions)

- 1. Break Mutual exclusion:**
By replicating critical resources, mutual exclusion becomes unnecessary (only applicable in very specific cases).
- 2. Break Hold and wait:**
Allocation of all required resources in one request. Processes can either hold *none* or *all* of their required resources.
- 3. Introduce Pre-emption:**
Provide the additional infrastructure to allow for pre-emption of resources. Mind that resources cannot be pre-empted, if their states cannot be fully stored and recovered.
- 4. Break Circular waits:**
E.g. order all resources globally and restrict processes to request resources in that order only.

Mutual exclusion
 Hold and wait
 No pre-emption
 Circular wait

© 2003 Peter D. Zinnen, The Australian National University page 479 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

RAG = (V, E); Resource allocation graphs consist of vertices V and edges E.
 V = P ∪ R; Vertices V can be processes P or Resource types R.
 with processes P = {P₁, ..., P_n} and resources types R = {R₁, ..., R_k}
 E = E_c ∪ E_r ∪ E_a; Edges E can be "claims" E_c, "requests" E_r, or "assignments" E_a
 with claims E_c = { →, ... }
 requests E_r = { →, ... }
 and assignments E_a = { →, ... }

Note: any resource type can have more than one instance of a resource.

© 2003 Peter D. Zinnen, The Australian National University page 480 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

© 2003 Peter D. Zinnen, The Australian National University page 481 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

Two process, reverse allocation deadlock:

© 2003 Peter D. Zinnen, The Australian National University page 482 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

© 2003 Peter D. Zinnen, The Australian National University page 483 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

No circular dependency no deadlock:

© 2003 Peter D. Zinnen, The Australian National University page 484 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

© 2003 Peter D. Zinnen, The Australian National University page 485 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

Two circular dependencies deadlock:

1 → 2 → 3 → 3 → 2 → 1
 as well as: 2 → 3 → 3 → 2 → 2

Derived rule:
 If some processes are deadlocked then there are cycles in the resource allocation graph.

© 2003 Peter D. Zinnen, The Australian National University page 486 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Edge Chasing
(for the distributed version see Chandy, Misra & Haas)

blocking processes:
 Send a probe to all requested yet unassigned resources containing ids of: (the blocked, the sending, the targeted node).

nodes on probe reception:
 Propagate the probe to all processes holding the critical resources or to all requested yet unassigned resources - while updating the second and third entry in the probe.

a process receiving its own probe:
 (blocked-id = targeted-id)
 Circular dependency detected.

© 2003 Peter D. Zinnen, The Australian National University page 487 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

Knowledge of claims:

Claims are potential future requests which have no blocking effect on the claiming process - while actual requests are blocking.

© 2003 Peter D. Zinnen, The Australian National University page 488 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

Assignment of resources such that circular dependencies are avoided:

© 2003 Peter D. Zinnen, The Australian National University page 489 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

Earlier derived rule:
 If some processes are deadlocked then there are cycles in the resource allocation graph.

Reverse rule for multiple instances:
 If there are cycles in the resource allocation graph and there are multiple instances per resource then the involved processes are potentially deadlocked.

Reverse rule for single instances:
 If there are cycles in the resource allocation graph and there is exactly one instance per resource then the involved processes are deadlocked.

© 2003 Peter D. Zinnen, The Australian National University page 490 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs
(Silberschatz, Galvin & Gagne)

Reverse rule for single instances:
 If there are cycles in the resource allocation graph and there is exactly one instance per resource then the involved processes are deadlocked.

Actual deadlock identified

© 2003 Peter D. Zinnen, The Australian National University page 491 of 758 (chapter 7: "Safety & Liveness" up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Reverse rule for multiple instances:
If there are cycles in the resource allocation graph and there are *multiple* instances per resource then the involved processes are *potentially* deadlocked.

⇒ Potential deadlock identified

© 2002 Peter D. Zisner, The Australian National University page 492 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

Reverse rule for multiple instances:
If there are cycles in the resource allocation graph and there are *multiple* instances per resource then the involved processes are *potentially* deadlocked.

⇒ Potential deadlock identified
– yet clearly not an actual deadlock here

© 2002 Peter D. Zisner, The Australian National University page 493 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Deadlocks

Resource Allocation Graphs

(Silberschatz, Galvin & Gagne)

How to detect actual deadlocks in the general case?
(multiple instances per resource)

© 2002 Peter D. Zisner, The Australian National University page 494 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Deadlocks

Banker's Algorithm

There are processes $P_i \in \{P_1, \dots, P_n\}$ and resource types $R_j \in \{R_1, \dots, R_m\}$ and data structures:

- Allocated $[i, j]$ ⇒ the number of resources of type j currently allocated to process i .
- Free $[j]$ ⇒ the number of currently available resources of type j .
- Claimed $[i, j]$ ⇒ the number of resources of type j required by process i eventually.
- Requested $[i, j]$ ⇒ the number of currently requested resources of type j by process i .
- Completed $[i]$ ⇒ boolean vector indicating processes which may complete.
- Simulated_Free $[j]$ ⇒ Number of available resources assuming that complete processes deallocate their resources.

© 2002 Peter D. Zisner, The Australian National University page 495 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Deadlocks

Banker's Algorithm

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:
 $\forall j$: Simulated_Free $[j] \Leftarrow$ Simulated_Free $[j] +$ Allocated $[i, j]$;
 Completed $[i] \Leftarrow$ True;
3. If $\forall i$: Completed $[i]$ then the system is currently **deadlock-free!**
 else all processes i with \neg Completed $[i]$ are involved in a **deadlock!**.

© 2002 Peter D. Zisner, The Australian National University page 496 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Deadlocks

Banker's Algorithm

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:
 $\forall j$: 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!**

A **safe** system is a system in which future deadlocks can be avoided assuming the current set of available resources.

© 2002 Peter D. Zisner, The Australian National University page 497 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Deadlocks

Banker's Algorithm

Check potential future system safety by simulating a granted request:
(Deadlock avoidance)

```

if (Request < Claimed) and (Request < Free) then
  Free := Free - Request;
  Claimed := Claimed - Request;
  Allocated := Allocated + Request;
  if System_is_safe (checked by e.g. Banker's algorithm) then
    ⇒ Grant request
  else
    ⇒ Restore former system state: (Free, Claimed, Allocated)
end if;
end if;

```

© 2002 Peter D. Zisner, The Australian National University page 498 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Deadlocks

Distributed deadlock detection

Observation: Deadlock detection methods like Banker's Algorithm are too communication intensive to be commonly applied in full and at high frequency in a distributed system.

- ⇒ Therefore a distributed version needs to:
 - ⇒ Split the system into nodes of reasonable locality (keeping most processes close to the resources they require).
 - ⇒ Organize the nodes in an adequate topology (e.g. a tree).
 - ⇒ Check for deadlock inside nodes with blocked resource requests and detect/avoid local deadlock immediately.
 - ⇒ Exchange resource status information between nodes occasionally and detect global deadlocks eventually.

© 2002 Peter D. Zisner, The Australian National University page 499 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Deadlocks

Deadlock recovery

A deadlock has been detected ⇒ now what?
Breaking the circular dependencies can be done by:

- ⇒ Either *pre-empt* an assigned resource which is part of the deadlock.
- ⇒ or *stop* a process which is part of the deadlock.

Usually neither choice can be implemented 'gracefully' and deals only with the symptoms.

Deadlock recovery does not address the reason for the problem!
(i.e. the deadlock situation can re-occur again immediately)

© 2002 Peter D. Zisner, The Australian National University page 500 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Deadlocks

Deadlock strategies:

- **Deadlock prevention**
System prevents deadlocks by its structure or by full verification.
⇒ **the best approach if applicable.**
- **Deadlock avoidance**
System state is checked with every resource assignment.
⇒ **More generally applicable, yet computationally very expensive.**
- **Deadlock detection & recovery**
Detect deadlocks and break them in a 'coordinated' way.
⇒ **Less computationally expensive (as lower frequent), yet usually 'messy'.**
- **Ignorance & random kill**
Kill or restart unresponsive processes, power-cycle the computer, ...
⇒ **Move of a panic reaction than a method.**

© 2002 Peter D. Zisner, The Australian National University page 501 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Atomic & idempotent operations

Atomic operations

Definitions of atomicity:

An operation is atomic if the processes performing it ...

- (by 'awareness') ... 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 atomic operation.
- (by 'communication') ... do not communicate with other processes while the atomic operation is performed.
- (by 'means of states') ... cannot detect any outside state change and do not reveal their own state changes until the atomic operation is complete.

Short:
An atomic operation can be considered to be **indivisible and instantaneous.**

© 2002 Peter D. Zisner, The Australian National University page 502 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Atomic & idempotent operations

Atomic operations

© 2002 Peter D. Zisner, The Australian National University page 503 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Atomic & idempotent operations

Atomic operations

Important implications:

1. An atomic operation is either performed *in full* or *not at all*.
2. A failed atomic operation cannot have any impact on its surroundings (must keep or re-instantiate the full initial state).
3. If any part of an atomic operation fails, then the whole atomic operation is declared failed.
4. All parts of an atomic operations (including already completed parts) must be prepared to declare failure until the final global commitment.

© 2002 Peter D. Zisner, The Australian National University page 504 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Atomic & idempotent operations

Idempotent operations

Definition of idempotent operations:
An operation is idempotent if the observable effect of the operation are identical for the cases of executing the operation:

- once,
- multiple times,
- infinitely often.

Observations:

- Idempotent operations are often atomic, but do not need to be.
- Atomic operations do not need to be idempotent.
- Idempotent operations can ease the requirements for synchronization.

© 2002 Peter D. Zisner, The Australian National University page 505 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Reliability, failure & tolerance

'Terminology of failure' or 'Failing terminology'?

Reliability ⇒ measure of success with which a system conforms to its *specification*.
⇒ low failure rate.

Failure ⇒ a deviation of a system from its *specification*.

Error ⇒ the system state which leads to a failure.

Fault ⇒ the reason for an error.

© 2002 Peter D. Zisner, The Australian National University page 506 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Reliability, failure & tolerance

Faults during different phases of design

- Inconsistent or inadequate specifications
⇒ frequent source for disastrous faults
- Software design errors
⇒ frequent source for disastrous faults
- Component & communication system failures
⇒ rare and mostly predictable

© 2002 Peter D. Zisner, The Australian National University page 507 of 758 (chapter 7: 'Safety & Liveness' - up to page 517)

Safety & Liveness

Reliability, failure & tolerance

Faults in the logic domain

- Non-termination / -completion
Systems 'frozen' in a deadlock state, blocked for missing input, or in an infinite loop
⇒ Watchdog timers required to handle the failure
- Range violations and other inconsistent states
⇒ Run-time environment level exception handling required to handle the failure
- Value violations and other wrong results
⇒ User-level exception handling required to handle the failure

© 2002 Peter & Zdenek, The Australian National University page 107 of 108 (chapter 7: 'Safety & Liveness' - up to page 112)

Safety & Liveness

Reliability, failure & tolerance

Faults in the time domain

- Transient faults
⇒ Single 'glitches', interference, ... very hard to handle
- Intermittent faults
⇒ Faults of a certain regularity ... require careful analysis
- Permanent faults
⇒ Faults which stay ... the easiest to find

© 2002 Peter & Zdenek, The Australian National University page 108 of 108 (chapter 7: 'Safety & Liveness' - up to page 112)

Safety & Liveness

Reliability, failure & tolerance

Observable failure modes

© 2002 Peter & Zdenek, The Australian National University page 110 of 108 (chapter 7: 'Safety & Liveness' - up to page 112)

Safety & Liveness

Reliability, failure & tolerance

Fault prevention, avoidance, removal, ...

and / or

Ⓡ Fault tolerance

© 2002 Peter & Zdenek, The Australian National University page 112 of 108 (chapter 7: 'Safety & Liveness' - up to page 112)

Safety & Liveness

Reliability, failure & tolerance

Fault tolerance

- Full fault tolerance
the system continues to operate in the presence of 'foreseeable' error conditions, without any significant loss of functionality or performance – even though this might reduce the achievable total operation time.
- Graceful degradation (fail soft)
the system continues to operate in the presence of 'foreseeable' error conditions, while 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.

© 2002 Peter & Zdenek, The Australian National University page 112 of 108 (chapter 7: 'Safety & Liveness' - up to page 112)

Safety & Liveness

Summary

Safety & Liveness

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

© 2002 Peter & Zdenek, The Australian National University page 113 of 108 (chapter 7: 'Safety & Liveness' - up to page 112)

