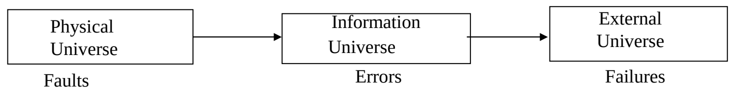
Fault Tolerance

- High performance systems must be *fault-tolerant*: they must be able to continue operating despite the failure of a limited subset of their hardware or software.
- They must also allow *graceful degradation:* as the size of the faulty set increases, the system must not suddenly collapse but continue executing part of its workload.

- Faults → errors → failures
  - A *fault* is a physical defect, imperfection or flaw that occurs within some hardware or software component. A fault can be caused by specification mistakes, implementation mistakes, component defects or external disturbance (environmental effects).
  - An *error* is the manifestation of a fault.
  - If the error results in the system performing its function(s) incorrectly, then a system *failure* occurs.

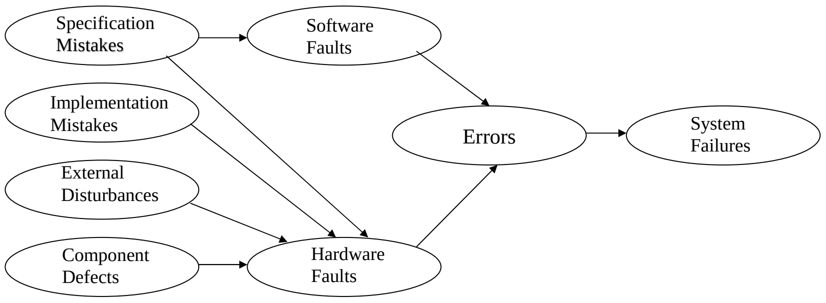
The Three universe model



Three-universe model representing the cause-and-effect relationship between faults, errors, and failures. Faults occur in the physical universe and cause errors to occur in the informational universe. Errors can result in failures that occur in the external universe.

Cause-and-effect relationship

(faults, errors, and failures in a system)



Types of faults

- A *permanent* fault remains in existence indefinitely if no corrective action is taken.
- A *transient* fault disappears within a short period of time
- An *intermittent* fault may appear and disappear repeatedly.

Dealing with Faults

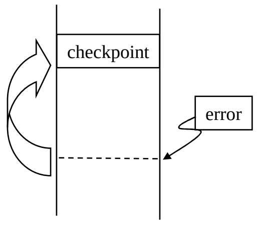
- *Fault avoidance* aims at preventing the occurrence of faults at the first place: design reviews, component screening, testing.
- *Fault Tolerance* is the ability of a system to continue to perform its tasks after the occurrence of faults
  - *Fault Masking*: preventing faults from introducing errors
  - *Reconfiguration:* Fault detection, location, containment and recovery

Types of Redundancy

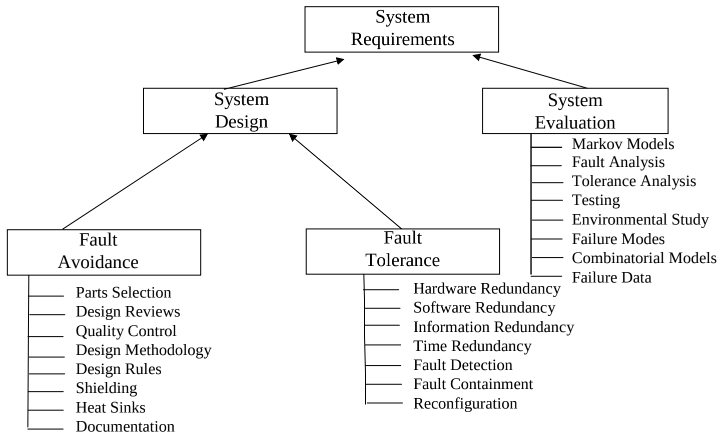
- *Hardware Redundancy:* Based on physical replication of hardware.
- *Software Redundancy:* The system is provided with different software versions of tasks, preferably written independently by different teams.
- *Time Redundancy:* Based on multiple executions on the same hardware in different times.
- *Information Redundancy:* Based on coding data in such a way that a certain number of bit errors can be detected and/or corrected.

Types of Recovery

- *forward-error recovery:* the error is masked without any computations having to be re-done.
- *backward-error recovery:* periodically take *checkpoints* to save a correct computation state. When error is detected, roll back to a previous checkpoint, restore the correct state and resume execution.



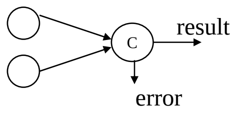
A top-level view of the system design process illustrating the importance of fault avoidance, fault tolerance, and system evaluation.



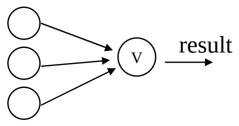
Redundant systems

- *Sparing:* Can have spares (hot or cold spares) and use a spare after a permanent fault is detected in the primary hardware.

- *Duplex systems:* can detect a fault by executing twice (on separate hardware on sequentially on the same hardware) and compare the results.

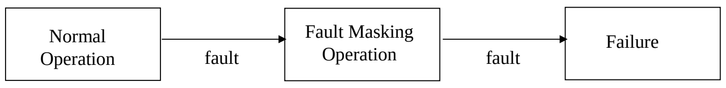


- *Triple modular redundancy (TMR):* can mask an error by executing three times and taking a majority vote.

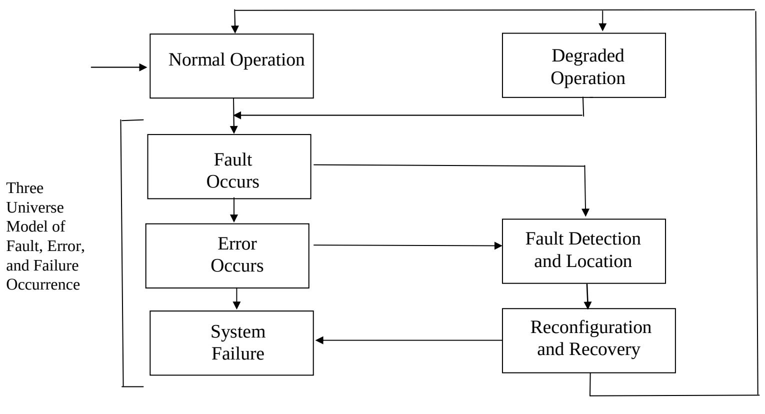


- *N modular redundancy (NMR):* can mask an error by executing *N* times and taking a majority vote. How many faults can be tolerated?

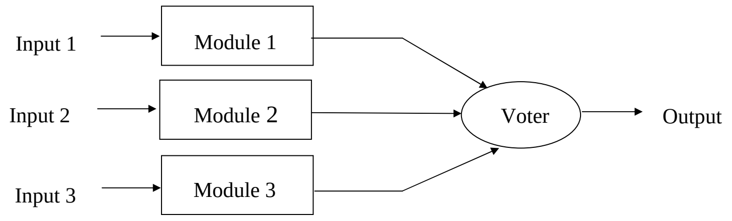
<u>Passive redundancy</u>



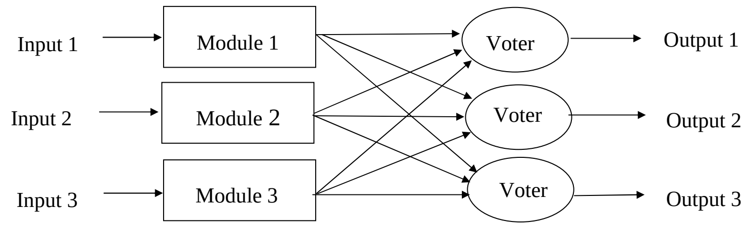
<u>Active redundancy</u>


Normal Operation
Degraded Operation
Fault Occurs
Error Occurs
System Failure
Fault Detection and Location
Reconfiguration and Recovery
Three Universe Model of Fault, Error, and Failure Occurrence

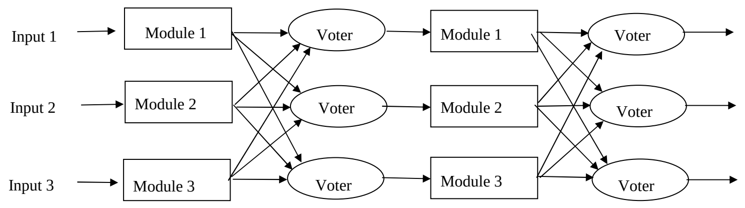
Passive Redundancy



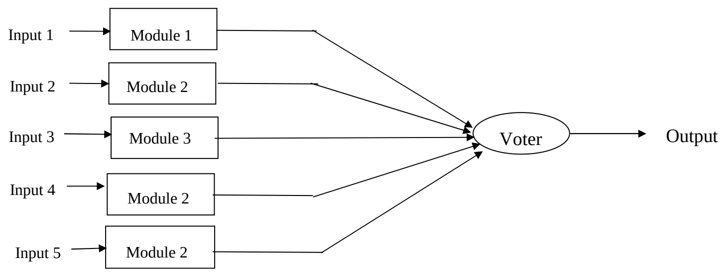
Triple modular redundancy (TMR) uses three identical modules, performing identical operations, with a majority voter determining the output.



Triple modular redundancy with **triplicated voters** can be used to overcome susceptibility to voter failure. The voter is no longer a single point of failure in the system.

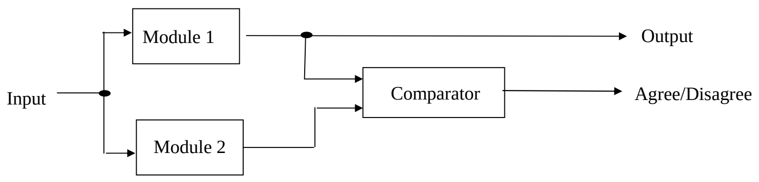


In multiple-stage TMR systems, voting occurs between each stage so that errors are corrected before being passed to a subsequent module.

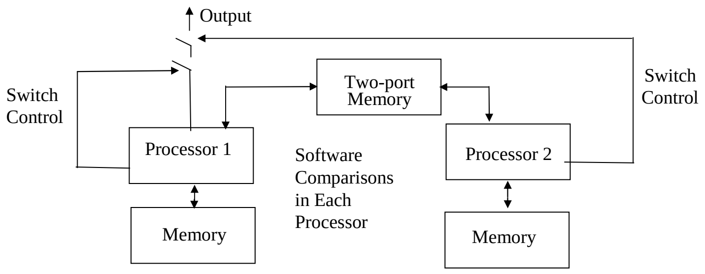


5MR is an example of NMR with five identical modules. Majority voting allows the failure of two modules to be tolerated.

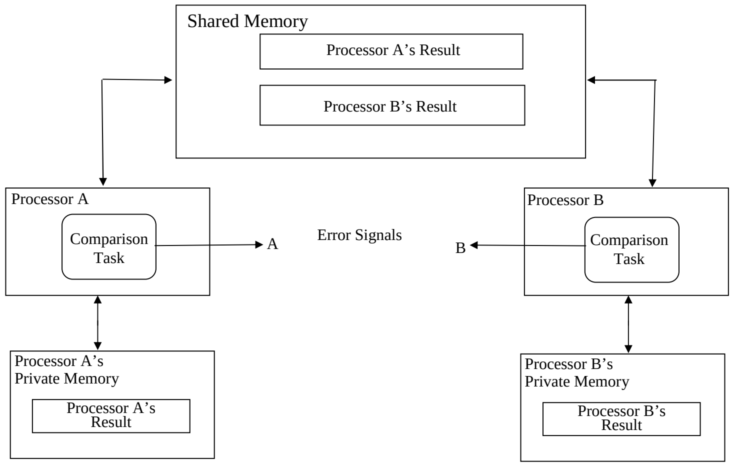
Active Redundancy



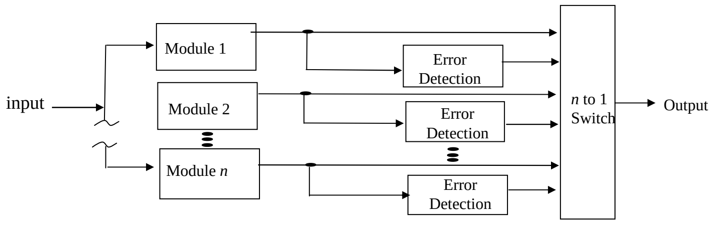
Duplication with comparison uses two identical modules performing the same operations and compares their results. **Fault detection** is provided but not fault tolerance.



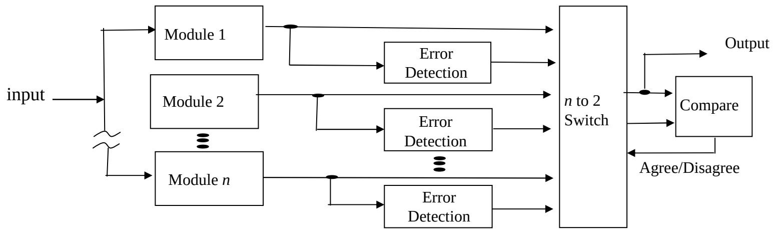
The necessary comparisons in **duplication with comparison** can be implemented in software. Both processors must agree that results match before an output is generated.



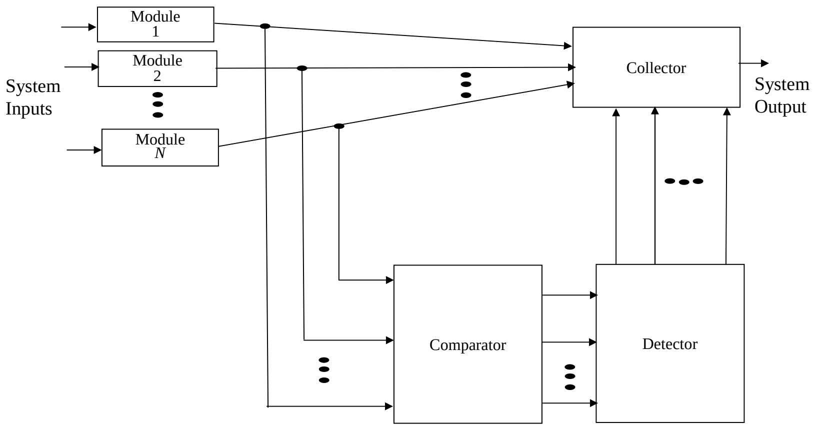
A software implementation of duplication with comparison (shared memory systems)



In standby sparing, one of n modules is used to provide the system's output, and the remaining n-1 modules serve as spares (Hot or Cold). Error detection techniques identify faulty modules so that a fault-free module is always selected to provide the system's output.

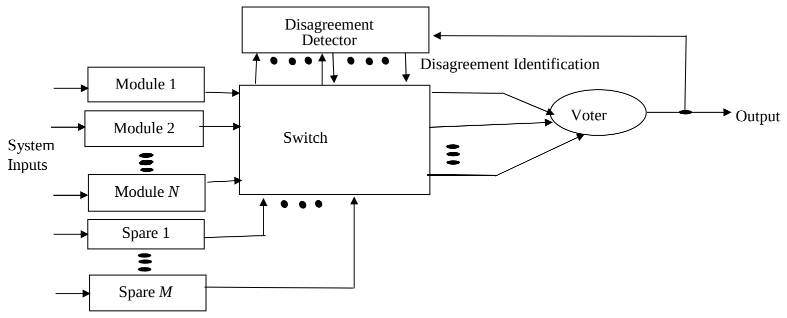


The **pair-and-a spare** technique combines duplication with comparison and standby sparing. Two modules are always online and compared, and any spare can replace either of the online modules.

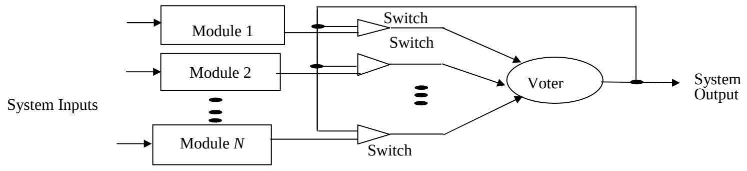


Sift-out modular redundancy uses a centralized collector to create the system output. All modules are compared to detect faulty modules - Faculty module can be put back in service if fault is transient.

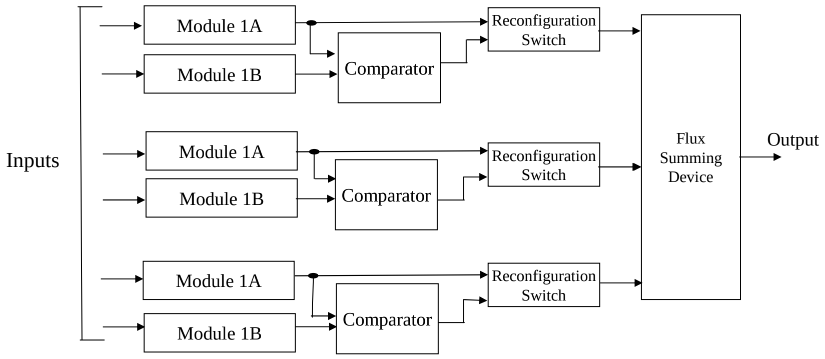
Hybrid Redundancy



***N*-modular redundancy with spares** combines NMR and standby sparing. The voted output is used to identify faulty modules, which are then replaced with spares.



Self-purging redundancy uses the system output to remove modules whose output disagrees with the system output.



The triple-duplex approach to hybrid redundancy.

Time Redundancy

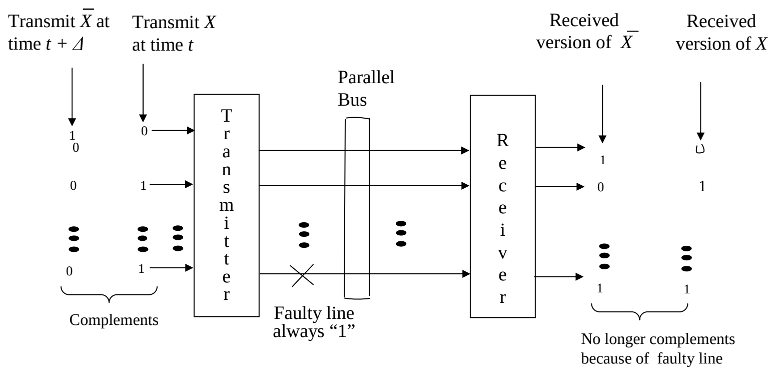
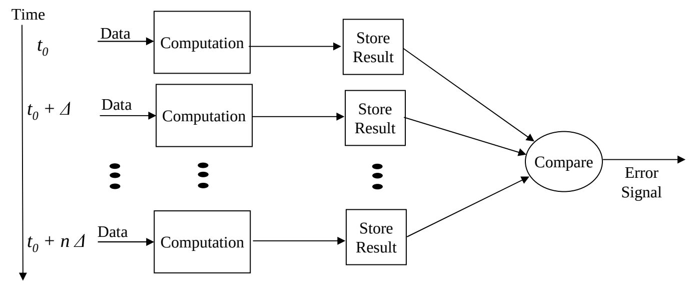
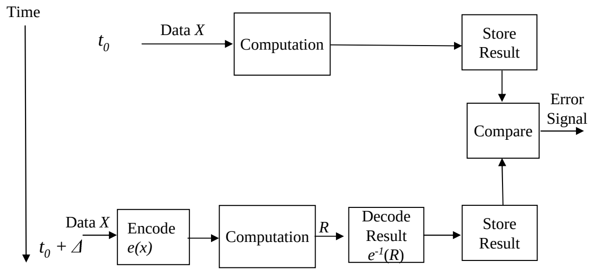


Illustration of **alternating logic** time redundancy – the second transmission is the complement of the first.



In time redundancy, computations are repeated at different points in time and then compared.

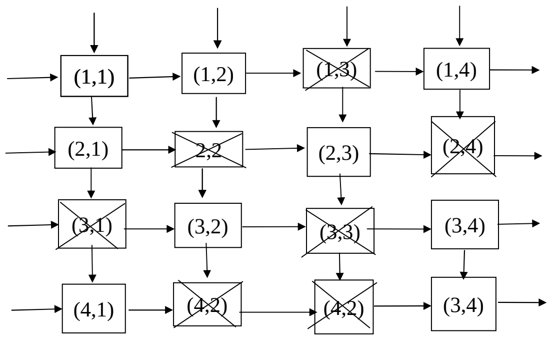


Permanent faults can be detected using time redundancy by modifying the way in which computations are performed.

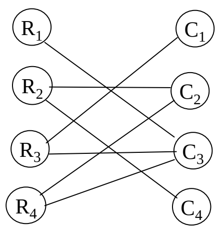
Information Redundancy

- Parity codes
- Hamming codes
- *M-of- n* codes
- BCH (cyclic) codes

Example of repair (reconfiguration of memory arrays)



4-by-4 Array with Seven Faulty Elements



The corresponding bipartite graph.

Repair Most

Fault-tolerant software

- *Consistency checks:* a software acceptance test to detect wrong results.
- *N-version programming*: Prepare N different versions and run them (in parallel or sequentially). The *voting* at the end will select the output of the majority.
- Sources of common-mode failures:
  - Ambiguities in the specification
  - Choice of the programming language
  - Choice of numerical algorithms
  - Common background of the software developers
- *Recovery block approach:*
  - Each job/task has a primary version and one or more alternatives.
  - When primary version is completed, an *acceptance test* is performed.
  - If the acceptance test fails, an alternative version can be invoked.

Reliability and availability

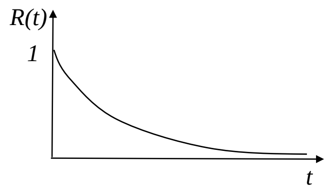
- *The reliability at time t, R(t),* is the conditional probability that the system performs correctly during the period [*0,t*], given that the system was performing correctly at time *0*.
- The *unreliability, F(t),* is equal to *1 – R(t).* Often referred to as *the probability of failure*.
- *The availability at time t, A(t)* is the probability that a system is operating correctly and is available to perform its functions at time *t*. Unlike reliability, the availability is defined at an instant of time.
- The system may incur failures but can be repaired promptly, leading to high availability.
- A system may have very low reliability, but very high availability!

Mean time to failure (FTTF)

- Let *R(t)* be the reliability of a system and *F(t) = 1 - R(t).*
- *F(t)* is the probability that the system is not functioning correctly at time *t.* *Hence,* $\frac{dF(t)}{dt} = -\frac{dR(t)}{dt}$ is the probability that the system fails exactly at time *t (failure density function).*
- The average time to failure is

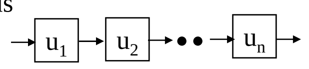
$$MTTF = \int_0^\infty t \frac{dF(t)}{dt} dt = -\int_0^\infty t \frac{dR(t)}{dt} dt = [-tR(t)]_0^\infty + \int_0^\infty R(t)dt = \int_0^\infty R(t)dt$$

- Example: if *R(t)* = $e^{-\lambda t}$, then
  - MTTF = 1 / λ ,
  - λ is the failure rate.

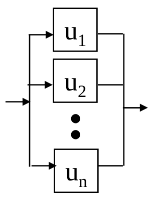


Combinatorial calculation of the reliability

- For n units connected in series, the system is functioning if all the units are functioning, thus the reliability of the system is
  $R(t) = R_1(t)\, R_2(t) \ldots R_n(t)$



- For n units connected in parallel, the system is functioning if at least one unit is functioning, thus
  $1 - R(t) = (1 - R_1(t))\,(1 - R_2(t)) \ldots (1 - R_n(t)),$
  and the system reliability is
  $R(t) = 1 - (1 - R_1(t))\,(1 - R_2(t)) \ldots (1 - R_n(t))$

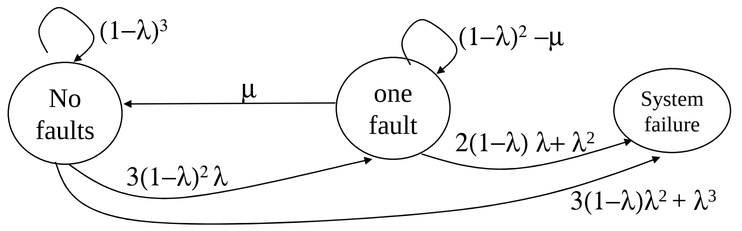


- Example: Reliability of a TMR system is
  $(3R_{unit}^2(1 - R_{unit}) + R_{unit}^3)R_{voter}$

Markov processes

- Is a process that can be represented by states and probabilistic state transitions, such that the probability of moving from a state s_i to another state s_j , does not depend on the way the process reached state s_i.

- Example: a TMR system with unit MTTF = $1/\lambda$, and mean time to repair equal to = $1/\mu$.



- Note that the failure state is an absorbing state.
- For discrete time processes, one transition occurs in every time unit.

Discrete Markov processes

- A Markov process with *n* states can be represented by an *nxn* probability matrix $A = [a_{i,j}]$, where $a_{i,j}$ is the probability of moving from state *i* to state *j* in one time unit.

- The sum of the elements in each row of *A* is equal to 1.

- If $p(t) = [p_i(t)]$ is a vector such that $p_i(t)$ is the probabilities of being in state *i* at time *t*, then, $p(t+k) = B^k\, p(t)$, where *B* is the transpose of *A*.

- Can use *the first step analysis* to find
  - the average number of transitions before absorption, and
  - the average time of being in a certain state (if steady state, then $Bp = p$).

Average # of transitions before absorption

- Consider an n state Markov process in which state n is an absorption state, and let v_i be the average number of steps to absorption if we start at state i.
- Hence, for every $i=1, \dots, n-1$ we have

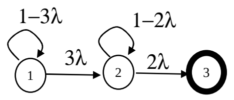
$$v_i = a_{i,1}(1+v_1) + \dots + a_{i,n-1}(1+v_{n-1}) + a_{i,n}$$

- Solve the above $n-1$ equations and find the values of $v_1, \dots, v_{n-1}$
- Given an initial probability distribution $p(0)$, the average time to absorption is $p_1 v_1 + \dots + p_{n-1} v_{n-1} + 0 \cdot p_n$
- Example: The TMR system without repair ($\mu = 0$) and ignoring λ^2 terms

$v_1 = (1-3\lambda)(v_1 +1) + 3\lambda (v_2 +1)$

$v_2 = (1-2\lambda)(v_2 +1) + 2\lambda$

Which gives $v_2 = 1 / 2\lambda$ and $v_1 = 5 / 6\lambda$



Another example of the first step analysis

- Consider an $n+2$ state Markov process in which states $n+1$ and $n+2$ are absorption states, we want to find out what is the probability that the process will end up in state $n+2$ (as opposed to $n+1$).
- Let u_i be the probability that the process will eventually end up in state $n+2$ assuming that the process starts at state i.
- Hence, for every $i=1, \dots, n$ we have

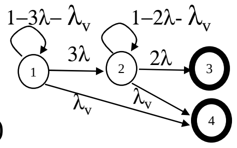
$$u_i = a_{i,1} u_1 + \dots + a_{i,n} u_n + a_{i,n+2}$$

- Solve the above n equations and find the values of $u_1, \dots, u_n$
- Given an initial probability distribution $p(0)$, the probability of absorption to state $n+2$ is $p_1 u_1 + \dots + p_n u_n$
- Example: The TMR system with a voter and voter failure rate λ_v

$u_1 = (1 - 3\lambda - \lambda_v) u_1 + 3\lambda u_2 + \lambda_v$

$u_2 = (1 - 2\lambda - \lambda_v) u_2 + \lambda_v$

Which gives $u_1 = \lambda_v (5\lambda + \lambda_v) / (2\lambda + \lambda_v)(3\lambda + \lambda_v)$



Cache-based checkpointing

Reg + cache ---------------------> Main Memory

Active ……………………... Stable

On cache miss = save registers (checkpoint)

Rollback = restore registers + invalidate dirty cache lines

Notes:

- Same principle applies to checkpoint in paged memory systems

    Main memory (active) ---------- > Disk (stable)

- Need write-back cache and not write through
- May trigger checkpoints at regular intervals

Need to consider faults during checkpointing

- save state twice and record time before and after checkpoints
- if fault occurs during last checkpoint, restore the previous one

(t1 .. State1 .. t2………………….t3 .. State2 .. t4)

Virtual checkpointing

- Memory pages are active ------ disk storage is stable
- Keeps two copies of each page in your virtual address
- Keep a global counter to register the number of the last checkpoint (V)
- Modify your TLB (translation lookaside buffer) to reflect one active page (L = 1 & v = V) and one backup page (L = 0 & v < V)

- Checkpoint → V++
- At reference → check v, and if v < V, interchange active and backup

    (L = 0) @ old active (L=1 , v = V) @ old backup

- Rollback → copy backup to active
- Overhead = check v at each reference

Using a second processor for checkpointing

- Checkpoint → send state to backup
- Periodically send (I am alive) messages --- heart-beats
- If a heart beat is missing, backup takes over from last checkpoint.

Checkpointing in shared memory system

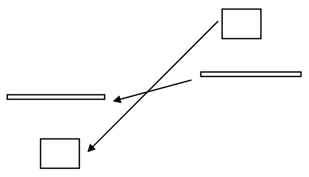
- Checkpoint each processor at cache miss (save registers)
- Works only with write back and not write through
- Cache coherence problems for shared variables??

Checkpointing in distributed shared memory systems

(Memory pages are active ------ disk storage is stable)

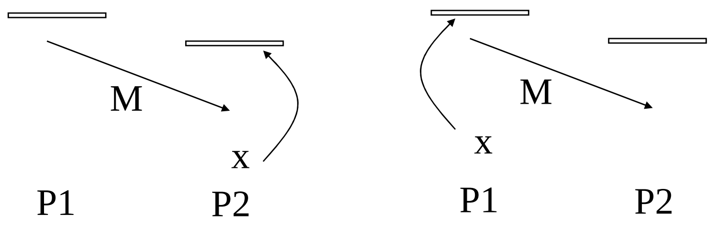
GLOBAL checkpointing:

- S initiates checkpoints and inform others
- Others take checkpoints
- To prevent incomplete migration: If a page is received after the ith checkpoint but was sent before the ith checkpoint, then a local checkpoint is taken.
- Recovery is by rolling back to ith checkpoint.



Checkpointing in distributed memory systems

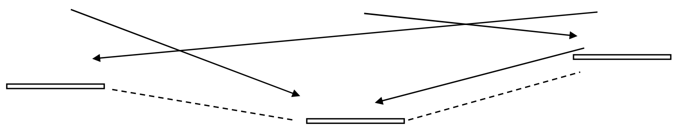
(Memory pages are active ------ disk storage is stable)



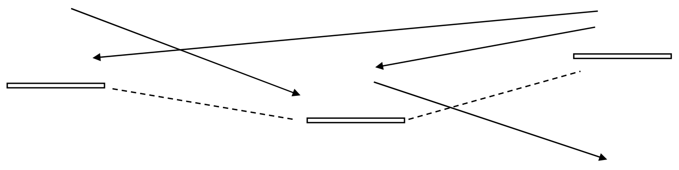
If P2 roll backs, then message M is lost

If P1 roll backs, then message M is an orphan

Consistent recovery line

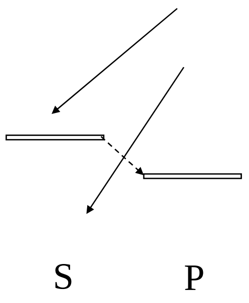


Inconsistent recovery line



1) Synchronous checkpointing (pessimistic)

- Initiator, S, send a request to checkpoint to every P, with the stamp of the last message received by S from P
- P determines if it has to checkpoint,
- If S gets any NOT OK, it aborts the checkpointing, else it asks everybody to make the checkpoint permanent.
- No messages are transmitted during checkpointing



ROLL BACK

• S is faulty and rolls back – It sends messages to every P, with the stamp of the last messages (at checkpoint) from S to P

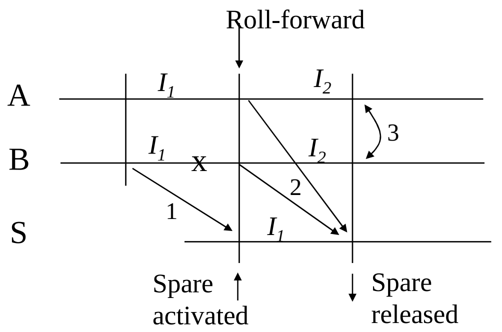
• P determines if it has to roll back, and if so, repeat the procedure.

2) Asynchronous checkpointing (optimistic)

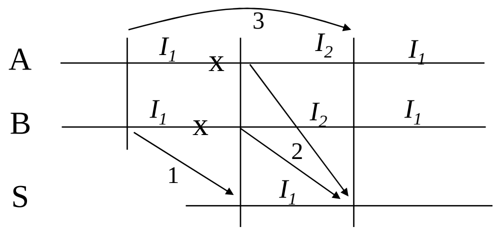
- lost messages can be retransmitted (if logged)
- orphan messages should not be permitted

- Each processor takes checkpoints asynchronously
- Dependencies are kept track of in messages
- If S rolls back, it notifies every P. Then P checks if it has to roll back, if so, it repeats recursively, if not it retransmit the messages P -> S.
- May roll-back to beginning of computation
- If received messages are logged in stable storage, then this minimizes retransmission.
- Can eliminate checkpoints when sure that they are not needed.

Forward recovery



- Spare acts as an arbiter to determine the faulty processor
- Direction of transfer in (3) is determined by fault location



- With double faults, roll back