SAFETY AND RELIABILITY

November-28-12 10:06 AM

IN TRADITIONAL SYSTEMS safety and reliability are normally considered to be independent issues -- there are traditional systems that are safe and unreliable and vice-versa.

EXAMPLE 1: a word processing software may not be very reliable but is safe but a failure to the software does not usually cause any significant damage or financial loss. **EXAMPLE 2**: a hand gun can be unsafe but reliable (i.e, a hand gun rarely fails). However, if it fails for some reason, it can misfire or even explode and cause damage.

IN REAL-TIME SYSTEMS safety and reliability are coupled together.

DEFINITIONS:

1. FAIL-SAFE STATE of a system is a state which if entered when the system fails, no damage would result. EXAMPLE: the fail-safe state of a word processing program is a state where the document being processed has been saved on the disk.
   - If no damage can result when a system enters a fail-safe state just before it fails, then through careful transition to fail-safe state upon a failure, it is possible to turn an extremely unreliable and unsafe system into a safe system.
2. SAFETY-CRITICAL SYSTEM is a system whose failure can cause severe damage. EXAMPLE: the navigation system on-board of an aircraft. In a safety-critical system, the absence of fail-safe states implies that safety can only be ensured through increased reliability.

HOW TO ACHIEVE HIGH RELIABILITY?

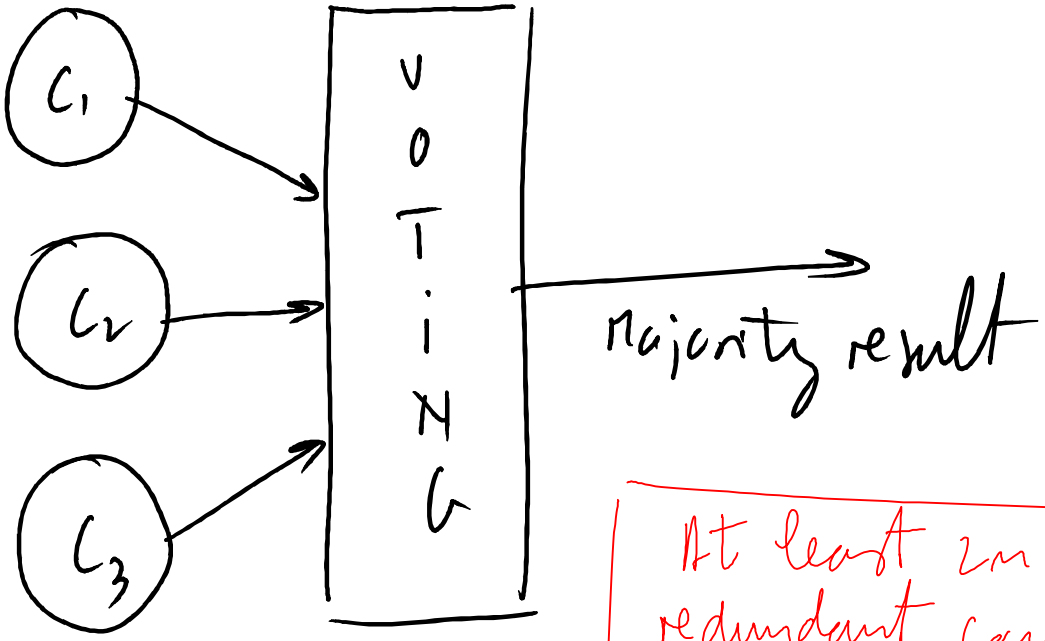
For safety-critical systems the issue of safety and reliability become interrelated such that safety can only be ensured through increased reliability. Highly reliable software can be developed by adopting the following techniques:

1. ERROR AVOIDANCE. Every possibility of occurrence of errors should be minimized as much as possible during product development. This can be achieved by adopting well-founded software engineering practices and sound design methodologies, etc.
2. ERROR DETECTION AND REMOVAL. In spite of using best available error avoidance techniques, many errors are still possible in the code. By conducting thorough reviews and testing, the errors can be detected and then removed.
3. FAULT-TOLERANCE. It is virtually impossible to make a practical software system entirely error free. Few errors still persist even after carrying out thorough reviews and testing. Therefore, to achieve high reliability, even in situations where errors are present, the system should be able to tolerate the faults and compute the correct results. This is called fault-tolerance. Fault tolerance can be achieved by carefully incorporating redundancy.

FAULT TOLERANCE IN HARDWARE

BUILT-IN SELF TEST (BIST). In BIST, the system periodically performs self tests of its components. Upon detection of a failure, the system automatically reconfigures itself by switching out of the faulty component and switching in one of the redundant good components.

TRIPLE MODULAR REDUNDANCY (TMR). In TMR, three redundant copies of all critical components are made to run concurrently.



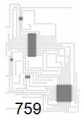
Redundant copies of the same component

At least 2n+1 redundant components are required to tolerate simultaneous failures of n components.

Fault Tolerance through Redundancy

Follows the Real-time System by C. M. Krishna

- If a system is kept running despite the failure of some of its parts, it must have spare capacity
  - Hardware redundancy: the system is provided with far more hw than it would need, typically, between 2 and 3 times as much
  - Software redundancy: the system is provided with different software versions of tasks, ...
  - Time redundancy: the task schedule has some slack in it, so that some tasks can be rerun if necessary and still meet critical deadlines
  - Information redundancy: the data are coded in such a way that a certain number of bit errors can be detected and/or corrected



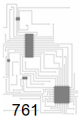
Additional hardware can be used in 2 ways:

(1) Fault detection, correction, and masking (this is a short term measure) -- multiple hardware units may be assigned to the same task in parallel and their results compared -- when units become faulty this will show as a disagreement in the results. We can mask the faults with the majority result (if only a minority of the units are faulty). Example. In a chemical plant, the computer is accessible to be repaired and replaced so we are interested in short-term measures to respond to failures.

(2) Replace the malfunctioning units (this is a long term measure). It is possible for systems to be designed so that spares can be switched in to replace any faulty units. Example: if a computer is used aboard an unmanned deep-space probe it must include sufficient spare modules and self-repair mechanism to sustain long time functionality.

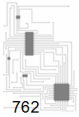
Voting and Consensus

- By using redundancy, multiple units can execute the same task and comp the output
  - if at least 3 units are involved, this comparison can choose the majority value – a process called voting
    - this process can mask some effects of failures
- The designer must decide whether exact or approximate agreement is expected
- If there are 3 units A, B, C, and both A and B produce the value x while C produces the value x±α, for what value of α is C considered faulty



Types of Voters

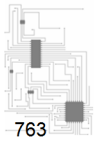
- Approximate agreement may be used in cases where sensors are measuring the physical environment. Ex – next page
- There are three main types of voters, which can function in cases where approximate agreement is required
  - the formalized majority voter
  - the generalized k-plurality voter
  - generalized median voter
- Each case uses the distance metric measurement: d(x1,x2) denotes the distance between outputs x1 and x2
  - if x1 and x2 are real numbers => d(x1,x2) = |x1-x2|
  - if they are vectors of real numbers Cartesian distance may be chosen



Radu Muresan

Table 7.1 Comparison of Voter Types			
Case	**Majority Voter**	***k*-plurality Voter**	**Median Voter**
All outputs and SE	Correct	Correct	Correct
Majority correct and SE	Correct	Correct	Correct
k correct and SE	No output	Correct	Possibly correct
All outputs correct but none SE	No output	No output	Correct
All outputs incorrect and not SE	No output	No output	Incorrect
Majority incorrect and SE	Incorrect	Incorrect	Incorrect
Note: SE = Sufficiently equal Adapted from Larczak, Caglayan, and Eckhardt			

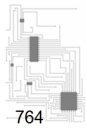
This table assumes that there are N outputs to be voted on, and that N is an odd number



Approximate agreement may be used in cases where sensors are measuring physical environment.
Ex: Temperature sensors placed in a boiler
→ impractical to expect to produce identical readings.
Why? - installed slightly apart
- They will have certain tolerance
To get a good output, the voter must take the median

Formalized Majority Voter

- If d(x1,x2) ≤ ε; x1 and x2 are sufficiently equal (not transitive relation) for all practical purposes. (SE)
- The voter constructs a set of classes, P1, …,Pn such that: → for all x & y
  - x,y are in Pi iff d(x,y) ≤ ε, and
  - Pi is maximal; that is, if z is not in Pi, then there exists some w in Pi such that d(w,z) > ε.
- Example. Let ε = 0.001 for some 5-unit system. Let the five outputs be 1.0000, 1.0010, 0.9990, 1.0005, and 0.9970. The classes will be: …



The classes are:

$P_1 = \{1.000, 1.0010, 1.0005\}$;

$P_2 = \{1.000, 0.9990\}$;

$P_3 = \{0.9970\}$

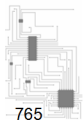
Note: 1.000 is in both P_1 & P_2

P_1 - largest class and has $3 > \lceil N/2 \rceil$ elements.

- The classes may share some elements. Take the largest Pi thus generated. If it has more than $\lceil N/2 \rceil$ elements in it, any of its elements can be chosen as the output of the voter.
- We say Pi is maximal in the sense that there are no other elements that can be added to its set.

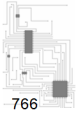
Generalized k-plurality voter

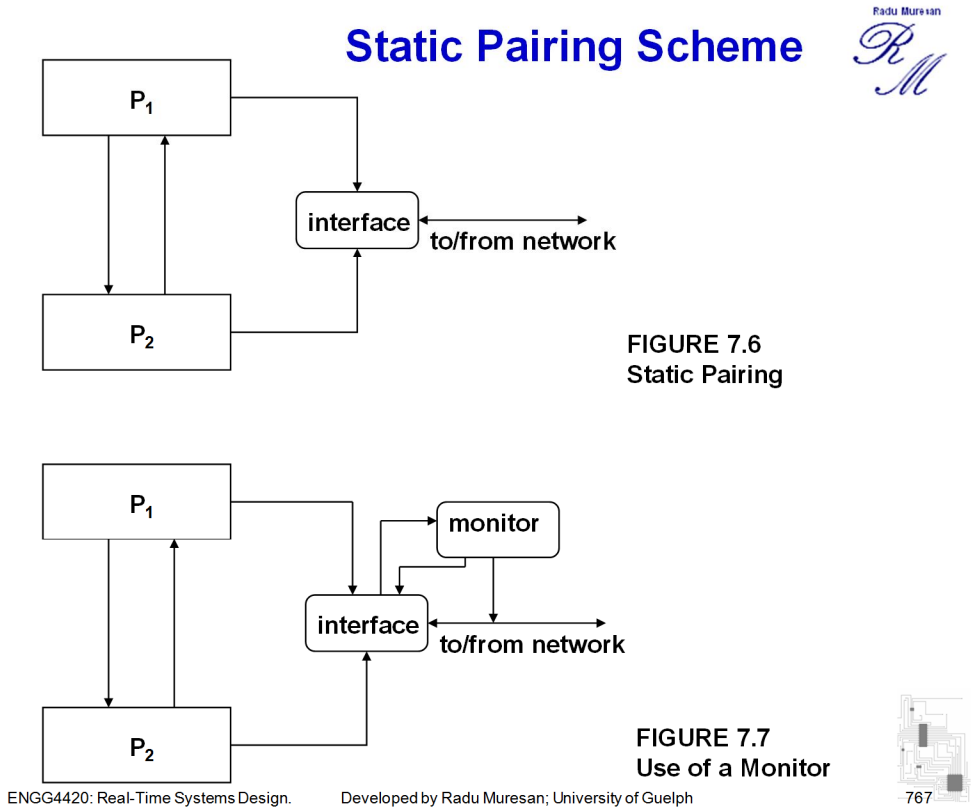
- The generalized k-plurality voter works along the same lines as the generalized majority voter, except that it simply chooses any output from the largest partition Pi, so long as Pi contains at least k elements
- k is selected by the system designer



Generalized Median Voter

- By selecting the middle value (N is odd)
  - successively throwing away outlying values until only the middle value is left
- Algorithm
  - Let outputs being voted on be the set S={x1,…,xn};
  - Step1. Compute dij=d(xi,xj) for all xi, xj in S for i ≠ j;
  - Step2. Let dkl be the maximum such dij (break any ties arbitrarily); define S = S – {xk,xl}; If S contains only one element, that is the output of the voter; else go back to step1.





TECHNIQUES FOR AUTOMATIC HARDWARE REPLACEMENT

STATIC PAIRING is a simple scheme that hardwires processors in pairs and discards the entire pair when one of the processors fails.

- The pair runs identical software using identical inputs, and compares the output of each task. If the outputs are identical, the pair is functional. If either processor in the pair detects non-identical outputs, that is an indication that at least one of the processors in the pair is faulty.
- The processor that detects this discrepancy switches off the interface to the rest of the system, thus isolating this pair.

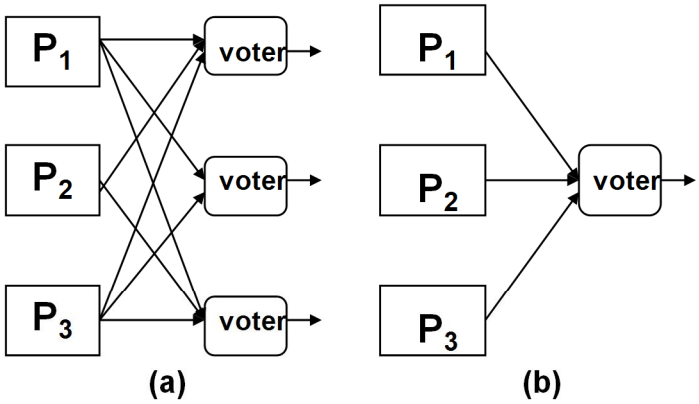
PROBLEMS WITH THIS SCHEME: 1) if the interface fails; 2) if both processors fail identically and around the same time.

- The interface problem can be solved by introducing an interface monitor. The monitor and interface can test each other.

N –MODULAR RUDUNDANCY. ***N*** **–modular redundancy (NMR) is a scheme for forward error recovery. It works by using *N* processors instead of one, and voting on their output. *N* is usually odd. Figure 7.8 illustrates this scheme for *N* = 3. One of the approaches is possible. In design (a), there are *N* voters and the entire cluster produces *N* outputs. In design (b), there is just one voter.**

To sustain up to *m* failed units the NMR system requires (2*m* + 1) units in all. The most popular is the *triplex*, which consists of a total of 3 units and can mask the effects of up to one failure. → TMR

Usually, the NMR clusters are designed to allow the purging of malfunctioning units.



A cluster of N=2m+1 processors is sufficient to guard against up to m failures

Figure
Structure of an NMR cluster.
(a) *N* voters (b) single voter.

NMR clusters are designed to allow the purging of malfunctioning units
– possibly a spare processor if available is introduced in the cluster upon failure
Purging can be done either by hardware or by the operating system.

SOFTWARE FAULT-TOLERANCE TECHNIQUES

Three methods are popular for software fault-tolerance: 1) N-version programming technique, 2) recovery block technique, and 3) roll-back recovery.

N-VERSION PROGRAMMING. This technique is an adaption of the TMR technique for hardware fault-tolerance. In the N-version programming technique:

- Independent teams develop N different versions (value of N depends on the degree of fault-tolerance required) of a software component (module) -- the central idea is that independent teams would commit different types of mistakes, which would be eliminated when the results produced by them are subjected to voting.
- The redundant modules are run concurrently (possibly on redundant hardware);
- The results produced by the different versions of module are subject to voting at run time and the result on which majority of the components agree is accepted.

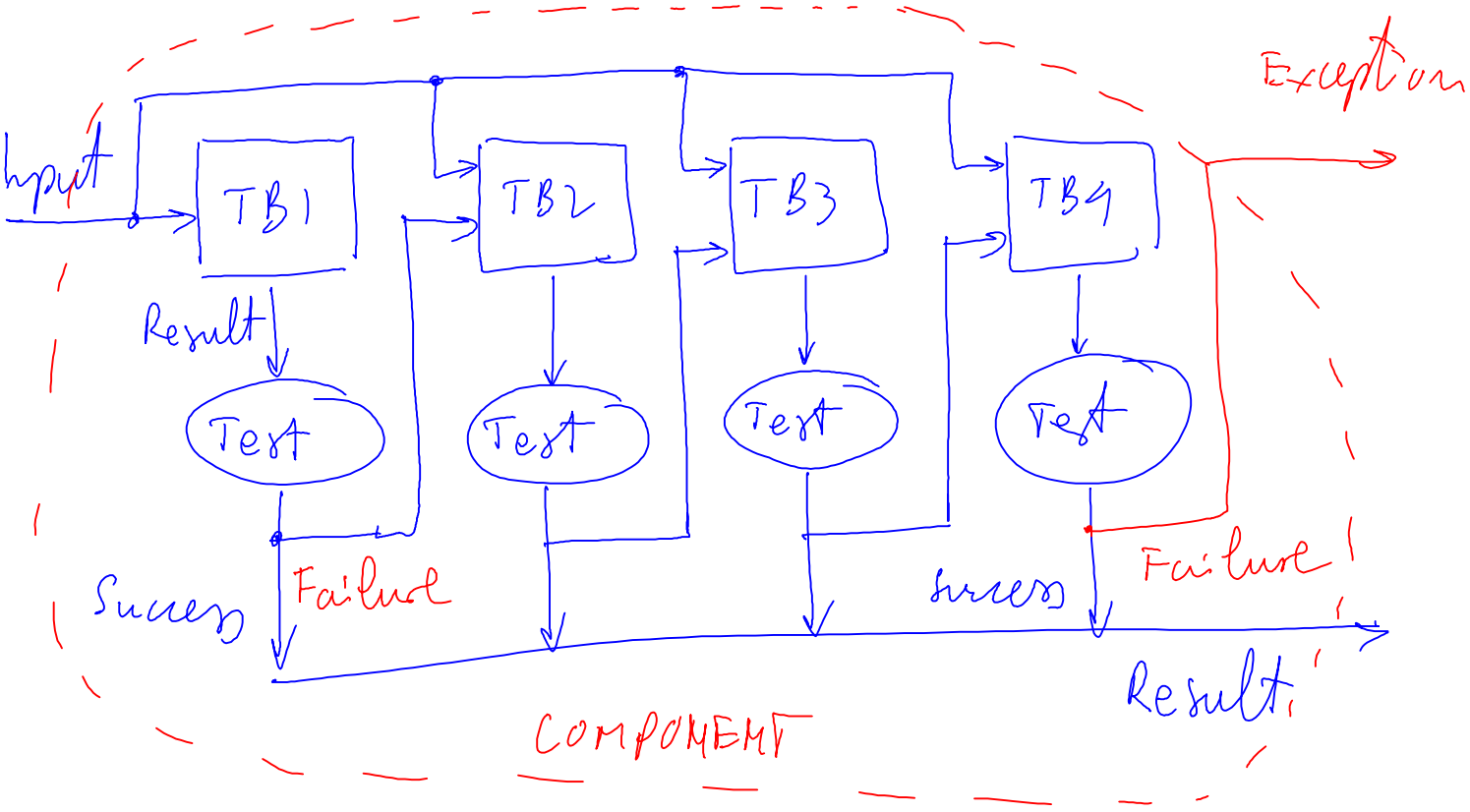
THE SCHEME is not very successful in achieving fault-tolerance and the problem can be attributed to "statistical correlation of failure" -- which means that even with independent teams developing different version the versions tend to fail for identical reasons.

FOR EXAMPLE it is easy to understand that programmers commit errors in those parts of a problem which they perceive to be difficult -- and what is difficult to one team is difficult to all teams. SO, identical errors remain in the most complex and least understood parts of a software component.

RECOVERY BLOCKS

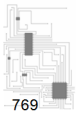
In the recovery block scheme, the redundant components are called "try blocks".

- Each try block computes the same end result as the others but is intentionally written using a different algorithm compared to the other try blocks.
- In this scheme the try blocks are run one after another
- The results produced by a try block are subjected to an acceptance test. If the acceptance test fails then the next try block is tried.
- The process is repeated in a sequence until the result produced by a try block successfully passes the test.
- The scheme can use a common test for all blocks



Software Redundancy

- To provide reliability in the face of software faults, we must use redundancy
  - simply replicating the same software N times will not work
  - instead, the N versions of the software must be diverse so that the probability that they fail on the same input is acceptably small
    - this can be done by having different software teams generating software for the same task
- There are 2 approaches in handling multiple versions of software: N-version programming; and recovery-block



Software Redundancy Structures

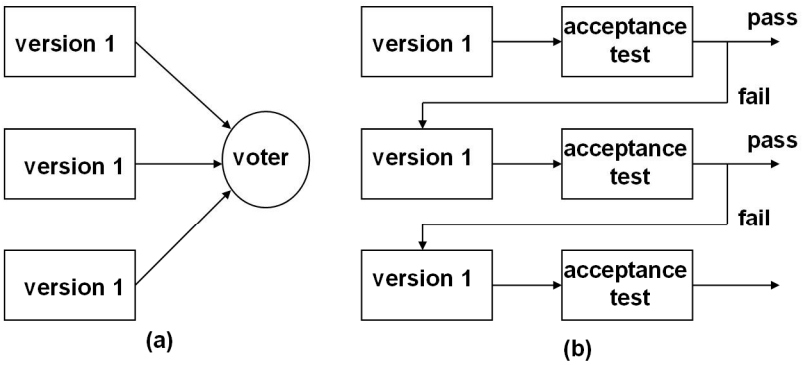
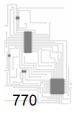


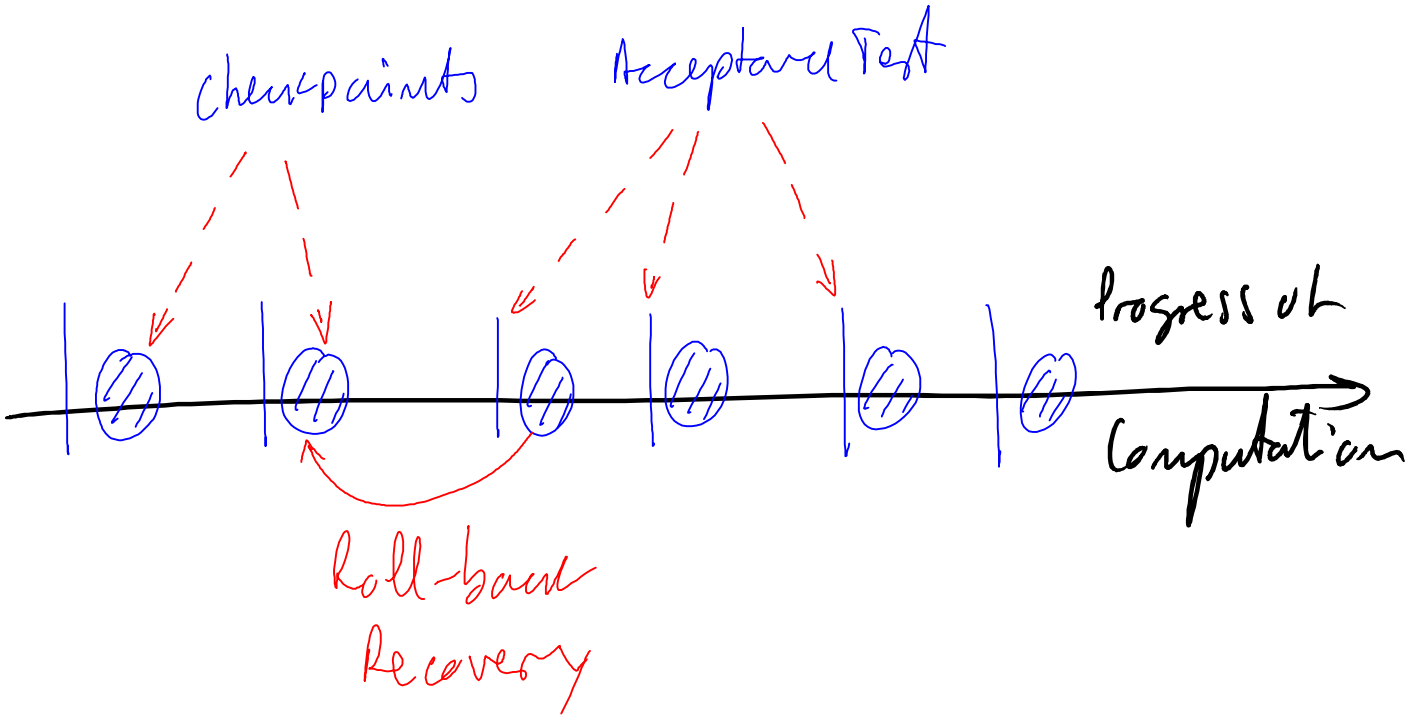
FIGURE
Software fault-tolerant structures; (a) *N* –version programming; (b) recovery-block approach.



CHECKPOINTING AND ROLL-BACK RECOVERY

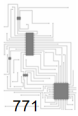
In this scheme as the computation proceeds, the system state is tested each time after some meaningful progress in computation is made.

- Immediately after a state-check succeeds, the state of the system is backed up on a stable storage
- If the next test does not succeed the system can be made to roll back to the last check-pointed state.
- After a roll back, from a check-pointed state a fresh computation can be initiated.
- This technique is especially useful, if there is a chance that the system state may be corrupted as the computation proceeds.



Time Redundancy-Implementing Backward Error Recovery

- Critical to the successful implementation of backward error recovery is the restoration of the state of the affected processor to what it was before the error occurred
- Corrective action, such as assigning another processor to carry on with the execution beyond this point or retrying on the same processor with the corrected state information, can be taken



Recovery Points

R_1, R_2, R_3 - points where we want to roll back.

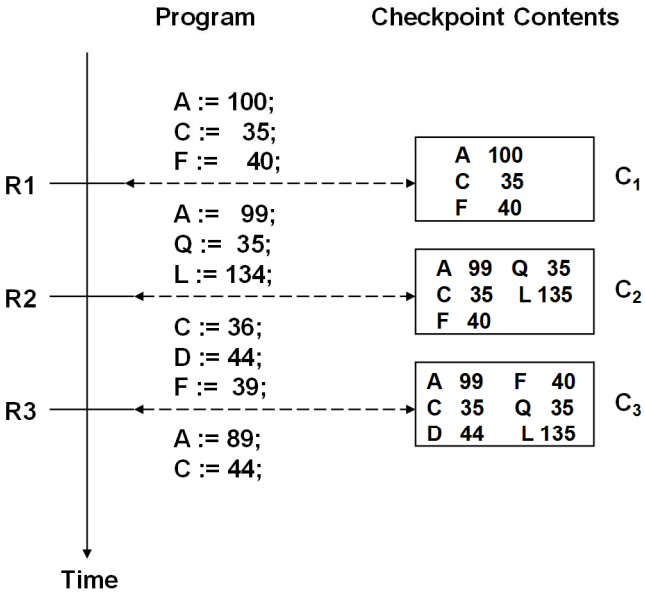
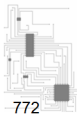


FIGURE
Checkpointing a program (Ri indicate recovery points; C_i indicate checkpoints).



Example

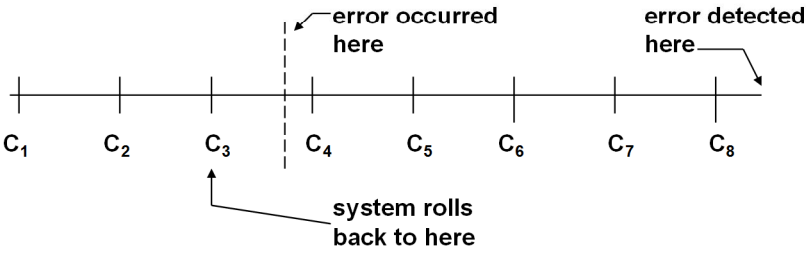


FIGURE
Checkpoints and rolling back the process (C_i indicate checkpoints);

Recovery Cache

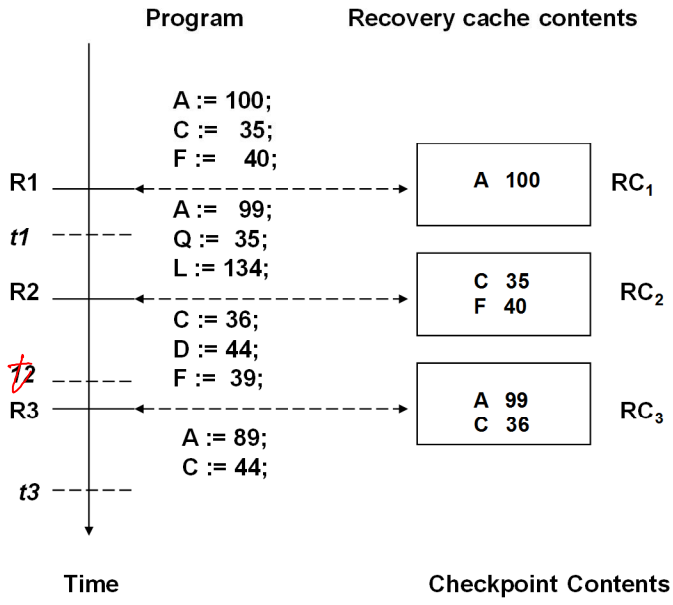


FIGURE
Recovery caches.

FAULT-TOLERANT SCHEDULING (READING)

November-28-12 9:38 AM

- We consider a static schedule --
- QUESTION: How a static schedule can respond to hardware failures?
  SOLUTION: ensure to have sufficient reserve capacity and a sufficient fast failure response mechanism to continue to meet critical-task deadlines despite a certain number of failures.

  PRACTICAL SOLUTION: use additional ghost copies of tasks, which are embedded into the schedule and activated whenever a processor carrying one of their corresponding primary or previously-activated ghost copies fails.
  - NOTE: the ghost copies do not need to be identical to the primary copies. They can be alternative copies that are simplified and produce poorer results but still acceptable.

ASSUMPTIONS:

1. A set of periodic critical tasks.
2. Multiple copies of each task version are assumed to be executed in parallel with voting or some other error masking mechanism.
3. Existence of the forward-error recovery mechanism to compensate for the loss caused by a task running on a processor that failed.
4. The fault-tolerant scheduling algorithm produces a schedule meant to find a substitute processor to run the future tasks that were scheduled on a failed processor.
5. Existence of a non-fault tolerant algorithm for allocation and scheduling. This algorithm can be called as a subroutine by the fault-tolerant procedure.
6. The allocation/scheduling procedure consists of an assignment part Π_a and an EDF scheduling part Π_s.

DEFINITION OF THE PROBLEM

PROBLEM: Requirements of the system are:

1. Run $n_c(i)$ copies of each version (or iteration) of task Ti;
2. Tolerate up to n_{sust} processor failures.

- The fault-tolerant schedule must ensure that, after some time for reacting to the failure(s), the system can still execute $n_c(i)$ copies of each version of task Ti, despite the failure of up to n_{sust} processors -- the processor failure may occur in any order.
- THE OUTPUT of the fault-tolerant scheduling algorithm will be a ghost schedule, plus one or more primary schedules for each processor.

- If one or more of the ghosts is to be run, the processor runs the ghosts at the time specified by the ghost schedule and shifts the primary copies to make room for the ghosts.
- There are two conditions that ghosts must satisfied in our schedule. These conditions are denoted with C1 and C2.

EXAMPLE: Figure below shows an example of a pair of ghost and primary schedules, together with the schedule that the processor actually executes if the ghost is activated. Of course, this pair of ghost and primary schedules is only feasible if, despite the ghost being activated, all the deadlines are met.

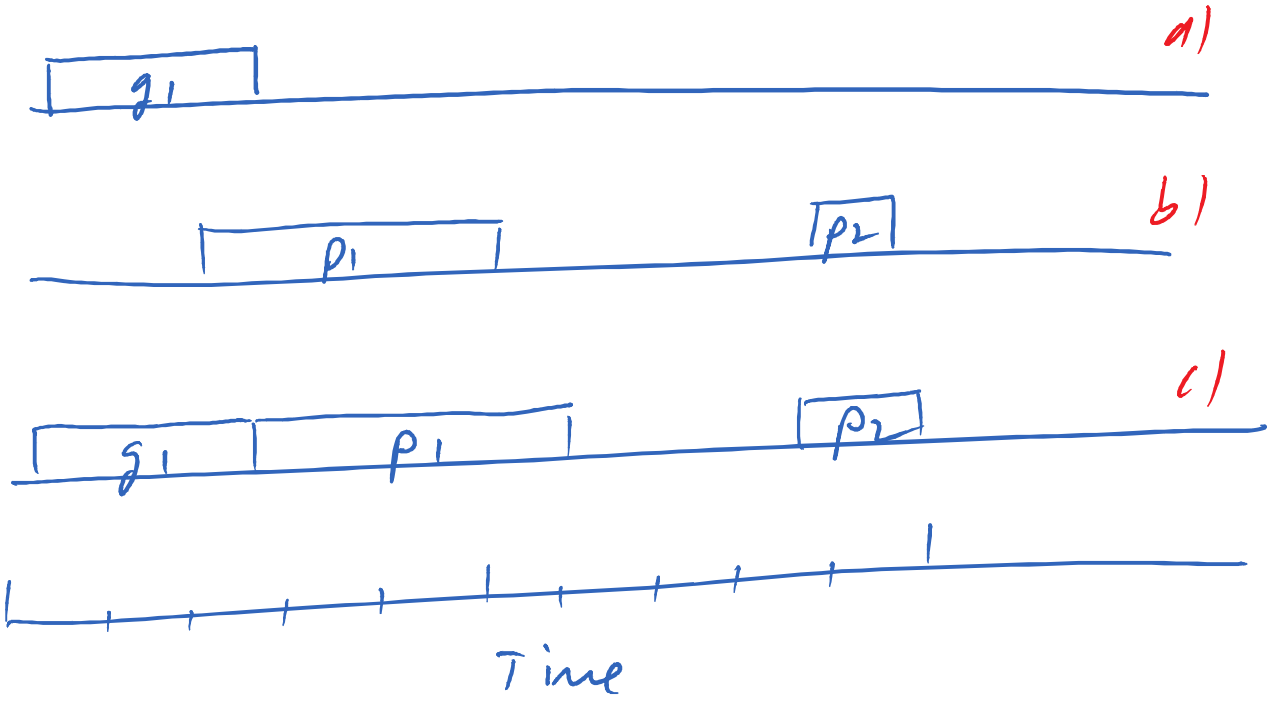


Figure. a) is the ghost schedule; b) is the primary schedule; c) is the schedule that results when g1 is activated.

- Note that here gi and pi refer to the ghost copy of a task and primary copy of a task, respectively.

FEASIBLE PAIR: a ghost schedule and a primary schedule are said to form a feasible pair if all deadlines continue to be met even if the primary tasks are shifted right by the time needed to execute the ghosts. Ghosts may overlap in the ghost schedule of a processor -- in this case only one ghost copy can be activated.

CONDITIONS FOR SCHEDULING COPIES

Conditions C1 and C2 for ghosts are the necessary and sufficient conditions for up to n_{sust} processor failures to be tolerated.

CONDITION C1 -- this condition assures the fault-tolerance of n_{sust} processor failures. Two or more copies (primary or ghost) of the same version must not be scheduled on the same processor.

CONDITION C2 -- this is the condition for describing overlapping requirements. Ghosts are conditionally transparent. That is, they must satisfy the following two properties:

1. Two ghost copies of different tasks may overlap in the schedule of a processor if no other processor carries a copy (either primary or ghost) of the tasks.
2. Primary copies may overlap the ghosts in the schedule only if there is sufficient slack time in the schedule to continue to meet the deadlines of all primary and activated ghost copies on that processor.

THEOREM: Conditions C1 and C2 are necessary and sufficient conditions for up to n_{sust} processor failures to be tolerated. (Proof -- See reference book).

The theorem above provides the conditions that the fault-tolerant scheduling algorithm must follow in producing the ghost and primary schedule.

- We present two simple fault-tolerant algorithms called FA1 and FA2.

FA1: under FA1, the primary copies will always execute in the positions specified in schedule S (this is the primary schedule), regardless of whether any ghosts happen to be activated, since the ghost and primary schedule do not overlap.

FA1 ALGORITHM

1. RUN Π_a to obtain a candidate allocation of copies to processors. Denote by π_i and θ_i the primary and ghost copies allocated to processor pi, i = 1, ..., np;
2. RUN $\Pi_s^*(\pi_i \cup \theta_i, i)$. If the resultant schedule is found to be infeasible the allocation as produced by Π_a is infeasible; return control to Π_a in step 1. Otherwise, record the position of ghost copies (as put out by Π_s^*) in ghost schedule Gi, and the position of the primary copies in schedule S.

DRAWBACK OF FA1: the primary task are needlessly delayed when the ghosts do not have to be executed -- while all the tasks will meet their deadlines, it is frequently best to complete execution of the tasks early to provide slack time to recover from transient failures.

ALGORITHM FA2

1. **Run** Π_a to obtain a candidate allocation of copies to processors. Denote by τ_i and θ_i the primary and ghost copies allocated to processor p_i, i = 1, ..., n_p. For each processor, p_i, do steps 2 and 3.
2. **Run** $\Pi_s^*(\tau_i \cup \theta_i, i)$. If the resultant schedule is found to be infeasible, the allocation as produced by Π_a is infeasible; return control to Π_a in step 1. Otherwise, record the position of the ghost copies (as put out by Π_s^*) in ghost schedule Gi. Assign static priorities to the primary tasks in the order in which they finish executing, i.e., if primary τ_i completes before τ_j in the schedule generated in this step, τ_i will have higher priority than τ_j
3. Generate primary schedule Si by running Π_{s2} on τ_i with priorities assigned in step 2

The drawback of FA1 does not occur in FA2 -- here an additional scheduling algorithm Π_{s2}, which is a static-priority preemptive scheduler is used -- given a set of tasks, each with its own unique static priority, Π_{s2} will schedule them by assigning the processor to execute the highest-priority task that has been released but is not yet completed.
THEOREM. The ghost schedule Gi and primary schedule Si form a feasible pair -- PROOF. The primary tasks will complete no later than the time specified in $\Pi_s^*(\tau_i \cup \theta_i, i)$ even if all the space allocated to ghosts in Gi is, in fact occupied by them.

EXAMPLE (Home Work). Consider the case where a processor p has been allocated ghosts g4, g5, g6, and primaries π1, π2, π3. The release times, execution times, and deadlines are given in table below.

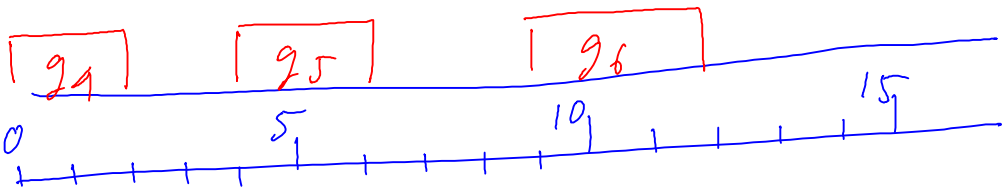
If g fails we need to run both g4 & g5

CASES TO BE SCHEDULED:

1. Suppose that there exists some processor q to which the primary copies of g4 and g5 have been allocated and we cannot overlap g4 and g5 in the ghost schedule of processor p.
2. There is other allocation of the same ghost and primary tasks to p. In this allocation the primary of g4 and g5 cannot be allocated to other processor. As a result, we can overlap g4 and g5.

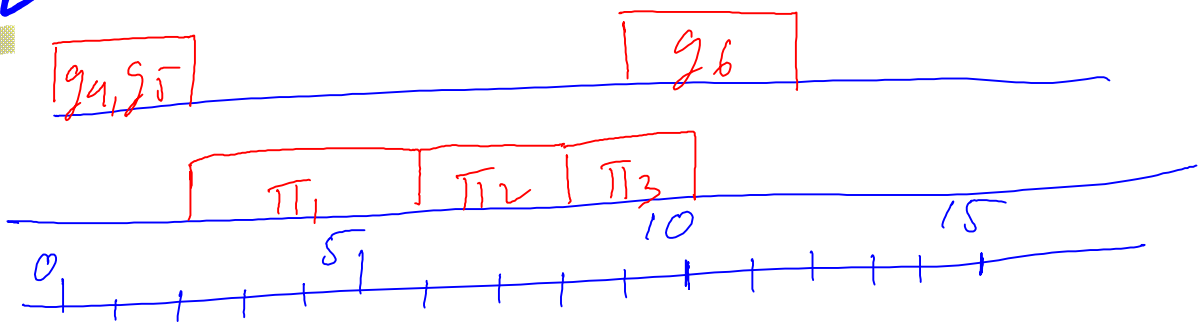
	π_1	π_2	π_3	g_4	g_5	g_6
Release time	2	5	3	0	0	9
Execution	4	2	2	2	2	3
Deadline	6	8	15	5	6	12

CASE 1



Allocation $\pi_1, \pi_2, \pi_3, g_4, g_5, g_6$ to p - infeasible

CASE 2



Feasible schedule for p

Cyclic Codes [Reading]

- Cyclic is carried out by multiplying the word to be coded by a polynomial, called the generator polynomial
- All additions in this process are modulo-2
- Multiplication by Xn essentially means shifting by n places
- Example …

Let us consider more complex multiplication. Multiply the word to be coded, $1+X+X^5$ (representing the number 100011), by the generator polynomial $1+X+X^2$ (representing 101). We have $1+(1+1)X+(1+1)X^2+1X^3+0X4+1X^5+1X^6+1X^7$. Doing the additions modulo-2 (which means putting them through exclusive-OR gates) results in $1+0X+0X^2+1X^3+1X^4+1X^5 = 1+X^3+X^5+X^6+X^7$, representing 11101001. The coded value corresponding to 100011 is therefore 11101001. The circuit in Figure 7.20 will carry out this coding operation. To begin, all flip-flops have their value set to 0. the flip-flops represent multiplication.

The coding circuit can be written down by inspection of the generator polynomial. Let us return to figure 7.20; note that the input is fed in serially, bit by bit, When we ask for the multiplication (with modulo-2 addition) by $1+X+X^2$, we are, in effect saying, "add, mudulo-2, the present input bit to the previous one (representing X) to the input before that one (representing X2)." the circuit follows immediately from what: The flip-flop produces that required delay, and the exclusive-OR gate carries out the modulo-2 addition.

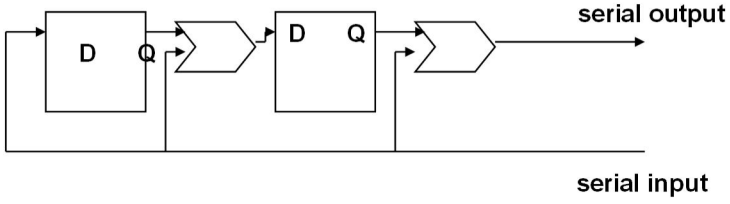


FIGURE:
Coding with the generator polynomial 1+X+X2.

THE END OF CHAPTER 5