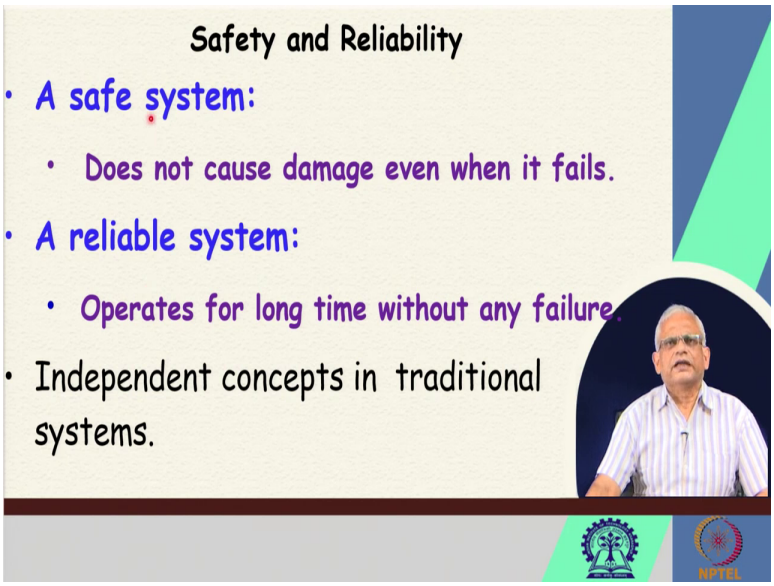
Real Time Systems
Professor Rajib Mall
Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur
Lecture 04
Characteristics of a real-time embedded system

(Refer Slide Time: 00:13)



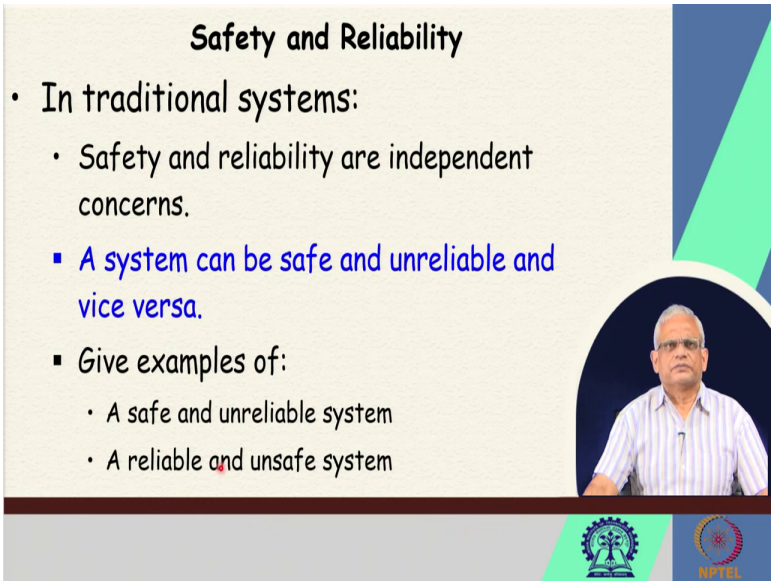
Welcome to this lecture. We are still in the introductory part of this course, where we are looking at some basic concepts and terminology, which will help us to understand the topics when we discuss the various important topics concerned with real time systems. Now, in this introductory lecture, in the last lecture, we are discussing about these important terms safety and reliability.

Now, let us just recapture. We said that a safe system is one where even if a failure occurs, there is no damage. On the other hand, a reliable system is one which can operate for long durations, maybe millions of hours of operation without failing. But a reliable system can be unsafe. In a traditional system, a reliable system can be unsafe. It operates for long time or expected to operate for a long time without failure. But even when there is a failure that can be a catastrophic. So, let us continue with this, let us develop this concept of safety and reliability.

So, a safe system does not cause damage even when it fails. On the other hand, a reliable system operates for long durations without any failure. And in the traditional systems, these are two

independent concepts. A system can be safe, but unreliable. And another system may be reliable, and unsafe. But we will see that in real time embedded systems, these two concepts are actually tied together.

(Refer Slide Time: 02:23)



A system can be safe and unreliable in a traditional system and vice versa. Now, can you think of an example of a traditional system, which is safe and unreliable and another example where it is reliable, but unsafe. So, one example we need, which is safe, a system which is safe, but is unreliable, and another system, which is extremely reliable, but it is unsafe system.

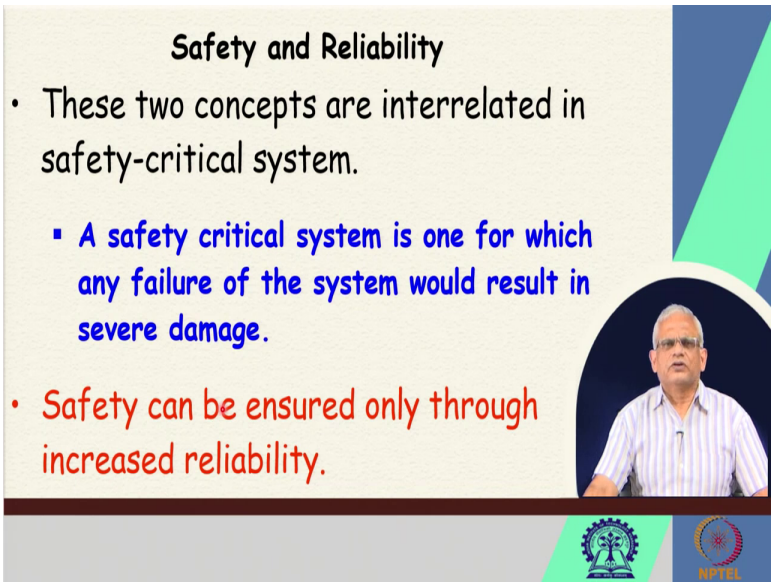
One example that I can give is that, let us say we have word processing software which we are using, which is extremely unreliable. It hangs every now and then. But one good thing about this word processing software is that, before it goes into a hang, it saves whatever work we had done. And we restart and we see that nothing is lost actually.

So, we can say this word processing software is unreliable. It fails quite frequently, maybe several times in a day. But then it is safe because it does not cause us damage. It has saved all the work that we had done. We did not lose anything. If it did not save it, it will be an unsafe system, because it would have incurred damages.

But now let us try to give the example of a system which is reliable, but unsafe. One example that I can think is let us say a gun or maybe a knife. Now, let us look at the gun. The gun is extremely reliable, works for several years without any problem. But now let us say the gun has got a bit of damage or maybe there is something stuck inside. Now, when we fire it, the gun explodes and the person using the gun might die. So, it is a reliable system, but when it fails, it causes a lot of damage. So, it is an unsafe system. Similarly, maybe a knife, it is very reliable, but let us say somebody was holding it incorrectly and it caused a lot of damage.

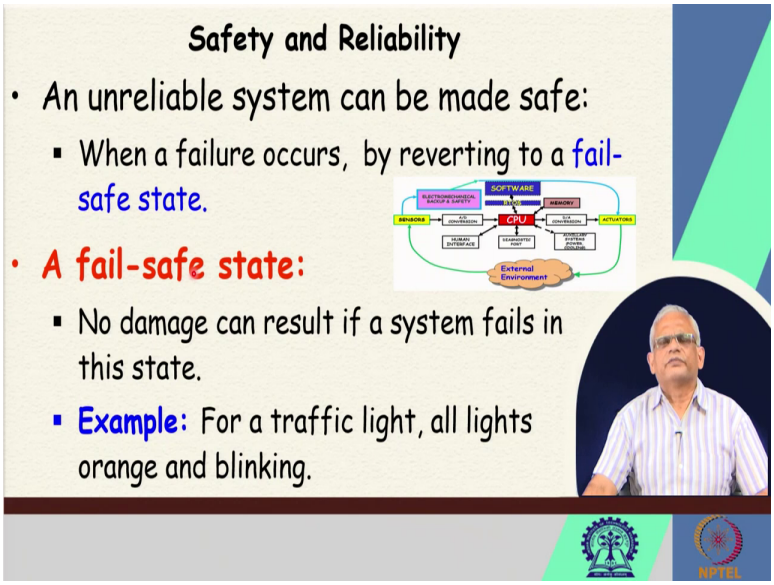
So, these are examples of reliable but unsafe systems and we had seen safe and unreliable systems. So, in traditional systems, we can find a system which is safe and unreliable, safe and reliable, reliable and unsafe, reliable and safe and so on. All possible combinations you can get. But now, let us look at real time embedded systems.

(Refer Slide Time: 05:50)



In the real time embedded systems, many times safety and reliability are interrelated and we call these safety critical systems, because here any failure would result in severe damage. So, we cannot have a failure here, because any failure will cause damage. So, it has to be, it should not fail at all. It should be extremely reliable such that there is no failure at all. And here safety is ensured through increased reliability. Now, let us just explore this little bit further.

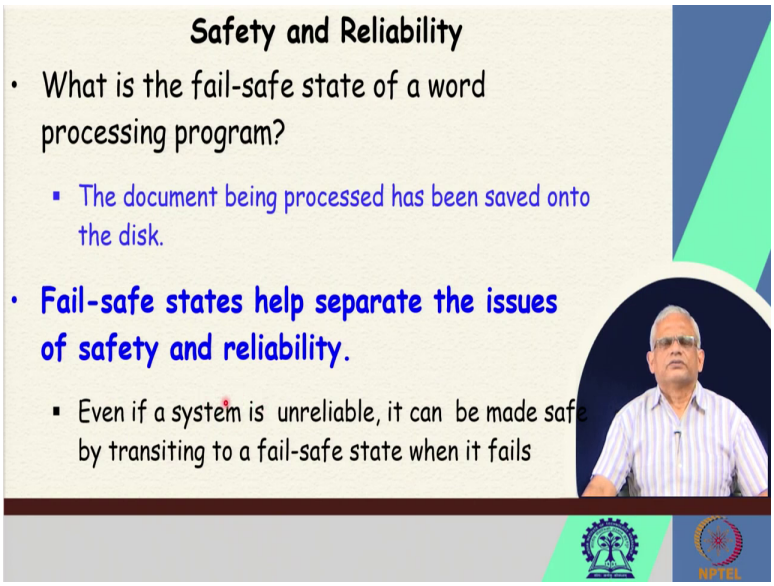
(Refer Slide Time: 06:47)



In the word processing software, we had an unreliable system. It failed very frequently, but we made it safe because it saved the data before failure. So, an unreliable system can be made safe by reverting to a failsafe state before the failure and look at this embedded system. Here we can have a failsafe system like let us say chemical plant; we have an electromechanical safety and backup arrangement. Where there is a failure, we enter into a failsafe state by switching on or by activating the electromechanical system. So, we have a failsafe system for the chemical plant.

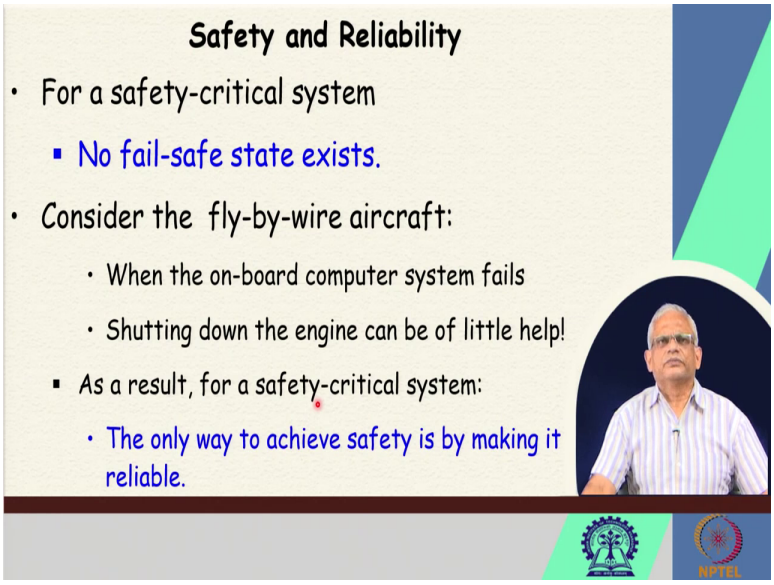
A failsafe state when it is entered, there will be no damage. Another example let us say look at this traffic light controller. These are also systems where if the system fails then there can be severe accidents. But there is a failsafe state here. The failsafe state is that all lights are orange and blinking and then the vehicles they know that the system is not working. They will take precaution. But if the traffic light fails when all lights are green, then there is going to be accidents. So, whenever there is a failsafe state, even when there is a failure, we may not have damages. But unfortunately for many of the real time embedded systems, there is no failsafe state.

(Refer Slide Time: 09:09)



Let us just look at the example of, ok this is an example of a word processing program where we had a failsafe state, we saved all data onto the disk. But what about, whenever we have a failsafe state for a system, safety and reliability become independent issues. Even if there is a failure, we still have safety. But what about an aircraft let us say the aircraft auto mode it fails.

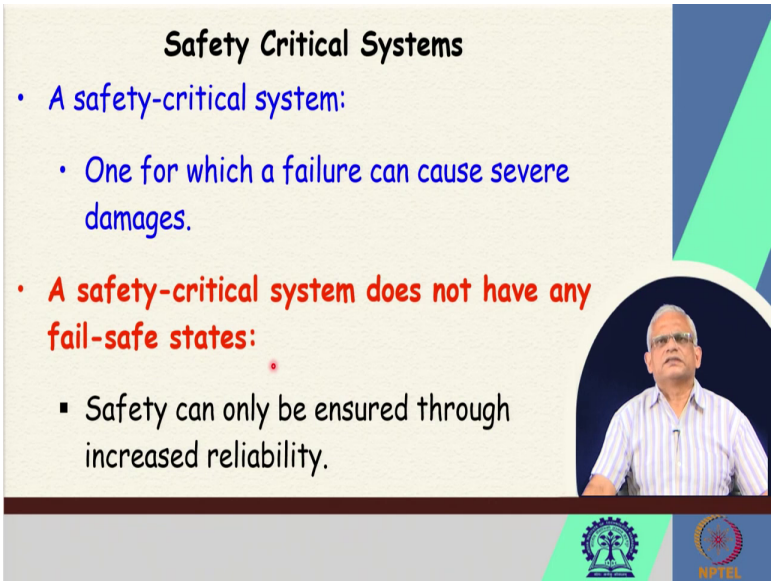
(Refer Slide Time: 09:56)



We do not have a failsafe state in that or a fly-by-air where aircraft, let us say there is a computer in control of this aircraft and there is a failure of the processor. In this case, we can just switch off the embedded system and activate some electromechanical device that would not help. So, the aircraft is going to crash. Even if we start at try shutting down the engine that does not help. So, that means that for these kinds of systems, there is no failsafe state.

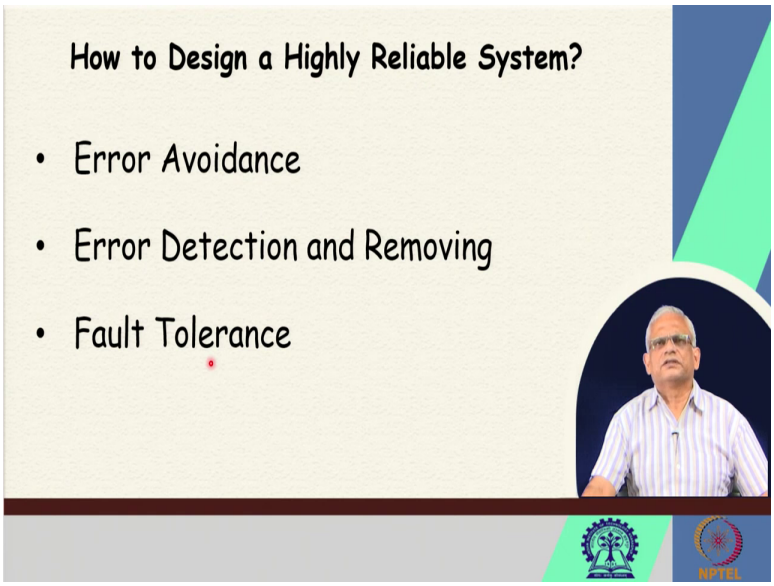
Now, let us say that a safety critical system is one where there is no failsafe state and there is going to be damaged when there is no failure. According to this definition, the chemical plant was not really safety critical because we had a failsafe state before the system fails, we could activate the electronic mechanical system. So, in those systems where there is no failsafe state and any failure is going to cause huge damage, we will call these as the safety critical systems. And in a safety critical system, the only way to achieve safety is by making them more and more reliable so that they never fail.

(Refer Slide Time: 11:47)



So, now let us come to a definition of a safety critical system. A safety critical system is that any failure can cause severe damages and there are no failsafe states. And safety can be ensured through increased reliability. Many applications of embedded real time systems are actually safety critical. Not only that failure can cause severe damage, but there is no failsafe state.

(Refer Slide Time: 12:26)

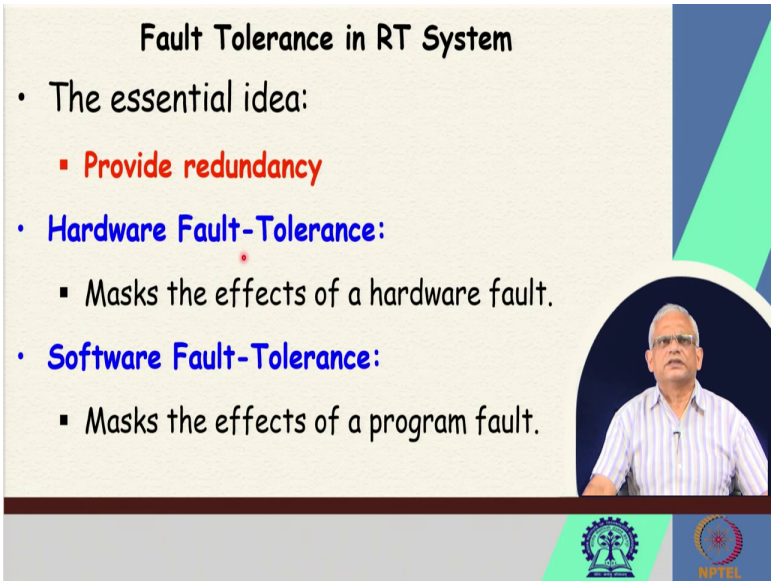


But how do we get a highly reliable system so that it will never fail or almost never fail? Let us say nuclear plant. We have to ensure that its reliability is such that even after million hours of operation there is no failure or maybe a billion hours of operation there is no failure. How can we achieve this? There are three main techniques to achieve a highly reliable system. One is error avoidance, how do we have error avoidance. We develop the system with care. We follow software engineering principles. We specify it carefully. We do formal verification. We design it carefully. Write the code using utmost precaution, and that was about error avoidance.

And the other way to achieve highly reliable system is even if there are some errors, after we have tried our best to have the error avoidance, the few errors that were remaining we should be able to detect and remove it. How do we do that? Testing, we test it thoroughly. And even after error avoidance, error detection and removing maybe just a few remain. For this we use fault tolerance. The failure can be in hardware or in the software. We will have fault tolerance in hardware and software. But as you will see that there are different degrees of success with respect to fault tolerance with respect to the hardware and software.

The hardware errors can be easily masked. Hardware failures can be easily masked by fault tolerance. It has been extremely successful, but not so successful for software. Fault tolerance achieving for software is very difficult problem. We will just briefly explore why so.

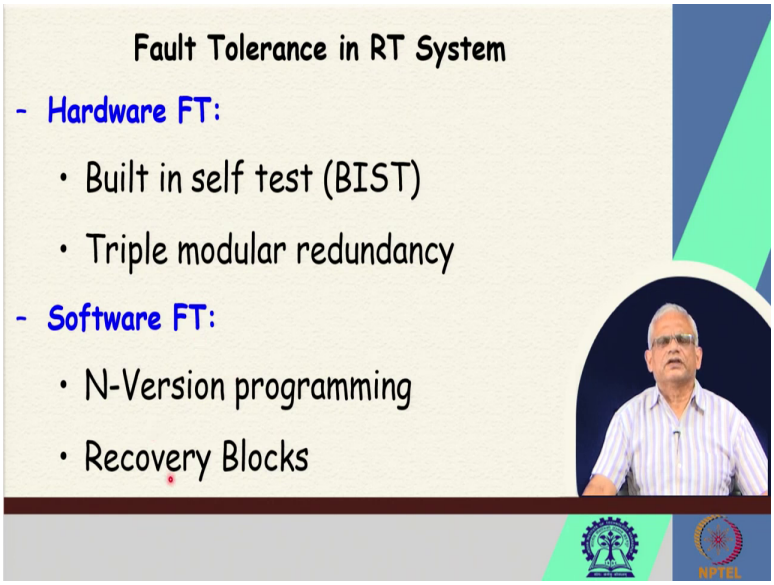
(Refer Slide Time: 15:23)



The essential idea between any fault tolerant system is that provide redundancy maybe two different hardware doing the same thing or multiple hardware, maybe three, four, redundant copies of the hardware are in operation, one may fail and then we can get the correct result from the other functioning hardware devices. So, even if there is a failure of one hardware, we can mask the effect, because the other hardware devices will be operating correctly.

On the other hand, the software fault tolerance, we need to mask a program fault which was undetected during testing. Somehow the fault remained and we need to mask it. Now, let us explore this a little bit further. What are the different ways to achieve hardware fault tolerance and software fault tolerance?

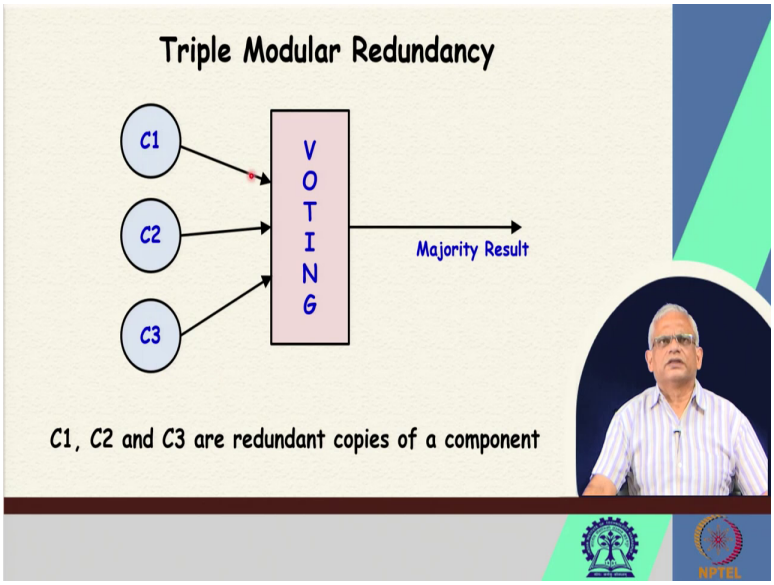
(Refer Slide Time: 16:44)



The hardware fault tolerance one is built in self-test, triple modular redundancy, and for software fault tolerance N-version programming and recovery blocks. So, these are popular techniques. So, built in self-test, it tests the hardware and isolates the one that is not working satisfactorily and switches in a correct working hardware.

Triple modular redundancy, here we have redundant hardware like 3 or 4 copies operating and there is a voting mechanism. And the result produced by the majority of the hardware is considered even if there is one failure that will get masked. In software fault tolerance, we have N-version programming and recovery blocks. We will just explore this in some detail.

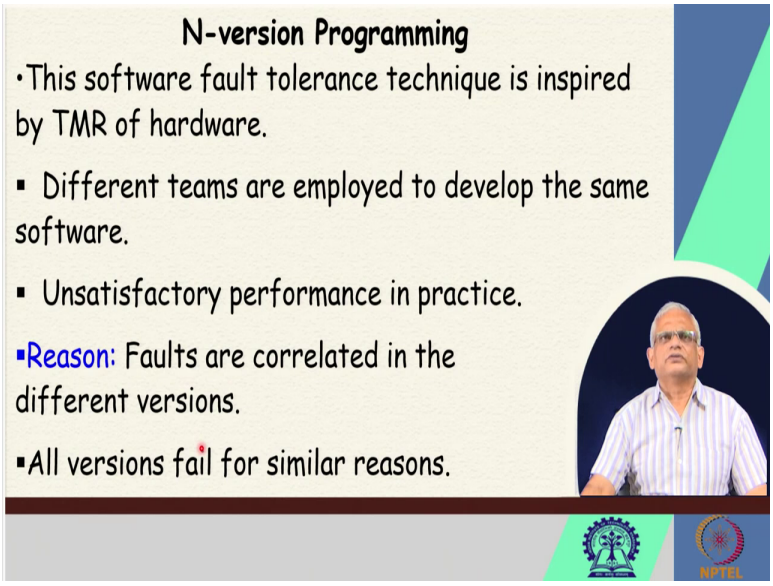
(Refer Slide Time: 17:53)



First, let us look at the triple modular redundancy. It is a popular hardware fault tolerance technique. Here we have redundant copies of the hardware C1, C2, C3 are the redundant copies. Now, the results they produce are voted. And even if one hardware failed, let us say it just did not produce any result or maybe it produced an erroneous result, still the majority would produce the correct result and this is considered.

And for hardware remember that the result is typically whether a signal is going high, signal is going low or maybe a number computation that whether the computed number matches, so this can be easily done by a voting mechanism. The voting mechanism is easy to design here for hardware and you can easily isolate the component which is faulty, it is not producing any results or is producing an erroneous result all these cases we can mask the failure by voting has been extremely successful for hardware for let us say disk storage, the raid disk storage, redundant array of disks. So, those are redundancy used to achieve reliability, fault tolerance.

(Refer Slide Time: 19:50)



Now, let us look at other techniques. Let us look at the software technique which is the N-version programming. This is the software fault tolerance technique. This is, if we think of it, it is actually an adaptation of the triple modular redundancy technique of hardware. Here, for a given problem, we develop 3 or 4 versions of the same software. And these versions of the software are developed by different teams. They develop the same software.

Maybe we can isolate these teams so that they do not discuss their design etc., do not become very similar. We can make them work at different places so that they produce independent copies of the software. And then we can have a voting mechanism here. The N-versions the result produced by them will be voted. And of course, let us say some version produces result very late, we can ignore that. Just like in hardware, there was a hardware which never produced result, and we just masked it. And here software may produce result late and we can ignore that. The other results we will consider.

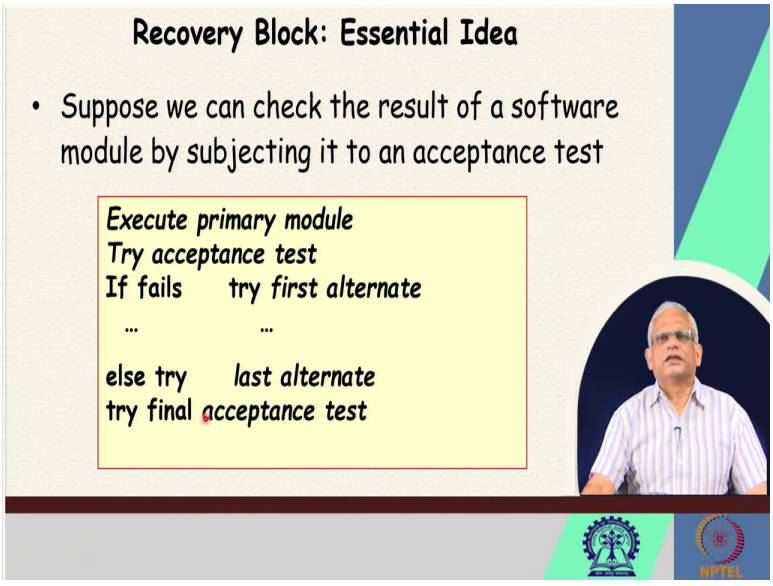
Otherwise, we will just take a voting and see the result which is produced by majority of the software versions, we will consider that. But in practice, in software, it has not been very successful. Even though in hardware triple modular redundancy is a very effective technique to

provide hardware fault tolerance, but software fault tolerance, this technique is not very successful. What is the reason?

One reason is that the faults are correlated in different versions. That is a fault which exists in one version also exists in the other versions. Even though there are independent teams working who never communicated with each other, they developed an isolated way, but still they commit the same mistakes. Why is that? Possibly because the teams they only do mistake in the difficult parts, the ones that are hard to grasp. The simple things they do not do mistake.

So, whenever one version fails, other versions also failed for identical reasons. Even though the teams worked independently, they had correlated faults. Another reason may be that the specification which was given to these different teams, maybe that had a problem. So, even though the teams worked independently, but the source of the problem was in the specification. So, that can also happen. But finally, we find that all the N-versions, they fail for similar reasons. And fault tolerance is not achieved. So, far software fault tolerance is still a very difficult problem. Now, let us look at the other way to achieve fault tolerance that is recovery blocks.

(Refer Slide Time: 24:05)



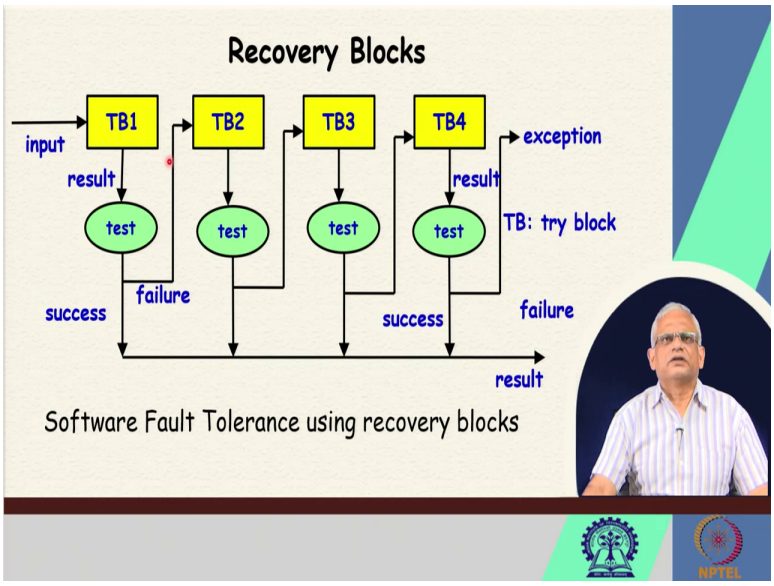
The main idea behind the recovery block is that, we assume here that we can have an acceptance test of a software component and decide whether the result is a pass or fail. If that test, we can

develop then what we can do is we can execute the primary module. That is the main module developed carefully. We execute and then try the acceptance test. And typically, it would pass the acceptance test and then our system works fine.

But if it fails, remember by that time lot of time has been taken up by this primary module execution and then trying out the acceptance test. But now, if it fails, then we have much less time. And here we can write a graceful degradation in the computation. We can write some modules, alternate modules, which take less time to compute and produce some workable result which may be doing the shutdown and things like that.

So, in the recovery block our primary module if it works fine no problem, but if it fails, we activate an alternate module and then try an acceptance test on that. If it passes the acceptance test then fine, otherwise we start another alternate module and so on. So, depending on the lacks time we have, we can try multiple alternate modules and once all the modules fail their acceptance test, we might do some exception handling. So, that is the main idea here.

(Refer Slide Time: 26:28)



And this one shows the recovery block mechanism through a diagram from the given input the try block 1 is the primary module. It is executed and the result is passed to an acceptance test. If there is a success, then the result is used. But if there is a failure, do not use the result and we

activate the try block 2. And once the try block 2 completes, then again it is subject to acceptance test. And if it passes, we can use the result, otherwise we activate the try block 3, otherwise try block 4. If all try blocks fail, then we invoke an exception handling mechanism.

So, this is the main idea behind the recovery block that is software fault tolerance using recovery block approach. Here also we have redundancy. And these blocks, because these are time constraint computation, the different try blocks use different algorithms and also, they successively take less and less time for computation. Maybe the first one took, let us say, 100 millisecond, the second one maybe 10 millisecond, the third one maybe 1 millisecond and so on much simpler algorithm just to initiate the graceful degradation of the system.

But one of the main problems of the recovery block approach is designing the acceptance test. Many times, it is very hard to design acceptance test, because we can know a very unusual result, let us say, it is producing a result which is several billions when the result was supposed to be produced within let us say a 6 to 7, 6 degree to 7 degree temperature, but it produced at a billion degree temperature. So, exceedingly high result or exceedingly low can be determined.

But what about result which is within the accepted range, but it is not correct. So, then it becomes very difficult to design the acceptance test. This is one problem with the software fault tolerance techniques, in general, and specifically, with respect to the recovery block techniques. So, we are at the end of this lecture. We will stop here and continue further into the introductory topic we will slowly explore more details of real time operating systems and so on. We will stop here and continue in the next lecture. Thank you.