Real Time System Testing
MIT 16.070 Lecture 32

hperry
5/4/01

Real Time System Testing (32)

- The next three lectures will focus on:
  - Lecture 30: *(R 11.3)*
    - How to minimize failure in real time systems
    - Methods used to test real time systems
  - Lecture 31: *(R 13)*
    - What is Software Integration?
    - Test Tools
    - An example approach for integration and test of the MIT 16.070 final project
  - **Lecture 32:** *(R 11.4)*
    - **Fault Tolerance**
    - **Exception Handling**
    - **Formal Test Documentation**

Fault Tolerance

What does it mean for a system to be fault tolerant?

The system can operate (although performance may be degraded) in the presence of a software or a hardware failure.

How do you design a fault tolerant system?

- Incorporate exception handling to tolerate missed deadlines or work around error conditions
- Design fault tolerant or redundant hardware or software into the system

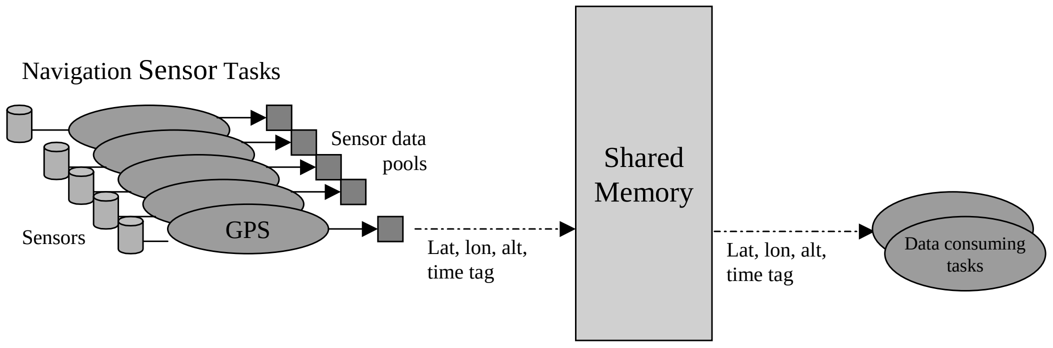
Ask yourself questions…how can the system fail?

hperry
5/4/01

Some Exception Handling Methods

Q: What if your system randomly misses data from a sensor?

A: Time tag your system data and use it only if the time tag has been updated since the last time it was used.



GPS data with time tag is written to shared memory
When other tasks come to read that data out of shared memory, they can discard the data if the new time tag = old time tag.

hperry
5/4/01

Some Exception Handling Methods

Q: What if your system randomly misses data from a sensor? (continued)

A: If data is too stale and the system cannot function properly without the new data, switch to a degraded mode of operation

- In a navigation system, this might be a backup navigation mode that operates on minimal inputs, separate from those that have failed
- In a space explorer robot system, this might be a zero-velocity state where the robot waits for communication of a set of new commands

Some Exception Handling Methods

Q: What if data goes bad for any number of reasons? What if bad data results in...

- Divide by zero in the software algorithm
- Data input from sensors out of a specified range (overflow condition for the algorithm or for the data type)

A: Add conditionals to your software to work around the problem.

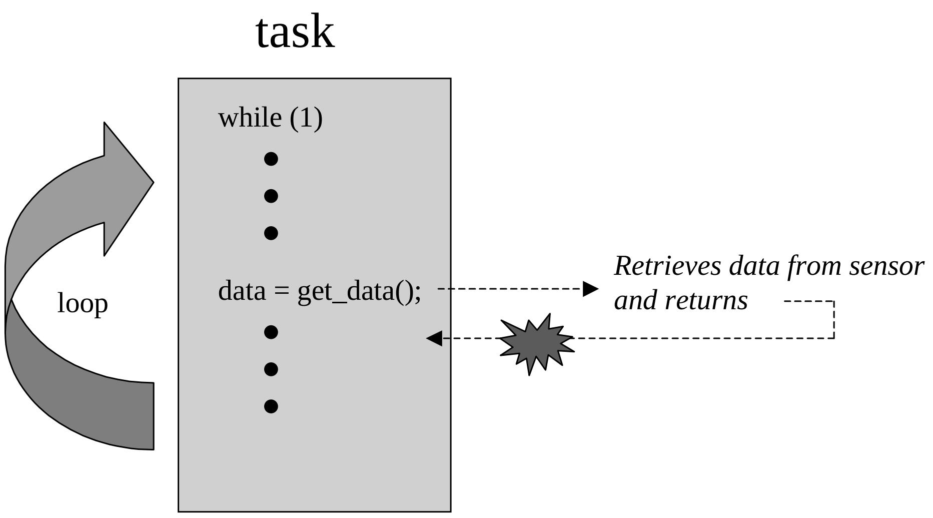
Instead of	Use
result = y/x;	if x !=0 then result = y/x;
data = get_data();	data = get_data();
	if data>1000, data = 1000;
	if data<0, data = 0;

hperry
5/4/01

Some Exception Handling Methods

Q: What if a critical task should hang during execution?

– For example, a task is waiting on data from a sensor, but the sensor loses its data link to the processor before it can provide the data)?



hperry
5/4/01

Some Exception Handling Methods

A: Relieve the task from waiting by…

- Designing functions with return values to indicate good/bad status
- Adding timeouts on retrieving the data to those functions to drive good/bad status return.

Note:

- This brings up the need for call by reference in C
- This concept can be expanded to operating system calls (posting mailboxes, releasing semaphores), library function calls, etc. provided the RTOS supports it.

hperry
5/4/01

Some Exception Handling Methods

The importance of "call by reference" to facilitate error handling

- C functions can return only one value
- If that value = status, how does the calling task (or function) get any information back from the task? The answer - Call by reference.

<u>Instead of</u>	<u>Use</u>
int get_data();	int get_data(*int x)
where the int returned is data:	*where int returned is the status and y is data:*
data = get_data();	status = get_data(&y)

hperry
5/4/01

Some Exception Handling Methods

<u>Without exception handling</u>:

```
int y;
int get_data();
void light_LED();
while (1)
        {
        y = get_data();                 /* get data from sensor software*/
        if (y>100)
                {
                light_LED();            /* turn on LEDs */
                }
          }                             /* end infinite loop */
```

hperry
5/4/01

Some Exception Handling Methods

<u>With exception handling</u>:

```
# include <stdio.h>
int y, **error;**
**int get_sensor_data(int* x)**
void light_LED();
while (1)
        {
        **error = get_sensor_data(&y);**                    /* pass the function the address of y, return error */
        if (!error)
                {
                    if (y >100) light_LED();              /* turn on LEDs */
                    }
          else printf("Error in data coming from sensor =  %d", error);

          }                                                /* end infinite loop */
```

hperry
5/4/01

Exception Handling Methods - The Tradeoff

- On the one hand, exception handling can guard against problems such as:
  - Erroneous mathematical conditions (divide by zero, overflow)
  - Tasks that hang waiting for inputs that will never come (due to failed hardware, poor communication link, software bug etc.)
  - Poor reactions to missed deadlines

- On the other hand, putting in all of this exception handling takes up resources (CPU time and memory) that must be worth the trade-off

- You must balance the two to achieve a robust software design that works within the timing and sizing constraints of the system

hperry
5/4/01

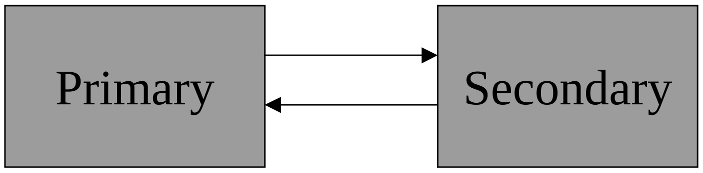
Fault Tolerance - Checking Hardware Resources

<u>How can a processor check its own status?</u>

- Built-In-Test (BIT)
  - Ongoing diagnostics of the hardware that runs the software
  - Interface checks
- CPU testing (done in the background)
- Memory testing
  - Checking for memory corruption due to vibration, power surges, electrostatic discharge, single event upsets, etc.
  - Use error detection & recovery schemes (CRC, Hamming Code)
- Watchdog Timers
  - Counting registers used to ensure that devices are still on line
  - CPU resets the timer at regular intervals. Timer overflow indicates a problem with the CPU

Fault Tolerance

Redundant Hardware Solutions - A two processor scheme

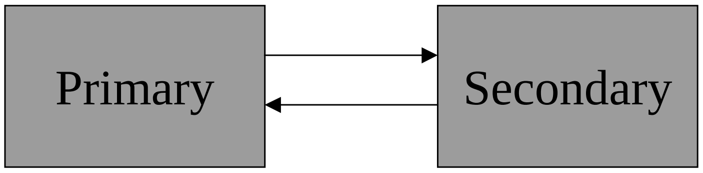


- Primary sends replica of all its inputs to Secondary
- Secondary runs same software as Primary
- Secondary checks for “pulse” from Primary to verify its health
- If pulse is absent, Secondary takes over the system
- Requires redundant communication lines to all system components
- Many military aircraft systems are built this way

hperry
5/4/01

Fault Tolerance

Redundant Hardware Solutions - A two processor scheme



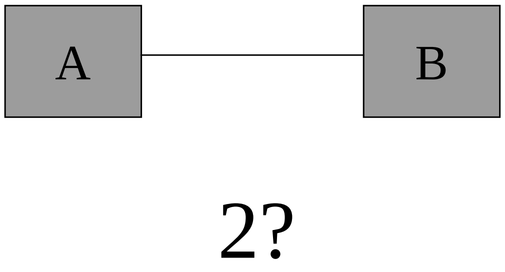
- When might this scheme fail?
  -
  -
  -

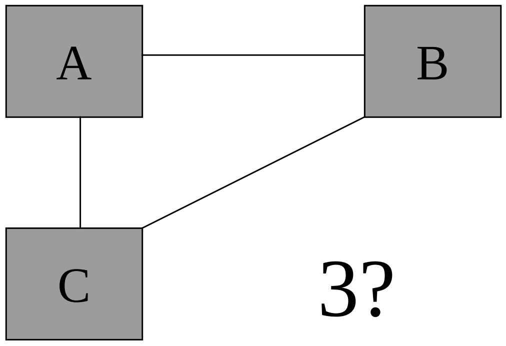
hperry
5/4/01

Fault Tolerance - Redundant Processors

- Computers can vote on who is worthy of staying in the system
  - Check “pulse” to be sure the computers are on-line
  - Compare data outputs from computations

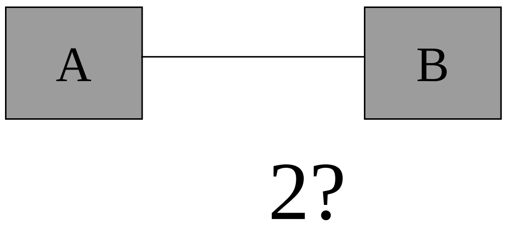
How many do you need?



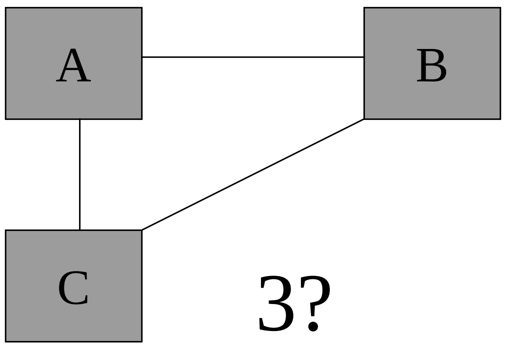


hperry
5/4/01

Fault Tolerance - Redundant Processors



A says B is sick
B says A is sick
Who is right?
Who should take over the system?



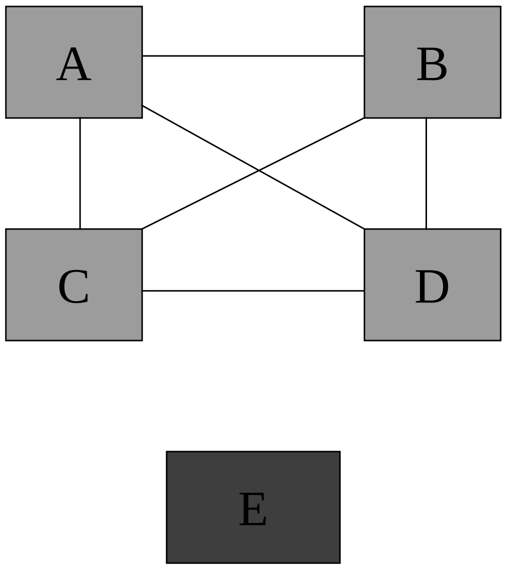
A says B is sick
B says A is sick
C says A is okay
C says B is sick
You might deduce that B is sick

But what if you lose one computer?
You must consider the probability of losing a computer given the catastrophe of being in a 2 computer case.

hperry
5/4/01

Fault Tolerance - Redundant Processors

- In some cases 4 computers are necessary, each checking the status of the other 3.



Is there any way that the 4 computer scheme can still fail?

hperry
5/4/01

Fault Tolerance - Redundant Processors

Who needs 4 computers and a backup? The Space Shuttle

- On ascent and landing/entry the Space Shuttle uses 4 identical computers and one backup. Why? A 1/2 second glitch in the guidance, navigation & control software will cause the shuttle to spin out of control.
- During rendez-vous and docking, 2 computers are used
- On orbit, only one computer is necessary
- In all cases, the backup computer is always available and runs a different set of software than the other 4, a technique known as N-version programming.

hperry
5/4/01

N-Version Programming

- Same system requirements for multiple implementations
- The different implementations of code are written by independent teams or contractors
- Eliminates the common software fault issue
- Often used as a backup system

hperry
5/4/01

TEST DOCUMENTATION

hperry
5/4/01

Real Time System Testing (32)

Test Documentation (Test Plan / Test Report)

- Includes an Executive Summary
- Describes test environment
- Identifies software to be tested
- Identifies tests that will be run on the software
- Includes a requirements traceability matrix
- Describes results of each test
- May have additional information provided as “notes”

hperry
5/4/01

Section 1 - Executive Summary

- System Overview
  - Purpose of the system
  - General operation of the system
  - History of system development
- Document Overview
  - Summarize contents of the test plan / report
  - Summarize key test runs and success/failure of tests
  - Includes an overall assessment of the project software

hperry
5/4/01

Section 2 - Software Test Environment

- Software Items Under Test
  - Identify what exactly is being tested (i.e. the workstation and handyboard software)
- Components in the Software Test Environment
  - Identifies operating systems, compilers, communications software, related application software, etc. <u>used to test the workstation & handyboard software</u>
  - Identify version numbers for the various software test environment
- Hardware and Firmware
  - Identifies computer hardware, interfacing equipment, extra peripherals, etc. used in the testing of the software
  - Describe purpose of each item
- Software Test Configuration (Diagram)

hperry
5/4/01

Section 3 - Test Identification

Identify planned tests for the….

- Serial I/O
- Thrust Controller
- Simulator
- Fuel Out Indicator
- Altitude and Velocity Display
- Integrated System

hperry
5/4/01

Section 4 - Requirements Traceability

- Identifies the method used for verification of the requirement:
  - Analysis
  - Inspection
  - Demonstration
  - Test
- Maps each software requirement to a test

hperry
5/4/01

MIT 16.070 Requirements Matrix for Final Project

Rqmt #	Description	Method	Test (if applicable)
1	The system shall be composed of a controller (implemented on a handyboard), a workstation simulation of the Mars Lander, a graphics package (which shall be supplied to the developer) and a serial interface (cable and software) to connect the controller to the workstation.	Inspection	n/a
2	The Mars Lander simulation shall begin to execute on the workstation.	Test	
3	The simulation ends when the Mars Lander softly comes to rest on the surface of Mars, when it crashes into the planet, or when it rises above its initial altitude.	Test **Demo**	
4	When the spacecraft is allowed to free fall (no thrust), it shall crash into the Mars surface in approximately 16 seconds.	Test **Demo**	
5	With thrust, the user should be able to land the spacecraft softly on Mars.	Test **Demo**	
6	The controller shall operate as the user's interface to the Mars Lander Vehicle.	Test	
7	Upon receipt of a data packet (1 byte) from the simulation, the controller shall: 1. Display vehicle altitude and vehicle velocity (both plus & minus) on the handyboard LCD.	Test **Demo**	
8	Upon receipt of a data packet (1 byte) from the simulation, the controller shall: 2. Light the LEDs if the "Fuel Out" bit is on.	Test **Demo**	
9	Upon receipt of a data packet (1 byte) from the simulation, the controller shall: 3. Format a data packet and send it to the simulation.	Test	
10	This data packet shall contain a "thrust on/off" indication. When the start button is depressed, the controller shall indicate a "thrust on" condition to the simulation. This condition will remain "on" until the stop button has been depressed (even through subsequent transmissions of the data packet to the simulation). When the stop button is depressed, the controller shall indicate a "thrust off" condition to the simulation. This condition shall remain until the start button is pressed.	Test	
11	The simulation will track fuel remaining and send a flag to the controller identifying when the fuel is out. If the vehicle has run out of fuel, the simulation will not process any more thrust commands from the controller.	Test	

hperry
5/4/01

Section 4 - Requirements Traceability (continued)

Rqmt #	Description	Method	Test (if applicable)
12	The simulator will have three types of output. It will: 1. send information to the controller specifying altitude, velocity, and whether there is fuel remaining;	Test	
13	The simulator will have three types of output. It will: 2. call a graphics package (provided) that will display a representation of the lander as it descends. The simulator must determine the condition of the vehicle using one of four conditions defined in the Graphics Package Interface section (see "vehicle-condition" variable);	Test	
14	The simulator will have three types of output. It will: 3. write altitude, velocity, thrust and fuel remaining information to a telemetry file using the same format as the simulation in PS8 (plus a fuel remaining column), and write an error message to the telemetry file if the graphics package returns an error condition.	Test	
15	The simulation will end when the vehicle reaches 0 meters of altitude or goes above its initial altitude.	Test (see rqmt 3)	
16	Simulation code will interface to the provided graphics package	Test **Demo**	
17	The serial interface should be used to continually send data back and forth between the simulation and the controller.	Test	
17	Data sent from the simulation to the controller shall correspond to the provided interface specification	Test	
18	Data sent from the controller to the simulation shall correspond to the provided interface specification	Test	

hperry
5/4/01

Section 5 - Test Results

- Includes an overall assessment of the software as demonstrated by the test results
- Identifies any remaining deficiencies, limitations or constraints that were detected by the tests
- Includes recommended improvements in the design, operation or testing of the software
- Includes a writeup of each test’s results

Section 6 - Notes

- Any general information that aids in the understanding of the test report
- May include
  - a list of abbreviations or acronyms
  - definitions
  - background information
  - test rationale
  - etc.

hperry
5/4/01

Real Time System Testing - Summary

- Over the last 3 lectures, you have learned that test and integration is an important part of developing a real time system.

  Why?

  A Real Time System must continue running in the presence of failure

  Therefore,

  We must design in fault tolerance
  We must find as many errors as possible before the system goes into use
  We must be methodical about how we test and document what we learn

hperry
5/4/01