Testing Issues for Real-Time Systems

- Part of the problem of testing any real-time system lies with the fact that they are for the most part asynchronous systems interfacing to asynchronous events and hence it is almost impossible to predict what they are doing at any one instant in time. For example what will an elevator be doing 5 seconds after we turn it on?

- Furthermore it may be practically or physically impossible to exhaustively test every aspect of the software/hardware design of a system such as a fly by wire airplane since this could theoretically take forever and cost a fortune.

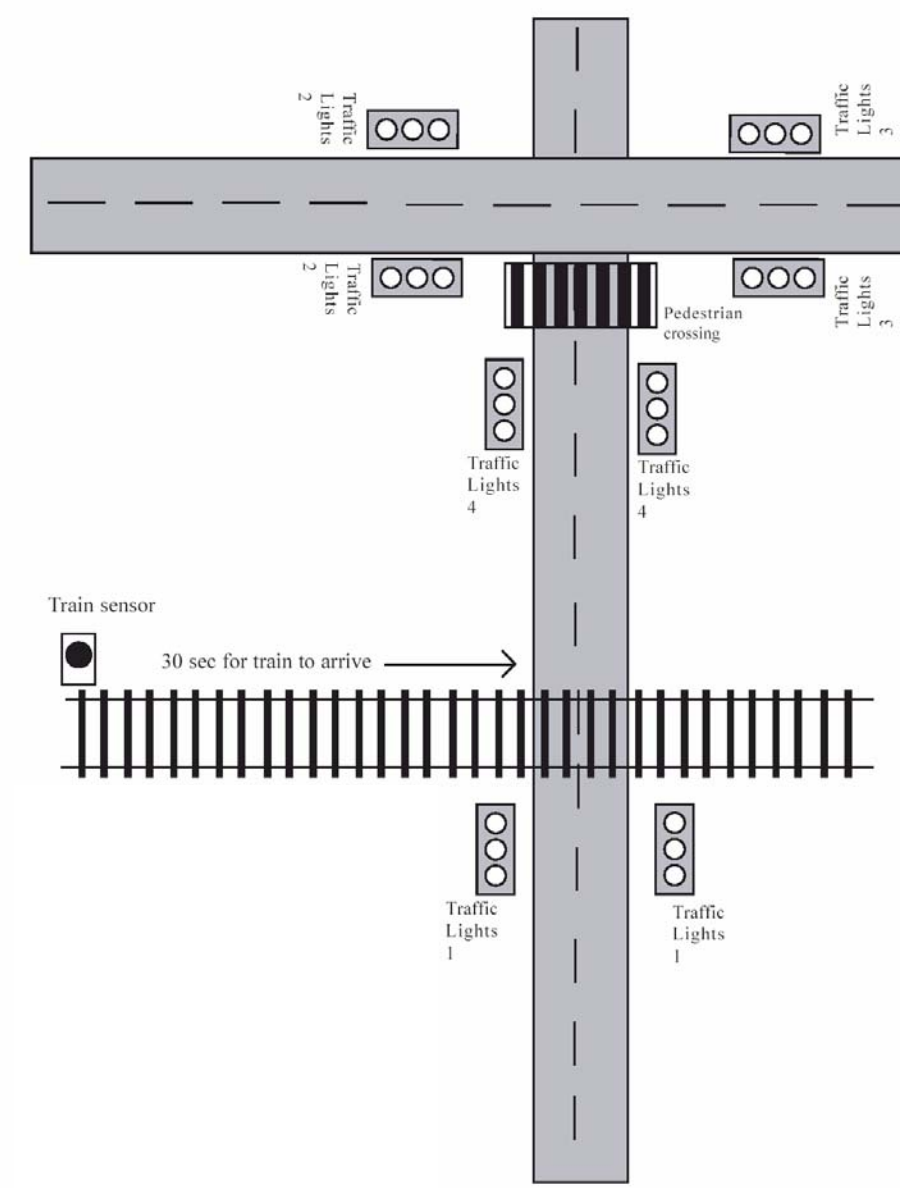
- For example, how do test the systems response to a pilot when he/she perform a sequence of operations on a Friday night flying over the equator, while changing time/zones or control towers, with the undercarriage up/down etc. ?

- As a practical example, it is well known that Motorola and Intel only test each of their CPUs for less than 1 Sec during production. Even though to exhaustively test each combination of instruction/addressing mode/register plus all combinations of 32 bit data could take several weeks. Even then such testing ignores asynchronous events such as what happens if an interrupt, exception of bus-error occurs while executing instruction 'X', using register Y and Data Z.

- Real-time systems are also difficult to exhaustively test since it may be virtually impossible to recreate the set of situations or circumstances that may have caused the system to fail in the first place and hence figure out where and why the system failed.

Real-life, Real-time system that killed due to inadequate testing

- As a train passes the sensor, traffic lights 1, 2 and 3 would turn red, while lights 4 would turn green to allow the stationary traffic on the railway tracks to clear out of the way.
- However, if a pedestrian happened to be crossing then lights 4 would be held up for a critical 30 secs before turning to green. If traffic happened to be on the line at this exact same time, it would be hit by the train. Think this cannot happen ? Think again.
- This is the exact same setup that occurred in one town in the USA resulting is several fatalities over a 2 year period, including a school bus whose length meant that whenever it drew up to lights 4, its tail was always hanging over the tracks .
- The timing and the circumstances leading to the crash were so rare and yet so critical that it was almost impossible to re-create the situation and analyse what was going on.
- Only a chance observation from a store holder allowed engineers to piece together events.



Single and Multi-tasking Real Time systems

- Irrespective of whether a system is 'hard' or 'soft', 'event' or 'time' driven. A further classification of real time systems exists which are based on the following concepts.
  - Single tasking real-time systems
  - Multi-tasking real-time systems (concurrent systems)

Single Tasking Real-Time Systems

- Such systems usually comprise a single program with a single function main() which is executed sequentially from start to finish. These are typical of the sort of programs you have written thus far in your studies
- Such systems are designed to carry out a single task or several tasks carried out sequentially and are by far the easiest systems to design, debug and test
- They are often used in small dedicated applications such as engine management, washing machines, microwave ovens etc.

Multi-Tasking Real Time Systems

- These systems involve the use of multiple co-operating processes, i.e. several programs, each with a function main() and each responsible for implementing a sub-set of the systems overall functionality.

- The important difference here is that all of these programs could well be running at the same time, i.e. concurrently and this in turn poses its own set of unique problems that make the design, debug and test of such system much more difficult than that of single tasking systems

Implementing Multi-tasking systems

- Multi-tasking systems can be realised in a variety of different ways

  - Pseudo Multi-Tasking (i.e. faking it)
  - Multiple CPUs (true multi-tasking)
  - Time sliced Single CPU (inexpensive compromise)

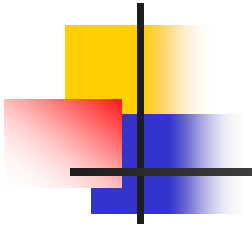
Pseudo Multi-tasking Systems:

- Here a form of fake multitasking is implemented by the system where by through clever programming it looks as if the system is executing several tasks in parallel.
- With a carefully crafted program decomposed into smaller tasks, each of which are brief and can be executed, repetitively, we can design a system that gives the illusion of 'concurrency'.
- The program below demonstrates pseudo multi-tasking using a loop built into a users program. "Processes" or activities/tasks are simulated via subroutines/functions called repetitively within the program.

```
int main(void)
{
        while(1)  {                          // forever loop
                  test_switch1() ;           // task 1
                  monitor_temperature() ;    // task 2
                  control_flow_rate() ;      // task 3
                  display_results() ;        // task 4
        }
}
```

Advantages of Pseudo-Multitasking Systems

- Processes are easy to design and control, being nothing more than simple functions or subroutines in 'C' etc.
- No complex or expensive operating system required, solution is smaller, cheaper and requires less of a learning curve, using existing skill sets.
- Process communication is limited but can easily be achieved using shared or global variables accessible by all parts of the program.
- Priority of processes can easily be changed by having the same process/function called or invoked more often than others in the system. That is, the function can be called more times with each iteration of the loop.
- Complex interactions between processes can be simplified, e.g. shared memory, mutual exclusion etc. The processes can resolve these issues themselves because they are part of the same physical program.



Drawbacks to Pseudo Multi-tasking Systems

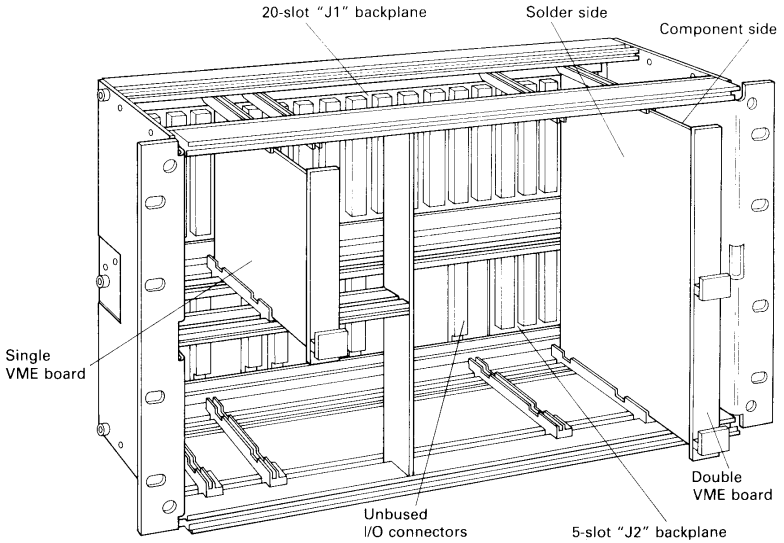
- "Processes" (implemented as functions) must be brief and must be designed such that they can be called repetitively within the system. A process cannot consume CPU time forever, neither can it get involved in some operation that would cause it to delay returning from the subroutine, such as getting input data from an operator, or over a network, otherwise the illusion of concurrency will be lost.

- A "process" that crashes due to bugs may have a severely detrimental effects on the system as a whole. In effect it may wipe out all the other processes, if not by corruption then due to the fact that it may not return from its function and thus gives rise to the problem above.

- Asynchronous processing via interrupts is difficult, since it is not easy to associate a particular process/function with an event/interrupt. This scheme works best with polled and hence soft real-time system. However a process cannot sit in a polling loop inside the function as this will suspend all other processes, the polling has to be done once for each call.

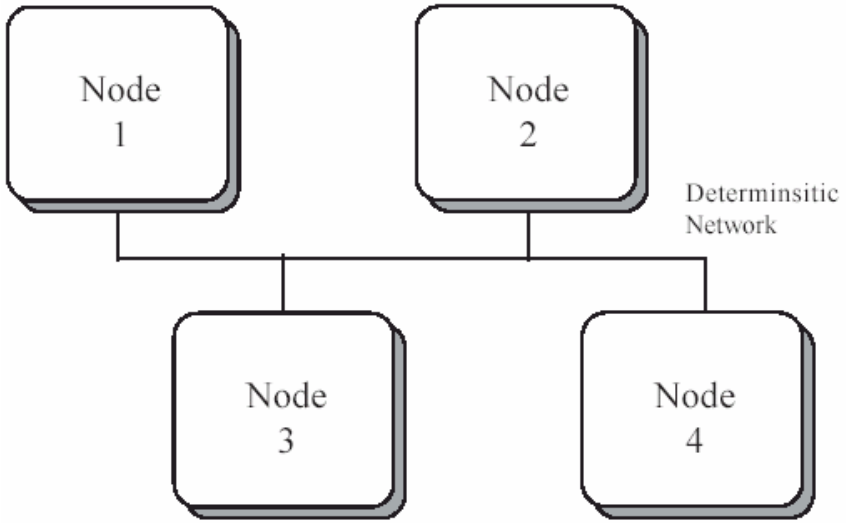
Multiple CPU Approach 1: Shared Backplane

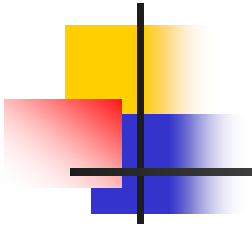
- Here, the system is composed of several CPU distributed on motherboards which plug into a backplane such as VME bus. That is, each CPU is dedicated to a single process/activity.
- CPU's (and hence processes) communicate with each other via shared memory and peripherals along the backplane (They may also have their own "on board memory and peripherals" to reduce bus activity)
- A bus arbitration module (BAM) arbitrates between CPU's when more than one wishes to communicate along the bus.
- This type of design is useful if the number of processes, I/O and memory requirements of the system are fixed and known at design time. It is not easy to accommodate dynamic process creation with this scheme as a change in the number of processes requires a change in number of CPU cards.
- Can be very expensive but performance very high
- Not very flexible. Using a backplane means all CPU's have to sit in same 19" rack leading to long cable runs back to the equipment being controlled. A case of all your eggs being in one basket.



Multiple CPU Approach 2: Networks

- Here each process still has it's own dedicated CPU but these are networked to allow the process to sit closer to the environment that it controls.
- Use of a real-time network with deterministic characteristics is important which pretty much rules out Ethernet.
- You want to be able to calculate worst case, how long a message will take to get from one node to another in the presence of several nodes who may all be transmitting.
- Best approach is to use token passing.
- Here, a node can transmit only when it has a token and can only transmit up to one packet of information before it has to release the token for use by any other node.
- Here the delay in getting a message from one node to another can be calculated and is a related to
  - Speed of network in Mbps
  - Size of message in Bytes
  - Size of a message Packet in bytes
  - Number of nodes on network





Multi-tasking Systems Using Time slicing and an Operating System

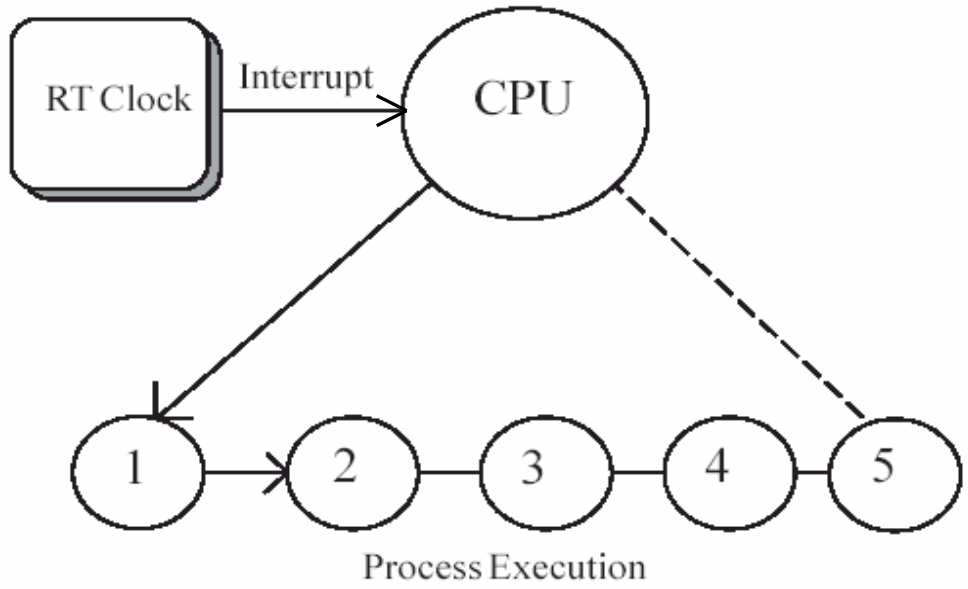
- A more practical approach to multi-tasking (which can also be applied to Multi-CPU and distributed networked systems) is to make use a Multi-Tasking Operating System (MTOS or Kernel) in conjunction with a single CPU and time slicing.

Time Slicing: A Definition

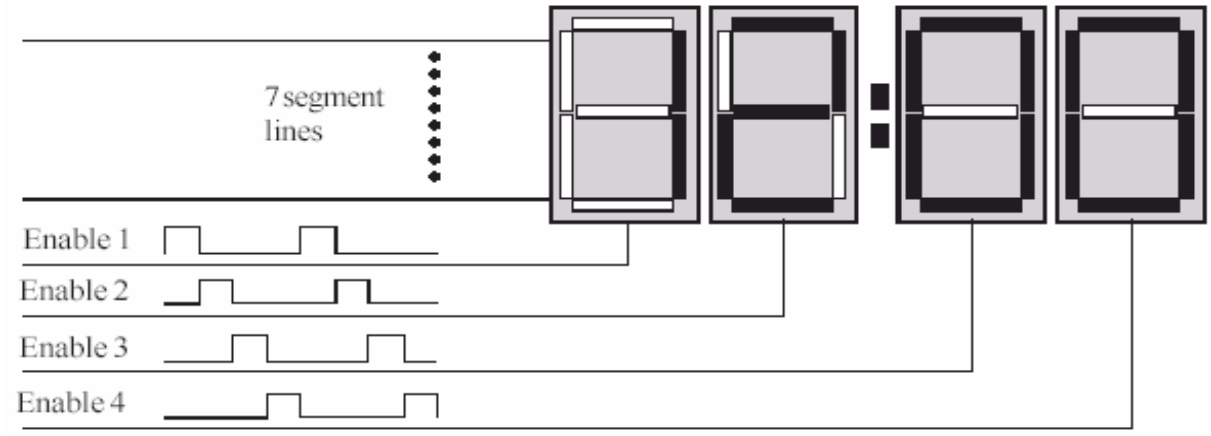
- Time slicing is the subdivision of available CPU time, into 'packets' or time slices which are then allocated under the control of an operating system kernel, to processes or programs that are able to make use of them.
- It is, in effect, an attempt to create concurrency by getting a CPU to forcible swap from executing one program to another very rapidly. It is of course, still an illusion, since with a single CPU, only one process can ever be running at any one instant.
- If the swapping is done frequently enough, it looks like all the processes are executing at the same time and a human operator will be unaware that their programs are being 'swapped in' and 'swapped out' of the CPU at regular intervals.
- Note that this is not the same as Pseudo Multi-tasking. In that case, we only had one process/program running and we swapped between functions within it on a voluntary basis. That is the tasks within needed to be aware of each other and co-operate so as not to 'hog' the CPU.

The Real-Time Clock (RTC)

- Time slicing is achieved with the aid of a real-time clock (RTC) which generates interrupts to the CPU periodically, say every 10-50mS depending upon system.
- Such interrupts are called Ticks.
- The purpose of the Tick is to force the CPU to break away from the executing its current program and force it, with the aid of the host operating system to resume processing of another task.
- Such an approach is analogous to time-division multiplexing. We can see this illustrated in the Diagram below.



- The illusion of concurrency can best be seen by considering an analogy with one of the old style digital clocks with LED displays. The diagram below illustrates the idea, with a clock showing 12:00
- Here, because of costs, each display must share the same 7 segment lines, that is, these lines are common to all displays.
- Obviously if all displays were to be enabled at the same time, then each display could not be differentiated from any other and they would all show the same information.
- However, by using time-division multiplexing, each display can in turn be enabled and the 7 segment lines changed to the data relevant for that display.
- The display can then, after a suitably brief period be turned off and the next one turned on, with new data on the 7 segment lines.
- If this multiplexing is done sufficiently quickly, it gives the illusion, to the eye at least, that all displays are lit at the same time, however a high speed camera would tell another story.
- This principle of rapidly swapping between displays to give the impression that all are active at the same time is exactly what a multi-tasking operating system is trying to achieve. So how is it managed?



The Role of the Operating System Kernel

- In order to manage concurrency, we need a means of managing the processes running on the system, dealing with the interrupts or ticks from the RTC and directing the CPU from executing one process to executing another.
- Such tasks are handled by an Multi-tasking Operating System (MTOS) Kernel of 'Executive'.

What is an Operating System Kernel ?

- A kernel is the heart of an operating system and is simply a collection of software subroutines that implement such as things as concurrency and solve problems associated with it, e.g. mutual exclusion, process communication, synchronisation, sharing memory etc.
- The kernel thus provides a set of services to the host programs running underneath it. That is, user programs, i.e. ones that you or I write can invoke operations within the kernel to carry out certain tasks such as opening files, communicating across networks and, in our case, creating and managing multiple tasks.
- Each operating system will have their own unique Kernel with different ways of doing things, and with differing functionality and size, but all doing essentially the same things.
- In Microsoft Windows the kernel is generally referred to as the Win32 Kernel after it was re-written for 32 bit operation.
- So what does the operating system kernel do ?

Function of the Operating System Kernel

- Handles the creation, termination and management of processes (activities) within the total system. Furthermore, these processes or threads of concurrent execution, may be prioritised giving varying amounts of CPU time to each based on its needs and importance to the system.
- Directs the CPU, when instructed by a multi-tasking clock, to break off the execution of one process and begin or pick up execution of another, preserving the processes 'volatile environment' so that it can be restored.
- Manages the shared resources of the system, such as memory and provides a transparent interface for processes to access the system resources e.g. disks, A to D converters via the provision or device drivers.
- Facilitates the creation and management of communication links between processes that which to communicate with each other, e.g. pipelines and datapools.
- Manages and facilitates 'mutual exclusion' between processes, i.e. preventing two or more processes accessing resources that are non sharable.
- Provides a means for one process to stimulate others into action using signals and timers.
- Provides facilities to allow the synchronisation of processes, semaphores, events, conditions and rendezvous.