Commercial Real-Time Operating Systems
Real-Time Systems Design (CS 6414)

Sumanta Pyne

Assistant Professor
Computer Science and Engineering Department
National Institute of Technology Rourkela
pynes@nitrkl.ac.in

March 15, 2022

Overview

- Ensures every real-time task meets timeliness requirements
- Use appropriate task scheduling techniques
- Provide flexibilty to programmers to select a scheduling policy
- Sharing critical resources and handling task dependencies
- Can general purpose OS (Unix or Windows) extended to RTOS?
- Fundamental problems associated with traditional OS
- Commercially available RTOS and their limitations
- Examine POSIX standard for RTOS and its implications
- Time service supports provided by RTOS
- Accurate and high precision clocks are important for RTOS
- Survey important features of commercially used RTOS
- Identify parameters on which RTOS can be benchmarked

Time services

- Clocks and time services - basic facility to programmers
- Provided by OS based on software (system) clock
- System clock by OS kernel receiving interrupt from hardware clock
- Clock resolution - time granularity by system clock
- Resolution - duration of time between two clock ticks
- Fine resolution for hard real-time systems
- Fine resolution system clock difficult in RTOS
- Presently resolution of hardware clocks finer than 1 nsec ($>$ 3GHz)
- Clock resolution modern RTOS order of several msec

Why fine resolution of system clock difficult in RTOS?

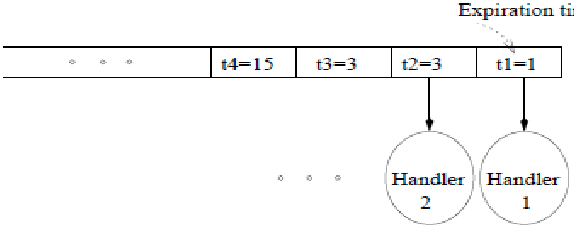
- Hardware clocks periodically generate time service interrupts
- After an interrupt kernel updates software clock
- A thread gets system clock time by system call (like clock-gettime())
- Finer resolution, more interrupts, higher kernel respond time
- This overhead is a limitation on fine resolution of system clock
- Response time of clock-gettime() is not deterministic due to jitter
- Since interrupts have higher priority than system calls

Why fine resolution of system clock difficult in RTOS?

- Processing of system call stalled during interrupt
- Worsens further
- System call preemption time vary as OS disable interrupts
- Jitter introduces error in accuracy of time value
- Calling thread gets inaccurate time value from kernel
- System call jitter in commercial OS can be several msec
- Jitter introduces error in time read by program
- Software clock resolution finer than this error - not meaningful

Clock interrupt Processing

- Each clock interrupt
    - Incrementing software clock
    - Three handler routine activities
- Process timer events



    - RTOS timer queue - per-process or single system-wide
    - All timers in queue arranged in order of expiration times
    - A handler routine for each timer - invoked when timer expires
    - At each interrupt kernel checks timer data structures for timer event
    - If event occurs, then it queues handler routine in ready queue

Clock interrupt Processing

- Update ready list
  - Since last clock event, some tasks arrive or become ready after waiting
  - Tasks in wait queue waits for some events like page fetch or semaphore
  - Tasks in wait queue are checked if any task has become ready
  - Tasks found ready are queued in ready queue
  - If a task found ready has higher priority than currently running task
    - Currently running task is preempted
    - Scheduler is invoked
- Update execution budget
  - At each clock interrupt, scheduler decrements task time slice (budget)
  - If remaining budget becomes zero and task incomplete
    - Task preempted
    - Scheduler invoked to select another task

Providing High Clock Resolution

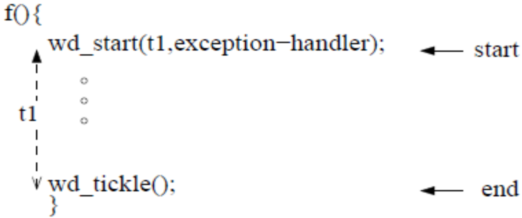
- Two main difficulties in providing a resolution timer
  - clock interrupt processing overhead excessive with finer resolution
  - jitter for lookup system call in order of several msec
- Not useful to provide clock resolution finer than several msec
- Some real-time applications deal time constraints of order of few nsec
- Is it possible to support measurement with nsec resolution?
- Fine resolution - mapping hardware clock to application address space
- An application can read hardware clock from memory, no system call
- On Pentium processor user thread reads Pentium time stamp counter
- Counter starts at 0 when system powered on
- Increments after each hardware clock interrupt
- Making hardware clock readable reduces portibilty of application
- An application running on Pentium ported to different process
- New processor may not have high resolution counter
- Memory address map and resolution would differ

Timers

- Periodic Timers
  - Used for sampling events at regular intervals or periodic activities
  - Once a periodic timer is set, it expires periodically
  - Implemented using timer queues
    - Each time periodic timer expires, hsndler routine is invoked
    - Timer data structure inserted back into timer queue
  - For example
    - A periodic timer may be set to 100 msec
    - Its handler set to poll temperature sensor after every 100 msec interval

Timers

- Aperiodic (or One Shot) Timers
  - Set expire only once
  - Watchdog timers are popular example of one shot timers
  - Watchdog timers in real-time programs to detect if task misses deadline
  - Initiate exception handling process upon a deadline miss
  - For example



    - If f() does not complete after t_1 time units have elapsed
    - Then the watchdog timer expires, indicating deadline missed
    - Exception handling procedure is initiated
    - Incase task completes before watchdog timer expires (within deadline)
    - Watchdog timer is reset using wd_tickle() call

Features of an RTOS

- Clock and Timer Support
  - With adequate resolution required in real-time programming
  - Hard real-time application requires time services in order of few msec
  - Finer resolutions required for certain applications
  - Clock and timer vitsl part of every RTOS
  - Traditional OS do not provide time services with high resolution
- Real-Time Priority Levels
  - RTOS must support static priority levels or real-time priority levels
  - Once programmer assigns priority value to a task, OS doesn't change it
  - Traditional OS - priority levels change dynamically to max. throughput
- Faster Task Preemtion
  - Higher priority critical task arrives, preempnt low priority task for CPU
  - Duration higher priority task waits for execution - task preemption time
  - Contemporary RTOS have task preemption time in order of few μsec
  - Traditional OS have task preemption time in order of a second
  - Significantly large latency caused by a non-preemptive kernel
  - RTOS needs preemptive kernel, task preemption time order of few μms

Features of RTOS

- Predictable and Fast interrupt Latency
  - Interrupt latency - delay between interrupt occurrence and running ISR
  - Upper bound of interrupt latency in RTOS less than a few μsec
  - Low latency by bulk ISR activities in Deffered Procedure Call (DPC)
  - DPC performs most of ISR, executes after ISR completes at low priority
  - Support for nested interrupts desired
  - RTOS be preemptive during execution of kernel routines and ISR
  - Important for hard real-time applications with sub-μsec requirements
- Support for Resource Sharing Among Real-Time Tasks
  - Traditional critical resource sharing - unbouned leading deadline misses
  - RTOS provide basic priority inheritance mechanism
  - Support of Priority Ceiling Protocol (PCP) desirable
  - PCP for large and medium sized applications

Features of RTOS

- Requirements on Memory Management
    - General purpose OS - virtual memory and memory protection
    - Embedded RTOS never support these features
    - Meant for large and complex applications
    - RTOS for large and medium-sized applications need virtual memory
    - Virtual memory reduces average memory access time
    - Degrades worst-case memory access time
    - Penalty - storing address in translation table and address translation
    - Fetching pages from secondary memory - page fault latency significant
    - Control paging (memory locking) for RTOS virtual memory support
    - Memory locking a page from being swapped from memory to hard disk
    - Large jitter due to absence of memory locking
    - Lack of memory protection - single address space for all tasks
    - Single address space - simple, save memory bits, lightweight sys calls
    - Small embedded applications - few kB/process overhead unacceptable
    - No memory protection - high cost of developing and testing program
    - Change in a module req retesting entire sys - high maintainance cost

Requirements on Memory Management

- Embedded RTOS do not support virtual memory
- Creates physically contiguous blocks of memory for an application
- Memory fragmentation and protection problems
- In many embedded sys kernel and user program run in same space
- System and function calls in an application indistinguishable
- Makes debugging applications difficult
- A runaway pointer can corrupt OS code, making system 'freeze'

Features of RTOS

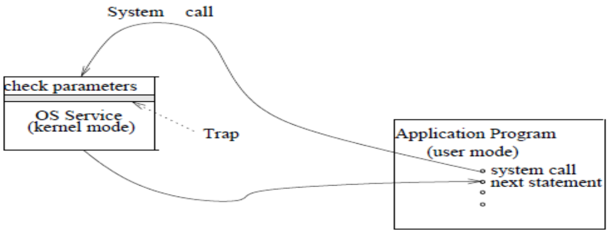
- Support for Asynchronous I/O
  - Non-blocking I/O
  - read() and write() system calls - synchronous I/O
  - Synchronous I/O - Process needs to wait until hardware completes I/O
  - aio_read() and aio_write() system calls - asynchronous I/O
  - System call will return immediately I/O request passed down
  - Execution of process not blocked, no need to wait for system call result
  - Continue execution, receive I/O results later when available
- Additional Requirements for Embedded RTOS
  - Cost, size and power consumption
  - Diskless systems
  - Flash memory or ROM
  - ROM or RAM

Unix as an RTOS

- Unix - popular general purpose OS originally developed for mainframe
- Unix and its variants used in desktop and handheld computers
- Shorcomings of traditional Unix used for real-time applications
    - non-preemptive Unix kernel
    - dynamically changing priorities of tasks
- All dicussions on Unix are based on original Unix System V

Non-preemptive Kernel

- Unix kernel cannot be preempted
- All interrupts disabled when any OS system routine runs
- Application programs invoke OS system service through system calls
- System calls - create process, interprocess comm, I/O operations
- After a system call invoked, arguments by application are checked
- A software interrupt called trap is executed
- Trap changes processor state from user to kernel (supervisor) mode



Non-preemptive Kernel

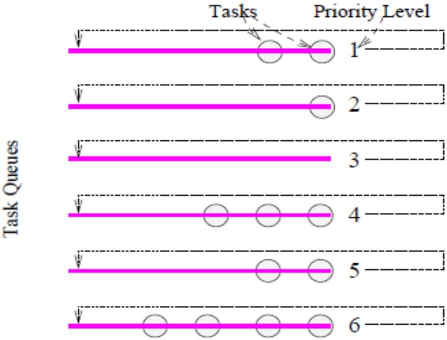
- Creating process, file operations in kernel mode only
- Application programs prevented, need to request OS by system calls
- In Unix running process in kernel mode cannot be preempted by others
- Unix system does preempt processes running in user mode
- System call of low priority process, high priority process waits
- For real-time applications this causes priority inversion
- Longest system calls may take several hundreds of msec to complete
- Worst case preemption times - several hundreds of msec
- Higher priority tasks with few msec deadlines
- System call make higher priority tasks to miss deadlines

Why non-preemptive Unix kernel?

- When Unix kernel runs, all interrupts disabled
- Interrupts enabled only after OS routine completes
- Very efficient way of preserving integrity of kernel data structures
- Save overheads - setting-releasing locks
- Low average task preemption time
- Non-preemptive kernel worst case response time 1 sec - acceptable
- Unix designers didn't foresee usage of Unix in real-time applications
- Correct kernel data structures - use locks instead disabling interrupts
- Increases average task preemption time

Dynamic Priority Levels

- Traditional Unix systems - no static priority value for real-time tasks
- Programmer sets priority value, OS alters it during task execution
- Difficult to schedule real-time tasks using RMA or EDF
- RMA and EDF consider static task priority
- Why Unix needs to dynamically change priority values of tasks?
- Unix uses round-robin scheduling with multilevel feedback
- Scheduler arranges tasks in multilevel queues



Dynamic Priority Levels

- At every preemption point, scheduler scans from top
- Selects task at head of first non-empty queue
- Each task allowed to run for a fixed time quantum (or slice) at time
- Unix normally uses one second time slice by default - reconfigurable
- If blocked or not completed within time slice, a process preempted
- Scheduler selects next task for dispatching
- Kernel recomputes preempted task's priority
- Inserts back it to one of priority queues
- Priority of task Ti at end of jth time slice
  pri(Ti,j) = Base(Ti) + CPU(Ti,j)+nice(Ti)
  - Base(Ti) - base priority of Ti
  - CPU(Ti,j) - weighted history of CPU utilization of Ti
    - Maximum weightage for activity of task in immediate concluded interval
    - If Tj uses CPU for full duration of j CPU(Ti,j) has a high value
    - High value of CPU(Ti,j) - lowering priority of Ti
  - nice(Ti) - non-negative nice value associated with Ti

Dynamic Priority Levels

- CPU(Ti,j) $=$ U(Ti,j-1)/2 $+$ CPU(Ti,j-1)/2 $+ \cdots$
  - U(Ti,j) - utilization of Ti for its jth time slice
- Recursively, CPU(Ti,j) $=$ U(Ti,j-1)/2 $+$ CPU(Ti,j-2)/4 $+ \cdots$
- pri(Ti,j) $=$ Base(Ti) $+$ U(Ti,j-1)/2 $+$ U(Ti,j-2)/4 $+ \cdots +$ nice(Ti)
- I/O transfer rate responsible for slow response time
- Processors much faster than I/O devices
- Delay of I/O transfer - a bottle neck
- A solution - keep I/O channels as busy as possible
- Achieved by assigning higher priority to I/O bound tasks
- In Unix set of priority bands assigned to different tasks
- Tasks in decreasing order of priority
  - Swapper
  - block I/O - during page fault, uses DMA, efficient use of I/O channel
  - file manupulation
  - character I/O - mouse and keyboard transfers
  - device control
  - user processes

Dynamic Priority Values

- Priority bands provide most effective use of I/O channels
- Any task performing I/O must not wait too long for CPU
- So as soon as a task blocks for I/O, its priority increased
- If a task makes full use of las assigned time slice
- Task is computation-bound, its priority is reduced
- In Unix interactive tasks have higher priority levels
- Processed at earliest - gives good response time
- Accepted for scheduling soft real-time tasks in general purpose OS
- Like Microsoft's Windows

Dynamic Priority Values

- Unix is very appropriate for maximizing average task throughput
- Provide good average response time to interactive soft real-time tasks
- Almost every modern OS does dynamic recomputation of task prioritie
- Maximize overall system throughput, good average response time
- Dynamic priority inappropriate for hard real-time tasks
- Prevents tasks being constanlty scheduled at higher priority levels
- Prevents real-time task scheduling using EDF and RMA

Insufficient Device Driver Support

- Unix device drivers run in kernel mode
- To support a new device driver module is linked to kernel modules
- Providing such support in already deployed application is cumbersome

Lack of Real-Time File Services

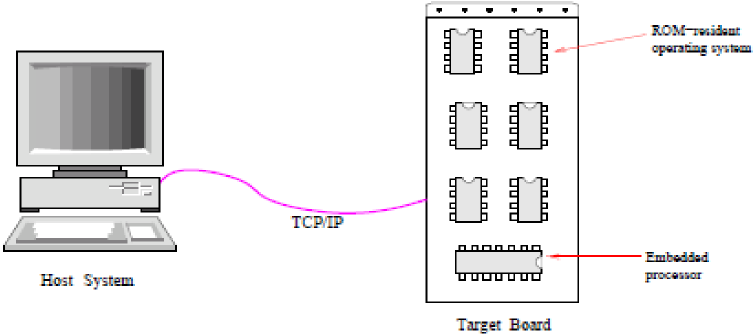
- File blocks allocated on request by an application
- While writing to a file a task may encounter disk out of space error
- No guarantee given for disk space available when task writes in file
- Traditional approaches result in slow writes
- Since required space has to be allocated before writing a block
- Blocks of a file block may not be contiguously located on disk
- Result unpredictable times for read operations - jitter in data access
- Real-time file systems store files contiguously on disk
- Achieves significant improvement in performance
- File system preallocates space
- Times for read and write operations more predictable

Inadequate Timer Services Support

- Insufficient timer support for hard real-time applications
- Clock resolution 10 msec - too coarse for hard real-time applications

Unix-based RTOS

- Extensions to the traditional Unix Kernel
  - Adding some real-time capabilities over basic kernel
  - Include real-time timer and task scheduler built over Unix scheduler
  - Extensions do not address fundamental problems
- Host-Target Approach



- Host-target OS popular in embedded applications
- Commercial examples - PSOS, VxWorks, VRTX

Host-Target Approach

- Real-time application developed on host machine with traditional OS
- Application downloaded on target board embedding real-time system
- ROM-resident small real-time kernel in target board
- OS on target board is as small and simple as possible
- No virtual memory, compilers, program editors required
- Processor on target board run RTOS
- Host Unix/Windows with editors, cross-compilers, library, debuggers
- Require virtual memory support
- Host connected to target using serial port or TCP/IP
- RTOS developed at host, cross-compiled for target processor code
- Executable module downloaded to target board
- Tasks run on target board, ctrld at host using symb. cross-debugger
- On success, RTOS fused on ROM or flash memory, ready to work

Preemption Point Approach

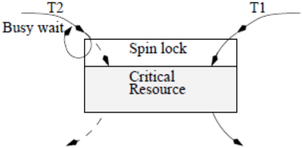
- Unix V mask interrupts during system call - unacceptable for RTS
- Improve real-time performance of non-preemptive kernel
- Introduce preemption points in execution of system routines
- Preemption points - instants at kernel data structures consistent
- Kernel safely preempted to run any waiting higher priority task
- Without corrupting any kernel data structures
- When execution of systen call reaches a preemption point
- Kernel checks any higher priority tasks have become ready
- If atleast one, it preempts processing of kernel routine
- Dispatches waiting highest priority task immediately
- Worst-case preemption time
  - longest time between two consecutive preemption points
  - improves several folds compared to traditional OS
- Preemption point-based OS suitable for hard real-time applications
- Not for hard real-time applications with preemption latency $\leq \mu$sec
- Requires minor changes to kernel code
- Past OSs like HP-UX and Windows CE taken preemption point

Self-host systems

- Real-time application developed on same OS on which it will run
- OS modules not needed excluded to minimize embedded system OS
- Minimize OS size for lesser cost, size and power consumption
- Application runs on host, fused to target board ROM or flash memory
- Current self-host OS based on micro-kernel architecture
- In a micro-kernel architecture
  - kernel mode routines - interrupt handler and process management
  - user mode modules - memory, file and device management
- Add-on modules easily excluded when not required
- Easy to configure OS, resulting in a small-sized system
- Monolithic - drivers, file system, kernel process same address space
- Single programming error cause fatal kernel fault
- Micro-kernel OS - components memory-protected
- Rare system crashes, very reliable

Problems overcome in Self-host systems

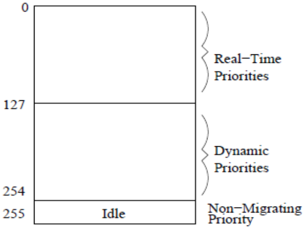
- Non-Preemptive Kernel
  - Kernel-level and spin locks for fully preemptive Unix systems
  - When a task waits for kernel-level lock held by another task
  - It is blocked and undergoes context switch, ready after lock available
  - Inefficient if duration critical resources needed $<$ context switching time



  - A critical resource protected by spin lock
  - Needed by T1 and T2 for very short times wrt context switching time
  - Suppose T1 acquires spin lock, meanwhile T2 requests resource
  - Since T1 has locked resource, T2 can't access it and busy waits
  - T2 is not blocked, no context switch occur
  - T2 gets resource as soon as T1 relinquishes resource
  - Spin lock in multiprocessor system using cache coherence protocol
  - Uniprocessor - mutual exclusion critical resource needs very short time

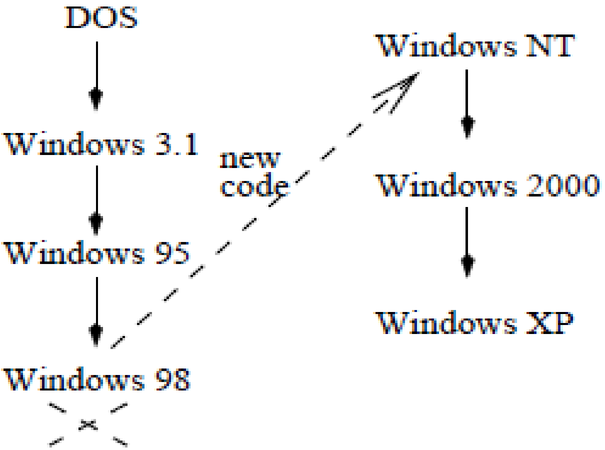
Problems overcome in Self-host systems

- Real-time Priorities



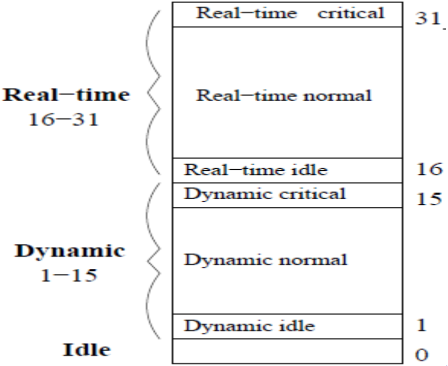
- Unix-based RTOS has three addition priority levels
- Idle (Non-Migrating) Priorities
  - Lowest priority level
  - Tasks run at this level when there are no other tasks to run
  - Idle tasks are static and are not recomputed periodically
- Dynamic Priorities
  - Recomputed periodically
  - To improve average response time of soft-real time (interactive) tasks
  - Ensures higher priority for I/O bound tasks, lower for CPU-bound tasks
- Real-Time Priorities
  - Static priorities, not recomputed during runtime
  - Hard real-time tasks operate at these levels

- Genealogy MS Windows



Windows NT for real-time applications

- Priority-driven preemptive scheduler
- Real-time threads precedence over all including kernel threads
- Shortcomings
  - Interrupt Processing
  - Support for Resource Sharing Protocols



Windows NT vs Unix

Real-Time Feature	Windows NT	Unix V
DPCs	Yes	No
Real-time priorities	Yes	No
Locking virtual memory	Yes	Yes
Timer precision	1 milli Sec	10 milli Sec
Asynchronous I/O	Yes	No

POSIX - Portable Operating System Interface

- Important standard for OS and RTOS
- Open software movement and emergence of POSIX
- Open Software catagories
  - Open Source - portibilty at source code level
  - Open Object - portability of unlinked objects
  - Open Binary - portability at executable level
- POSIX provides portibilty at source code level
- Originally developed by AT&T Bell Labs in early 70s
- UCB - earliest recipients of Unix source code free of cost
- AT&T came up with Unix V
- UCB incorporated TCP/IP with Unix using DARPA grant
- UCB's Unix version - Berkeley Software Distribution (BSD)
- Rapid growth in commercial importantance of Unix
- Extensions of Unix
  - IBM AIX, HP HP-UX, Sun Solaris, Digital Ultrix, SCO SCO-Unix
- To solve portibilty problem ANSI/IEEE yielded POSIX

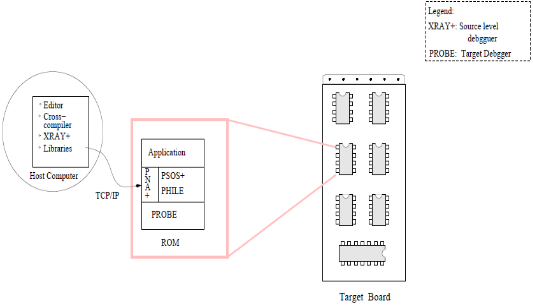
POSIX standard

- Defines only interfaces to OS and semantics of these services
- Does not specify how exactly services are implemented
- Source-code level portability - POSIX specifies
  - System calls needed by OS
  - Exact parameters of system calls
  - Semantics of system calls
- Leaves OS vendors freedom to implement system calls
- Does not specify
  - OS kernel single-threaded or multithreaded
  - Priority level at kernel services are executed
  - Programming language to be used
- POSIX standard parts
  - POSIX.1: system interfaces and system call parameters
  - POSIX.2: shells and utilities
  - POSIX.3: test methods for verifying conformance to POSIX
  - POSIX.4: real-time extensions

Real-time POSIX Standard

- POSIX.4 - real-time extensions of POSIX also called POSIX-RT
- Requirements for OS to be POSIX-RT compliant
  - Execution scheduling - Support for real-time (static) priorities
  - Performance requirements on system calls - Worst-case execution time
  - Priority levels - atleast 32
  - Periodic and one shot timers - CLOCK_REALTIME for RT-POSIX
  - Real-time files - Stored in contiguous blocks on disk
  - Memory locking - deterministic memory access
    - mlockall()/munlockall() - lock/unlock all pages of a process
    - mlock()/munlock() - lock/unlock a range of pages
    - mlockpage()/munlockpage() - lock/unlock only current page
  - Multithreading - Real-time threads schedulable with time constraints

- PSOS
  - A RTOS for embedded applications from Wind River Systems
  - Host-target type RTOS
  - Used in commercial embedded systems
  - Example- PSOS in base stations of cell phone systems
  - Host - desktop Unix/Windows
  - Target board - embedded processor, RAM, ROM



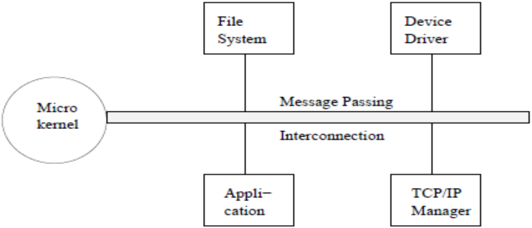
VRTX

- POSIX-RT compliant OS from Mentor Graphics
- Certified by US FAA for use in life-critical applications like avionics
- Available in two multitasking kernels: VRTXsa and VRTXmc
- VRTXsa
  - Large and medium-sized applications
  - Support - virtual memory, POSIX-compliant library, priority inheritance
- VRTXmc
  - Optimized for power consumption, ROM and RAM sizes
  - Kernel 4-8 KBytes of ROM and 1 kb RAM
  - No virtual memory support
  - Cell phones and handheld devices

VxWorks

- A Wind River Systems product
- Host-target type RTOS
- Windows or Unix machine as host
- Conforms POSIX-RT and uses an IDE called Tornado
- Tornado has editor, cross-compiler, cross-debugger, VxSim, WindView
- VxSim simulates a VxWorks target in absence of actual target board
- Deployed in Mars Pathfinder sent to Mars in 1997
  - After landing responded to ground commands, sent scientific data
  - Repeatedly reset itself
  - Real-time tasks miss deadlines due to unbouned priority inversion
  - As a result, the exception handler reset the system each time
  - Determined at ground using trace generation, logging, debugging tools
  - VxWorks supports priority inheritance
  - Debug tool found out priority inheritance disabled in configuration file
  - This problem was fixed by enabling it

QNX

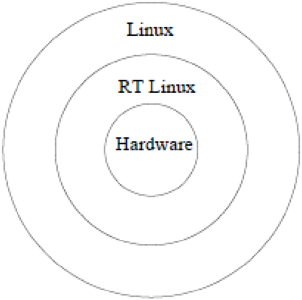


- A product from QNX Software Systems Ltd
- For mission-critical applications
- Medical-instrumentation, Internet routers, telemetrics
- Process control applications, air traffic control systems
- QNX Neutrino - POSIX compliant APIs, uses microkernel architecture
- Configured to a very small size in high volume devices
- Extremely fast on very small memory footprint portable devices
- Web app., MP3 players, set-top boxes, medical/insdustrial app.
- Runs on Intel x86, MIPs, PowerPC, ARM processor family

μC/OS-II

- From Micrium Corporation, written in ANSI C and assembly code
- Ported to 8-64 bit μPs/μCs and DSPs
- Let programmers to opt few or select entire range of services
- Fully preemptive kernel - ensures higher priority task execution
- Allows upto 64 tasks, 64 priorities levels, use PID for tasks
- Fixed sized memory blocks - deterministic constant time alloc/dealloc
- FAA certified for use in commercial aircraft - robust and safe

RT Linux



- A self-host OS that runs along with a Linux system
- Real-time kernel sits between hardware and Linux system
- To standard Linux RT Linux appears to be the actual hardware
- RT Linux kernel intercepts all interrupts generated by hardware
- Hardware interrupts for non real-time activities
  - are held
  - passed to Linux kernel as s/w interrupts when RT Linux kernel is idle
  - standard Linux kernel runs

RT Linux

- An interrupt to cause a real-time task to run
  - Real-time kernel preempts Linux, if it is running
  - Lets real-time task run
- Linux runs a a low priority background task of RT Linux
- Real-time applications are written as loadable kernel modules
- Real-time applications run in kernel space
- Dual kernel approach - independent real-time kernel and Linux kernel
- Real-time kernel implemented outside Linux kernel
- Tasks requiring deterministic scheduling run as real-time tasks
- These tasks preempt Linux whenever they need to execute
- Yield CPU to Linux when no real-time tasks are ready to run

Shortcomings of dual-kernel approach

- Duplicated Coding Efforts
  - Real-time tasks invoking Linux service are subject to preemption
  - Linux processes behave non-deterministically
  - Separate drivers and system services for real-time kernel
  - Real-time application development - duplication of coding effort
  - Real-time tasks need network communications or file accesses
  - Require appropriate drivers written in real-time kernel
- Fragile Execution Environment
  - Tasks running in real-time kernel not MMU-protected
  - Coding error (corrupt C pointer) - fatal fault for safety-critical systems
- Limited Portability
  - Different implementations of dual kernels use different APIs
  - RT programs written on one vendor's RT Linux may not run on another
- Programming Difficulty
  - RT Linux supports limited subset of POSIX APIs - more effort and time

Lynx

- A self-host RTOS available from www.lynuxworks.com
- Lynx 3.0 is microkernel-based RTOS, earlier versions monolithic
- Fully Linux compatible
- A Linux program's binary image directly run on Lynx
- Other Linux compatible OSs like QNX - recompile applications to run
- Microkernel is 28 kB
- Services - task scheduling, interrupt dispatch, synchronization
- Other services provided as Kernel Plug-Ins (KPIs)
- KPIs to microkernel for I/O, file systems, sockets
- Multipurpose Unix with full configuration for hard & soft RT tasks
- Supports memory protection

Windows CE

- Stripped down version of Windows OS, minimum footprint of 400 kb
- 256 priority levels
- To optimize performance all threads run in kernel mode
- Timer accuracy 1 msec for sleep and wait related APIs
- Kernel functionalities broken into small non-preemptive sections
- During system call preemption turned off for short durations
- Interrupt servicing preemptable
- Supports nested interrupts and virtual memory management
- Priority inheritance to avoid priority inversion problem in Windows NT
- Kernel thread handle page fault run at higher priority than NORMAL
- NORMAL priority thread page fault - priority of kernel thread raised
- A thread not blocked by a lower priority thread during page fault

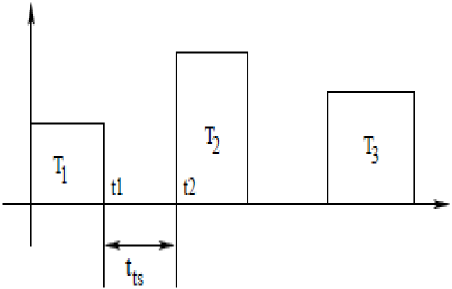
Benchmarking Real-time Systems

- Design and platform evaluation - benchmark computer systems
- Two traditional performance metrics - MIPS and FLOPS
- Based on what type of program is not specified by vendor
- MIPS and FLOPS ratings are misleading
- Synthetic benchmarks - large number of practical programs
- Determine statistical distribution of various instruction in an average
- Ex - ALU instructions 20%, I/O 10%, register transfer 10%, etc
- No meaningful results
- Synthesize using instruction with statistical distribution
- Performance results of synthetic very closely related to practical
- Synthetic benchmarks - Whetstone, Linpack and Dhrystone
- SPEC since 1980s - publicizes application specific benchmark suites
- SPECWEB - benchmark for web applications
- SPEC benchmarks are available at www.spec.org

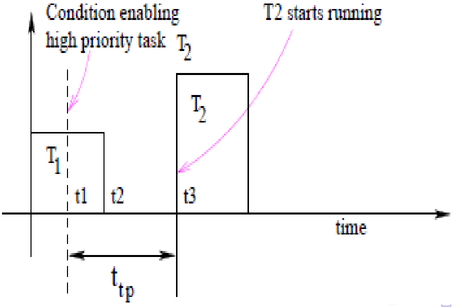
Rhealstone Metric

- A popular benchmarking for real-time systems
- Considers six important parameters of real-time systems

1. Task Switching Time (t_ts)
    - Time for one context switch among equal priority tasks
    - T1, T2, T3 are equal priority tasks with round-robin scheduling
    - t_ts = t2 - t1, is time after which T2 starts execution after T1
    - where T1 may complete its time slice, blocked or complete
    - Determined by the efficiency of kernel data structure

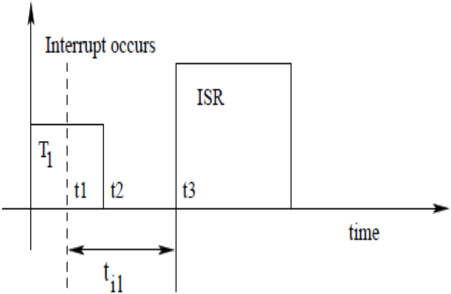


2. Task Preemption Time (t_pt)
   - Time taken to start execution of a higher priority task
   - Compared to currently running task
   - Three components
     - task switching time t_ts
     - time to recognize the event enabling higher priority
     - time to dispatch it
   - Larger than t_ts



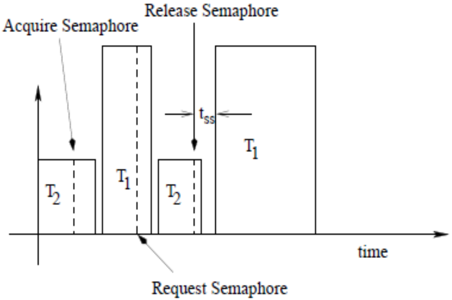
Rhealstone Metric

3. Interrupt Latency Time (t_il)
   - Time taken to start execution of ISR after an interrupt occurs
   - Components
     - hardware delay in CPU recognizing the interrupt
     - time to complete the current instruction
     - time to save the context of the currently running task
     - time to start ISR



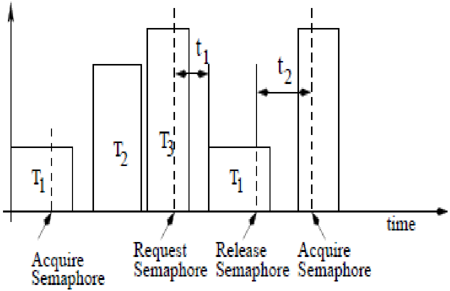
Rhealstone Metric

4. Semaphore Shuffling Time (t_ss)
   - Time lower priority task release semaphore & higher priority task starts
   - T2 holds semaphore, T1 requests for semaphore and blocks
   - Interval between T1 returns semaphore and T2 starts running



Rhealstone Metric

5. Unbounded Priority Inversion Time (t_up)
   - t1 - time taken by OS to recognize priority inversion
   - t2 - time to run T1 holding CR and start T2 after T1 completes
   - t_up = t1 + t2



Rhealstone Metric

6. Datagram Throughput Time (t_dt)
   - Number of kB of data transferred between two tasks
   - Without using shared memory or pointers
   - Measure efficiency of data structures handling message passing

- Computation of Rhealstone Metric
- a_1*t_ts + a_2*t_tp + a_3*t_il + a_4*t_ss + a_5*t_up+a_6*t_dt
- a_1, . . . , a_6 are empirically determined constants
- Figure of merit characterizes system's kernel-hardware combination
- A comprehensive benchmark
- Drawbacks
  - ability of tasks to meet deadlines not considered at all
  - no proper justification behind choosing six measurements

Interrupt Processing Overhead

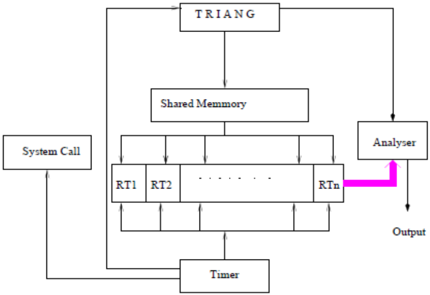
- t_0 - time to complete a task with no interrupts
- t_{20000} - time to complete same task with 20,000 interrupts
- Overhead due to 20,000 interrupts, $I_{20000} = (t_{20000} - t_0)/t_0$
- Consider $t_{20000} = 1.6$ nsec, t_0 is 1 sec
- $I_{20000} = 0.6$ (or 60%) of execution time
- Overhead considered for processing immediate parts of interrupt
- Deferred parts of interrupt not considered

Tridimensional Measure (TM)

- Three factors affect performance of real-time systems
- MIPS1 - Millions of Instructions per Second
- MIPS2 - Millions of Interrupts Processed per Second
- NIOPS - Number of I/O Operations per Second
- TM = $(\text{MIPS1*MIPS2*NIOPS})^{\frac{1}{3}}$

Determining Kernel Preemptability

- Extent to which OS kernel preemptable to higher priority RT tasks
- An experimental set up can indicate kernel preemptability



- TRIANG module generates random triplets - three vertices of triangle
- Real-time tasks RT1, RT2, . . . , RTn read triangle values
- n real-time tasks carry out certain processing
- Output triangle vertices to analyzer module through message passing

Determining Kernel Preemptability

- TRIANG operates at higher priority than real-time tasks
- Timer first invokes TRIANG, then invokes system call (create process)
- Subsequently triggers one of RT1, RT2, . . . , RTn each time
- Analyzer compares triangle vertices received from TRIANG and RT
- If system kernel non-preemptable, then
  - even before RT can read triangle value,
  - next triangle values overwrite last generated values
- Timer frequencies varied and analyzer results are observed
- Analyzer identifies timer frequency at which vertices get missed
- Indicates worst-case kernel preemption time

Thank you