CSCE 990: *Real-Time Systems*

Real-Time Operating Systems

Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/RealTimeSystems

Real-Time Operating Systems

(Sections 12.1 - 12.2 and 12.6 - 12.7 of Liu)

◆ **Outline:**

» Threads and Tasks (Section 12.1.1 of Liu).

» The Kernel (Section 12.1.2 of Liu)

- Most Real-Time operating systems contain a micro-kernel whose functionality can be extended with modules.

» Time services and scheduling mechanisms (Section 12.2 of Liu).

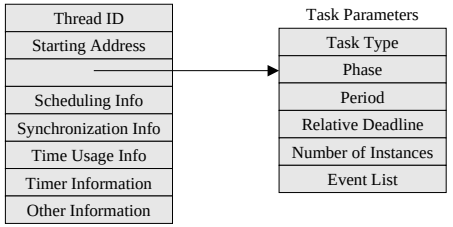
» A brief survey of commercial real-time and non-real-time operating systems (Sections 12.6-12.7 of Liu).

Threads and Tasks

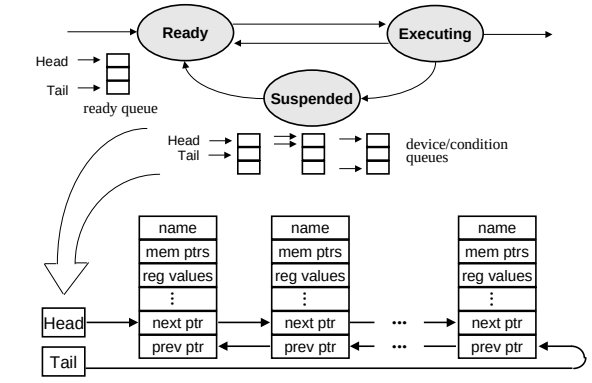
- **Thread:** A basic unit of work handled by the scheduler.
- **Task:** Threads implement the jobs of a task. Usually the same thread is re-used for each job of a task.
- **Thread Context:** The values of registers and other volatile data that define the state and environment of the thread.

Threads and Tasks (continued)

- **TCB:** The thread control block (TCB) is the data structure created when the kernel creates a thread.
  - » The TCB stores the context of the thread when it is not executing.



Implementing and Managing State Transitions



Liu lists five major states: Sleeping, Ready, Executing, Suspended (or Blocked), and Terminated.

Inserting a task on a queue, like the ready queue, really means you insert a pointer to the TCB for that queue.

Only the middle three states are shown here. Liu claims that a task that finishes executing, but will execute again (at the start of its next period) is placed in the sleeping state. Since most OS do not support periodic tasks, this is often implemented as a suspended stated where the task is blocked waiting for the event to occur that releases the next job.

Periodic Tasks and Threads

- A periodic tasks is implemented as a thread that executes periodically.
- A **periodic thread** is reinitialized by the kernel and put to sleep (i.e., transitioned to the sleeping state) when the thread completes. It is then released again at the beginning of the next period (i.e., transitioned to the ready state).
- The task parameters (e.g., phase and period) are stored in the TCB.
- Most commercial operating systems do not support periodic threads
  - » They are implemented at user level as a thread that sleeps until the start of the next period after it finishes executing.

Aperiodic and Sporadic Tasks

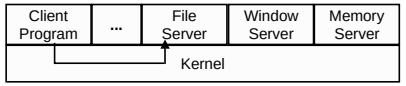
- An aperiodic or sporadic task is implemented as a thread that executes in response to specified types of events, which are usually represented as an external interrupt.
- An **aperiodic or sporadic thread** is reinitialized by the kernel and put to sleep (i.e., transitioned to the sleeping state) when the tread completes.
- Most commercial operating systems do not support aperiodic or sporadic threads
  - » They are implemented at user level as a thread that blocks for a specified event.

Operating System Structure

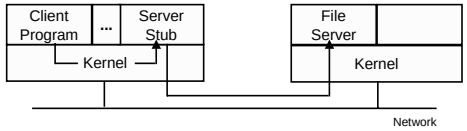
- Most real-time operating systems are used in embedded systems with limited space.
  - » the OS should be modular and extensible
  - » often built around a microkernel providing
    - scheduling
    - synchronization
    - interrupt handling

Operating System Structure

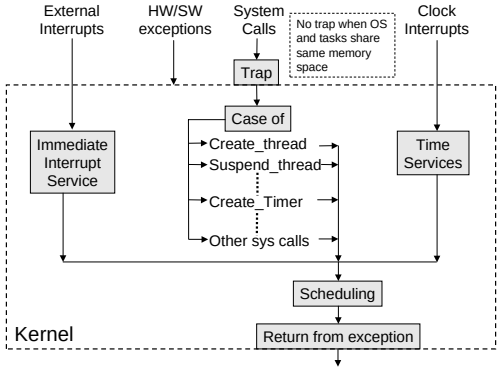
- Microkernel (or Client/Server)
  - » Centralized



  - » Distributed



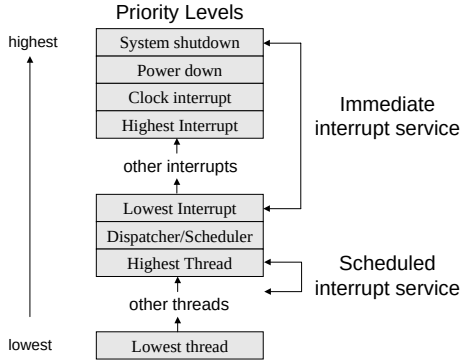
The Microkernel Structure



External Interrupts

- Hardware interrupts represent external events that trigger sporadic task or I/O activities.
- Depending on the source of the interrupt, the amount of time required to process the interrupt varies.
- Thus, interrupt processing is usually divided into two phases:
  - » Immediate interrupt service: executed by the kernel with higher priority than threads.
  - » Scheduled interrupt service: may be implemented as a preemptive aperiodic or sporadic thread.
- Why do this?
- We call it split interrupt handling.

Interrupt and Thread Priorities



Processing Interrupts

- When an interrupt occurs:
  - » The processor pushes the PC and SR on the interrupt stack and branches to the Kernel's interrupt handling code
  - » The Kernel then
    - disables interrupts,
    - saves the processor state on the interrupt stack,
    - enables higher priority interrupts (if possible),
    - calls the immediate interrupt service routine (ISR) of the interrupting device, and
    - enables interrupts (actually, it should restore the interrupt mask that was in effect when the interrupt occurred).
- Interrupt Latency: the time between when the interrupt occurs and when the immediate ISR starts to execute.

Processing Interrupts (continued)

- The immediate interrupt service routine (ISR) performs the minimum amount of work necessary to re-enable the external device.
- Most of the real interrupt processing is done by a scheduled interrupt handling routine (IHR), which is released/triggered by the immediate ISR.
- The scheduled IHR is executed at an appropriate application-level priority (usually preemptive).
- When the OS provides memory protection (e.g., LynxOS and Solaris), the scheduled IHR is executed by a kernel thread (rather than an application thread).

System Calls

- A system call is a call to the operating system's application program interface (API).
- Most real-time operating systems provide their own API and the POSIX (Portable Operating Systems Interface) standard 1003.1 API.
  - » POSIX 1003.1 defines the basic functions of a Unix OS.
  - » Many support at least a portion of the real-time (POSIX 1003.1b) and thread (POSIX 1003.1c) extensions.
- The system call may be synchronous or asynchronous
  - » synchronous: the calling thread is blocked until the kernel completes the called function
  - » asynchronous: the thread continues after making the call and the call is performed by a separate thread.

Time Services

- Every system has at least one hardware device that contains a counter.
  - » The counter is initialized with a specified value and counts down to 0. When it reaches 0, the counter generates an interrupt.
- A counter can (usually) be programmed to act as either:
  - » periodic timer: generates an interrupt every *p* time units.
  - » one-shot timer: must be re-initialized after each interrupt.
- The counter can serve as a clock and/or a timer.
  - » A clock is a counter and ISH that together keep (system) time.
  - » a timer is a counter and ISH that triggers an event *e* at time *t* (relative or absolute).

Clocks and Time

- The resolution of a clock is the granularity of time provided by the counter.
  - » The hardware resolution may be on the order of nanoseconds.
  - » However, the resolution seen by the application is usually in the order of hundreds of microseconds or milliseconds.
- A clock is implemented with a periodic timer.
- When a system has only one counter, it serves as both a clock and a timer.
  - » In this case, the kernel tracks timer expirations using the clock.
- At each clock period, a time-service interrupt is processed:
  - » The kernel updates the software clock, and
  - » checks the clock's timer queue for timer expirations, moving any to a pending queue.

Clocks and Time

- One out of every *x* time-service interrupts is considered a clock interrupt, which results in the scheduler executing.
- The clock interrupt is (almost) always processed using split interrupt handling techniques.
  - » The immediate ISR processes the time services interrupt
  - » The scheduled IHR, the Dispatcher/Scheduler thread, is released/triggered by the immediate ISR and performs scheduling services:
    - Processes all timer events in the pending queue by performing the specified event action.
    - Updates the execution budget for RR time-slice scheduling.
    - Updates the ready queue.
    - Dispatches the highest priority thread (assuming preemptive scheduling).

Timers

- Most operating systems (including all Real-Time POSIX compliant systems, NT, and Solaris) allow a thread to have it own timer.
- Threads can create, set, cancel, and destroy timers.
- When a timer is created by a thread, the kernel creates a timer data structure, which includes:
  - » the clock associated with the timer
  - » the thread that created the timer
  - » the expiration of the timer in absolute or relative time, once set
  - » the timer type: one-shot or periodic
  - » an event handler: a routine to execute when the timer expires.

Timers

- Synchronous Timers suspend the calling thread until the timer has expired.
  - » For example, timer_sleep(relative_time) or timer_sleep_until(absolute_time)
- Asynchronous Timers count down while the calling thread executes.
  - » When the timer expires, the associated handler is called.
  - » Threads may have multiple asynchronous timers active.
  - » Asynchronous timers can be used as a timing monitor to log missed deadlines (see Figure 12-4 in Liu).

Timer Resolution

- Nominal timer resolution is the granularity of absolute time or time interval specified as an argument to the timer function.
  - » If the nominal timer resolution is *x* microseconds, then the OS will not mistake two timer events set *x* microseconds apart as a single timer event.
- Actual timer resolution is the granularity of time measured by application threads using timers.
  - » If timers are implemented with a periodic timer interrupt, a counter in periodic timer mode, the actual timer resolution can be no greater than period of the counter.
  - » Worse, if the kernel only schedules at clock interrupts, the actual resolution may be as high as the period of clock interrupts (rather than the higher frequency of timer-service interrupts).

Timer Resolution

- Timer resolution can be improved using One-Shot Timer Interrupts rather than periodic timer interrupts.
  - » The counter raises an interrupt at each timer expiration.
  - » The kernel must re-set the counter for the next timer interrupt.
  - » The kernel either invokes or schedules the handler as soon as the interrupt occurs.
  - » Resolution is limited by the time the kernel takes to set and service the counter.

Scheduling Mechanisms

- All operating systems support Fixed Priority Scheduling.
  - » Many real-time operating systems provide 256 priority levels.
    - We have already seen that 256 levels approximates an ideal system.
  - » General purpose operating systems usually provide 10-20 levels.
    - NT has only 16 real-time priority levels.
  - » The priority is usually set (once) when the thread is created and stored in the TCB.
  - » The kernel (usually) maintains a ready queue for each priority level.
  - » Dispatching the highest priority ready thread requires finding the highest priority nonempty queue.
    - How do we do this?
    - What is the complexity of this?

Dispatching Threads

- An efficient way of doing this:
  - » Data structures:
    - Assume a K-bit queue-status word and Ω priority queues
    - Let each bit of the bit of the word represent the status of Ω/K priority queues
    - If a bit is set, one of the Ω/K associated queues has a ready thread
  - » Finding a ready thread:
    - If the queue-status word > 0 then a thread is ready
      - perform a binary search on the word (comparing its value to powers of 2) to find the highest-order bit set
      - find the highest priority non-empty queue of the Ω/K queues associated with the highest-order bit set, and dequeue the job at the head of the queue
      - clear the associated bit in the queue-status word if all Ω/K queues are empty
  - » Complexity: at most $\Omega/K + \log_2 K$
    - at most 13 comparisons for a 32-bit queue-status word and 256 priorities

EDF Scheduling Mechanisms

- Few operating systems support EDF Scheduling.
- However, only 2 changes to most kernels are required:
  - » The relative deadline parameter should be stored in the TCB.
  - » An absolute deadline is calculated from its release time and relative deadline and then stored in the TCB.
- Each priority represents a relative deadline value.
- A released thread is appended to the FIFO priority queue corresponding to its relative deadline.
  - » Thus, each of these FIFO queues are sorted by absolute time.
  - » The scheduler only needs to search among the threads at the head of each queue for the earliest deadline.
  - » Moreover, the threads at the head of each FIFO queue can also be kept in a separate priority queue sorted by absolute deadline.

Capabilities of Commercial Real-Time Operating Systems

- We will look at these real-time operating systems:
  - » LynxOS, pSOSystem, QNX, VRTX, and VxWorks
- Each of these systems shares the following attributes:
  - » Compliant or partially compliant to the Real-Time POSIX API Standard:
    - Preemptive, fixed priority scheduling
    - Standard synchronization primitives (mutex and message passing).
    - May support only threads or processes, but not both.
    - Each also has its own API.
  - » Modular and scalable
    - The kernel is small so that it can fit in ROM in embedded systems
    - I/O, file, and networking modules can be added

Shared Attributes (continued)

- Fast and efficient:
  - Most are microkernel sytems
  - Low overhead
  - Small context switch time, interrupt latency, and semaphore get/release latency: usually one to a few microseconds.
  - Nonpreemptable portions of kernel functions are highly optimized, short, and as deterministic as possible.
  - Many have systems calls that require no trap: applications run in kernel mode.
- Support Split Interrupt Handling
- Flexible Scheduling:
  - All offer at least 32 priority levels: min required by Real-Time POSIX.
  - Most offer 128 or 256 priority levels.
  - FIFO or RR scheduling for equal priority threads
  - Can change priorities, but EDF scheduling is not supported.

Shared Attributes (continued)

- Relatively High Clock and Timer Resolution:
  - Nominal timer resolution in the nanosecond range
  - Actual timer resolution in the microsecond range.
- No Paging or Swapping:
  - May not offer memory protection: often the kernel and all tasks execute in kernel mode, sharing one common address space.
  - May provide logical to physical memory translation.
- Optional Networking Support:
  - Can be configured to support TCP/IP with an optional module.

LynxOS

- Latest release, LynxOS 3.0, is based on a 28KB microkernel, which provides:
  - » scheduling, interrupt dispatch, and synchronization
- Kernel Plug-Ins (KPIs) are lightweight multi-threaded kernel service modules that can be added so that:
  - » LynxOS can serve as a multi-purpose Unix OS.
  - » LynxOS can be configured as a self-hosted system.
    - Embedded applications are developed on the same system on which they are deployed and run.
    - Greatly simplifies developing and debugging embedded systems.
- Thus, LynxOS also provides optional memory protection (with an MMU) and demand paging.

pSOSystem

- Object oriented operating system.
- pSOS+ is a preemptive, multitasking kernel that executes on a single processor.
- pSOS+m is a distributed multiprocessor kernel.
- Tasks can be scheduled with either preemptive priority-driven or time-driven scheduling algorithms.
- Device drivers are outside the kernel and can be loaded and removed at run-time.
  - » When an interrupt occurs, the processor jumps to the ISR via a vector table rather than going through the kernel.

QNX/Neutrino

- Capable of scaling from a 12KB microkernel on a single processor to a self-hosted, multi-processor OS executing on networked SMP machines with gigabytes of RAM.
- In Neutrino, the most recent version of the microkernel, the kernel only provides threads, context switching, and messaging services.
- Optional modules, called resource managers, can provide additional services ranging from memory protected processes to network services.
- QNX is a message-passing OS: messages are the basic means of IPC.
  - » QNX messages are blocking send/receive: no buffer copying

VRTX

- Two multitasking kernels: VRTXsa and VRTXmc
- VRTXsa is for medium and large real-time applications
  - » POSIX compliant library, even supports POSIX real-time extensions.
  - » System calls are deterministic and preemptable.
- VRTXmc is optimized for small embedded platforms:
  - » Cellular phones and other hand held devices.
  - » Provides only basic functions.
  - » Kernel requires 4-8KB of ROM and 1 KB of RAM.
- VRTX is the first commercial real-time OS certified by the FAA for mission and life-critical systems.
  - » Used for the avionics on the Boeing MD-11 aircraft.

VxWorks

- Used on the Mars Pathfinder
  - » A feature of the OS, called priority inheritance, fixed the problem Pathfinder was having after it landed.
  - » More on this story later in the semester!
- VxWorks is one of the few real-time operating systems that is a monolithic system rather than being based on a microkernel.
- However, it does allow major functions, such as memory protection and priority inheritance, to be disabled.
- Supports POSIX and most POSIX real-time extensions.
- The (native) VxWorks API is very popular and powerful.
  - » Many VxWorks API functions have a timeout feature.

VxWorks (continued)

- VxWorks is not a multiprocessor OS, but it provides shared-memory multiprocessing support as an option.
- Provides virtual-to physical address mapping using an MMU if one is available.
  - » Can make portions of memory non-cacheable.
  - » All task use a common context at system start time.
  - » However, a task can create a private virtual memory for memory protection.
- The Tornado tools create a cross-platform development environment:
  - » These tools execute on a host machine and communicate with the target over an I/O interface.

General Purpose Operating Systems used in Real-Time Systems

- General purpose operating systems are frequently being used to execute (soft or firm) real-time applications such as multimedia programs.
- We will look at Windows NT and Linux.
- Guess which is least suitable for real-time applications.
- Wrong! It is Linux.

Windows NT

- The Good:
  - » Supports threads, priority interrupts, and events.
  - » High timer and clock resolutions.
- The Bad:
  - » Large memory footprint (lets face it, this thing is a beast!)
  - » Weak support for real-time scheduling and resource access control. (We will not cover resource access control since we haven't yet talked about resource sharing.)
  - » Unpredictable interrupt handling and IPC mechanisms.
    - Can be somewhat controlled by keeping utilization low.
- The Ugly:
  - » Its Microsoft! ✪

Windows NT: Scheduling

- Limited Priority Levels: only 32 total
  - » Priorities 0-15 are for time-shared applications.
  - » Priorities 16-31 are for real-time applications--the OS never changes these priorities.
  - » Many kernel threads execute at priority level 16 (the lowest real-time priority level). Thus, memory management, file systems, and other services may be delayed by higher priority real-time threads.
  - » Worse, in NT 4.0, the SetThreadPriority() function only allows a user thread to specify 7 of the 16 real-time priority levels.
    - There is a kludge to work around this limitation.
    - NT 5.0, aka Windows 2000, fixes this deficiency.
      - – But then Windows 2000 requires 128MB RAM and recommends 256MB!

Windows NT: Scheduling (cont.)

- Scheduling within a Priority Level:
  - » NT 4.0 only supports RR (time-sliced) scheduling for threads with the same real-time priority.
  - » Windows 2000 RR or FIFO scheduling for threads with the same real-time priority, but you have to jump through hoops to enable the FIFO option.
- The scheduler/dispatcher thread may be blocked by scheduled IHRs, called Deferred Procedure Calls (DPC).
  - » Immediate ISRs are short and let DPCs do most of the device handling--just as all of the real-time operating systems do.
  - » DPCs execute below interrupt priorities, but higher than any scheduled threads, including the scheduler/dispatcher.
  - » Worse, their execution time is unbounded!

Windows NT: IPC Mechanisms

- Events are used to synchronize threads.
  - » A thread is suspended while it waits for one or more specified events. (Thus, events are a synchronous IPC mechanism.)
  - » Like Real-Time POSIX, events are queued. Thus, they will not be lost if not handled immediately.
  - » Unlike Real-Time POSIX, events are delivered in FIFO order and do not carry data.
- The kernel and device drivers use Asynchronous Procedure Calls (APC) to move data to/from user space.
  - » Kernel-mode APCs interrupt a thread to move data on behalf of the thread, but then the thread is nonpreemptable by higher priority threads while the APC executes.
  - » POSIX signals are implemented using APCs.

Linux

- The Good:
  - » 100 total priority levels by default.
  - » Linux provides flexible scheduling policies.
  - » You have the source. So if you don't like something, change it!
- The Bad:
  - » Most Linux subsystems disable priorities in critical sections.
    - For the disk subsystem this may be a few hundred microseconds.
  - » Timer error can be large and unpredictable.
- The Ugly:
  - » What could be ugly about Linux! ✪
    - Well, besides its still not based on a microkernel.

Linux: Scheduling

- 100 Priority Levels
- Linux provides SCHED_FIFO, SCHED_RR and SCHED_OTHER scheduling policies
  - » SCHED_FIFO and SCHED_RR are fixed priority algorithms for real-time processes.
  - » SCHED_OTHER is a time sharing algorithm for non-real-time processes, which execute at a lower priority than real-time processes.
- LinuxThreads, by X. Leroy, provides a kernel-level threads package that provides most of the POSIX thread extensions.

Linux: UTIME Extension

- Linux provides low resolution timers
  - » Timers are serviced during clock interrupts, normally every 10 milliseconds using the periodic timer mode of the clock device.
  - » Timer handlers are executed just before the kernel returns control to the application.
  - » This results in large and unpredictable timer error.
- UTIME is a high resolution time service developed by Douglas Niehaus and his students at Kansas University.
  - » Provides microsecond clock and timer granularity
  - » Uses one-shot timer mode of the clock device to generate an interrupt when the next timer expiration occurs.
  - » No free lunch: the execution time of the timer ISR is several times larger than in standard Linux.
  - » Timer handlers are still executed just before the kernel returns control to the application.

KURT: Kansas University Real-Time Linux

- Linux extension that supports firm real-time applications.
- Three modes:
  - » Focused mode - only real-time processes run when the system is in focused mode. These processes are scheduled in a time-driven manner using a pre-computed table.
  - » Normal mode - standard Linux mode.
  - » Mixed mode - non-real-time processes run in the background of real-time processes.
- Source of Unpredictability:
  - » KURT allows large schedules that may need to be paged.
  - » All of the normal stuff: interrupts disabled for extended time, disk accesses, etc.

Real-Time Linux from New Mexico Tech

- Extends Linux to supports hard real-time applications.
- Divides applications into two parts.
  - » A real-time part that executes on the RT kernel.
  - » A non-real-time part that executes on “normal” Linux.
- Linux is actually executed as the idle task for the RT Kernel.
  - » The RT Kernel captures all interrupts and then forwards interrupts destined for Linux via software emulated interrupts.
  - » Thus, Linux is fully preemptable for the real-time tasks.
- All communication between real-time tasks and non-real-time tasks is done via FIFO buffers locked in memory.
- The most recent version supports a POSIX API and SMPs.

Real-Time Linux Structure