Real-Time Support in Operating Systems

Real-Time and Embedded Systems (M)

Lecture 9

UNIVERSITY
of
GLASGOW

Lecture Outline

- Real time vs. general purpose operating systems
  - Real time constraints
  - Implications on **r**eal **t**ime **o**perating **s**ystem (RTOS) design
- Real time operating system concepts
  - Overall system architecture
  - Time services and scheduling
  - Interrupts, hardware, and system calls
- Examples of real-time operating systems

- The material in lectures 9-12 corresponds to parts of chapter 7, chapter 12, and the appendix of Liu's book

Practical Real Time Systems: Constraints

- Key fact from scheduler theory: *need predictable behaviour*
  - Raw performance less critical than consistent and predictable performance; hence focus on scheduling algorithms, schedulability tests
  - Don't want to fairly share resources – be unfair to ensure deadlines met
- Need to run on a wide range of – often custom – hardware
- Often resource constrained
  - Limited memory, CPU, power consumption, size, weight, budget
- Embedded and may be difficult to upgrade
  - Closed set of applications, trusted code
  - Strong reliability requirements – may be safety critical
  - How to upgrade software in a car engine? A DVD player? After you shipped millions of devices?

Implications on Operating Systems

- General purpose operating systems not well suited for real time
  - Assume plentiful resources, fairly shared amongst un-trusted users
  - Exactly the opposite of an RTOS!

- Instead want an operating system that is:
  - Small and light on resources
  - Predictable
  - Customisable, modular and extensible
  - Reliable

  …and that can be *demonstrated* or *proven* to be so

Real-Time Operating System Concepts

- Real-time operating systems typically *microkernel* or *nanokernel* designs, rather than a traditional monolithic kernel
  - Limited and well defined functionality
  - Easier to demonstrate correctness
  - Easier to customise
- Provide rich scheduling primitives
- Provide rich support for concurrency
- Expose low-level system details to the applications
  - Control of scheduling
  - Power awareness
  - Interaction with hardware devices

Nanokernel Architecture

- The simplest real-time systems use a limited *nanokernel* system
  - Provides a minimal time service: scheduled clock pulse with fixed period
  - No tasking, virtual memory/memory protection, etc.
  - Allows implementation of a static cyclic schedule, provided:
    - Tasks can be scheduled in a frame-based manner
    - All interactions with hardware to be done on a polled basis
- Operating system becomes a single task cyclic executive

```
setup timer
c = 0;
while (1) {
        suspend until timer expires
        c++;
        do tasks due every cycle
        if ((c % 2) == 0) do tasks due every 2nd cycle
        if (((c+1) % 3) == 0) {
                do tasks due every 3rd cycle, with phase 1
        }
}
```

Microkernel Architecture

- The nanokernel idea is widely used in low-end embedded devices
  - 8 bit processors with kilobytes of memory
  - Often programmed in C via cross-compiler, or assembler
  - Simple hardware interactions
  - Fixed, simple, and static task set to execute
  - Cyclic scheduler

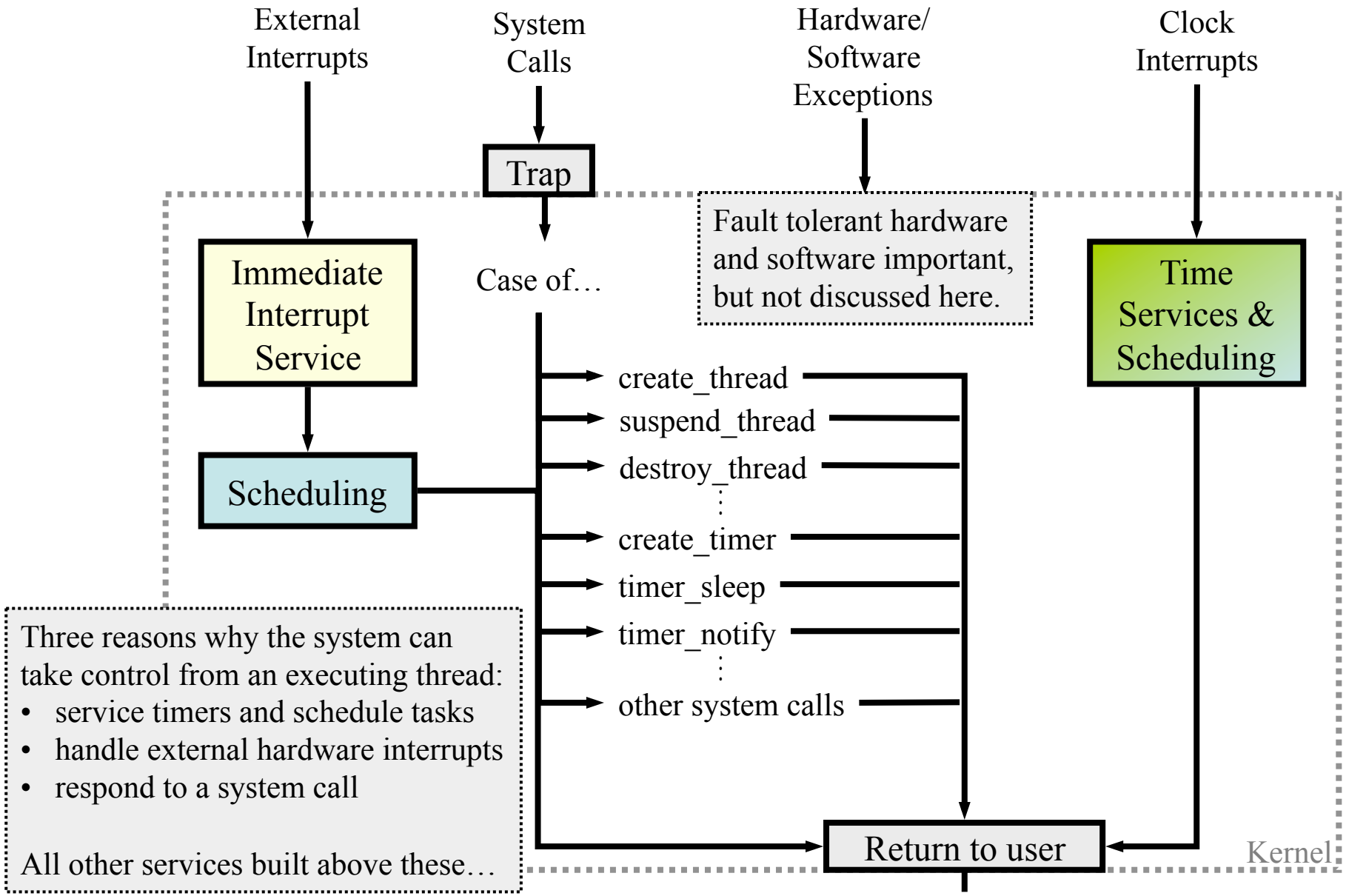
- But… many real-time embedded systems are more complex, and need a more sophisticated operating system
- Common approach: a *microkernel* based system
  - Configurable and robust, since architected around interactions between cooperating system servers, rather than a monolithic kernel with ad-hoc interactions

Microkernel Architecture

- A microkernel RTOS typically provides a number of features:
  - Scheduling
  - Timing services, interrupt handling, support for hardware interaction
  - System calls with predictable timing behaviour
  - Messaging, signals and events
  - Synchronization and locking
  - Memory protection

- These features often differ from non-RTOS environments

Microkernel Architecture

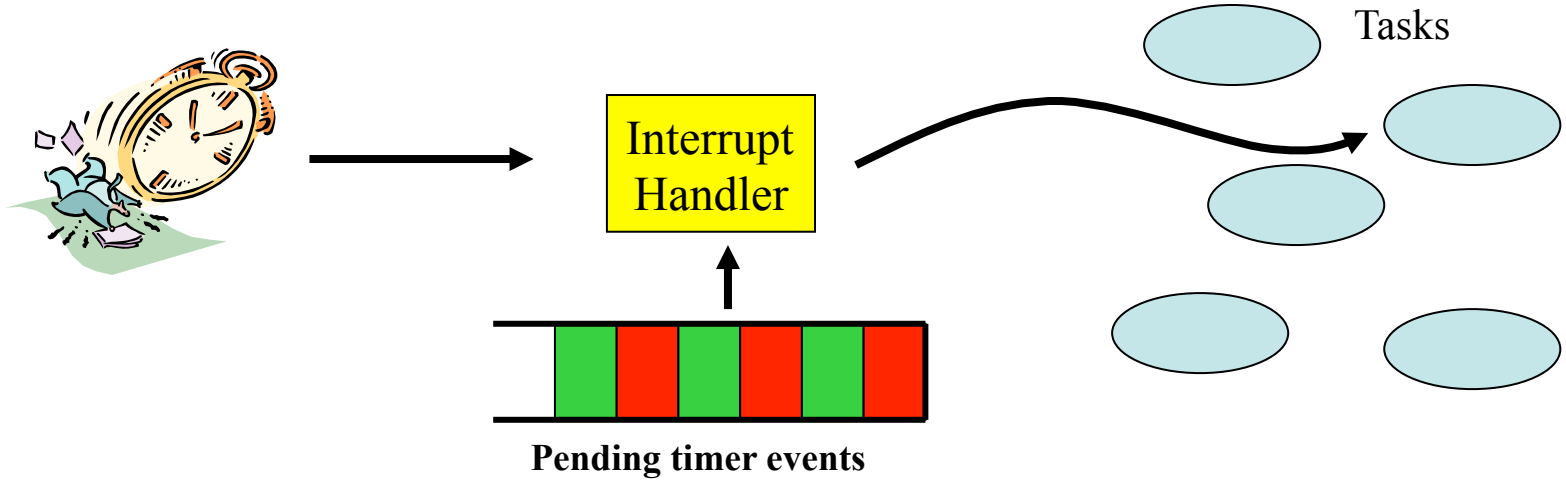


Scheduling

- Cyclic executives trivial to implement, but inflexible and support only simple clock-driven scheduling
- To implement other scheduling algorithms, need system support:
  - System calls to create, destroy, suspend and resume tasks
  - Implement tasks as either *threads* or *processes*
    - Processes (with separate address space and memory protection) not always supported by the hardware, and often *not useful*
  - Scheduler with multiple priority levels, range of periodic task scheduling algorithms, support for aperiodic tasks, support for sporadic tasks with acceptance tests, etc.

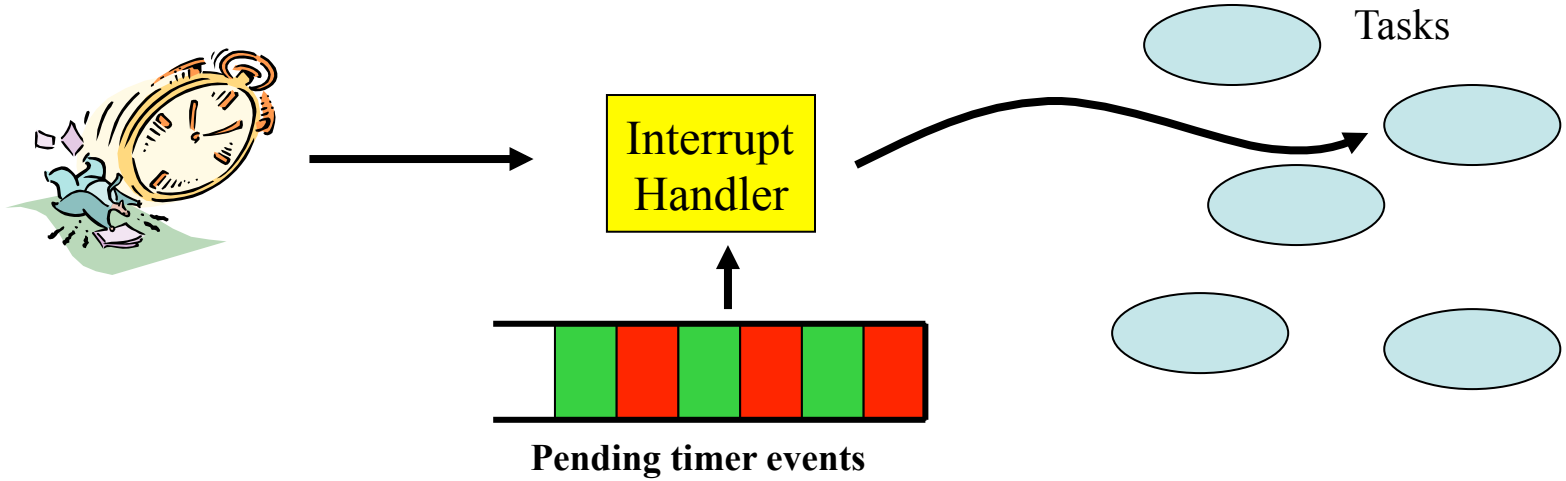
Lecture 10 will discuss implementation of schedulers, common APIs for task/job creation, management, etc.

Time Services



- Most systems support *time services interrupts* derived from high resolution hardware timers
  - Resolution and accuracy fixed, depend on hardware
    - Resolution: minimum interval that can be distinguished
    - Accuracy: is that interval correct? too large? too small?
  - Stability depends on environment and hardware
    - Is the accuracy constant? Does it depend on temperature or system load?
    - Worst-case behaviour depends where the system is installed; can affect design

Time Services



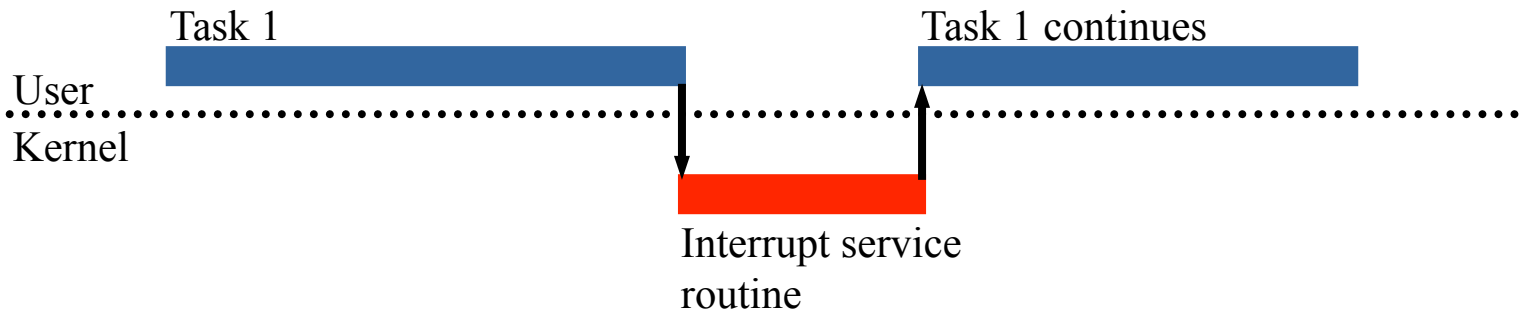
- Handling timers and other interrupts has latency
  - Depends on the number of pending timer events
  - Decreases perceived accuracy of the clock
- May be non-deterministic, depending on system and load
  - On general purpose system, may be 10s of milliseconds
    - Problematic for soft real time on general purpose systems
  - Smaller *and with guaranteed deterministic bounds* on dedicated RTOS

Interrupts and Device Interaction

- In addition to driving the scheduler and timing services, interrupts are often used to signal that hardware devices require attention
  - Useful for devices that require sporadic or aperiodic attention
  - Polling is an alternative, that may be lower overhead, for devices that need periodic attention
- Interrupts have priority, asserted by the hardware
- The system may support more devices than interrupt request lines
  - Require the interrupt service routine to poll multiple devices

Interrupts and Device Interaction

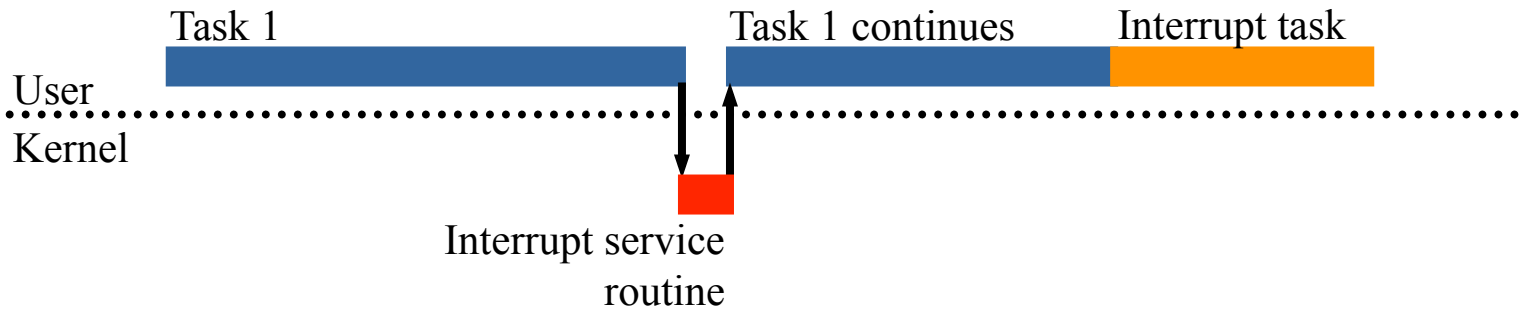
- Interrupts may be serviced immediately
  - Fast response to device
  - Significant delay to currently scheduled task, but common in non-RTOS
    - [Recall lecture 7]



- Interrupts disabled during the interrupt service routine
  - To safely modify the interrupt management data structures
  - Problematic if there are multiple interrupt sources, since high priority interrupts can be blocked

Interrupts and Device Interaction

- Alternatively, an interrupt service routine may schedule a task to complete the interaction
  - Slower to service the device
  - Potentially less impact on interrupted task, although this depends on the priority of the interrupt handler task



- Reduces blocking time during the interrupt service routine
  - Allows higher priority interrupts to activate while interrupt task executes
  - Allows us to schedule device handling as any other task, *and reason about correctness*

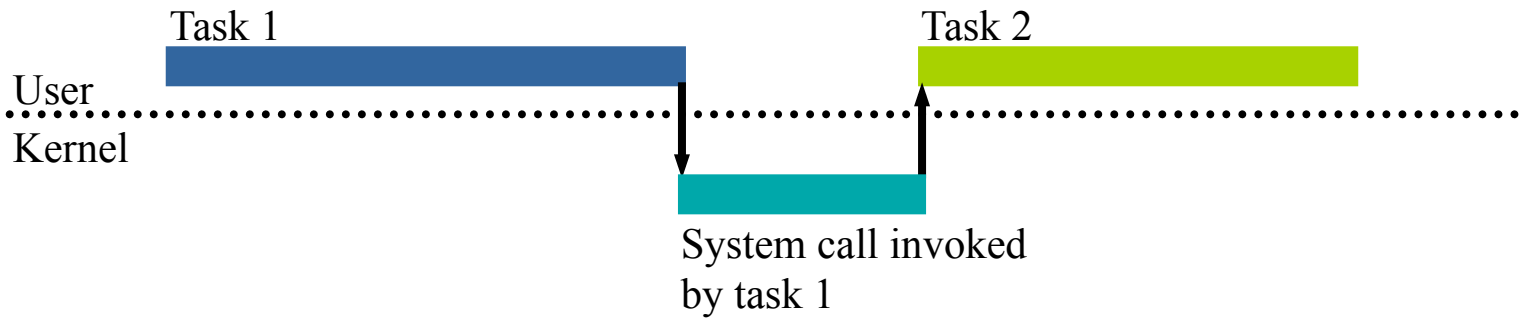
Interrupts and Device Interaction

- Interrupt latency varies:
  - Depending how many devices must be polled to locate the interrupt source
  - Depending on the tasks executing when the interrupt arrives
  - If multiple interrupts occurs at once
- Affects response time to hardware interrupts
- Affects clock and time service latency
- Implications:
  - Jitter on job release times
  - Inaccuracy on timers

…will affect scheduling

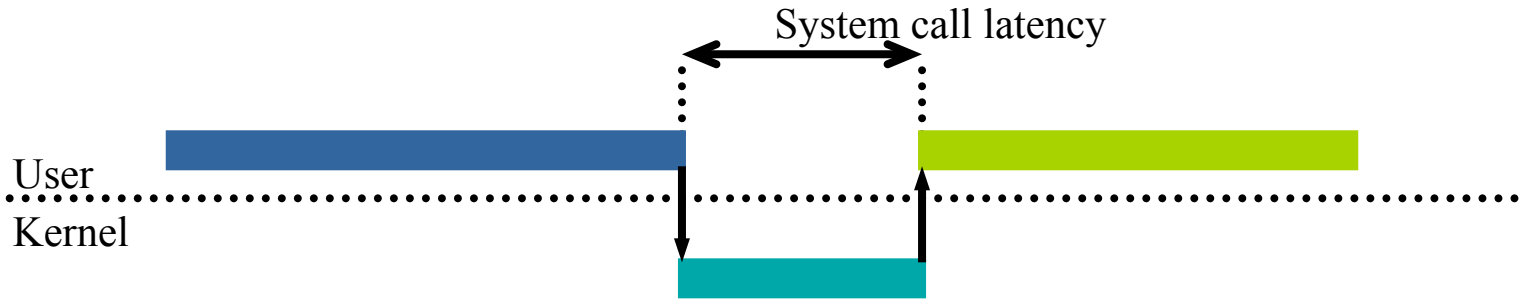
Synchronous System Calls

- Tasks invoke microkernel through *system calls*
- The kernel usually executes in separate a memory space, protected from user tasks
  - System calls traps to kernel mode, switches protection domains, and saves the context of the user thread
  - Reduces to a function call if the system doesn't implement memory protection
- Call name and arguments retrieved from the stack, and executed by the kernel on behalf of the thread
- When a system call completes:
  - The kernel saves the result
  - Switches protection domains and returns to user mode (if necessary).
  - The highest priority thread is executed (may differ from the thread that initiated the call)



Synchronous System Calls

- Some systems cannot be pre-empted during system calls
  - Worst case system call latency must be taken into account when scheduling
    - Systems calls might block for 100s of milliseconds, if they result in access to a hardware device
  - Example: low latency patches for Linux to reduce audio skip
- An RTOS will have a guaranteed bound on system call latency, a general purpose OS will not



Other Operating System Features

- Many standard operating system features also useful in real time operating systems:
  - Messaging, Signals and Events
  - Synchronisation and Locking
  - Memory Access and Protection

- Concepts and services often need to be slightly modified from their general purpose equivalents
- Typically implemented as *system service providers* outside the microkernel
  - Operating system literature refers to these as *servers*; we use the term system service provider to avoid confusion with other server tasks scheduled at the application level
  - Allows them to be optional, only implemented when needed

Messaging, Signals and Events

- In addition to typical inter-process communication features, real time operating systems provide:
  - Message queues (synchronous, as an alternative to pipes)
    - Messages delivered in priority order
    - Priority inheritance based on message reception
    - Notification and wakeup on message arrival
  - Asynchronous event notification (application defined signals)
  - Real time signals, priority queued, carrying data

- Key point: inter-process communication prioritised, scheduled, as other real-time functions

Synchronisation and Locking

- In addition to mutexes, locks, condition variables and semaphores a real-time operating system may support:
  - Priority inheritance and restoration
  - The priority ceiling protocol, or the similar ceiling priority protocol
  - Ability to disable priority inheritance to reduce overheads, if not needed

- Again: key point is to allow reasoning about timing behaviour of a system even in the presence of resource locking

[Details in lectures 13 and 14]

Memory Access and Protection

- Many embedded real time systems *do not use* memory protection
  - Any task can directly access any other task's memory
  - Any task can directly access kernel memory
- If a closed system has been proved correct, memory protection is unnecessary overhead
  - Time overheads, and unpredictability
    - Context switch overhead
    - Unpredictable access times, especially if virtual memory used
  - Memory overheads
    - Protection provided on a per-page basis, leads to wastage
    - Overhead of maintaining the page tables and protection maps
- Most commercial RTOS provide memory protection as an option, if supported by the hardware
  - Requires the system to fail-safe if an illegal access trap occurs
  - Useful for complex (or reconfigurable) systems

Summary of RTOS Features

- Modular and scalable
  - Configurable microkernel
- Efficient: low overhead, highly optimised
- System calls and interrupt handling optimised
  - To make the non-premptible regions small and predictable
  - Many actions deferred to kernel threads, with appropriate priority
- Scheduling
  - Fixed priority scheduling, many priority levels, reconfigurable
  - Limited support for deadline scheduling
- Priority inversion control
- Clock and timer resolution on the order microseconds
- Memory management
  - No paging, no cache, protection optional

RTOS Standards

- Operating systems are notoriously non-standard
- IEEE 1003 POSIX
  - Portable Operating System Interface
    - "The name POSIX was suggested by Richard Stallman. It is expected to be pronounced *pahz-icks*, as in positive, not *poh-six*, or other variations. The pronunciation has been published in an attempt to promulgate a standardized way of referring to a standard operating system interface."
  - Started as a subset of Unix functionality, various (optional) extensions have been added to support real-time scheduling, signals, message queues, etc.

- Widely implemented…
  - Widespread support in RTOS
  - Partial support in Unix variants
  - Limited support in Linux and Windows

  …but often in second place to "native" facilities

Example Real Time Operating Systems

Commercial RTOS:

- QNX/Neutrino
- VxWorks
- RTLinux
- Real-time Java

General purpose systems with real-time extensions:

- Windows NT
- Unix/Linux

Example Real Time Operating Systems

Commercial RTOS:

- QNX/Neutrino
- VxWorks
- RTLinux
- Real-time Java

Traditional RTOS:
- Dedicated real-time operating systems
- Provide all the features discussed
- POSIX compliant, with extensions

The book talks about these, and others, in some detail

General purpose systems with real-time extensions:

- Windows NT
- Unix/Linux

Example Real Time Operating Systems

Commercial RTOS:

- QNX/Neutrino
- VxWorks
- RTLinux
- Real-time J

• Non-traditional microkernel RTOS with hard real-time support
• Supports most features described, unusual programming model
• Runs Linux as a background periodic server
  - Linux tasks communicate with the RTOS through non-blocking and non-real time communications channels

General purpose systems with real-time extensions:

- Windows NT
- Unix/Linux

Example Real Time Operating Systems

Commercial RTOS:

- QNX/Neutrino
- VxWorks
- RTLinux
- Real-time Java

Recent extensions provide real-time support in Java:

- Real-time code cannot use the garbage collector
- Provides high-resolution timers
- Provides abstractions for periodic, aperiodic and sporadic tasks; real-time priority scheduler
- Provides access to physical memory/hardware

⇒ Reasonable real-time support, but limits access to other Java features and controls interactions with non-real time Java threads

General purpose systems with real-time extensions:

- Windows NT
- Unix/Linux

Example Real Time Operating Systems

Commercial RTOS:

- QNX/Neutrino
- VxWorks
- RTLinux
- Real-time Java

General purpose systems with real-time extensions:

- Windows NT
- Unix/Linux

- Support a subset of RTOS features
- Suitable for *soft* real time applications, no effective guarantees on scheduling performance, latency, memory access

Summary

- Outlined some features of real-time operating systems, differences from general purpose systems
- Outlined examples of commercial RTOS

In lecture 10: implementation and use of schedulers…