REAL TIME OPERATING SYSTEMS

Adhokshaja V. Madhwaraj, 160010032
Venkata Kowsik T, 160010035

Motivation behind RTOSs

- Embedded computing Applications - cell phones to home automation, automobiles
- Functionality, performance, and reliability that simple assembly cannot handle
- Large part of computation is “real-time” - time constrained or external environment driven
- Important to understand the basics of OSs to delve further into RTOSs (Obviously, duh!)

What is an Operating System?

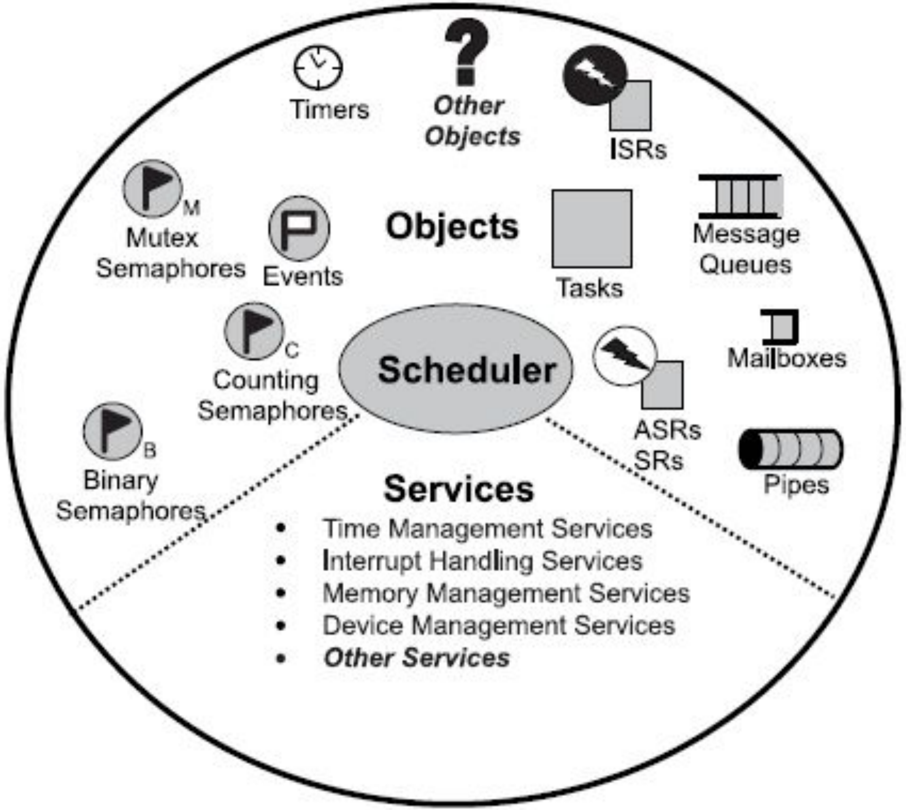
We all know this, but wouldn’t hurt to have a bit of a recap, would it?

Real-Time Operating Systems

- Special features for RTOSs
- Basic support for - Scheduling, Resource management, Synchronisation, etc.
- Additional features for precise timing
- Proprietary kernels for Embedded systems like Atmega, composition based kernels, Linux RT, Windows CE
- Correctness not only dependant on logical result, but also result delivery time

RTOS KERNEL

- **Scheduler** - explained in detail further
- **Objects** - tasks, semaphores, message queues
- **Services** - General operations like Timing, Interrupt Handling, Resource Management



HARD vs SOFT RTOSs

Hard Real-Time System

- Can never miss a deadline
- Missing - disastrous consequences
- Usefulness decreases abruptly as tardiness increases

Soft Real-Time System

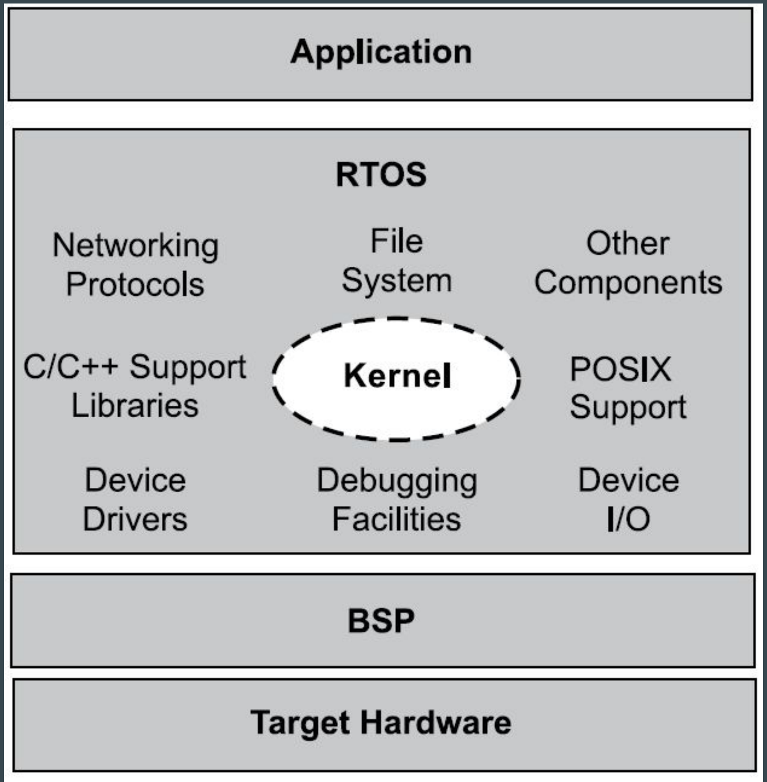
- Can miss a deadline occasionally
- No disastrous consequences
- Usefulness decreases gradually as tardiness increases

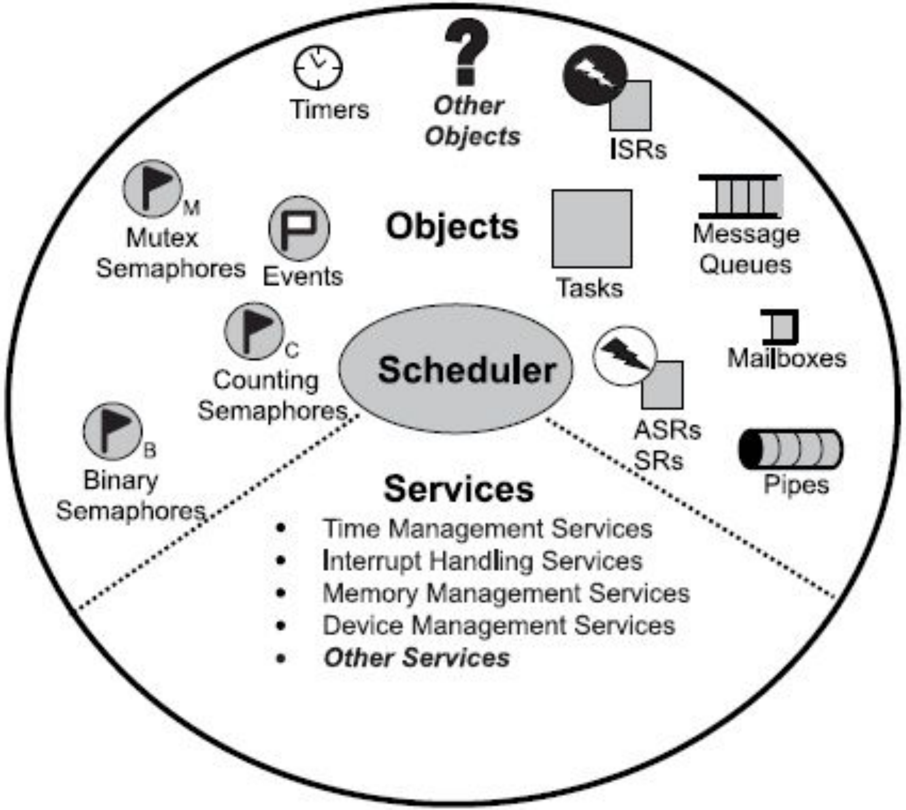
TARDINESS???

INSIDE AN RTOS KERNEL

A deeper look at the scheduler, the heart of the Kernel…

RTOS KERNEL


Application
RTOS
Networking Protocols
File System
Other Components
C/C++ Support Libraries
Kernel
POSIX Support
Device Drivers
Debugging Facilities
Device I/O
BSP
Target Hardware


Timers
Other Objects
ISRs
Mutex Semaphores
Events
Objects
Tasks
Message Queues
Counting Semaphores
Scheduler
Mailboxes
ASRs SRs
Binary Semaphores
Pipes
Services
• Time Management Services
• Interrupt Handling Services
• Memory Management Services
• Device Management Services
• Other Services

Schedulable Entities

- A kernel object that competes for execution time
- Tasks and Processes…

Tasks	• Independent threads of execution containing sequence of independently schedulable instructions • No protection, increased performance
Processes	• **Similar** - independently compete for CPU • **Differ** - better memory protection, reduced performance and higher memory overhead

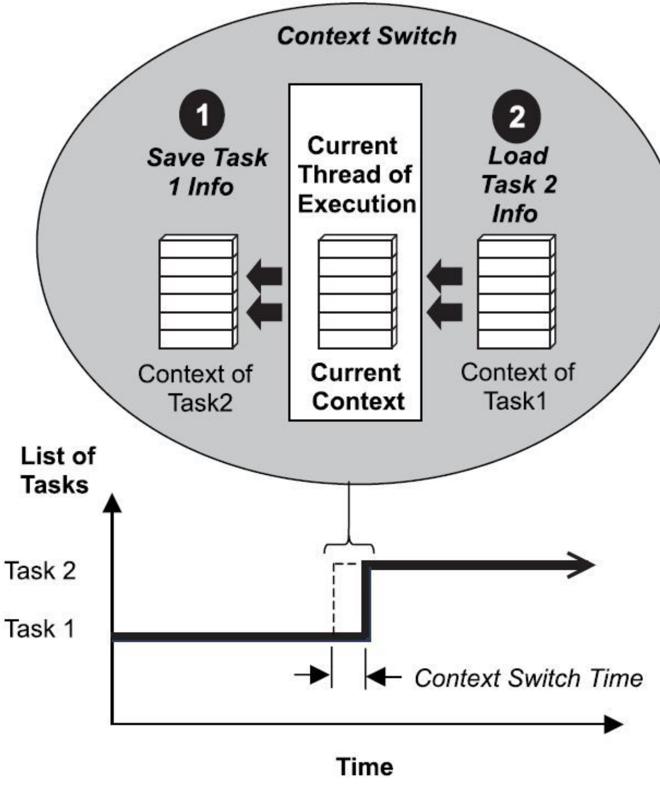
Multi-Tasking

- Handle multiple activities - **within set deadlines**
- Illusion of concurrent execution - interleaving executions sequentially
- How? **Scheduling Algorithms**

- Appropriate task should run at the exact right time - Most important feature of an RTOS
- Tasks : Kernel Scheduling Algo : : ISR : Hardware Interrupts & pre-Established priorities
- Next up - Context Switch... Increased CPU usage

CONTEXT SWITCH

- Context switch happens every time a process switches
- **TCB - Task Control Block** ~ PCB
- Three step process followed for a context switch
- Frequent Context Switch - Incur extra performance overhead
- Dispatcher - Associated module to make the switch
- **Task control functions** to spawn, initialize and activate new tasks
- Task naming, state checking, etc...
- Deletion needs precautions



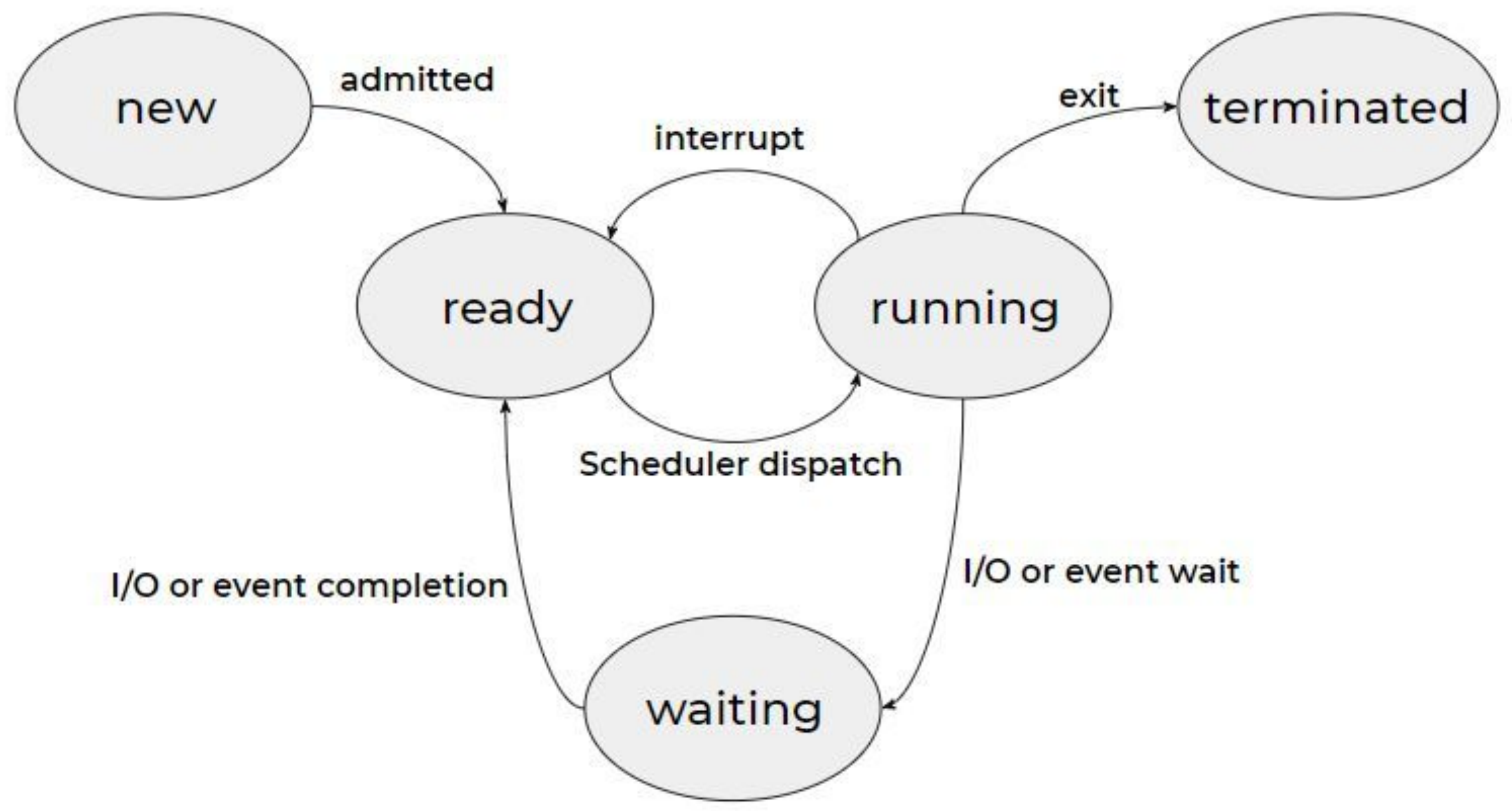
TASK CONTROL BLOCK

TASK CONTROL BLOCK

- **Task ID**: The unique identifier for a task
- **Address Space**: The address ranges of the data and code blocks of the task loaded in memory
- **Task Context**: PC, CPU registers (optional), FP registers, dynamic variables in a stack, Stack Pointer, I/O device assignments
- **Task Parameters**: includes task type, event list
- **Scheduling Information**: priority level, relative deadline, period
- **Synchronisation Information**: Semaphores, pipes, mailboxes, message queues, file handles, etc.
- **Parent** and **Child** Tasks

The Dispatcher

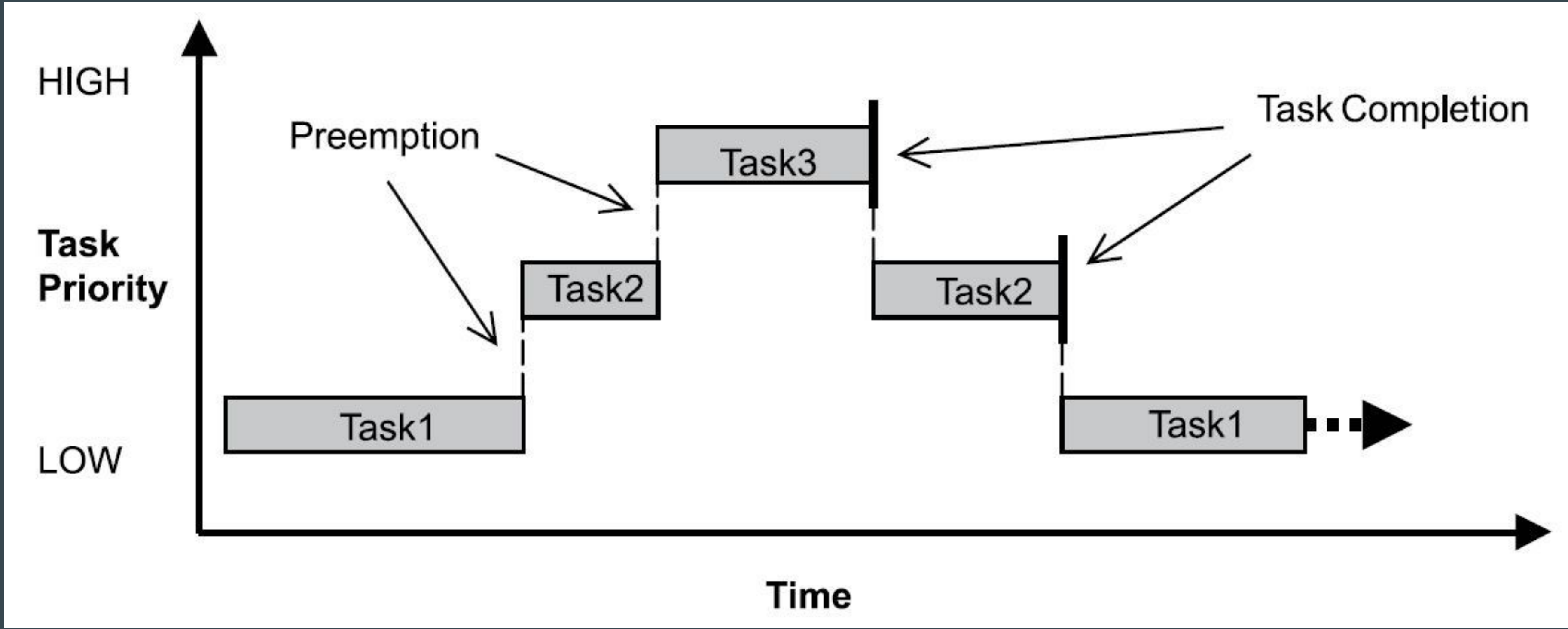
- Actually performs the context switch
- Flow of control -> 3 areas, namely:
    - **Application Task**
    - **Interrupt Service Routine (ISR)**
    - **Kernel**
- Task or ISR makes a system call - control passes to the kernel - executes a system routines provided by the kernel
- Dispatcher can be used on a call-by-call basis - coordinate task-state transitions that any of the system calls might have caused
- Why? More than one process in the READY Queue


new
admitted
terminated
exit
interrupt
ready
running
Scheduler dispatch
I/O or event completion
I/O or event wait
waiting

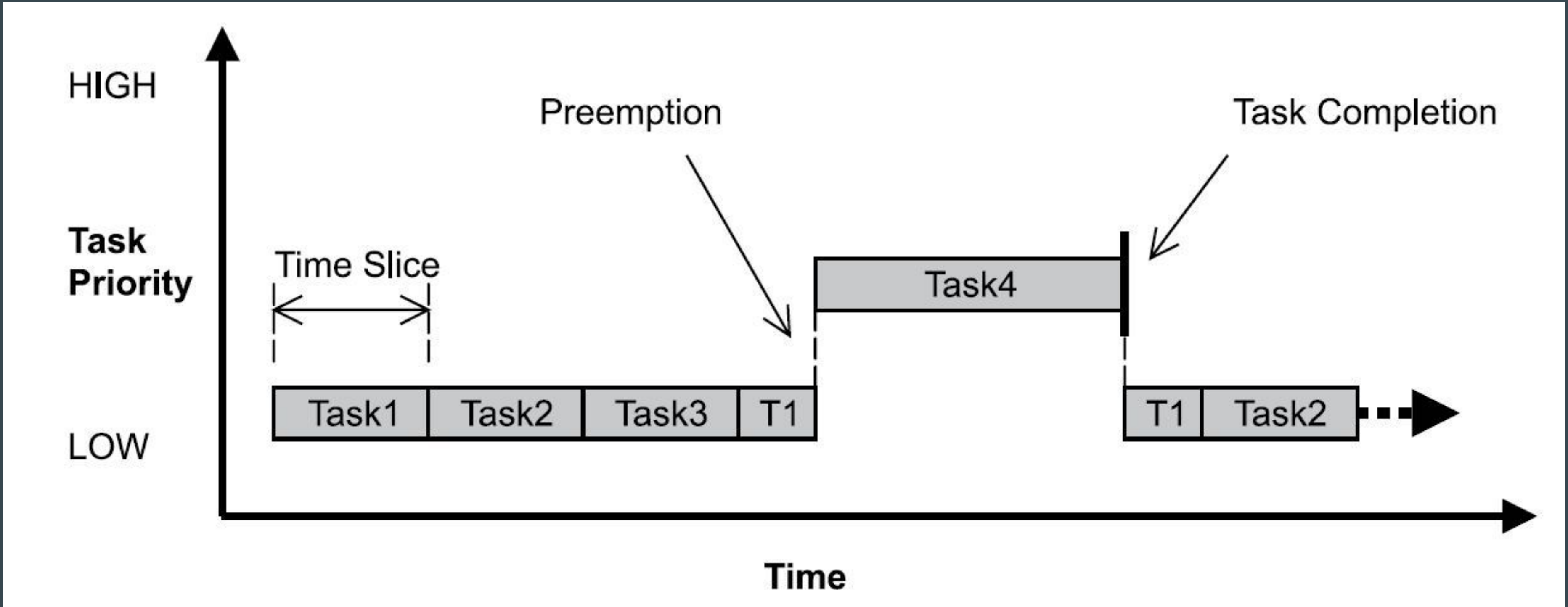
What's different in RTOS...

- ISR makes system call -> dispatcher is bypassed until ISR completes routine
- **ISR has highest priority**
- ISR should complete its work without getting interrupted
- Once done, kernel exits through the dispatcher, so that the next correct task is continued with.
- Very important feature, especially for embedded systems. **Why?**

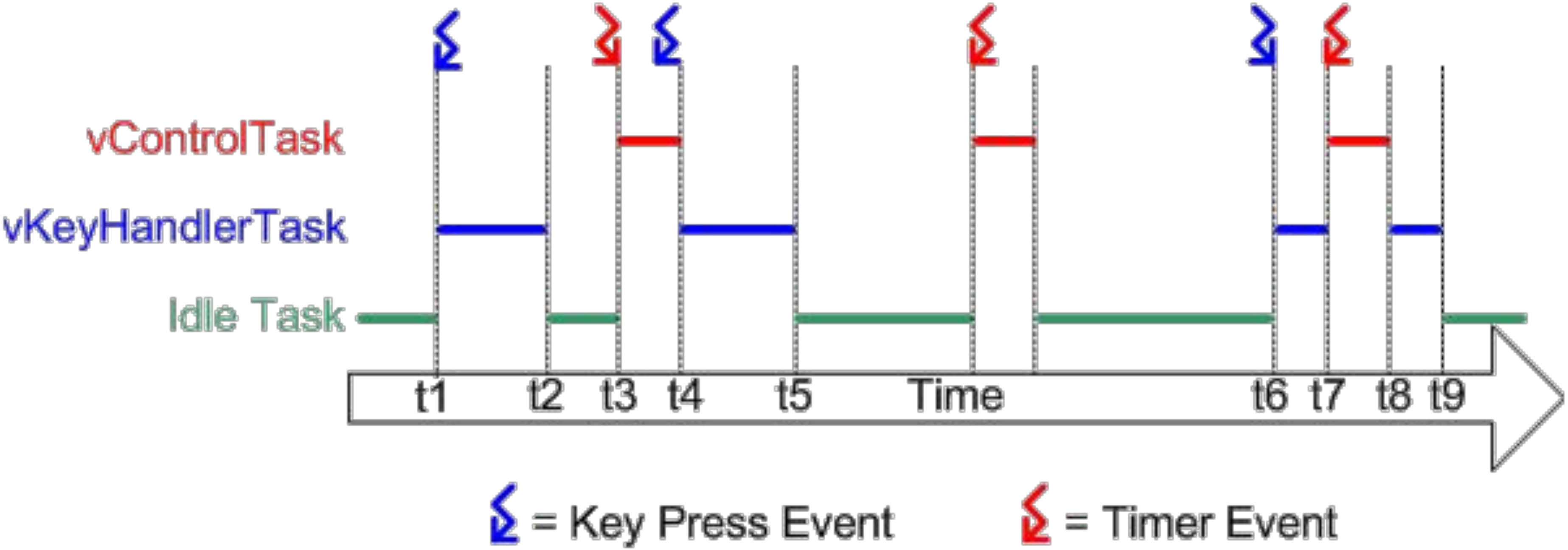
Scheduling Algorithms



Scheduling Algorithms



Scheduling Algorithms An example from FreeRTOS



Multitasking Models for RTOSs

- Explicit implementation of a scheduling policy is through a **module**
- Schedule itself is a task which executes when -
  - External / Internal interrupt - evaluates state of all non-terminated processes
  - Decision taken based on **priority level**, **availability of resources**
- Current priorities re-evaluated based on **deadline, computational dependencies, waiting times**, etc
- Dispatcher affects state transition of tasks based on scheduler:
  - Saving computational context of currently executing task
  - Enable next task by loading context - Also makes **short term decisions** in response to interrupts and I/O

Entry Conditions to Dispatcher

Real time clock interrupt -OR- **I/O completion interrupt**

- Search for work starting with highest priority
- **High repetition rate** ~ High priority - Clock-Level tasks, pretty much highest...
- Run **first** during each system clock period

Task suspension due to delaying, completion, request of I/O

- Search for work started at lowest priority to the task that was just running
- Can not be a higher priority task being READY since this would preempt the current process - INTERRUPTED...

Levels of Priority in a typical RTOS

Interrupt Level	Hard Real-Time Level	Soft Non-Real-Time
• Require very fast response (in **ms**) • No scheduling, immediate execution • Predictable system behaviour • Special common contexts • System Clock, WatchDog timers	• Tasks that are periodic Carried out based on the real-time sysClock • Virtual Software Clock Maintained based on sysClock • New task dispatched based on scheduling Algo - Lowest level task is scheduler...	• Tasks either have no deadlines / wide error margin • Low priority - executed only when no other higher priority task pending • Priority that of Base Scheduler - Round Robin fashion

Key Characteristics of an RTOS

- Reliability

Number of 9s	Downtime per year	Typical Application
3 Nines (99.9%)	~9 hours	Desktop Computer
4 Nines (99.99%)	~1 hour	Enterprise Server
5 Nines (99.999%)	~5 minutes	Carrier-Class Server
6 Nines (99.9999%)	~31 seconds	Carrier Switch Equipment

Table: Categorizing highly available systems by allowable downtime

Key Characteristics of an RTOS

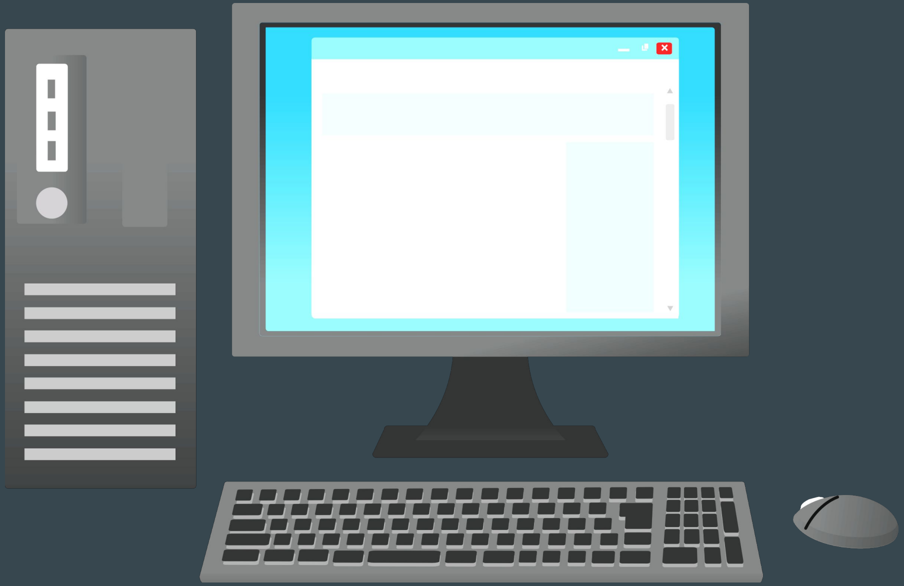
- Predictability
- Performance
- Compactness
- Scalability

DIFFERENCE BETWEEN A **GPOS** AND AN **RTOS**

This is the last thing we'll cover, so pay attention xD

Time Criticality - An Example

Personal Computer vs Automated Teller Machines



Task Scheduling

- **GPOS** - task scheduling is **not** based on **priority** always
- Priority is handled to manage high throughput - total number of processes that finish per unit time.
- In this case, the execution of a higher priority process may be slightly delayed to accommodate around 5-6 lower priority processes
- **RTOS** - scheduling is **always based on priority**.
- A high priority process execution will get override only if a request comes from an even high priority process

Hardware and Economic Factors

- **GPOS** - Generally high end systems like a PC, Work Station, etc.
- **RTOS** - Light weight and small in size compared to GPOS - Kiosks, ATMs, Vending Machines, etc.
- Main difference - Hardware Factors
- **GPOS** - GHz processors, TBs of Storage, GBs of Memory
- **RTOS** - MHz processors, GBs of Storage, MBs of Memory
- Not economical to run a GPOS on an ATM, does only basic functionality like Money transfer, Withdrawal, Balance check, etc.

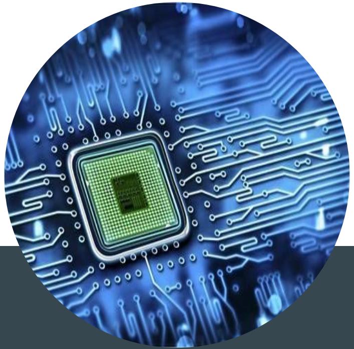
Latency Issues

- GPOS - Unbounded dispatch latency
- Number of threads to schedule increase, latencies also add up
- RTOS - No such issues due to bounded latencies
- This means every process / thread gets executed within a specified time limit

Preemptible Kernel

- **GPOS** - Kernel is not preemptible, **RTOS** - Kernel is preemptible
- If the kernel is not preemptible, then a request/call from kernel will override all other process and threads.
- In an RTOS the **kernel is kept very simple** and only very important service requests are kept within the kernel call. All other service requests are treated as **external processes and threads**.
- All such service requests from the kernel are associated with a bounded latency in an RTOS.
- This ensures a **highly predictable** and **quick response** from an RTOS.

TakeAway Points



Embedded Systems

RTOSs suited for Real-Time, application specific embedded systems

Timely Execution

Manage system resources, consistent foundation for application code

Kernels

Core Module
Deploy algorithms, objects, services, dispatching

Key Characteristics

Reliable, Predictable, High Performance, Compact, and Scalable

Thank You!