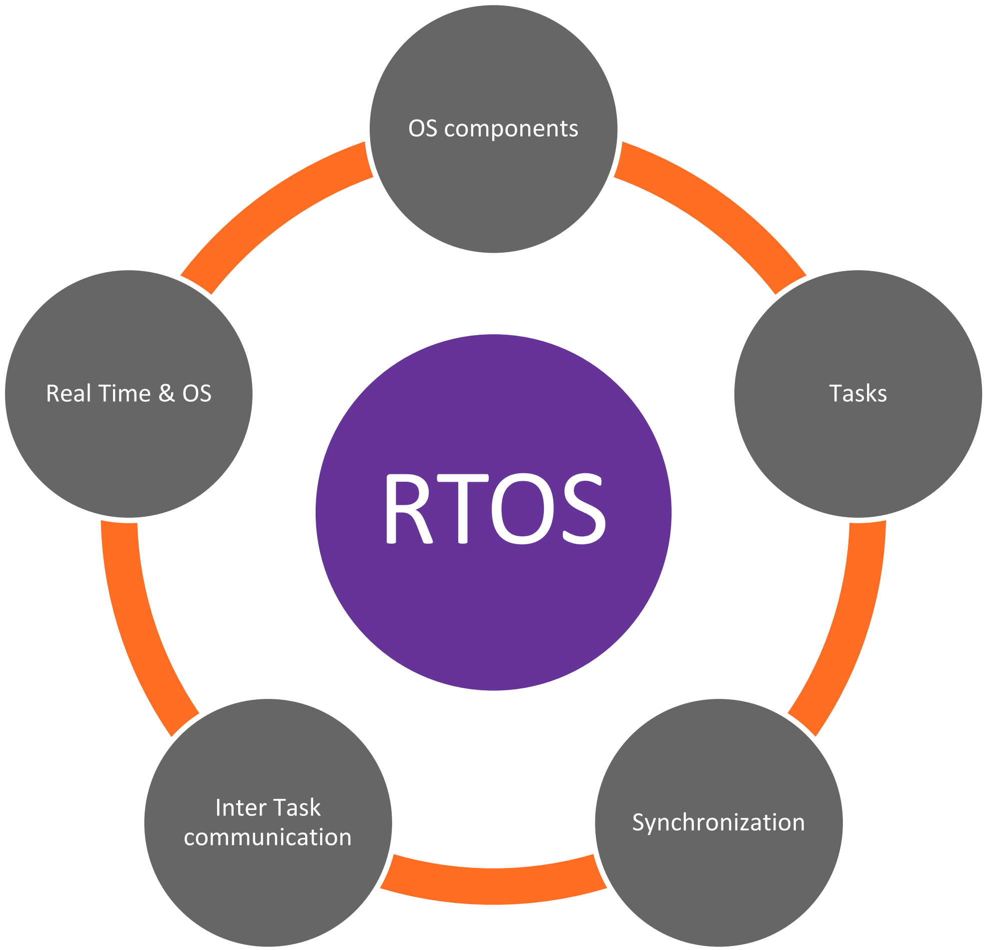
Real Time Operating System (RTOS)

Team Embedded
Emertxe Information Technologies

Course span-out



What is an Operating System (OS)?

Let us ponder…

- What exactly is an Operating System (OS)?
- Why do we need OS?
- How would the OS would look like?
- Is it possible for a team of us (in the room) to create an OS of our own?
- Is it necessary to have an OS running in a Embedded System?
- Will the OS ever stop at all?

Types of OS

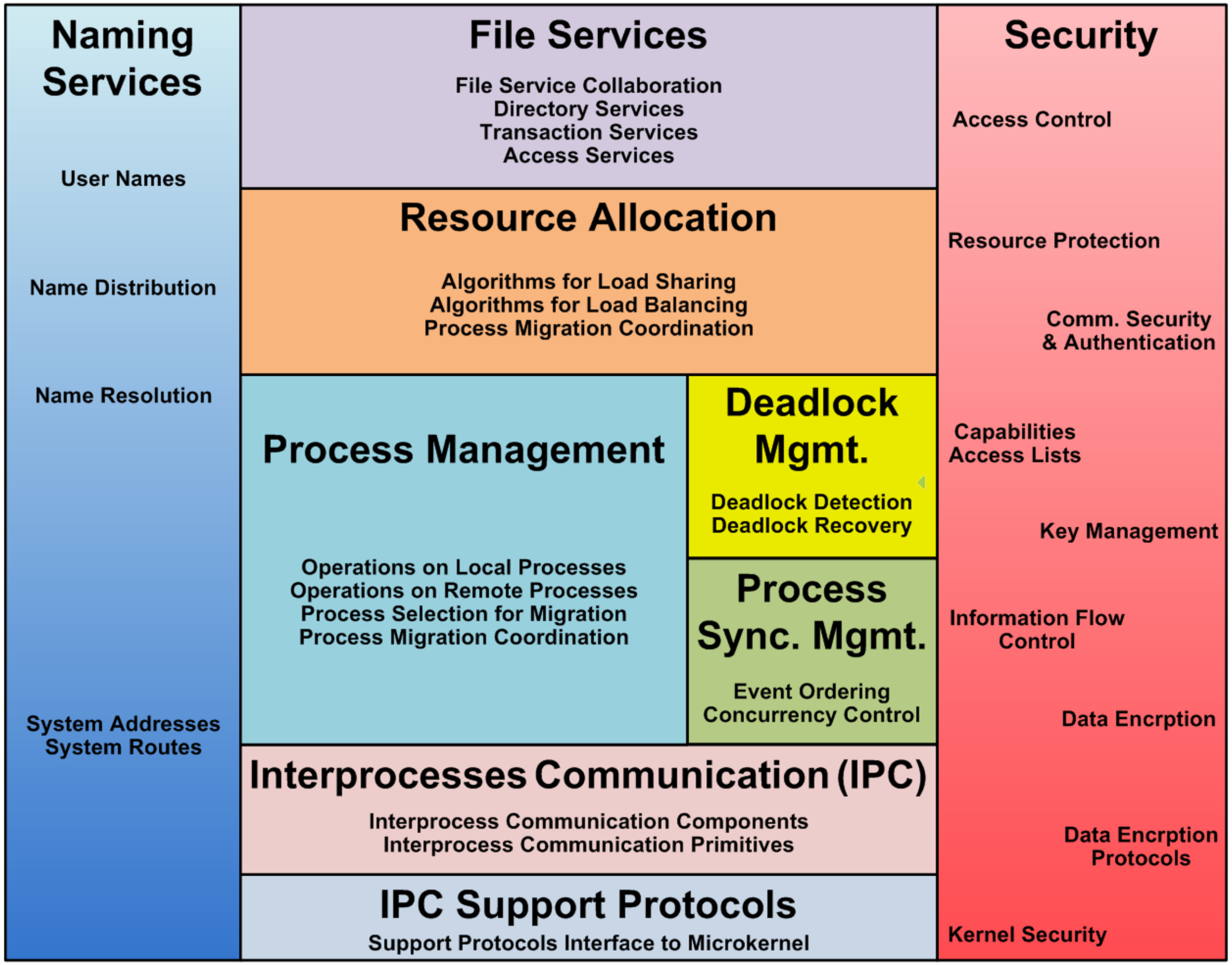
- ✓ OS can be of two types as follows:
  - ✓ General Purpose Operating Systems (GPOS)
  - ✓ Real Time Operating Systems (RTOS)

- ✓ Where and all the OS can run?
  - PC
  - Server
  - Embedded device

- ✓ OS characteristics:

OS type	PC	Server	Embedded
Size	↑	↑	↓
User Interface	↑	↓	↑
Security	↑	↑	↔
Reliability	↔	↑	↔
Cost	↔	↑	↓

OS Responsibilities



Co-ordinates:

- ✓ Security
- ✓ Communication
- ✓ Resource mgmt.

Service provider:

- ✓ File system
- ✓ Device access
- ✓ System utilities
- ✓ System services

OS – In action

- ✓ CPU loads boot program from ROM (e.g. BIOS in PC's)
- ✓ Boot program:
  - ✓ Examines/checks machine configuration (number of CPU's, how much memory, number & type of hardware devices, etc.)
  - ✓ Builds a configuration structure describing the hardware
  - ✓ Loads the operating system, and gives it the configuration structure

OS – In action

After basic processes have started:

- ✓ The OS runs user programs, if available
- ✓ Otherwise enters the idle loop

In the idle loop:

- ✓ OS executes an infinite loop (UNIX)
- ✓ OS performs some system management & profiling
- ✓ OS halts the processor and enter in low-power mode (notebooks)

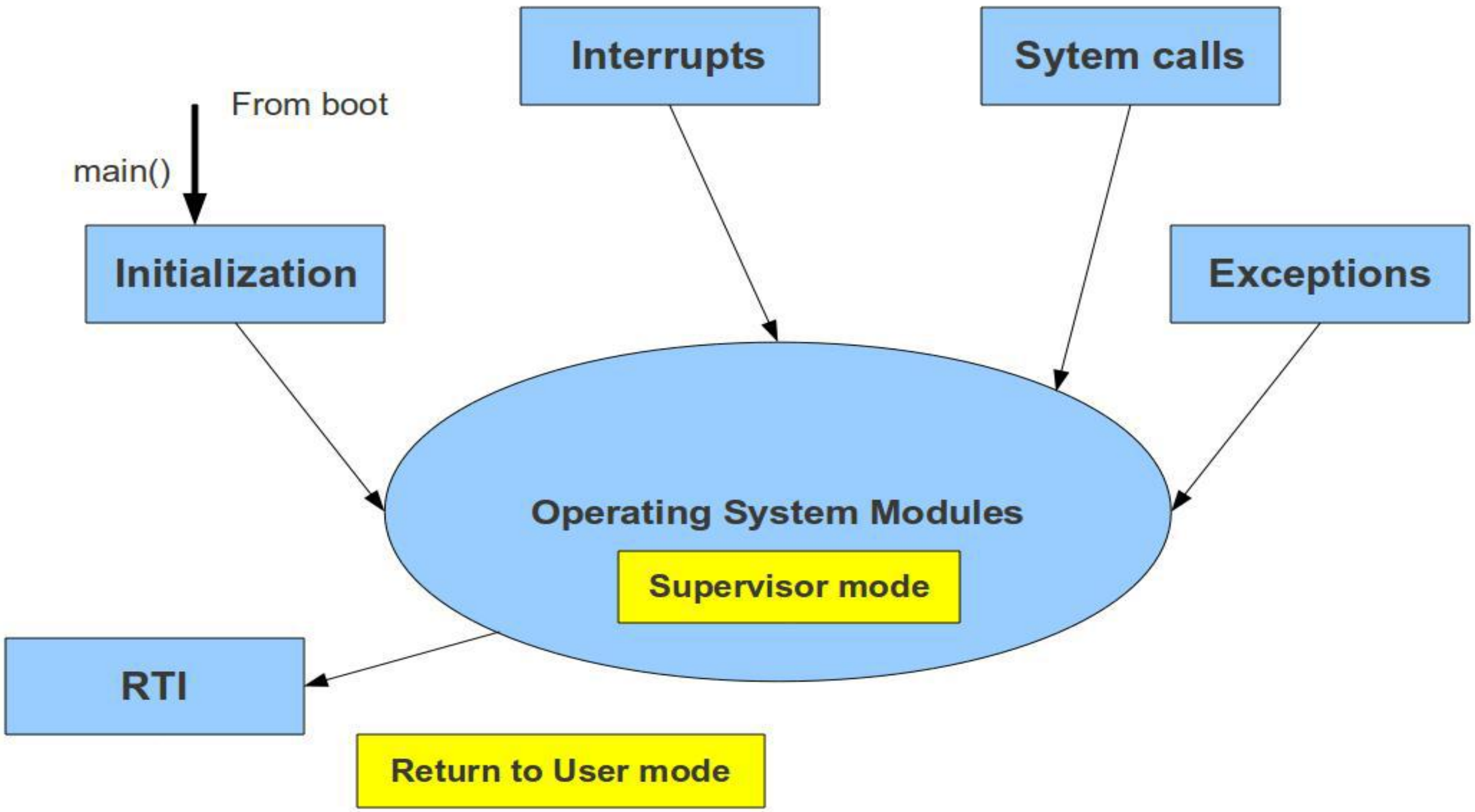
OS wakes up on:

- ✓ Interrupts from hardware devices
- ✓ Exceptions from user programs
- ✓ System calls from user programs

Two ***modes*** of execution:

- ✓ **User mode**: Less privilèges
- ✓ **Supervisor mode**: Unrestricted access to everything (OS)

Control flow in OS



Control flow - Interrupts

- ✓ Hardware calls the OS at a pre-specified location
- ✓ OS saves state of the user program
- ✓ OS identifies the device and cause of interrupt
- ✓ Responds to the interrupt
- ✓ OS restores state of the user program
- ✓ Execute an RTI instruction to return to the user program
- ✓ User program continues exactly at the same point it was interrupted.

Key Fact: None of this is visible to the user program

Control flow - Exceptions

- ✓ Hardware calls the OS at a pre-specified location
- ✓ OS identifies the cause of the exception (divide by 0)
- ✓ If user program has exception handling specified, then OS adjust the user program state so that it calls its handler
- ✓ Execute an RTI instruction to return to the user program
- ✓ If user program did not have a specified handler, then OS kills it and runs some other user program, as available

Key Fact: Effects of exceptions are visible to user programs and cause abnormal execution flow

Control flow – System calls

- ✓ Hardware calls the OS at a pre-specified location
- ✓ User program executes a trap instruction (system call)
- ✓ Hardware calls the OS at a pre-specified location
- ✓ OS identifies the required service and parameters (e.g. open(filename, O_RDONLY))
- ✓ OS executes the required service
- ✓ OS sets a register to contain the result of call
- ✓ Execute an RTI instruction to return to the user program
- ✓ User program receives the result and continues

Key Fact: To the user program, it appears as a function call executed under program control

Summary

- ✓ An OS is just a program:
  - ✓ It has a main() function, which gets called only once (during boot)
  - ✓ Like any program, it consumes resources (such as memory), can do silly things (like generating an exception), etc.
- ✓ But it is a very strange program:
  - ✓ It is “entered” from different locations in response to external events
  - ✓ It does not have a single thread of control, it can be invoked simultaneously by two different events (e.g. system call & an interrupt)
  - ✓ It is not supposed to terminate
  - ✓ It can execute any instruction in the machine

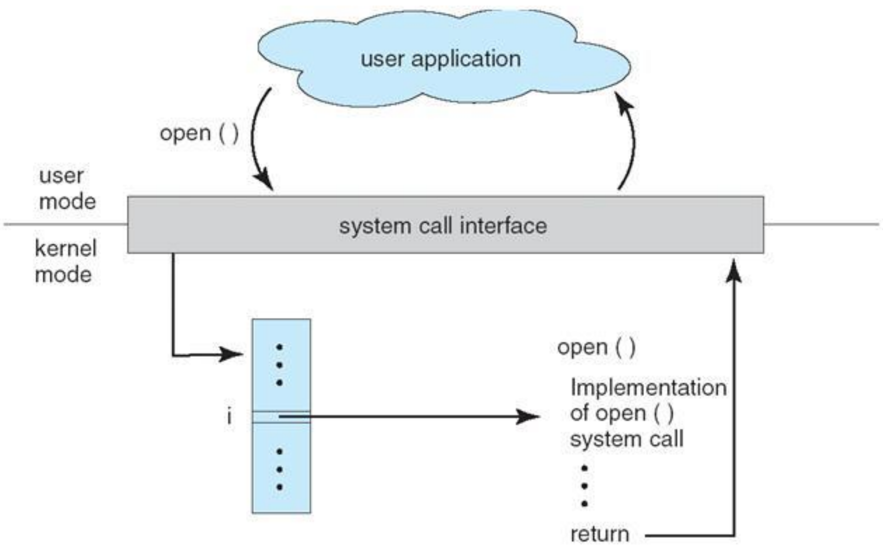
What is a system call?

A set of interfaces to interact with hardware devices such as the CPU, disks, and printers.

Advantages:

- ✓ Freeing users from studying low-level programming
- ✓ It greatly increases system security
- ✓ These interfaces make programs more portable

System call - Usage



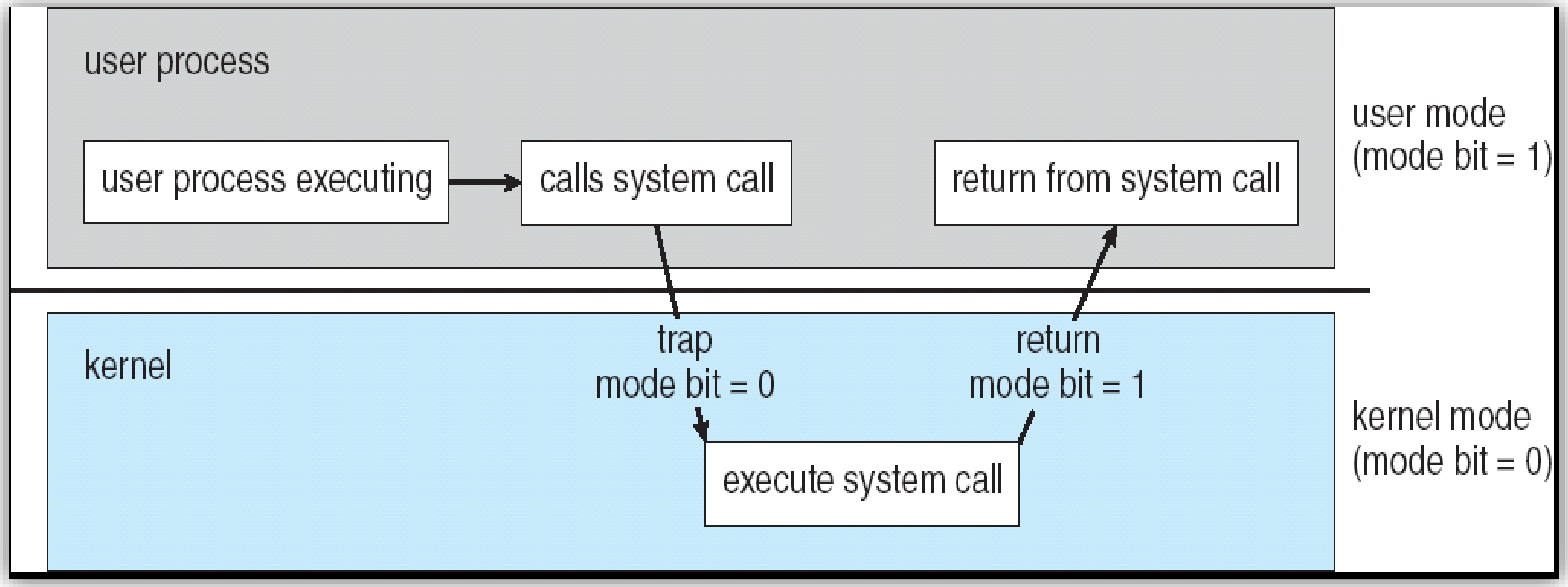
A set of interfaces to interact with hardware devices such as the CPU, disks, and printers.

Advantages:

- ✓ Freeing users from studying low-level programming
- ✓ It greatly increases system security
- ✓ These interfaces make programs more portable

For a OS programmer, calling a system call is no different from a normal function call. But the way system call is executed is way different.

Calling sequence



Logically the system call and regular interrupt follow the same flow of steps. The source (I/O device v/s user program) is very different for both of them. Since system call is generated by user program they are called as ‘Soft interrupts’ or ‘traps’

What is a task?

What is a Task?
- ✓ A task is an individual execution unit of an application.

A task:
- ✓ Starts with some parameters or inputs.
- ✓ It does a specific job assigned to it.
- ✓ It either runs forever in a loop or terminates.
- ✓ It communicates results to other tasks of the application.

Types of Tasks:
- ✓ Pre-emptive
- ✓ Non-Pre-emptive
- ✓ Round Robin

Application to tasks

- ✓ Any application can be split into a number of tasks based only on parallel operations possible for the application:
- ✓ For example, a car can have different tasks for controls such as for the following:
  - Driving Gears
  - Windows
  - Temperature
  - Door-lock/Theft-lock-alarm
  - Brakes/Accelarator
  - Air-Bag, Accident/Emergency
  - Ignition, Fuel Tank/Brake Oil
- ✓ These parts can run independently. However, at times they will need to communicate with each other. Alternatively, there can be a “master” task supervising these tasks which it creates, manages and destroys.

What is a task?

What is a Task?

- ✓ A task is an individual execution unit of an application.

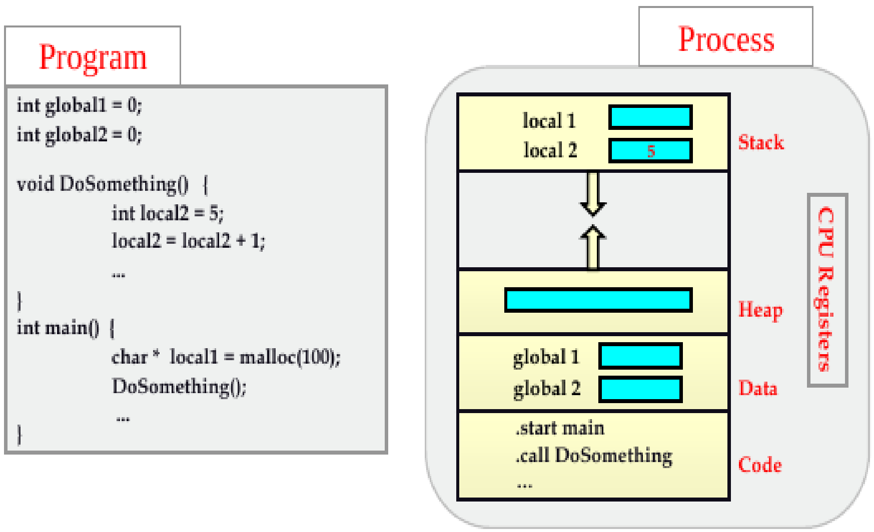
A task:

- ✓ Starts with some parameters or inputs.
- ✓ It does a specific job assigned to it.
- ✓ It either runs forever in a loop or terminates.
- ✓ It communicates results to other tasks of the application.

Types of Tasks:

- ✓ Pre-emptive
- ✓ Non-Pre-emptive
- ✓ Round Robin

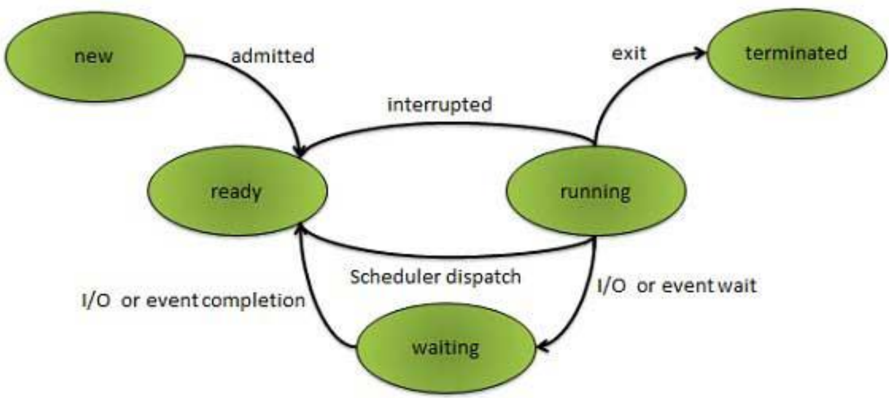
Task v/s Program



- ✓ A Task is a ***running*** Instance, whereas a Program is just the code which resides in primary or secondary memory.
- ✓ A program is a ***passive*** entity, such as file containing a list of instructions stored on a disk, whereas a task is a ***active*** entity, with a program counter specifying the next instruction to execute and a set of associated resources.
- ✓ A program gives rise to one or more active tasks when an executable file is loaded into main memory and execution of the tasks begin

EMERTXE

Task states



- ✓ A task goes through multiple states ever since it is created by the OS

- ✓ new: The task is being created.
- ✓ running: Instructions are being executed.
- ✓ waiting: The task is waiting for some event to occur.
- ✓ ready: The task is waiting to be assigned to a processor.
- ✓ terminated: The task has finished execution

Task descriptor

- ✓ To manage tasks:
  - OS kernel must have a clear picture of what each task is doing.
  - Task's priority
  - whether it is running on the CPU or blocked on some event
  - what address space has been assigned to it
  - which files it is allowed to address, and so on.
- ✓ This is the role of the task descriptor
- ✓ Usually the OS maintains a structure whose fields contain all the information related to a single task

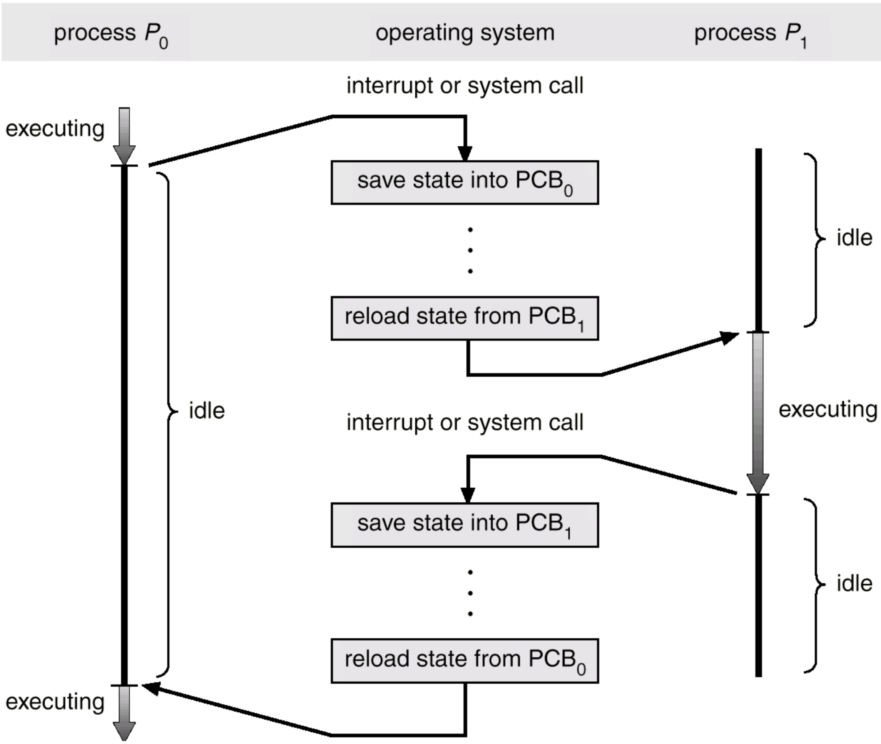
Task descriptor - Fields

Pointer	Task State
Task Number	
Program Counter	
Registers	
Memory Limits	
List of Open Files	
.	
.	
.	

✓ Information associated with each task:

- Task state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- I/O status information

Context switching



- ✓ Switching the CPU to another task requires saving the state of the old task and loading the saved state for the new task.
- ✓ The time wasted to switch from one task to another without any disturbance is called context switch or scheduling jitter.

Why Synchronization?

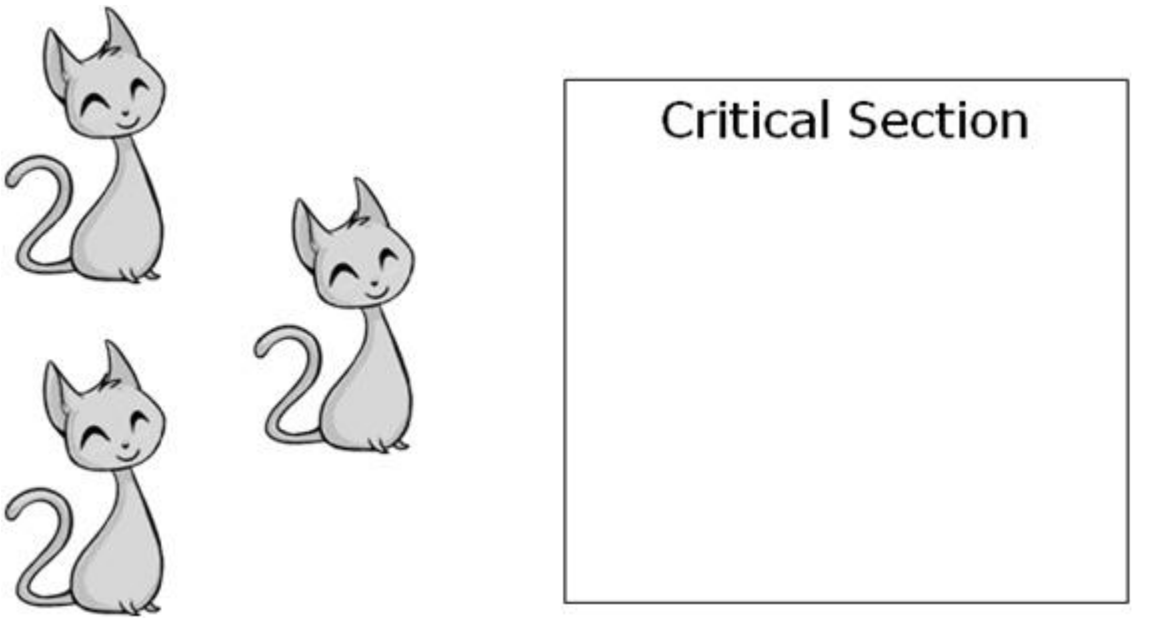
- ✓ When multiple tasks are running **simultaneously**:
  - ✓ either on a single processor, or on
  - ✓ a set of multiple processors
- ✓ They give an appearance that:
  - ✓ For each task, it is the only task in the system.
  - ✓ At a higher level, all these tasks are executing efficiently.
  - ✓ Tasks sometimes exchange information:
  - ✓ They are sometimes blocked for input or output (I/O).
- ✓ This **asynchronous** nature of scheduled tasks gives rise to **race conditions**

Race condition

- ✓ The ultimate cause of most bugs involving multiple-tasks is that the tasks are accessing the same (shared) data.
- ✓ If one task is only partway through updating a data structure when another task accesses the same data structure, it’s a problem.
- ✓ These bugs are called ***race conditions***; the tasks are racing one another to change the same data structure.
- ✓ Debugging a muti-tasking application is difficult because you cannot always easily reproduce the behavior that caused the problem.
- ✓ You might run the program once and have everything work fine; the next time you run it, it might crash.
- ✓ There’s no way to make the system schedule the tasks exactly the same way it did before.

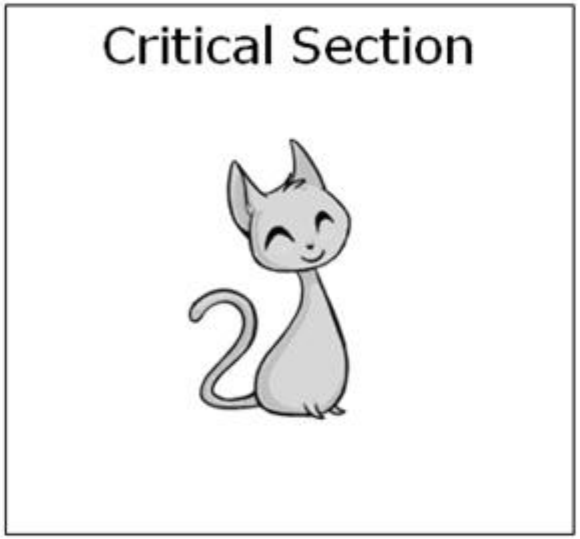
Critical section

- ✓ A piece of code that only one task can execute at a time.
- ✓ If multiple tasks try to enter a critical section, only one can run and the others will sleep.



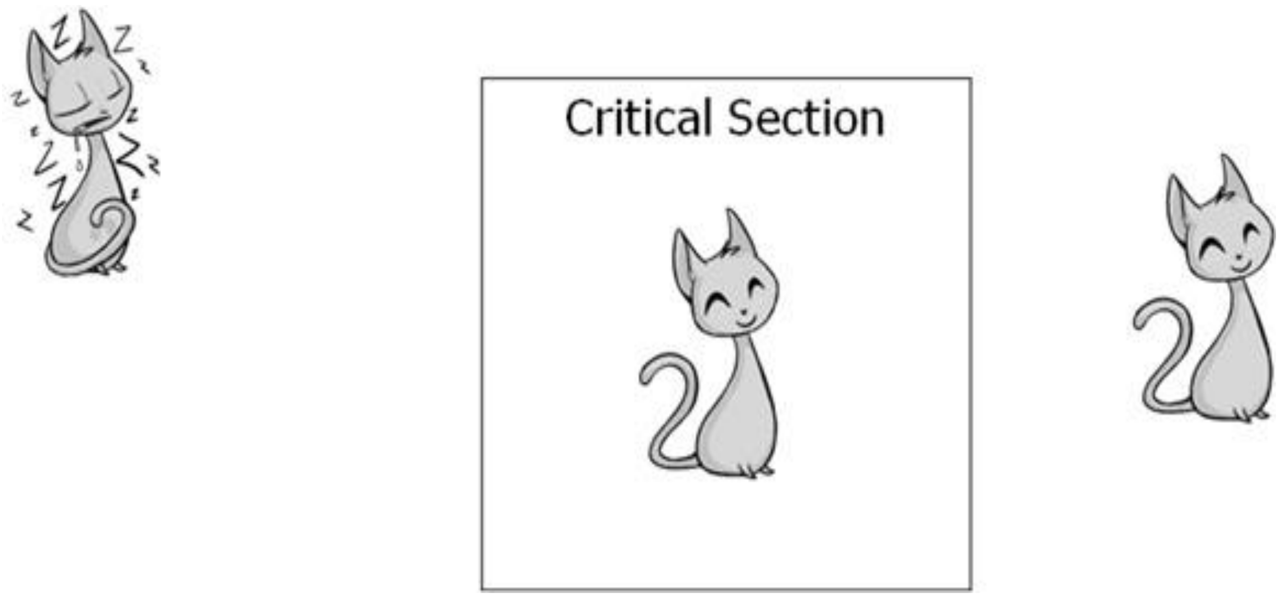
Critical section

- ✓ Only one task can enter the critical section; the other two have to sleep.
- ✓ When a task sleeps, its execution is paused and the OS will run some other task.



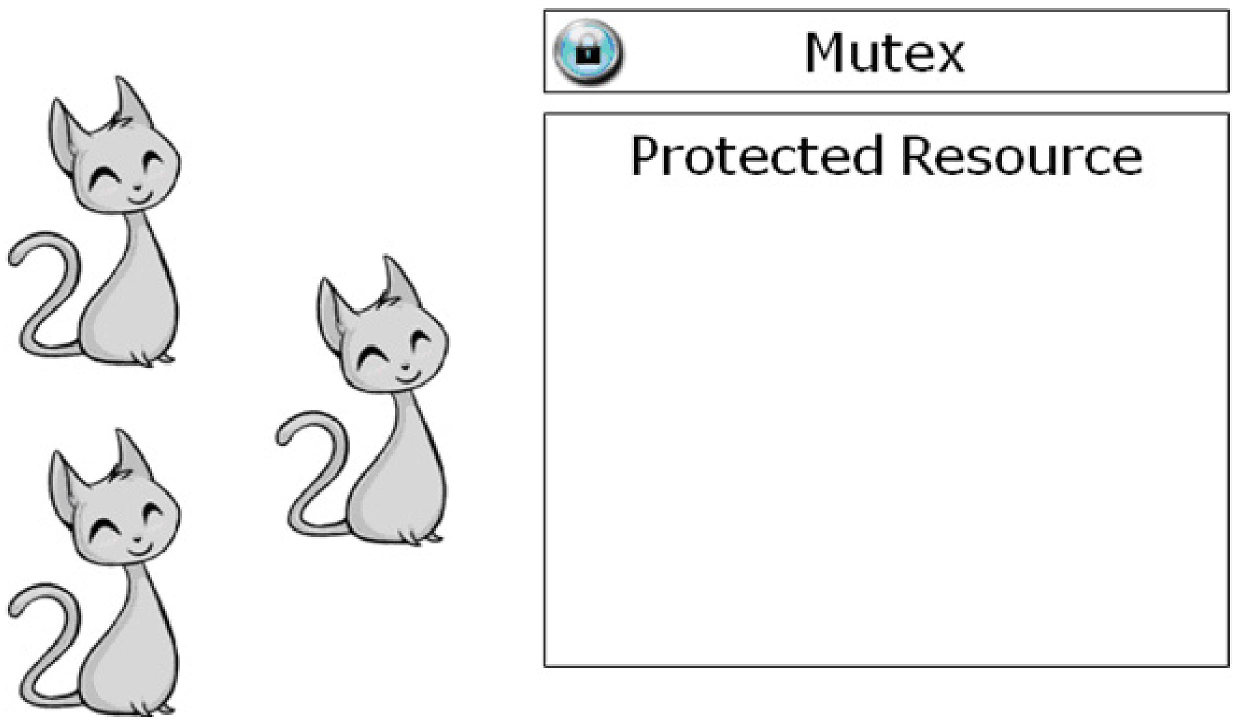
Critical section

- ✓ Once the thread in the critical section exits, another thread is woken up and allowed to enter the critical section.
- ✓ It is important to keep the code inside a critical section as small as possible



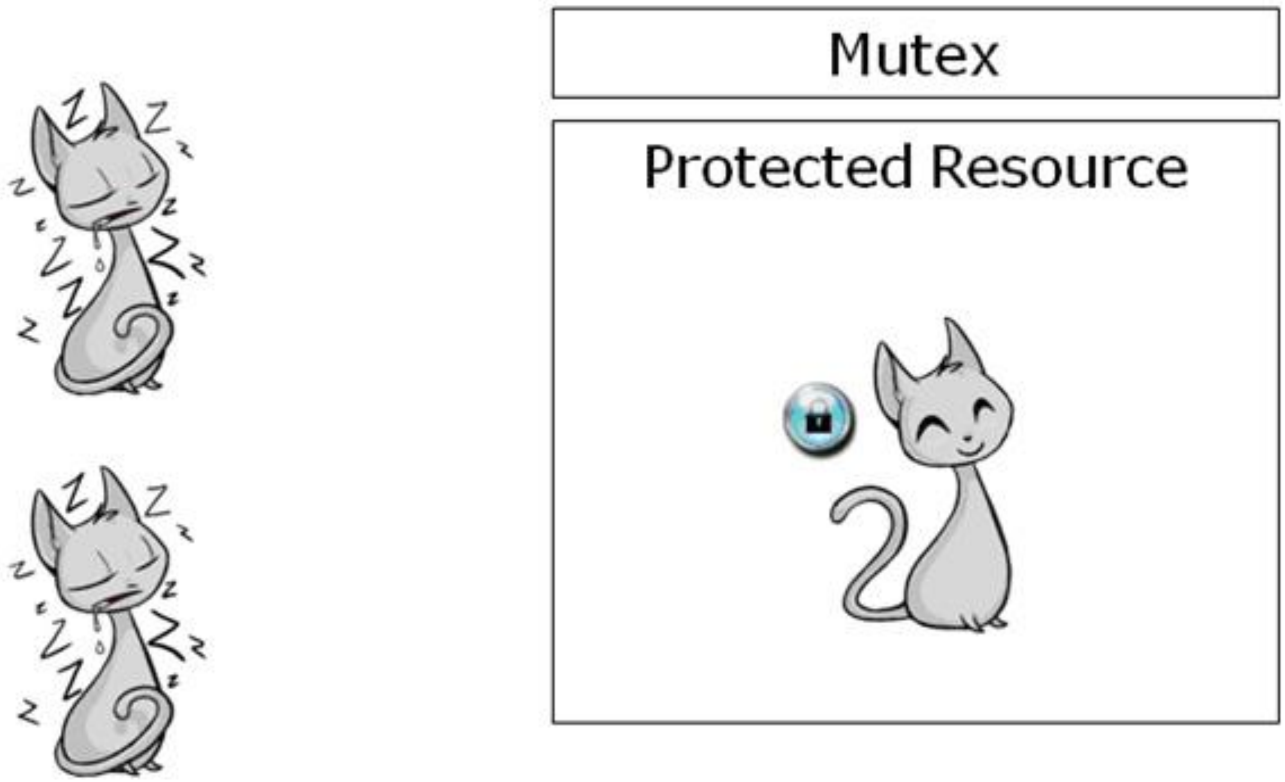
Mutual Exclusion

- ✓ A mutex works like a critical section.
- ✓ You can think of a mutex as a token that must be grabbed before execution can continue.



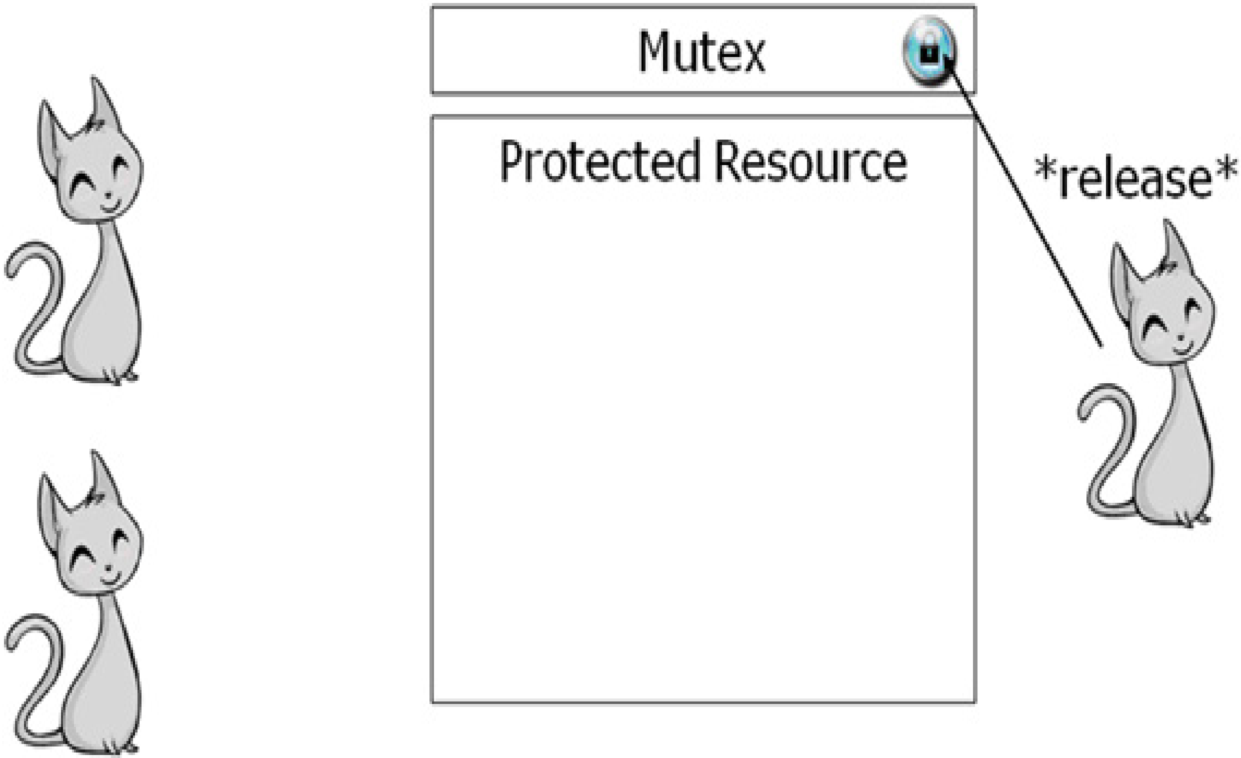
Mutual Exclusion

✓ During the time that a task holds the mutex, all other tasks waiting on the mutex sleep.



Mutual Exclusion

- ✓ Once a task has finished using the shared resource, it releases the mutex. Another task can then wake up and grab the mutex.



Locking & Blocking

- ✓ A task may attempt to lock a mutex by calling a ***lock*** method on it.
- ✓ If the mutex was unlocked, it becomes locked and the function returns immediately.
- ✓ If the mutex was locked by another task, the *locking function* ***blocks*** execution and returns only eventually when the mutex is ***unlocked*** by the other task.
- ✓ *More than one task* may be blocked on a locked mutex at one time.
- ✓ When the mutex is unlocked, *only one* of the blocked tasks is unblocked and allowed to lock the mutex; the other tasks stay blocked.

Deadlocks

- ✓ Mutexes provide a mechanism for allowing one task to block the execution of another.
- ✓ This opens up the possibility of a new class of bugs, called deadlocks.
- ✓ A deadlock occurs when one or more tasks are stuck waiting for something that never will occur.

Semaphores

- ✓ A semaphore is a counter that can be used to synchronize multiple tasks.
- ✓ As with a mutex, OS guarantees that checking or modifying the value of a semaphore can be done safely, without creating a race condition.
- ✓ Each semaphore has a counter value, which is a non-negative integer.

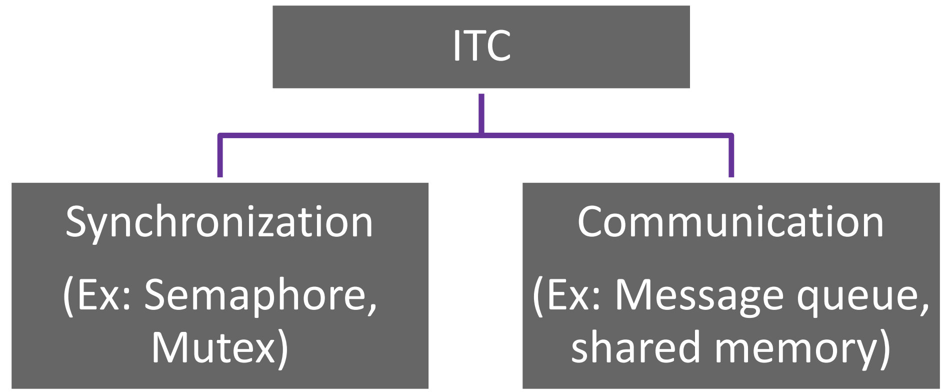
Semaphore - Operations

- ✓ A ***wait*** operation
- ✓ decrements the value of the semaphore by 1.
- ✓ If the value is already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other task).
- ✓ When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns.

- ✓ A ***post*** operation
- ✓ increments the value of the semaphore by 1.
- ✓ If the semaphore was previously zero and other tasks are blocked in a wait operation on that semaphore, only one of those tasks is unblocked and its wait operation completes (which brings the semaphore's value back to zero).

Why ITC?

- ✓ ITC allows tasks to communicate and synchronize their actions without sharing the same address space
    - Data Transfer
    - Sharing Data
    - Event notification
    - Resource Sharing and Synchronization



- ✓ In Synchronization, there is always a resource in contention which needs to be handled properly failing which will result in a race condition or deadlock

- ✓ In communication, it is about message or information exchange between two tasks offering various types depending on the need

ITC mechanisms

- ✓ Mechanisms used for communication:
  - Message Passing:
  - Mailboxes
  - Message Queues
  - Sockets
  - Pipes
  - Shared Memory

- ✓ Synchronization:
  - Mutex
  - Semaphore
  - Monitors
  - Event Notification/Signals

Let us focus on some of these mechanisms and build better understanding

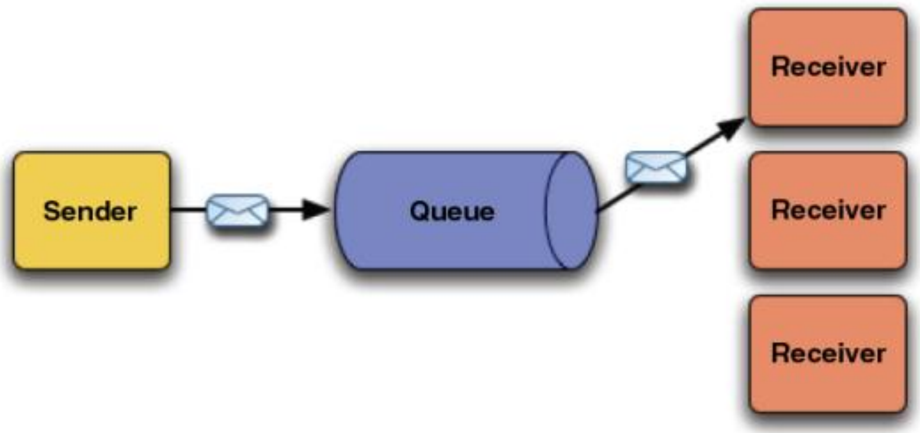
Message passing

- ✓ In a Message system there are no shared variables.
- ✓ Two operations for fixed or variable sized message:
  - send(message)
  - receive(message)

- ✓ If tasks P and Q wish to communicate, they need to establish a communication link exchange messages via send and receive

- ✓ Implementation of communication link
  - physical (e.g., memory, network etc.)
  - logical (e.g., syntax and semantics, abstractions)

Message passing

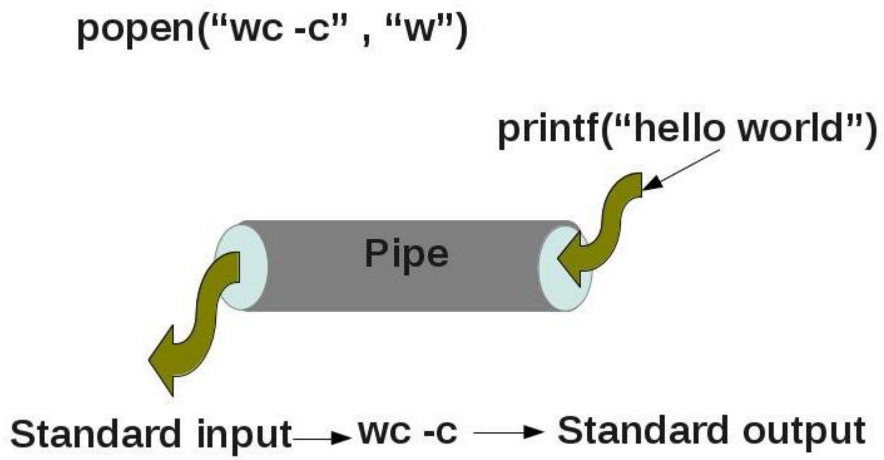


- ✓ Message queues pass message in both directions thereby making it as a ‘bi-directional’ mechanism of communication
- ✓ In a multi-processor or multi-tasking system this comes handy for ITC
- ✓ Before tasks starts communicating, they need to create appropriate queues to establish the connection
- ✓ Can be related with protocol request-response mechanism

Pipes

- ✓ A pipe is a communication device that permits unidirectional communication.
- ✓ Data written to the “write end” of the pipe is read back from the “read end.”
- ✓ Pipes are serial devices; the data is always read from the pipe in the same order it was written.
- ✓ A pipe’s data capacity is limited. If the writer task writes faster than the reader task consumes the data, and if the pipe cannot store more data, the writer task blocks until more capacity becomes available.
- ✓ If the reader tries to read but no data is available, it blocks until data becomes available. Thus, the pipe automatically synchronizes the two tasks.

Pipes

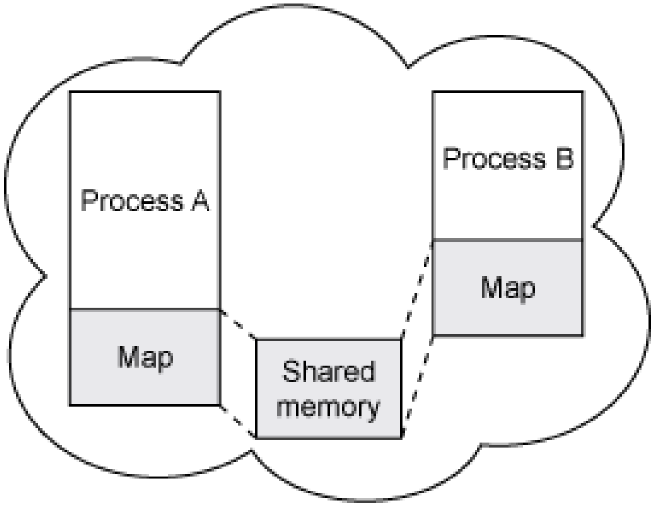


- ✓ Pipes are used in implementation where output of one process to be input of another, not vice-versa
- ✓ Some of the popular Linux utilities implement pipes for command handling
- ✓ Several powerful functions can be in a single statement using Pipes
- ✓ Streams of processes can be redirected to user specified locations using Pipes

Shared memory

- ✓ Shared memory allows two or more tasks to access the same memory.
- ✓ When one task changes the memory, all the other tasks see the modification.
- ✓ Shared memory is the fastest form of Inter-Task communication because all tasks share the same piece of memory.
- ✓ It also avoids copying data unnecessarily.

Shared memory



- ✓ To use a shared memory segment, one task must ***allocate*** the segment.
- ✓ Then each task desiring to access the segment must ***attach*** the segment.
- ✓ A task can make use of the shared memory using the attached location (pointer). It must use the memory in conjunction with synchronization methods using mutex and semaphore.
- ✓ After finishing its use of the segment, each task ***detaches*** the segment.
- ✓ At some point, one task must ***deallocate*** the segment.

Quick re-cap….

- ✓ Operating system is a program that runs on a super loop
- ✓ OS has come critical components – Scheduler, Task, Memory, System call interface, File systems etc…
- ✓ All of these components are very much part of Embedded and Real-time systems
- ✓ However some of the parameters need to be tuned/changed in order to meet the needs of these systems
- ✓ Fundamentals remain same, only specifics change
- ✓ Real time & Embedded systems – Coupling v/s De-coupling
- ✓ Engineers need to understand these differences in order to be effective during the product development

Real Time systems

✓ Characteristics:
- Capable of guaranteeing timing requirements of the processes under its control
- Fast – low latency
- Predictable – able to determine task's completion time with certainty
- Both time-critical and non time-critical tasks to coexist

✓ Types:
- Hard real time system
  - Guarantees that real-time tasks be completed within their required deadlines.
  - Requires formal verification/guarantees of being to always meet its hard deadlines (except for fatal errors).
  - Examples: air traffic control , vehicle subsystems control, medical systems.
- Soft real time system
  - Provides priority of real-time tasks over non real-time tasks.
  - Also known as "best effort" systems. Example – multimedia streaming, computer games

Classification

- ✓ The type of system can be either a:
  - Uni-processor,
  - Micro processor
  - Distributed System.
- ✓ There are two different execution models:
  - In a preemptive model of execution a task may be interrupted (preempted) during its execution and another task run in its place.
  - In a non-preemptive model of execution after a task that starts executing no other task may execute until this task concludes or yields the CPU.

Characteristics

- ✓ RTOS for Embedded systems are:
  - ✓ Single purpose
  - ✓ Small size
  - ✓ Inexpensively mass-produced
  - ✓ Specific timing requirements
- ✓ Features missing in an RTOS:
  - Support for variety of peripheral devices.
  - Protection and Security mechanisms
  - Multiple Users
  - Multiple Modes
  - Dynamic Allocation of memory

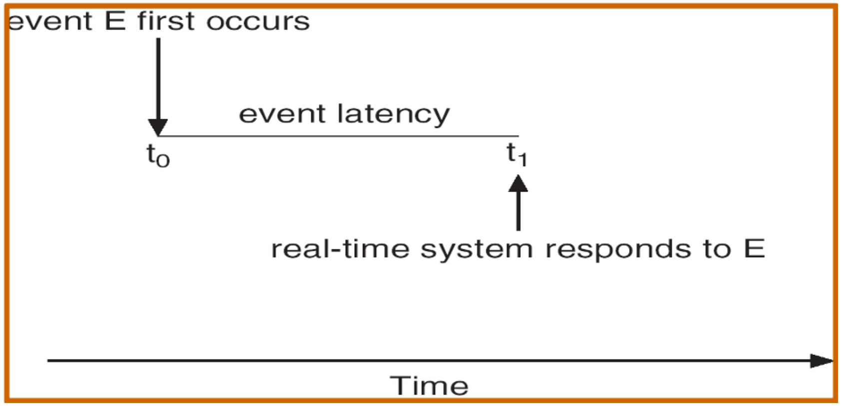
Why so?

- ✓ Real-time systems are typically single-purpose.
- ✓ Real-time systems often do not require interfacing with a user.
- ✓ Features found in a desktop PC require more substantial hardware than what is typically available in a real-time system.
- ✓ High overhead required for protected memory and for switching modes.
- ✓ Memory paging increases context switch time.
- ✓ Creates fragmentation adding to timing unpredictability.

Scheduling in RTOS

- ✓ Handling Priority & Scheduling
- ✓ Scheduling algorithms
- ✓ Meeting deadlines
- ✓ Avoiding conflicts like deadlocks
- ✓ Real-Time Systems Development

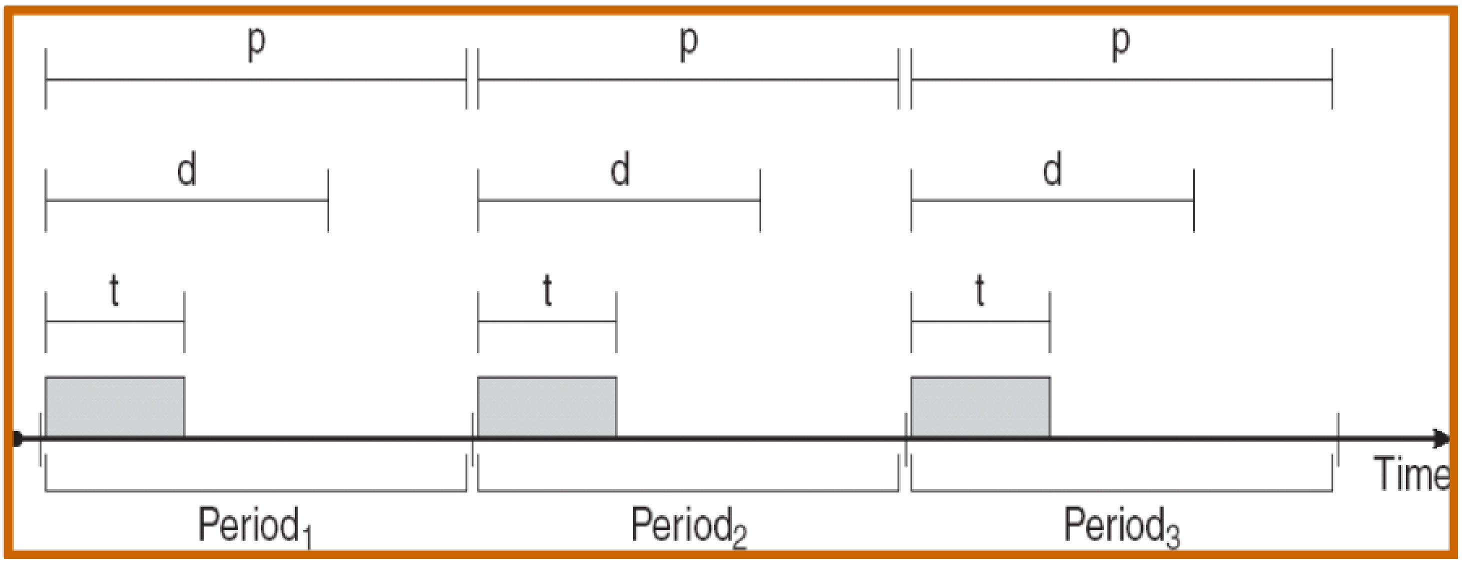
Event latency



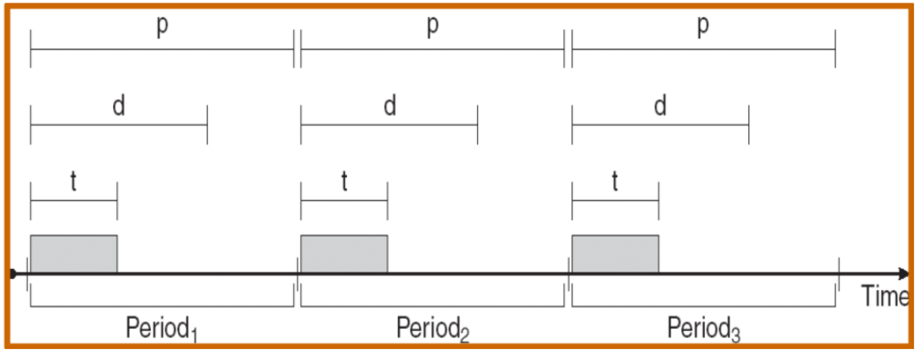
- ✓ The Event latency is nothing but the "amount of time from when and event occurs to when it is serviced"
- ✓ In a Real-Time system, this latency should always be within a particular limit
- ✓ This is known as responding in "real-time"
- ✓ Scheduler part of the OS to be configured to respond in this manner

Real Time CPU scheduling

- ✓ Periodic processes require the CPU at specified intervals (periods)
  - p is the duration of the period
  - d is the deadline by when the process must be serviced
  - t is the processing time
- ✓ By ensuring CPU responds within given time interval, real-time response can be guaranteed



Real Time CPU scheduling



- ✓ Periodic processes require the CPU at specified intervals (periods)
  - p is the duration of the period
  - d is the deadline by when the process must be serviced
  - t is the processing time
- ✓ By ensuring CPU responds within given time interval, real-time response can be guaranteed
  - Priority based scheduling
  - Earliest deadline first scheduling

RTOS - Issues

- ✓ Interrupt Latency should be very small
  - Kernel has to respond to real time events
  - Interrupts should be disabled for minimum possible time
- ✓ For embedded applications Kernel Size should be small
  - Should fit in ROM
- ✓ Sophisticated features can be removed
  - No Virtual Memory
  - No Protection

RTOS - Characteristics

- ✓ Reliability
- ✓ Predictability
- ✓ Performance
- ✓ Compactness
- ✓ Scalability
- ✓ User control over OS Policies
- ✓ Responsiveness
  - Fast task switch
  - Fast interrupt response

Examples of RTOS

- ✓ LynxOS
- ✓ OSE
- ✓ QNX
- ✓ VxWorks
- ✓ Windows CE
- ✓ RT Linux