Operating Systems and RTOS

Prof. P.H.Dave
DDU, Nadiad;
-
Prof. H.B.Dave
eInfochips, Ahmedabad.

March, 2015

Operating Environment

- Processes and Threads
- Mutex and Semaphores
- Scheduling in Embedded Systems
- Real Time systems
- Determinism and Deterministic system
- Methods of Scheduling
- Event-triggered and Time-triggered Scheduling
- Hard and Soft Real-Time systems
- Interrupt latency and its effects
- Operating Environment for an Embedded System

Variety of Computer Systems

- **General Purpose Computer Systems**:
  - **Office and School**: Desk-Tops and Laptops used for office work and scientific, engineering and professional computations
  - **Development Systems**: computers used for developing computer software and/or hardware simulation
  - **Servers**: File, Data, Software and Platform servers
  - **Work-Stations**: used for engineering, structural and architectural designs, system simulations
- **Embedded Computer Systems**:
  - **Control Systems**: chemical plants, factories, aeroplanes, automobiles, robots

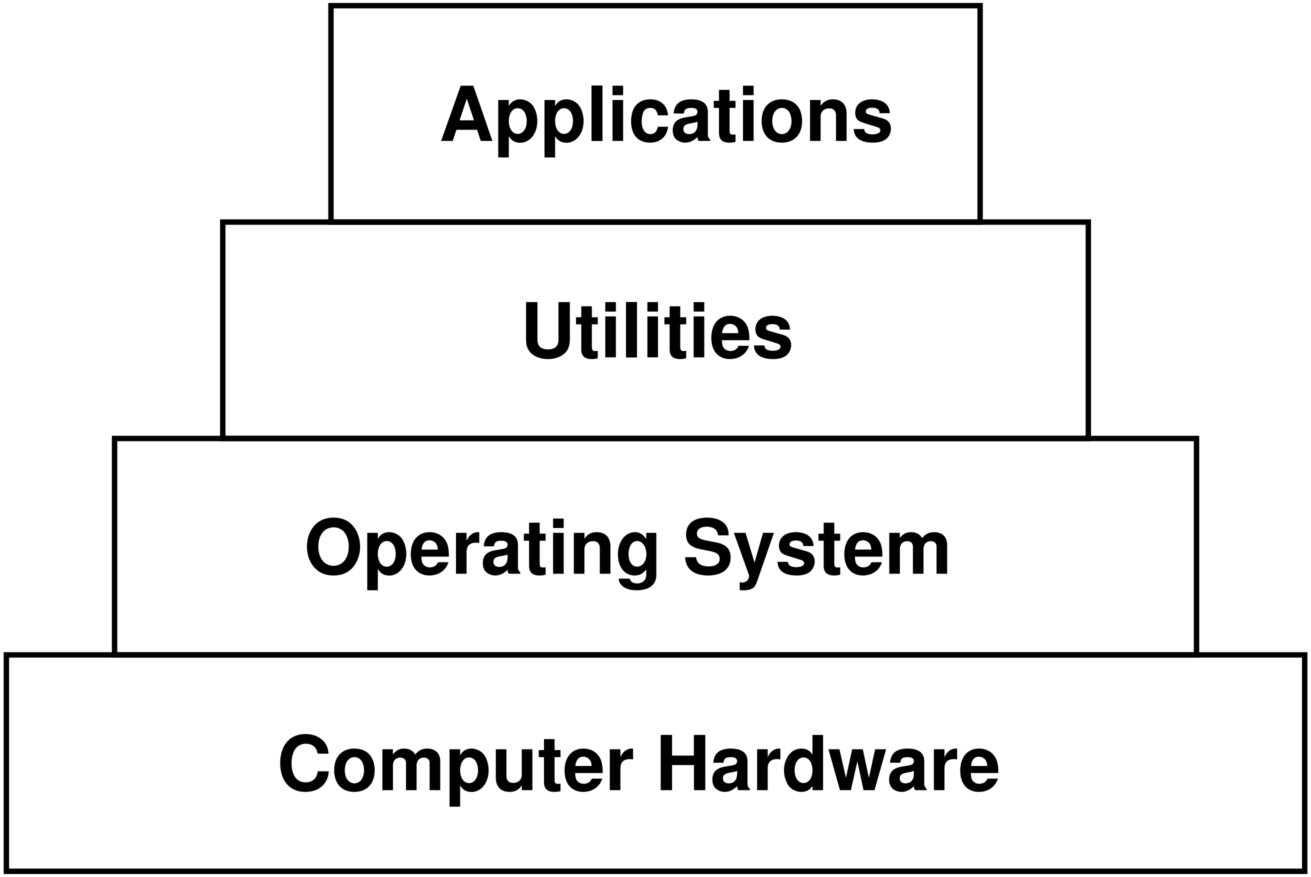
- **Application Oriented Systems**: Smart-phones, Tablets,
- **General Micro-controller Systems**: a large range of micro-controller based devices – toys, personal assistants of various types, wearable devices, home and entertainment gadgets, medical instruments and equipment, etc.

What is an Operating System (O/S)

- Ultimately it is the computer system hardware which executes various jobs
- All such jobs require certain functions to do common actions: reading an input device, outputting to an output device, organizing the information stored in the system so that it is easily accessible, decide which task to do at this moment, etc.
- A special set of programs is used to provide these functions in a working computer system
- It is called an *Operating System*, O/S, (considered to be a single word).
- Without such an O/S it would be extremely difficult to use a computer.

- On one side it is in direct communication with the hardware, including use of the CPU at a privileged level, memory management and I/O control
- The other side is in contact with various system utilities, commonly required by the prospective users

An O/S is positioned between the Hardware and user Software

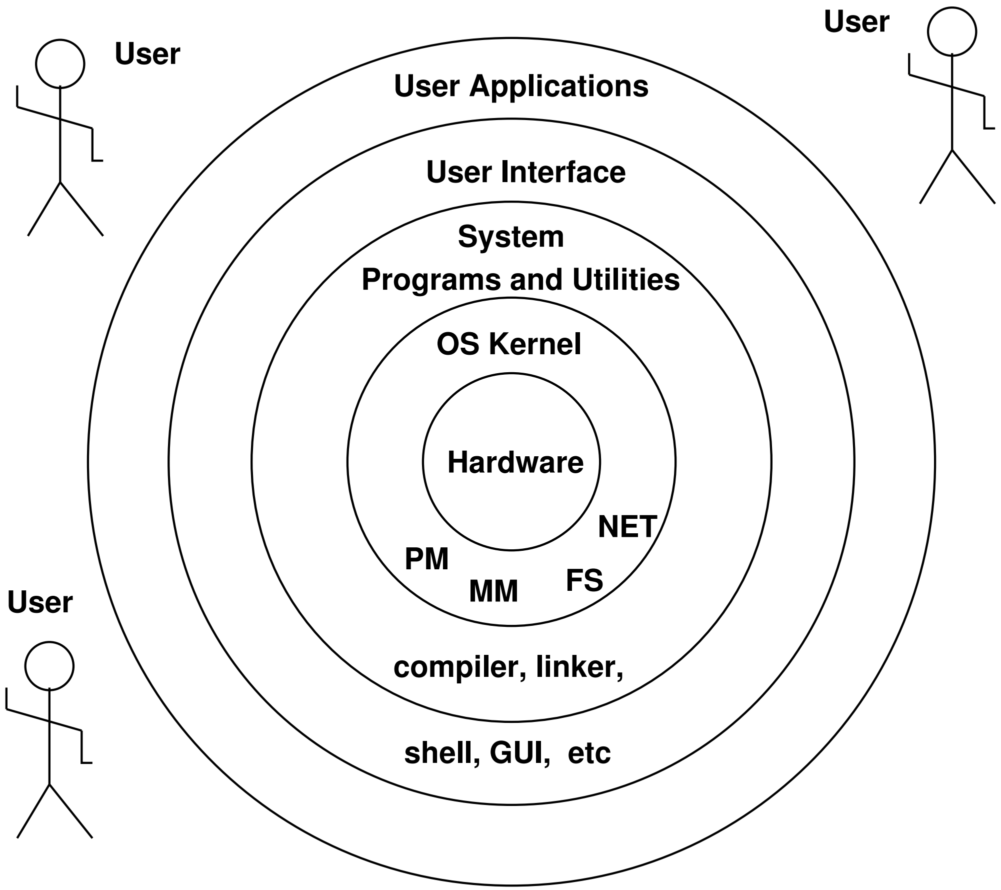


O/S- Several View-points

- **Interface between a user and Hardware**: a *user* says so. A general user will prepare his job using development utilities, store the resultant executable program in the system using File System, request execution of his program via Command Line Shell Shell, get the input and output done via I/O Control - all these facilities are provided by insulating the user from the tedious details of the hardware of the system.
- **A Resource Manager**: a System Manager says so. He is concerned with arranging for satisfactory service to maximum number of users within a given time period, at the same time maximizing the *Utilization Factor*s of all the components of the computer system - CPU, memory, I/O devices and stored data.
- **An Extended Machine**: an application developer says so. He views the O/S as consisting of layers of functions, each layer extending the capability of the underlying layers.

- **Set of Concurrent processes**: O/S own view of itself and an Operating System designer says so. In fact it is the self-view of the O/S itself.
- **A task scheduler**: an embedded system designer says so. As we shall subsequently discuss in detail below, the "Operating System" of most of the embedded systems consists of a task scheduler and a set of I/O Device Drivers.

O/S of a GP Computer as a layered enhancement of capabilities



Functions provided by the Kernel of a G.P. O/S

- **Processor and Process Management**: allocate enough processor time to each *process* and application so that they can run as efficiently as possible. At the same time maximize the utilization of the CPU (and other processors, if any.)
- **Memory Management**: ensure that each process has enough memory to execute, while also ensuring that one process does not encroach upon the memory allocated to another process.
- **Device management and File System**: manages the input to and output from the computer. Usually a File System is provided, to organize the stored information like data, source and executable programs, etc. On some O/S like Linux, Kernel contains a Virtual File System (VFS), as a kind of link to actual File Systems which are implemented as utilities.
- **Network Management**: the kernel of a good O/S would have a built-in component for communicating over a network.

Types of operating Systems

- **Single-user, Single task**: simplest O/S, only one user can effectively do one thing at a time. Microsoft MSDOS (a legacy O/S) was of this type. There, if you gave a PRINT command, for example, the computer does not do anything else except manage the printing operation.
- **Single-user, Multi-tasking**: let a single user have several programs in operation at the same time. Microsoft's Windows and Apple's MacOS are examples of this type of O/S. A user may be editing a document while the e-mail client may download mails.
- **Multi-user, Multi-tasking**: allows many different users to take advantage of the computer's resources simultaneously. The O/S must make sure that the requirements of the various users are balanced, and that each of the programs they are using has sufficient and isolated resources so that a problem with one user doesn't affect the entire community of users.

- **Multi-user, Time-Shared**: similar in its aim as Multi-user, Multi-tasking O/S, but sharing of the computer system resources is controlled purely by time-slicing, rather than events.
- **Real-Time**: (RTOS), a multitasking operating system that aims at executing real-time applications, which are required to achieve a deterministic behavior with respect to time. They often use specialized scheduling algorithms to meet this requirement. They are used to control machinery, scientific instruments, and industrial systems. In general, the user does not have much control over the functions performed by an RTOS.
- **Distributed operating System**: use multiple central processors, communicating with one another through various communication lines, abscence of a global common clock.

- **Network operating System**: runs on a server and allows shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks. UNIX, Linux, Mac OSX, are examples of this type of O/S.
- **Embedded**: designed to operate on small machines, with limited resources like memory, CPU speed, secondary storage, etc.

Processes or Task

A rough and working definition of a Process:

- An instance of a running program
- An application may have several processes working concurrently (``simultaneously'' in simple English)
- Runs and provides an environment for a program
- Consists of an address space and control point, i.e. the address of the next instruction to be executed
- It is the basic scheduling entity, only one process runs on the CPU at a time
- It contends for and owns various system resources like memory
- It requests system services, provided by the Kernel

- It has a life-cycle - *creation, execution, waiting, terminated*
- Only a running process can create a new process
- It has an exclusive address space, independent of other processes in the system

Threads

- A simpler scheduling entity compared to a Process
- It is sometimes called Light Weight Process (LWP)
- It runs strictly sequentially, in other words, it represents a single thread of control
- It has its own Program Counter (control point) and a stack
- It shares CPU in time-sliced manner with other threads
- Only on multiple CPU (multiprocessor or multi-core) can they really execute in parallel
- It can create child threads
- It blocks (waits or sleeps) on a system calls, i.e. calls for Kernel service

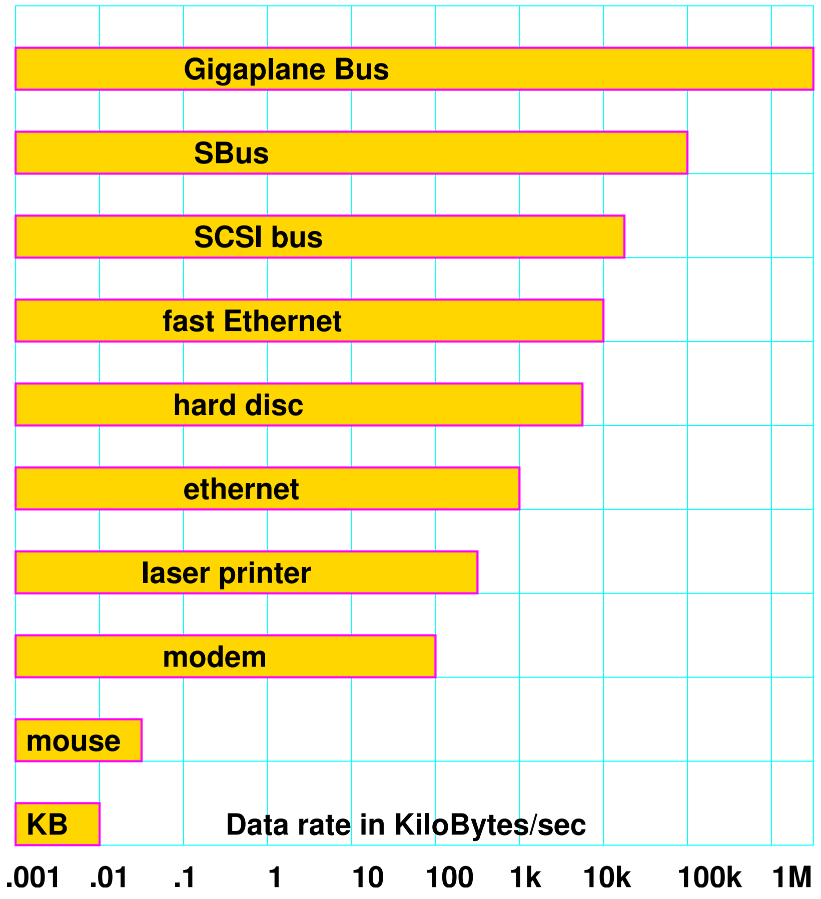
- While one thread is blocked, other in the same process can run
- It shares the address space with other threads in the *same process*
- It is a member of the set of several threads in a process

Processes and Threads - view point of Embedded Systems

- The address space of each Process are separate, but Threads share the address space of the containing process
- It is impossible for a Process to access memory of another Process
- One Process can communicate some information to another, only via the Inter-Process Communication (IPC) provided by the O/S
- The Process structure is designed for isolation and robustness
- All Threads share the address space of the containing Process
- It is very easy for one Thread to access memory of another within the same process

- This requires special care on part of the programmer to co-ordinate the working of Threads which use common resources
- It requires that thread synchronization be done carefully

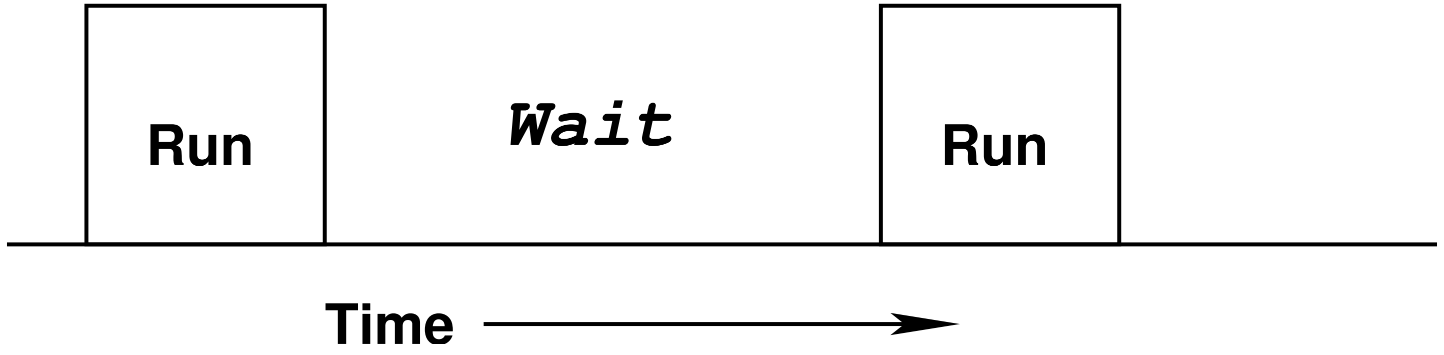
Data Rates of Some I/O devices and communication links



Why Multi-tasking or Multi-threading?

- Maximize the utilization of basic system resources like CPU time, memory space and critical I/O devices
- It increases the thru-put of a system for a given cost and power consumption
- If there is only a single task, which periodically does some data input, running on our machine, the activity of the CPU will be as shown next
- For most of the I/O devices, the speed at which data access takes place is much slower than the CPU speed
- In what follows, we use the word *task* to represent both a Process and a Tread, unless otherwise noted

The Time Chart of a CPU executing a single task only

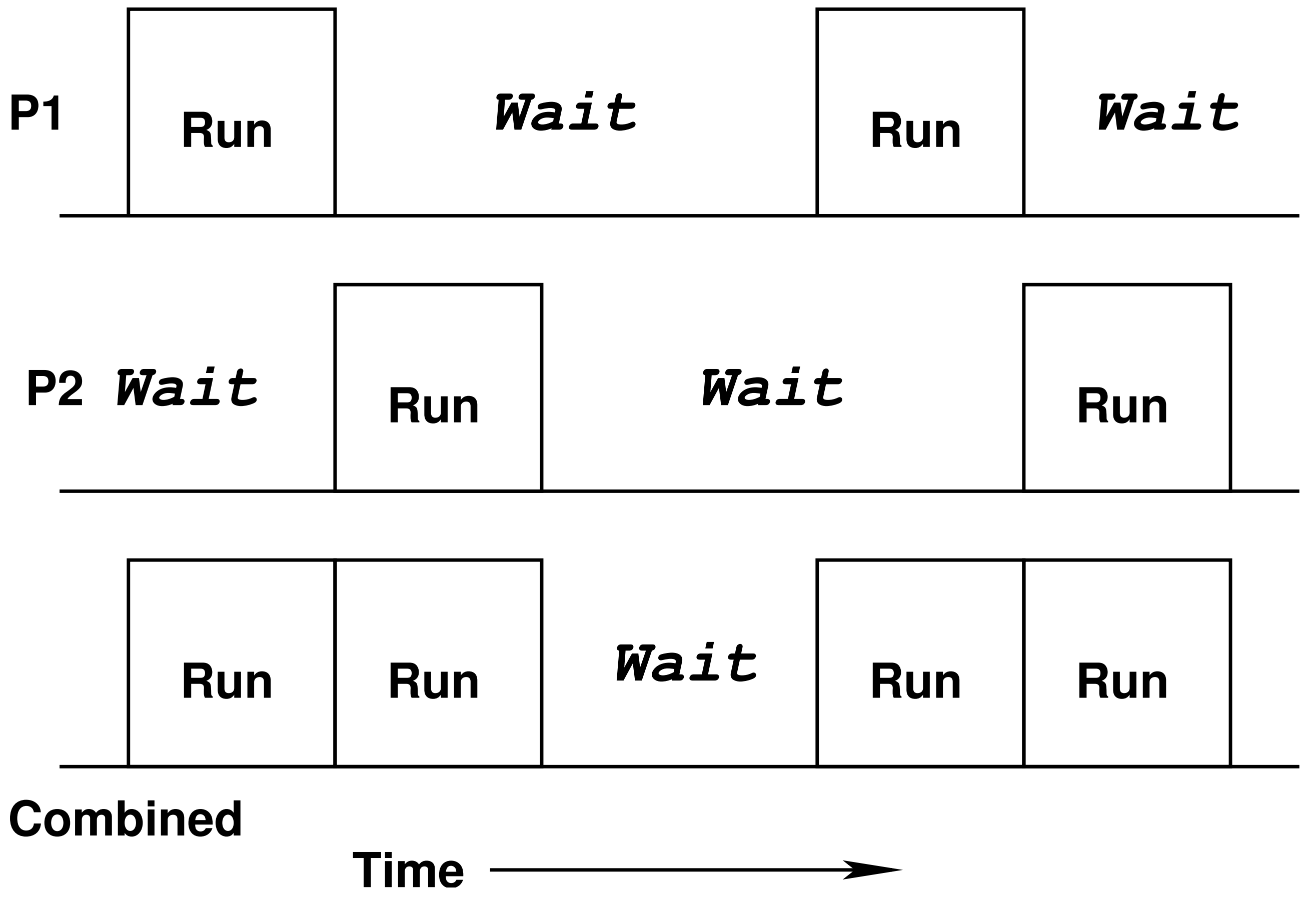


Multi-tasking

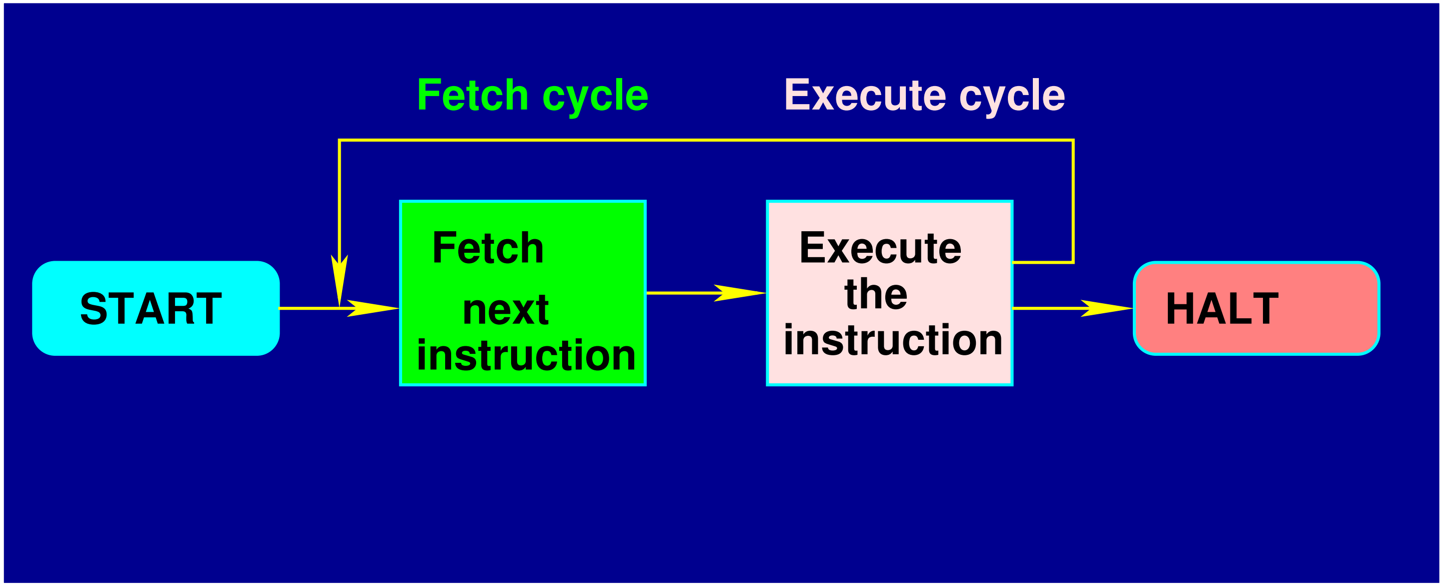
- Arrange matters such that when one task P1 is waiting for some I/O, CPU automatically starts running another task P2
- Now the system is able to almost double the computation work and the utilization of the CPU is doubled
- The idea can be extended to multiple tasks
- In order to utilize the CPU time when it is not executing either P1 or P2 ...
- Introduce a third task P3, called back-ground task
- P3 needs only a small amount of I/O operations, but has a lot of computation work to be done

- A function in the O/S Kernel, known as Scheduler decides which task to run next
- A companion function known as the Dispatcher actually does the job of switching the tasks

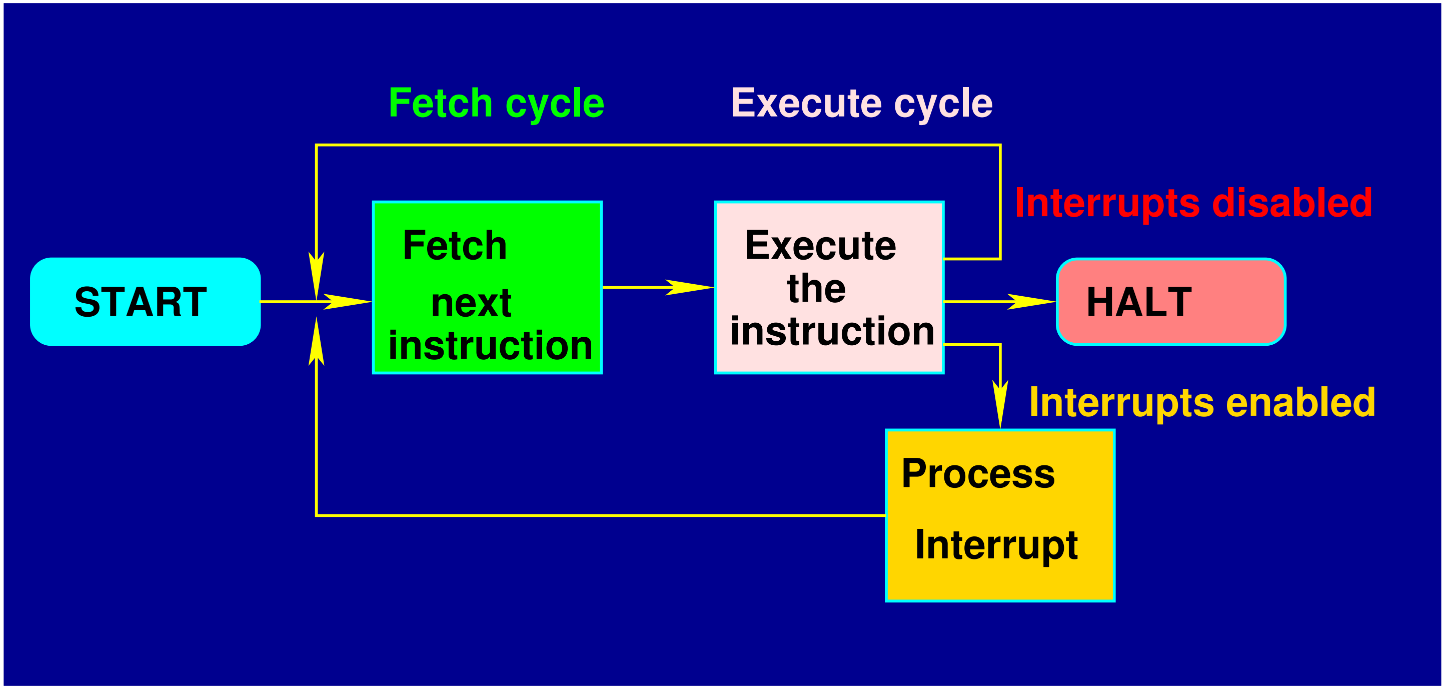
The Time Chart of a CPU executing two tasks



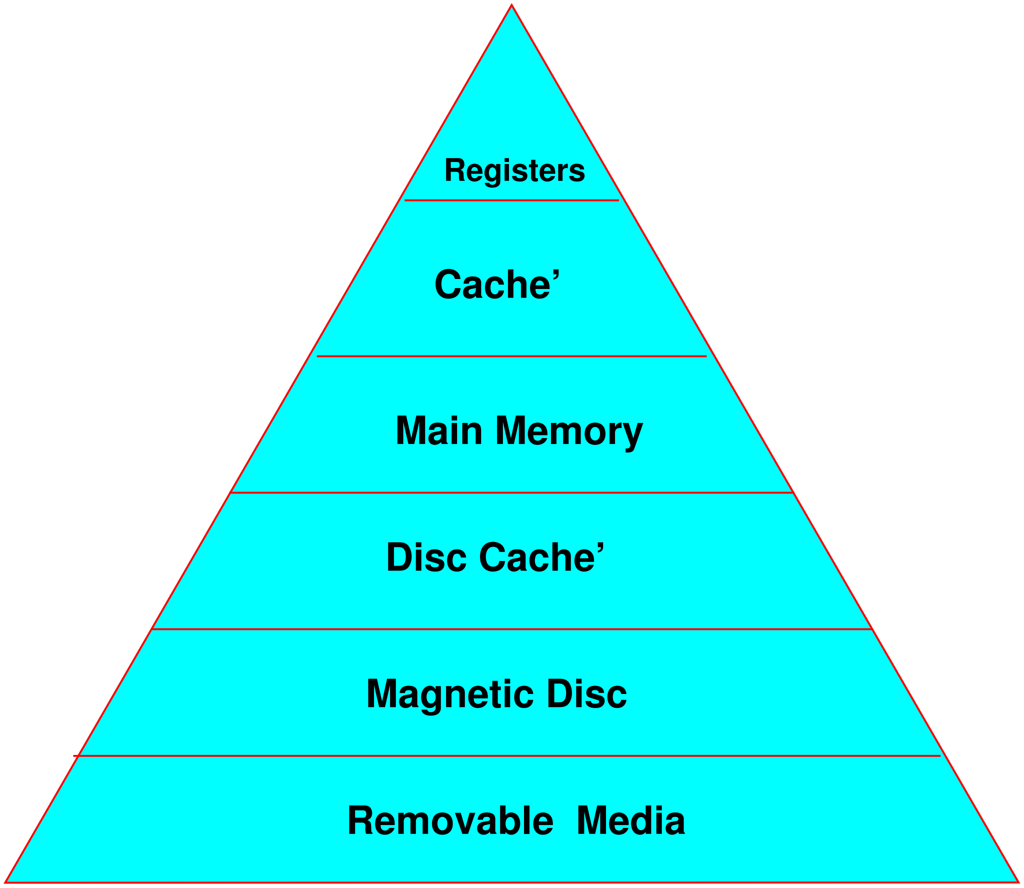
Execution rhythm: normal flow



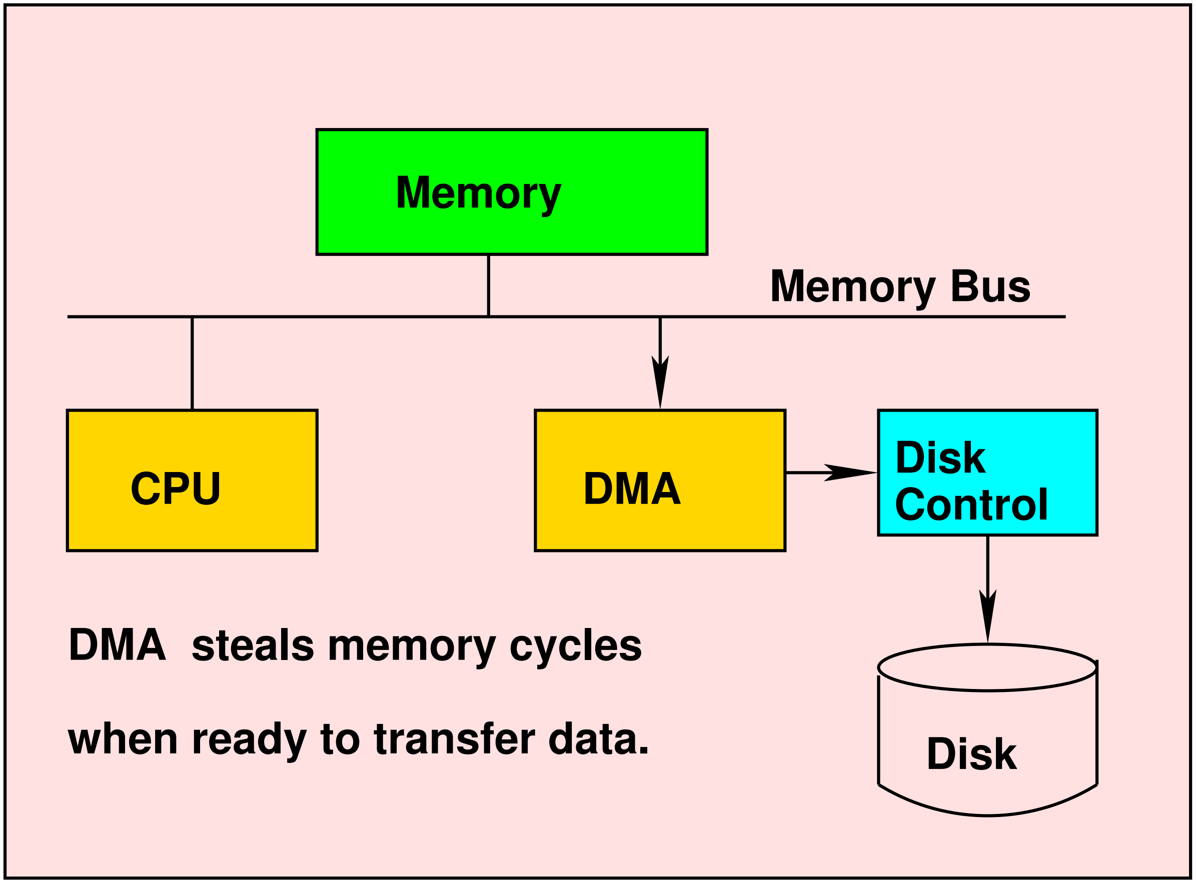
Execution rhythm: When an Interrupt occurs



Computer Memory Hierarchy



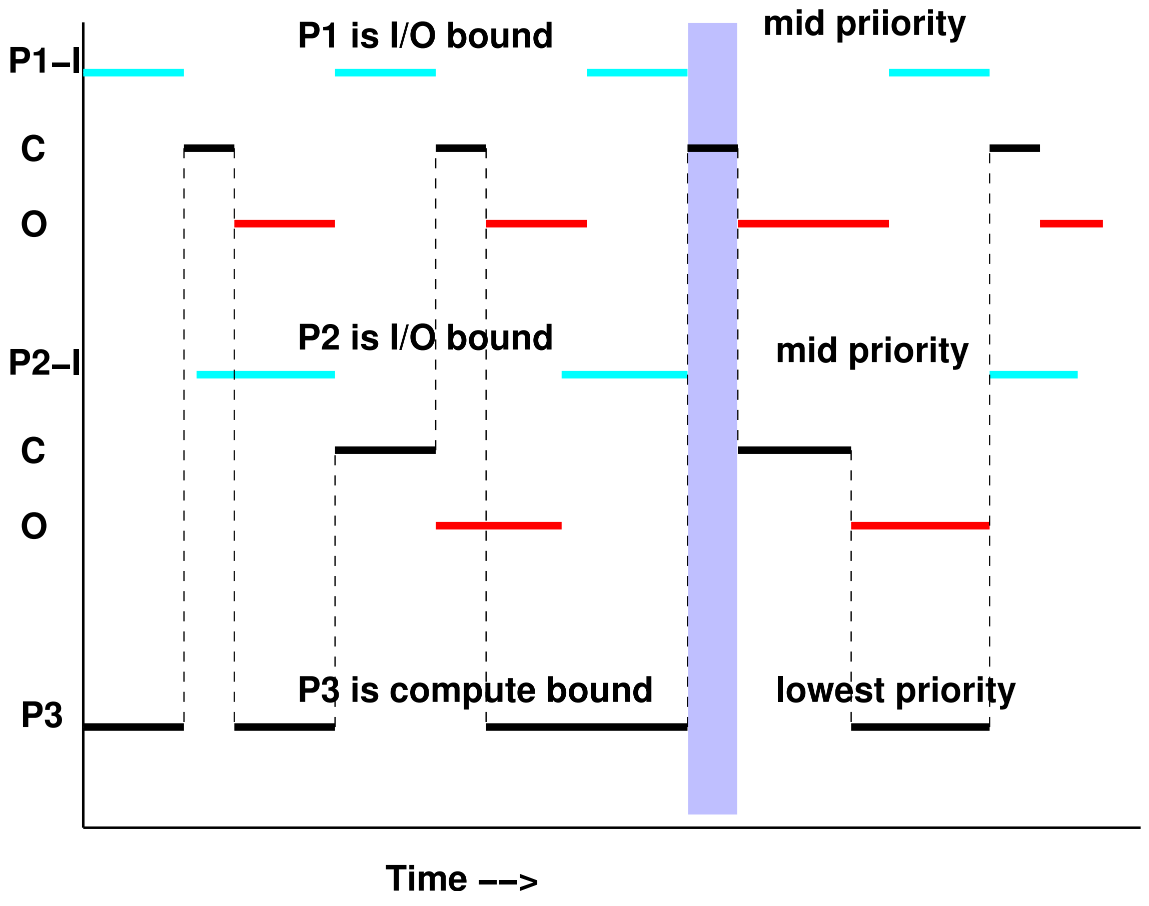
Direct memory Access (DMA)



DMA steals memory cycles

when ready to transfer data.

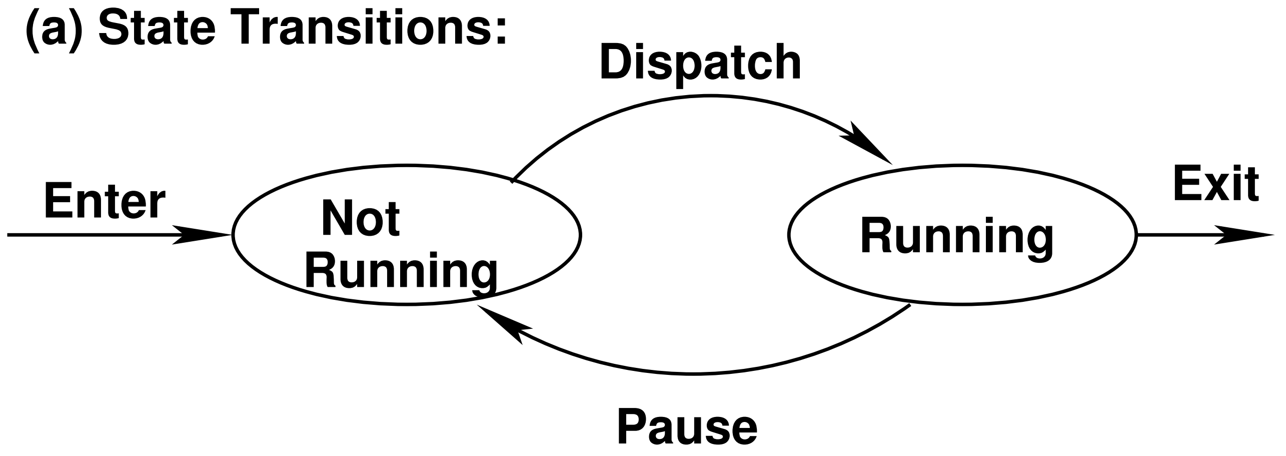
The Time Chart of a CPU executing several tasks

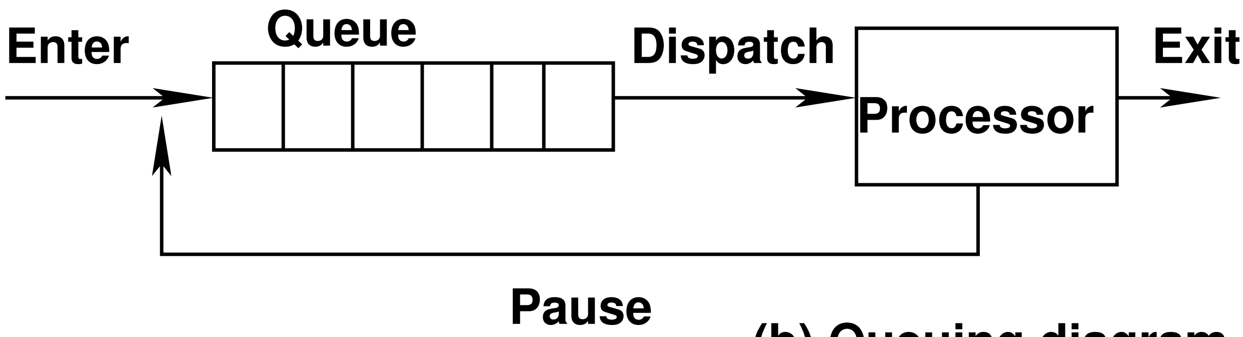


Scheduling

- Described above is called short-term scheduling, as it occurs at the finest level of time
- In a O/S of a general purpose computer, mid-term (concerned mainly with Demand-paging in Virtual memory system) and long-term (concerned with external user job queuing) are also going on
- They are not of much relevance in an embedded system
- Scheduling activity has two view-points - view-point of individual task and view-point of the system
- The CPU acts as a server which gives execution service to the tasks in a queue
- It is known as the Ready-queue, which contains tasks which are otherwise ready to run if CPU were available to them

Basic State Transitions of a Task



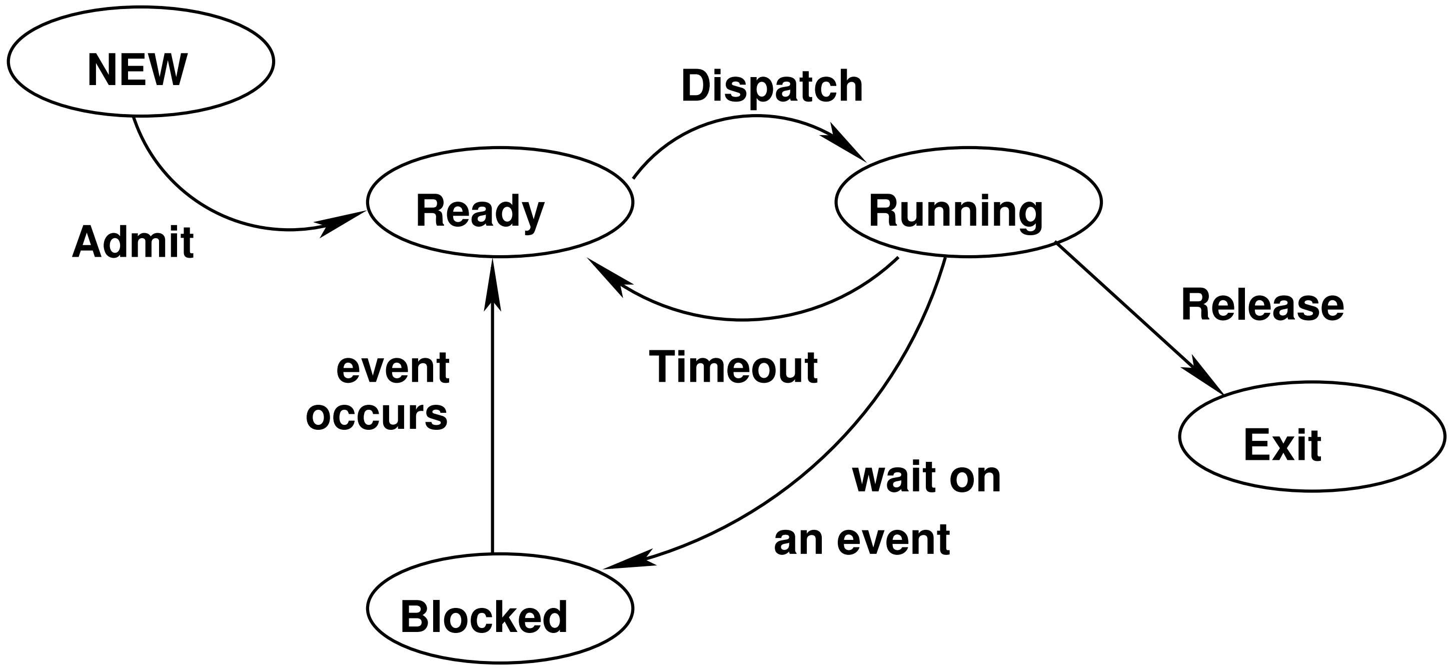


(b) Queuing diagram

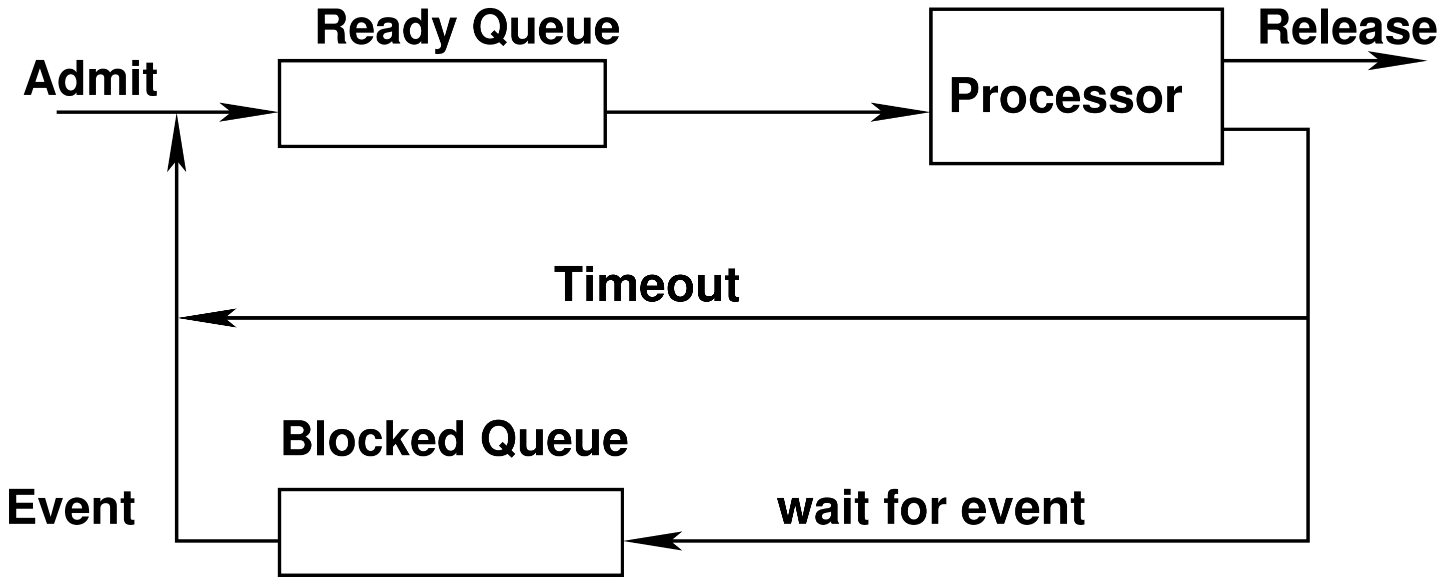
Working of simple Scheduling

- Why should the CPU suspend a task, say P1, and take up another task P2 from the Ready-queue? Two reasons:
- 1. The Scheduler has decided that the process has run out its allocated time-slice (happens in what is known as Round-Robin scheduling)
- 2. The process has requested a Kernel service, like some I/O operation (happens in Event-driven scheduling)
- The state-diagram of individual task and corresponding queuing diagram are ...

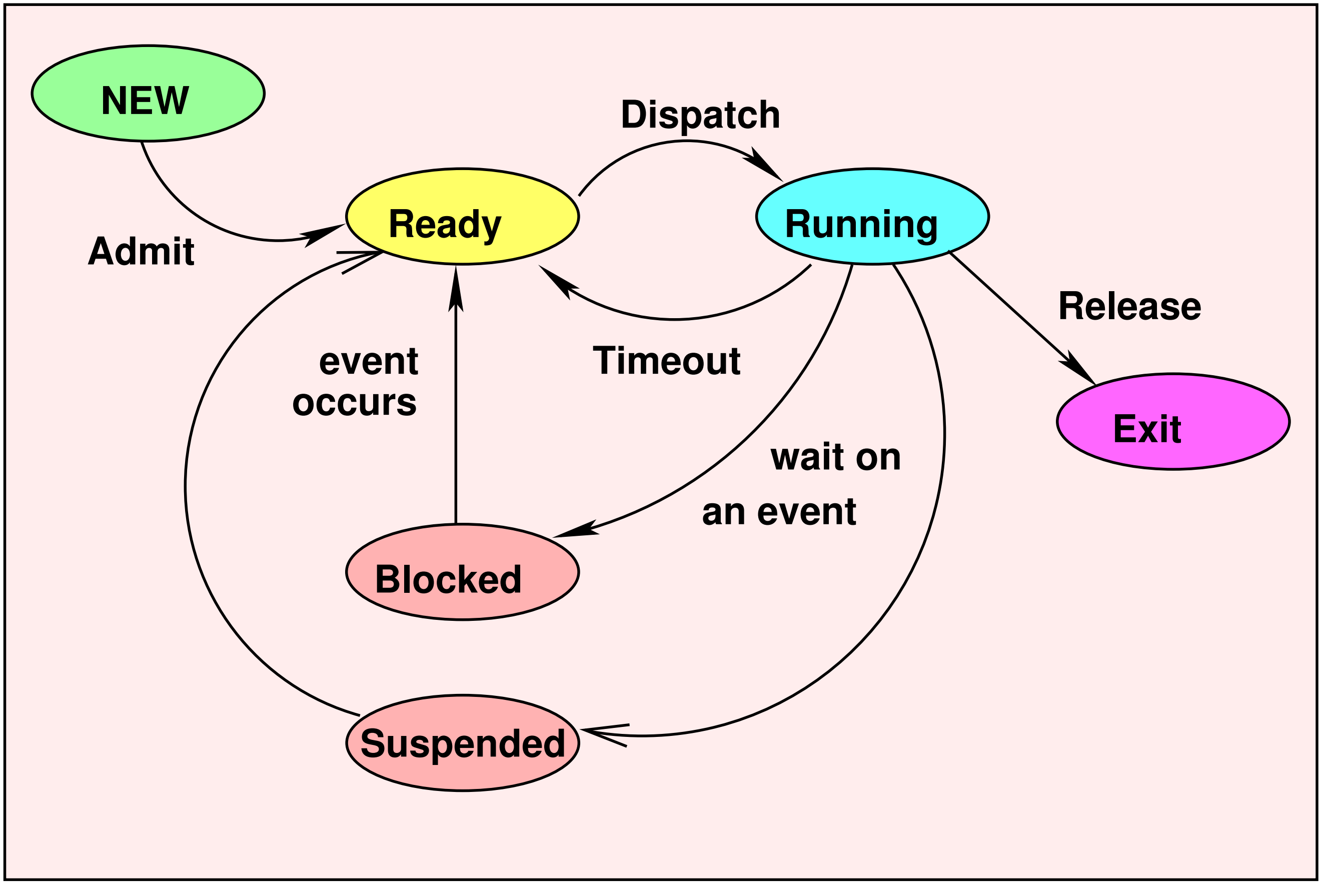
State Transitions of Task, including the Wait state



Queuing Diagram of a system, including the Wait queue



Six State Model of a Task



Race and Dead-lock

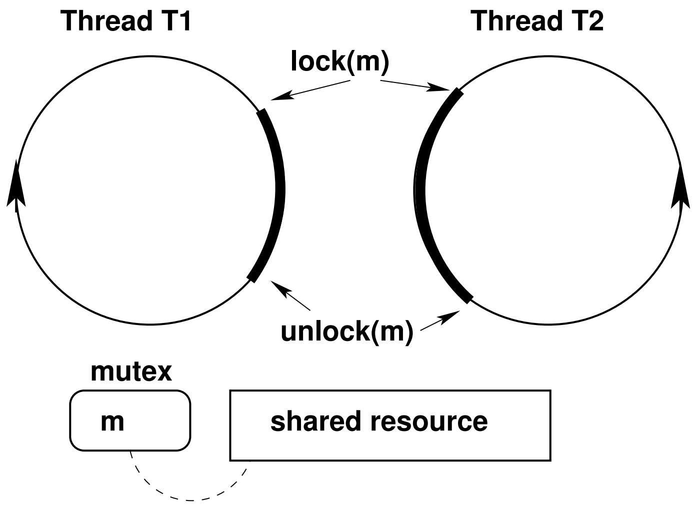
- Two severe problems can arise for shared data - Race and Dead-lock
- Race refers to non-deterministic result obtained when two or more threads read and/or write some shared data
- Dead-lock condition can arise, for example, if further progress in a thread P1 is determined by a check of value (say 0 or 1) of some shared variable x, and progress of another thread P2 depends upon similar check of another shared variable y
- If it is P1 which sets the value of y and P2 sets value of x, we have a potential Dead-lock, as both the threads may wait for the other to set the values of x and y

- Dead-lock condition is very dangerous in Embedded Systems, as many of such systems are unattended and only possible way out would be power cycling
- Many modern micro-controllers have a Watch-dog timer, which can be used to recover from a dead-lock condition

Mutex

- We require a method of co-ordinating the activities of threads which work concurrently and access shared resources
- Several methods are used, all of which ultimately depend upon the concept of an atomic or indivisible operation
- We want Mutual Exclusion between certain operation within a set of threads, for example, they will not try to change a shared variable at the same time
- The critical operations are called Critical Section and we use a special type of variable, known as Mutex, to lock out all other threads, except one
- If P1 reaches a point in its execution cycle where it would enter its critical section, it will try to lock a Mutex m and only if it is able to lock m will it be able proceed further to its own critical section

- If P1 can not lock m, it will wait just before entry to its critical section
- We say P1 is suspended or is waiting on m
- Once m is locked by any thread, it can not be locked again, it can only be unlocked by some thread



Two threads need to access a shared resource in their cyclic operation. The period during which a thread tries to access a shared resource is called Critical Section or critical Region, and we want that not more than one thread is in its critical section at a time. A Mutex m is generally associated with a shared resource.

Semaphore

- A special kind of variable, created in the O/S Kernel, used for implementing software functions needing atomic operations
- It can be considered to be an integer variable plus a pointer to a queue of processes, with certain special operations defined on it
- Apart from initial setting up of a semaphore, two operations are defined:
  - **signal**: (or V) - increments the value of the integer
  - **wait**: (or P) - *tries* to decrement the integer
- These operations are themselves atomic, i.e. indivisible and there is no other way that value of a semaphore can be changed
- **wait**: Decrements the value of semaphore variable by 1. If the value becomes negative, the process executing wait is blocked, i.e., added to the semaphore's associated queue.

- **signal**: Increments the value of semaphore variable by 1. After the increment, if the pre-increment value was negative (meaning there are processes waiting for a resource), it transfers a blocked process from the semaphore's waiting queue to the Ready queue.

Considerations for an O/S for an embedded system

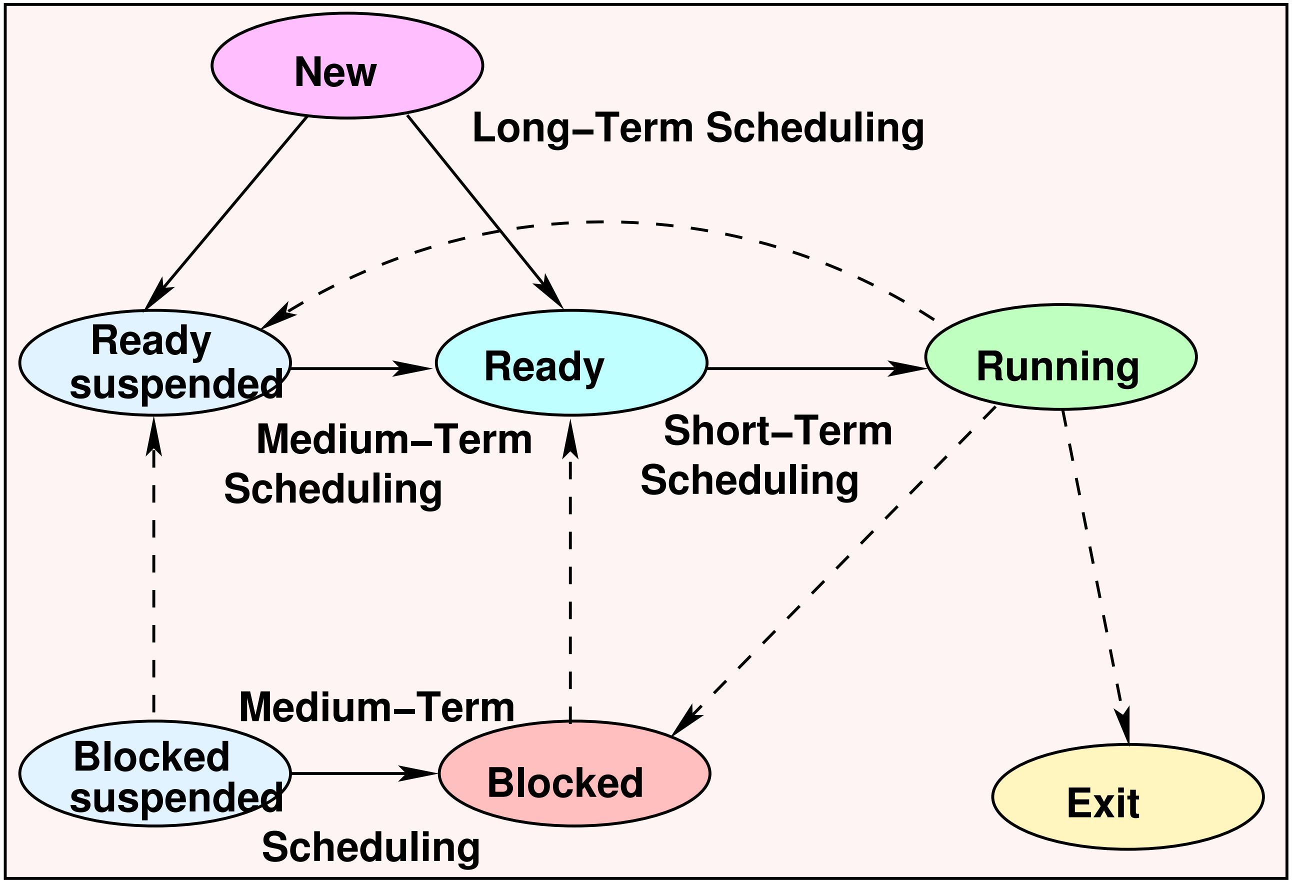
- It should take up only a small portion of program memory.
- It should use only a small amount of time to handle important requests (interrupts) or switch between tasks.
- Its behaviour should be predictable.
- Most of the resources are pre-assignable, only CPU cycles and possibly RAM requirements may be data dependent.
- Memory isolation, which is a major characteristics of processes, is not critical. Thread based organization of the application programs may be sufficient.
- Usually one thread per I/O device can be used.

- Initialization of special purpose I/O, like UART, I^2C, USB, Ethernet, etc. is mostly fixed.
- Initialization of general purpose I/O, like parallel ports, timer/counters, etc. are partly fixed.
- These considerations may be utilized to greatly simplify the O/S required for a particular range of applications

Task Scheduling

- the part of the Operating Environment that responds to the requests by programs and interrupts for processor attention and gives control of the processor to the processes dealing with those requests
- Scheduling algorithm is an algorithm followed by the Scheduler to decide which task gets the next turn on CPU.
- Tasks could either be periodic or aperiodic
- Real-time scheduling algorithms may assign priorities statically, dynamically, or in a hybrid manner, which are called fixed, dynamic and mixed scheduling algorithms, respectively.
- These algorithms may allow preemption to occur or may impose a non-preemptive method.

Task Scheduling in a G.P. System



Task Scheduler Types

- Endless Loop: No Tasks, Polled I/O only
- Basic Cyclic Executive: Tasks executed as often as possible, Polled I/O only
- Time Driven Cyclic: Single Frequency, Polled I/O
- Multi-rate Cyclic: Multiple Freq, Polled I/O
- Multi-rate for periodic Tasks: higher precision
- Multi-rate with interrupts: polled nd Interrupt driven
- Priority-based Pre-emptive: periodic and non periodic
- Dead-line Scheduler: better handling of Real-Time
- Partition Scheduler: better handling of Real-Time

Endless Loop

- The kind of task scheduling used in very simple Embedded systems.
- The control path may possibly have branches and nested loops.
- The endless loop keeps on working, no matter what.
- Interrupts can not be dealt with by the loop itself, they must be disabled, otherwise can create problems with the loop operation.
- It can be slow for many tasks, still, it is a model for what we use in many simple applications.
- ARDUINO uses generally this model.

A typical pseudo-code for Endless loop

```
while(1){
    Request Input Device to make a Measurement;
    Wait for the Measurement to be ready;
    Fetch the Value of the Measurement;
    Process the Value of the Measurement;
    if(Value is Reasonable){
        Prepare new Result using Value;
    } else {Report an Error;
    }
    Request Output Device to deliver the Result;
    Wait for the Result to be output;
    Confirm that output is OK;
}
```

Arduino approach in C language

```
void setup() {
  // setup code here,  runs once
  // initialize I/O devices, variables, etc.
}
void loop() {
  // main code here, runs repeatedly
  // this is the outer while(1) loop
}
```