Module V

RTOS BASED DESIGN

RTOS based Design – Basic operating system services. Interrupt handling in RTOS environment. Design Principles. Task scheduling models. How to Choose an RTOS. Case Study – MicroC/OS-II.

Real Time Operating System

A real time is the time which continuously increments at regular intervals after the start of the system and time for all the activities at difference instances take that time as a reference in the system.

A real time operating system (RTOS) is multitasking operation system for the applications with hard or soft real time constraints. Real-time constraint means constraint on occurrence of an event and system expected response and latency to the event.

Basic OS Functions

- Process Management
- Resources Management
- Device Management
- I/O Devices subsystems
- Network Devices and subsystems Management

OS Services

1. OS Goal
2. Processor modes(User & Supervisory)
3. System structure
4. Kernel

1. *OS Services Goal*

OS goals are perfection, correctness, portability, interoperability and providing a common set of interfaces for the system and orderly access and control when managing the processes.

- Easy sharing of resources as per schedule and allocations
- Easy implementation of the application program with the given system hardware.
- Scheduling of processes.
- Management of the processes.
- Common set of interfaces
- Maximizing the system performance

2. *Mode structure*
    - User mode: User function call, which is not a system call, is not permitted to read and write into the protected memory allotted to the OS functions,data and heap.
    - Supervisory mode: The OS runs the privileged functions and instructions in protected mode and the OS only accesses the hardware resources and protected area memory. Only a system call is permitted to access protected memory.
3. *System structure*

    Layered model.
    1. Application software
    2. Application Programming Interface(API)
    3. System software other than one provided at the OS
    4. OS interface
    5. OS
    6. Hardware-OS interface
    7. Hardware
4. *Kernel*

    The OS is the middle layer between the application software and system hardware. An OS includes
    1. A kernel with device management and file management as a part of kernel in the given OS.
    2. Kernel without device management and file management.

The kernel is the basic structural unit of any OS in which the memory space of the functions, data and stack are protected from access by any call other than the system-call. It can be defined as a secured unit of an OS.

Functions of the kernel

1. Process management
2. Memory management
3. File management
4. Device management and device drivers
5. I/O sub system management

Interrupt handling in RTOS environment

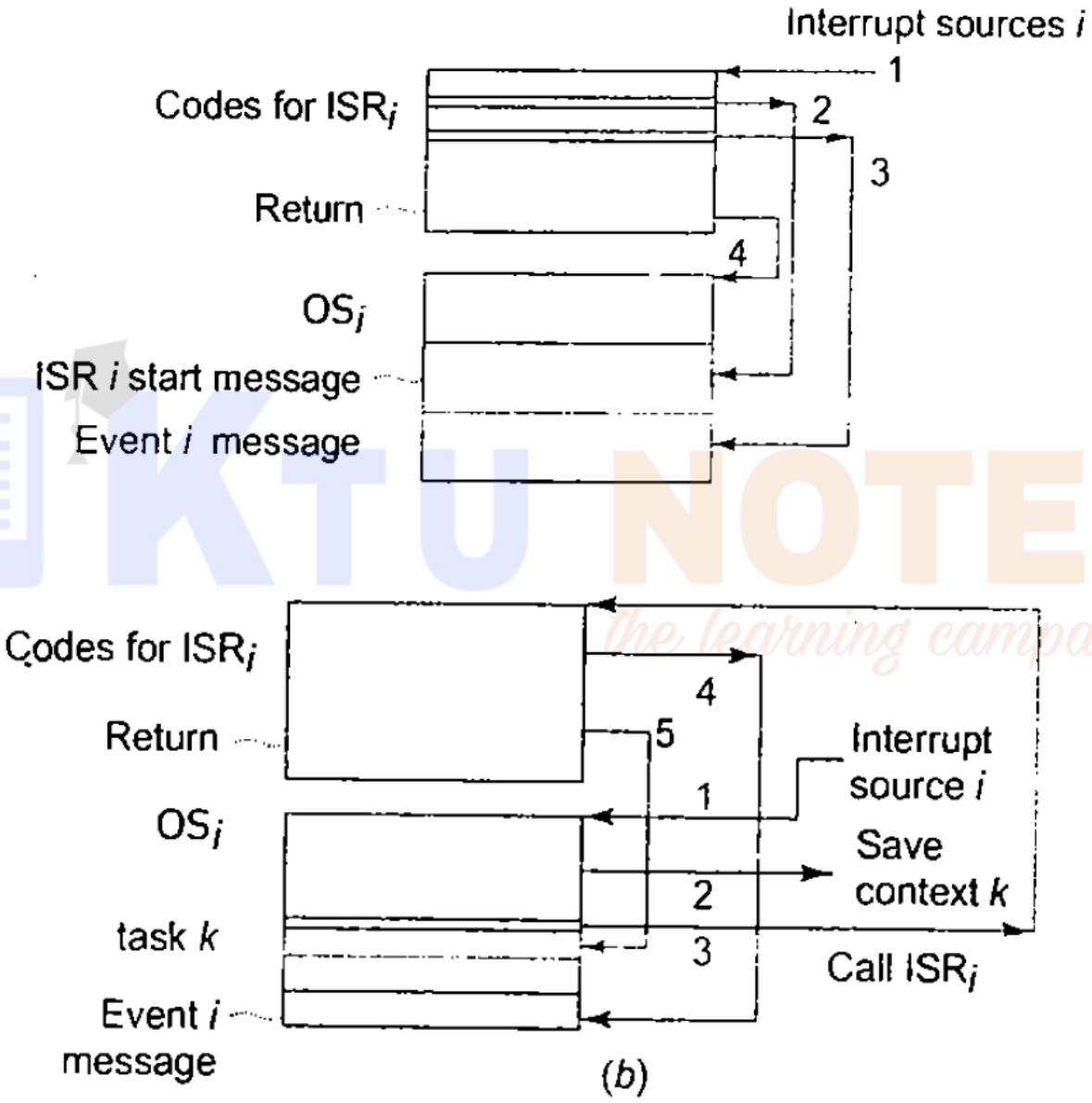
In a system, the ISRs should function as following.

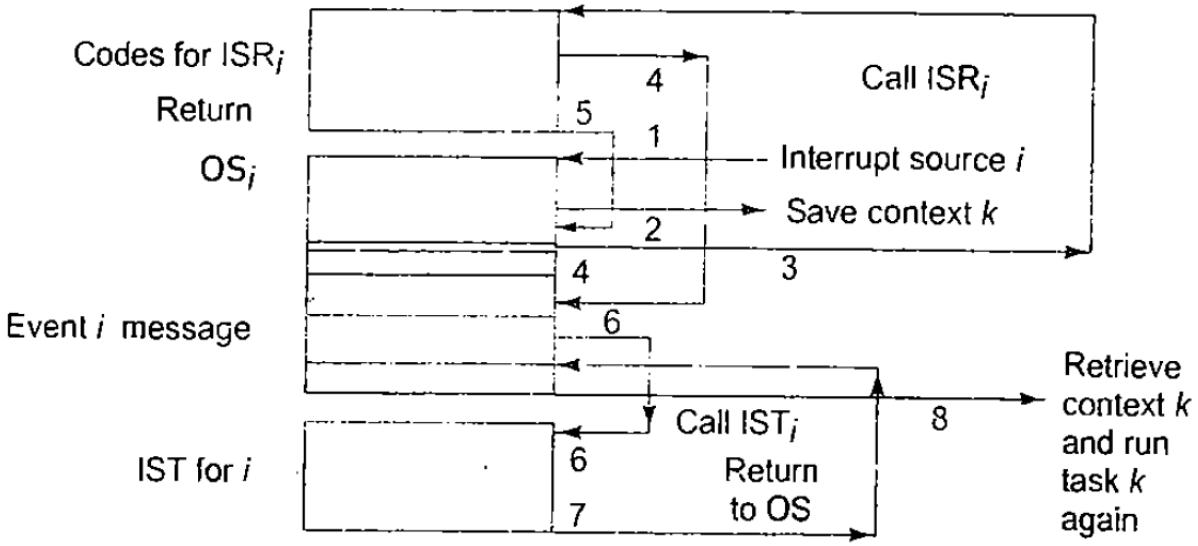
- ISRs have higher priorities over the OS functions and the application tasks. An ISR does not wait for a semaphore, mailbox message or queue message.

- An ISR does not also wait for mutex.

  There are three alternative systems for the OSes to respond to the hardware source calls from the interrupts. Figure below shows the three systems.

  1. Direct call to ISR by interrupting source.
  2. RTOS first interrupted on an interrupt, then OS calling ISR
  3. RTOS first interrupted on an interrupt then OS calling a fast ISR and the fast ISR calls a slow ISR(IST)





RTOS Services

Basic OS functions, Process priorities management; priority allocation and priority inheritance, Process management; preemption, Process predictability, Memory management; protection and MMU, Memory allocation, RTOS scheduling and interrupt latency control functions, Timer functions and time management, Asynchronous IO functions, IPC Synchronization mechanisms, Spin locks, Time slicing, Hard and soft real time operability, etc.

RTOS Design Principles

Basic design using an RTOS

An embedded system with a single CPU can run only one process at an instance. The process at any instance may either be an ISR, kernel function or task. An RTOS use in embedded system facilitates the following:

1. An RTOS provides running the user threads in kernel space so that they execute fast.
2. An RTOS provides effective handling of the ISRs, device drivers, ISTs, tasks or threads.
3. An RTOS provides memory allocation and de-allocation functions in fixed time and blocks of memory and restricting the memory accesses only for the stack and other critical memory blocks.
4. An RTOS provides for effectively scheduling and running and blocking of the tasks in case of many tasks.
5. IO management with devices, files, mailboxes, pipes and sockets becomes simple using an RTOS.
6. Effective management of the multiple states of the CPU.

RTOS design principles

- Design with ISRs and tasks
- Each ISR design consisting of shorter code.
- Design with using interrupt service threads or interrupt service tasks.
- Design each task with an infinite loop from start up to finish.
- Design in the form of tasks for the better and predictable response time control.
- Design in the form of tasks for modular design.
- Design in the form of tasks for data encapsulation.
- Design with taking care of the time spent in the system calls.
- Limit the number of tasks and select the appropriate number of tasks to increase the response time to the tasks, better control over shared resource and reduced memory requirement for stacks.
- Use appropriate precedence assignment strategy and use preemption in place of time slicing.
- Avoid task deletion.
- Use idle CPU time for internal functions.
- Design with memory allocation and de-allocation by the task.
- Design with taking care of the shared resources or data among the tasks.
- Design with hierarchial and scalable limited RTOS functions.

Task scheduling models

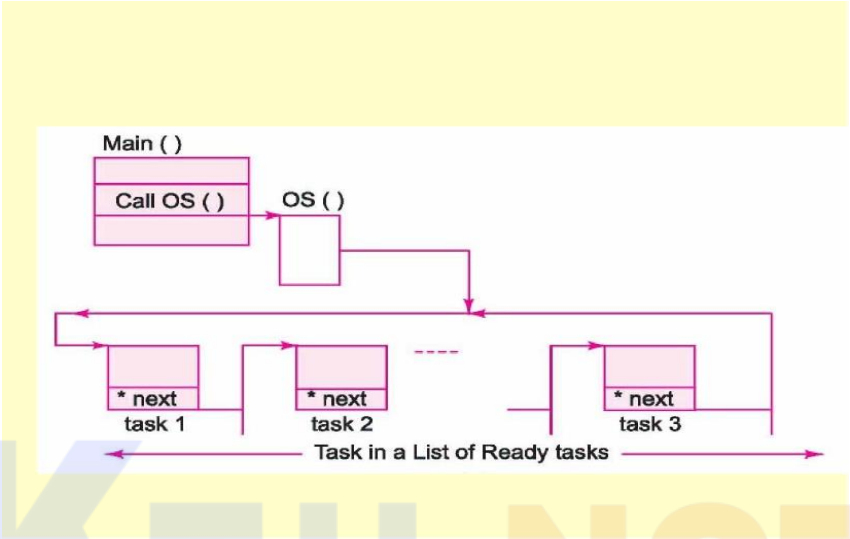
1. Cooperative scheduling model
2. Cooperative with precedence constraints
3. Cyclic and round robin with time slicing scheduling models
4. Preemptive scheduling model
5. Scheduling using Earliest deadline first(EDF) precedence
6. Rate monotonic schedulers (RMS) using higher rate of events occurrences first precedence
7. Fixed (static) time scheduling models
8. Scheduling of periodic, sporadic and aperiodic tasks
9. Advanced scheduling algorithm using the probabilistic time Petri nets (stochastic) or multithread graphs. For multiprocessor and distributed systems

   An RTOS commonly executes the codes for the multiple tasks as priority based preemptive scheduler.

1. Cooperative scheduling model

   Each task cooperate to let the running task finish. Cooperative means that each task cooperates to let the running one finish. None of the tasks does block in-between anywhere during the ready to finish states. The service is in the cyclic order. Worst case latency is same for every task.

   Scheduler inserts into a list, the ready tasks for a sequential execution in a cooperative mode



2. Cooperative with precedence constraints

   Scheduler using a priority parameter, taskPriority does the ordering of list of the tasks— ordering according to the precedence of the interrupt sources and tasks. The scheduler first executes only the first task at the ordered list, and the total, equals to the period taken by the first task on at the list. It is deleted from the list after the first task is executed and the next task becomes the first. The insertions and deletions for forming the ordered list are made only at the beginning of the cycle for each list. Worst-case latency is not same for every task. Cooperative Priority based scheduling of the ISRs executing in the first layer and Priority based ready tasks at an ordered list executing in the second layer.

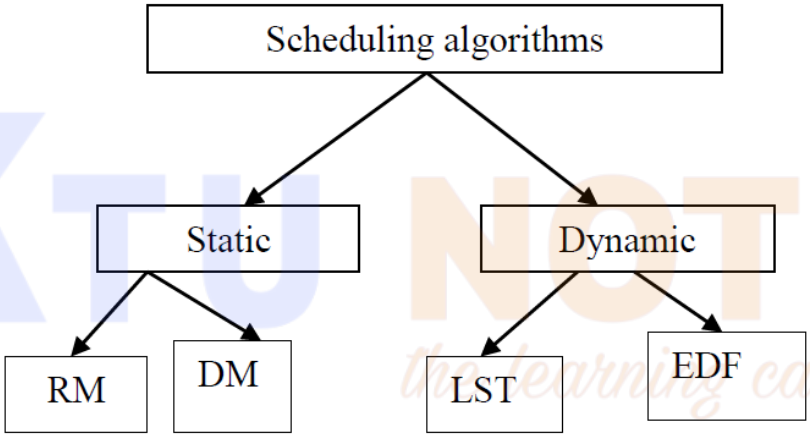
3. Cyclic and round robin with time slicing scheduling models

   Round-robin (RR) is one of the algorithms employed by process and network schedulers in computing. As the term is generally used, time slices (also known as time quanta) are assigned to each process in equal portions and in circular order, handling all processes without priority(also known as cyclic executive). Round-robin scheduling is simple, easy to implement, and starvation-free. Round-robin scheduling can also be

applied to other scheduling problems, such as data packet scheduling in computer networks. It is an operating system concept. The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn.

4. Other scheduling algorithms

In real time systems scheduling algorithms are classified into two categories: Static algorithm and Dynamic algorithm. Based on execution attributes of tasks, dynamic algorithm assigns priorities at runtime. This algorithm allows switching of priorities between tasks. In contrast with dynamic algorithm, a static algorithm assigns priorities at design time. All assigned priorities remain fixed throughout the execution of task. Figure below gives the classification of available scheduling algorithms for real time systems.



Rate Monotonic (RM): In RM algorithm, tasks have to be periodic in nature and deadline must be equal to its period. Tasks are scheduled according to their period. The rate of task is the inverse of its period. This algorithm implemented by assigning fixed priority to tasks, the higher its rate, higher the priority.

Deadline Monotonic (DM): The other algorithm for scheduling all the real time tasks based on their deadline is known as deadline monotonic. In this algorithm, priorities are decided by considering relative deadline. Tasks with shortest deadline get the highest priority. If one task will get higher priority than the other then its relative deadline must be shorter as compared to other tasks. RM and DM are identical except priorities are automatically computed from rate of task or deadline.

Least Slack Time First (LST): It is type of dynamic algorithm which assigns priority dynamically. Tasks are scheduled according to their slack: the smaller the slack, the higher the priority. Slack is computed by using the difference between the deadline, the ready time and the run time.

Earliest Deadline First (EDF): The most common dynamic priority scheduling algorithm for real time systems is the EDF. Here priorities are dynamically reassigned at runtime based on the time still available for each task to reach its next deadline. Both static and dynamic systems are scheduled by EDF algorithm. A queue of tasks is maintained in ascending order of deadline and whenever a processor gets free, the head of the queue will be assigned to the processor according to EDF algorithm. When new task arrives, its deadline will be compared with the deadline of currently executing task, and in case if deadline of newly arrived task is closer to the current time, it will receive the processor and the old task will be pre-empted and placed back in the queue. As compared to other algorithms EDF is optimal and simple to implement, giving much better utilization of processor. This research work proposes an EDF scheduling algorithm which helps to schedule tasks dynamically, enhance the throughput, reduce the overload and also helps to decrease energy consumed in data transmission. EDF improves the system's performance and allows a better exploitation of resources.

How to Choose an RTOS

Functional

- Processor support
- Memory requirement
- Real time capabilities
- Kernel and interrupt latency
- IPC and task synchronization
- Modularization support
- Support for networking and communication
- Development language support

Non Functional

- Custom developed or off the shelf
- Cost
- Development and debugging tools availability
- Ease of use
- After sales

Case Study – MicroC/OS-II

- MicroC/OS-II is a portable, ROMable, scalable, preemptive, real-time, multi-tasking, priority-based OS.
- Open source ANSI C and free for academic use.
- Was ported to 40+ architectures (8 to 64 bit) since 1992.

MicroC/OS-II

- Task creation and Management
- Kernal Functions and initialization
- Task scheduling
- Inter-task communication
- Mutual exclusion and task synchronization
- Timing and reference
- Memory management
- Interrupt handling

Task creation and Management

Five possible states for a task to be in

- Dormant – not yet visible to OS (use OSTaskCreate(), etc.)
- Ready
- Running
- Waiting
- ISR – preempted by an ISR

- A task is usually created as a non ending function

```
//creates a task with name MicroCtask
void MicroCTask(void *arg)
{
        while(1)
        {
                //code corresponds to task
        }
}
```

- Need to pass this task to MicroC kernel to make ready for execution (supports upto 64 tasks)
- OSTaskCreate() in os_task.c

- Create a task
- Arguments: pointer to task code (function), pointer to argument, pointer to top of stack , desired priority (unique)
- OSTaskCreateEx() in os_task.c
    - *Create a task*
    - *Arguments: same as for OSTaskCreate(), plus*
    - *id: user-specified unique task identifier number*
    - *pbos: pointer to bottom of stack. Used for stack checking (if enabled).*
    - *stk_size: number of elements in stack. Used for stack checking (if enabled).*
    - *pext: pointer to user-supplied task-specific data area (e.g. string with task name)*
    - *opt: options to control how task is created.*

Task Management

- OSTaskSuspend()
    - Task will not run again until after it is resumed ; kernel reschedules the next highest priority for execution
    - Sets OS_STAT_SUSPEND flag, removes task from ready list if there
    - Argument: Task priority (used to identify task)
- OSTaskResume()
    - Task will run again once any time delay expires and task is in ready queue
    - Clears OS_STAT_SUSPEND flag
    - Argument: Task priority (used to identify task)
- OSTaskDelRequest()
- OSTaskDel()
    - Sets task to DORMANT state, so no longer scheduled by OS
    - Removed from OS data structures: ready list, wait lists for semaphores/mailboxes/queues, etc.
- OSTaskChangePrio()
    - Identify task by (current) priority
    - Changes task's priority
- OSTaskQuery()

- Identify task by priority
- Copies that task's TCB into a user-supplied structure
- Useful for debugging

- OSTaskStkCheck()
  - Checks stack overflow for creation of tasks
  - OSTasknameGet()
    - Returns the name associated with a task
  - OSTaskNameSet()

Application requested delays

- OSTimeDly()/OSTimeDlyHMSM()
  - Task A calls OSTimeDly or OSTimeDlyHMSM() in os_time.c
  - TCB->OSTCBDly set to indicate number of ticks to wait;usually-2^{16} ticks
  - Otherwise H(0-255),M(0-59),S(0-590),M(0-999)
  - Note that OSTickISR() in os_cpu_a.a30, OSTimeTick() in os_core2.c decrement this field and determine when it expires
- OSTimeDlyResume()
  - Task B can resume Task A by calling OSTimeDlyResume()
  - Resumes a task which is delayed by a call to either OSTimeDly()/OSTimeDlyHMSM()
- Tasks example
  - Task 1
    - Flashes red LED
    - Displays count of loop iterations on LCD top line
  - Task 2
    - Flashes green LED
  - Task 3
    - Flashes yellow LED

<u>Kernel Functions and initialization</u>

- Kernel needs to be initialized and started before executing the user tasks.
- OS kernel initialization function OSInit() is executed first. It performs
  - Initialize the different OS kernel data structure
  - Creates the idle task OSIdle() –low priority
- Code snippet for OS initialization and start operations

```
#include<includes.h>
//main task
void main(void)
{
//initializes the kernel
OsInit();  //create the first task .
OSStart();  //Start OS by executing the highest priority task
}
```

Task scheduling

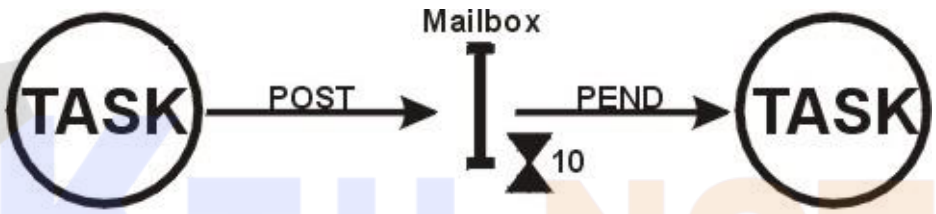
- MicroC/OS-II supports preemptive priority based scheduling (64 level 4-60 based)
- When the kernel starts,it first executes the highest priority task
- A task rescheduling happens when
    - Whenever a higher priority task becomes 'ready' or when task enters waitng state due to OsTaskSuspend() ,OSTaskResume(),OSTaskDel(),OSTimeDly() etc
    - Whenever a interrupt occurs ,the task are rescheduled upon return from the ISR
- Task level scheduling is executed by the kernel service OS_Shed() and ISR level by OSIntExit()
- Scheduler runs highest-priority task using OSSched()
    - OSRdyTbl has a set bit for each ready task
    - Checks to see if context switch is needed
    - Macro OS_TASK_SW performs context switch
        - *Implemented as software interrupt which points to OSCtxSw*
        - *Save registers of task being switched out*
        - *Restore registers of task being switched in*
- Scheduler locking
    - Can lock scheduler to prevent other tasks from running (ISRs can still run)
        - *OSSchedLock()*
        - *OSSchedUnlock()*
    - Nesting of OSSchedLock possible
    - Don't lock the scheduler and then perform a system call which could put your task into the WAITING state!
- Idle task

- Runs when nothing else is ready
- Automatically has prioirty OS_LOWEST_PRIO
- Only increments a counter for use in estimating processor idle time

Inter-task communication

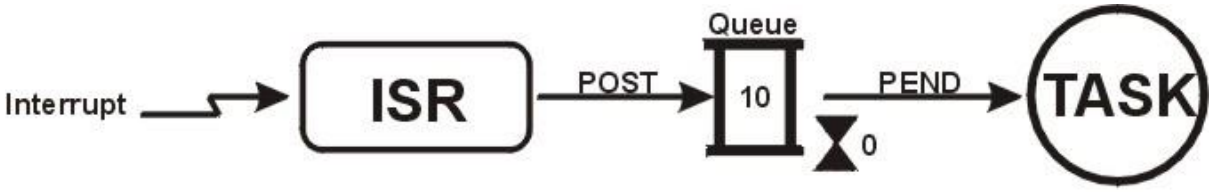
MicroC/OS-II supports following IPC techniques for data sharing and co-operation among tasks.

- Message Mailboxes
  - For passing of pointers
  - *Create, pend, post, post_opt, accept, query*
- Message Queues
  - For queuing pointer size messages
  - *Create, del, pend, post, post_front, post_opt, accept, flush, query*



Note: POST deposites a pointer size variable in the mailbox

Message Mailbox



Note: POST deposites a pointer size variable in the queue

Message Queue

Message Mailboxes

- OSMboxCreate()
- OSMboxPost()/OSMboxPostOpt()
  - A message is poseted in the mailbox

    - If there is already a message is there in the mailbox,an error is returned(not overwritten)
    - If tasks waiting for a message from the mailbox,the task with highest priority is removed from the wait list and scheduled to run
- OSMboxPend()
    - Used for retrieving messages from mailbox.
    - If mailbox is empty ,the task is immediately blocked and moved to the wait list
    - A time-out value can be specified
- OSMboxAccept()
    - Its also used for retrieving message from the mailbox
    - The calling task is not blocked('pended') if there is no message to read in the mailbox
- OSMboxQuery()
    - Retrieving information(whether message is there in inbox,is there any task is waiting ,how many tasks etc) from mailbox
- OSMboxDel()

Message Queue

- OSQCreate()
- OSQPost()/OSQPostFront()
- OSQPostOpt()
- OSQPend()
- OSQAccept()
- OSQQuery()
- OSQFlush()
- OSQDel()

<u>Mutual exclusion and task synchronization</u>

- In multitasking system, multiple tasks executes concurrently and share the system resources and data
- The access to shared resources should be made mutually exclusive to prevent ata corruption and race conditions
- Task synchronization can be used by the tasks to synchronize their execution
- Microc/OS-|| provides access to shared resources through the different stept
    1. Disabling task scheduling

2. Disabling interrupts
3. Semaphores for mutual exclusion

   Binary and Counting Semaphores

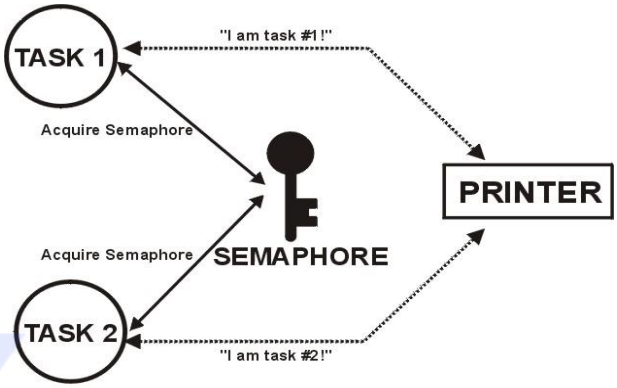
   For Event signaling, control access to shared resource, and synchronization

   accept,create,del,pend,post,query

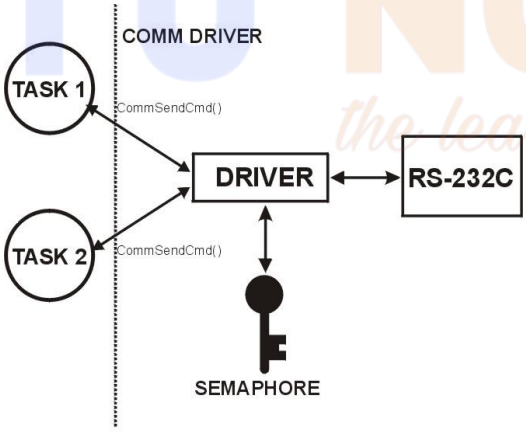
   Mutual Exclusion Semaphores

   For controlling access to a shared resource (with priority inheritance)

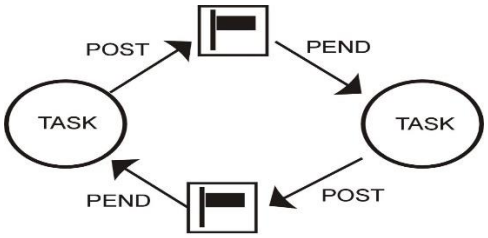
   accept, create, del, pend, post, query
4. Events (flags) for synchronization



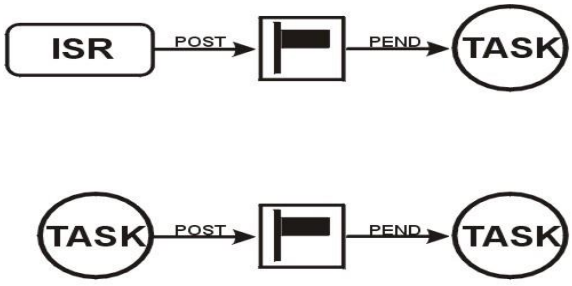
Two tasks competing for a printer



Comm Driver Example



Synchronizing two Tasks



Signaling Events through Semaphores

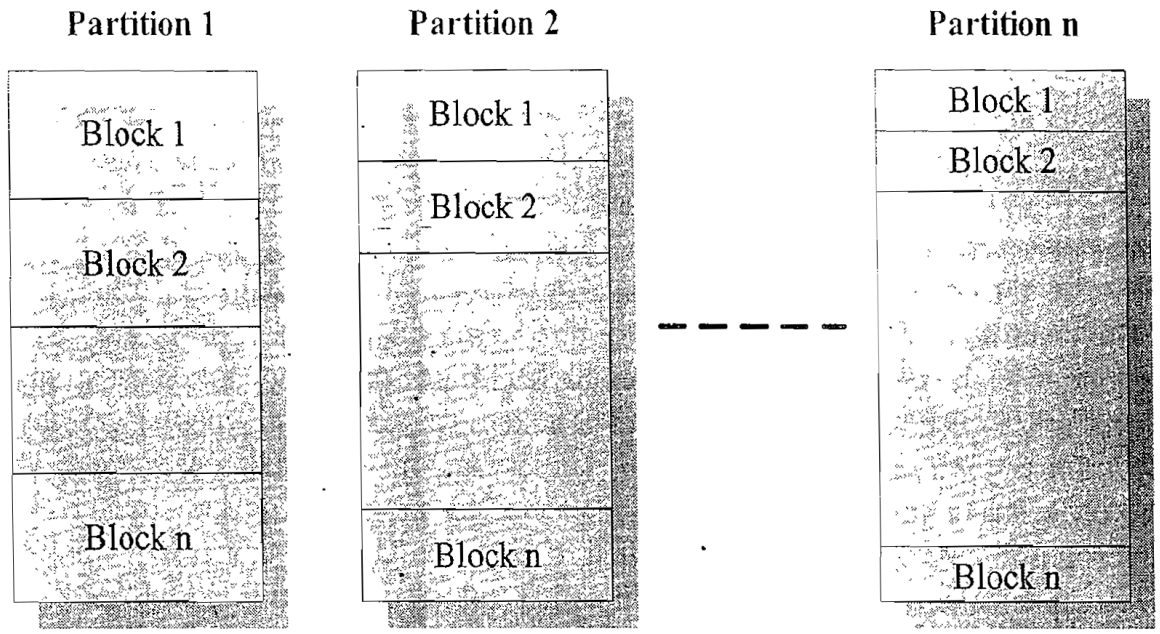
Timing and reference

The 'clock tick' acts as the time source for providing timing reference for time delays and timeouts. It generates periodic interrupts

- OSClockTick, 10 – 100 Hz, increments a 32 bit counter
- OSTimeDly()
- OSTimeDlyHMSM()
- OSTimeDlyResume()
- OSTimeGet() -counter
- OSTimeDlySet()

Memory management

- Alternative to malloc() and free()
- Memory is divided into multiple sectors(partitions)
- Partitions memory into blocks of equal sizes
- OSMemCreate(), get, put, query, nameget, nameset



Memory partitioning under Micro C/OS-II

Interrupt handling

- Structure needed
  - Save CPU registers – *NC30 compiler adds this automatically*
  - Call OSIntEnter() or increment OSIntNesting (faster, so preferred)
    - *OSIntEnter uses OS_ENTER_CRITICAL and OS_EXIT_CRITICAL, so make sure these use method 2 (save on stack)*
  - Execute code to service interrupt – *body of ISR*
  - Call OSIntExit()
    - *Has OS find the highest priority task to run after this ISR finishes (like OSSched())*
  - Restore CPU registers – *compiler adds this automatically*
  - Execute return from interrupt instruction – *compiler adds this automatically*
- Good practices

  Make ISR as quick as possible. Only do time-critical work here, and defer remaining work to task code.

  Have ISR notify task of event, possibly send data

  OSSemPost – raise flag indicating event happened

  OSMboxPost – put message with data in mailbox (1)

  OSQPost – put message with data in queue (n)

  Example: Unload data from UART receive buffer (overflows with 2 characters), put into a longer queue (e.g. overflows after 128 characters) which is serviced by task