Real-Time Programming Languages (EI5064)

Introduction to Concurrent Programming and OS

Samarjit Chakraborty

Processes

- A *process* consists of executable program code, the *state* of which is controlled by the OS
- The state during running of a process can be *running*, *blocked*, or *finished*
  - It is represented by *process-status*(), a *process structure* (data, objects and resources), and the *process control block* (PCB)
  - The PCB contains the program counter
- A process is scheduled to run by the OS
  - The OS gives the process control of the CPU as a result of a *system call* by the process
  - While running the process has access to the system memory and other system resources (e.g., network, display, etc.)

Application Programs Versus Processes

- An application program can consist of a number of processes
- Example – a mobile phone
  - Consists of a number of functions, interrupt service routines (ISRs), process threads, device drivers, … all of which should concurrently run on a single processor
  - Example processes – voice encoding and convolution (capture analog voice signals, digitize them, and encode them using a specific codec), modulation process, display process, GUIs, interrupts from keys (so that the user can receive a call or make one)

Process Control Block (PCB)

- PCB is a data structure that is associated with each process
- It is used by the OS to control/change process state (the PCB contains information about the *state* of the process)
- Stored in a protected area of the memory (so that other processes cannot change it)
- Information in the PCB
  - Process ID
  - Process Priority
  - Parent Process (if any)
  - Child Process (if any)
  - Address of the next PCB (denoting the next process that will run)

Process Control Block (PCB)

- PCB also contains information about the data used and generated by a process
- Signal/message dispatch table for inter-process communication
- Contains descriptors for open files
- Security restrictions and permissions

Process Context

- When a process starts running, its *context* is *loaded* into appropriate registers from the memory (e.g., program counter and stack pointer)
- At the time of a *context switch* (i.e., when a process is *blocked*), the context of the running process is saved in the memory and that of the next process is loaded from the memory (into the appropriate registers)

Threads

- A sub-process within a process that has its own program counter, stack pointer, stack and priority parameter
- A process can have multiple threads that share the same memory region
- Example – *Display* process of a mobile phone application might have
  - Display_Time_Date thread
  - Display_Battery thread
  - Display_Signal thread
  - Display_Profile thread
  - Display_Call Status thread
  - Display_Menu thread
  - They share the same memory region and resources allocated to the *Display* process

Threads

- Similar to a process, a thread might also have different states such as *running*, *blocked*, *finished*, etc.
- Different threads are scheduled according to some policy by the thread scheduler

Task, Processes, Functions and ISRs

- The terms *Task* and *Process* are often used interchangeably
- *Functions* are called from Tasks or Threads
- An *interrupt* is an event that might be triggered by either a task or a thread or by hardware and is processed by an *interrupt service routine* (ISR)

Sharing Resources

- Semaphores provide a mechanism to let a task/thread wait until another one finishes an action
- This is to avoid, for example, the following scenario
  - A thread T1 writes to a memory location, but is preempted before it completes
  - A second thread T2 reads from the same memory location (incorrect data)
- A *mutex* is a binary valued semaphore
  - typically used to serialize access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section

Sharing Resources

- Access to the shared resource appears in the code as a critical section
- Only one task is allowed to enter its critical section at a time
- This is done using a `get()` method to gain access to the semaphore
- The task that gets the semaphore continues execution
- After exiting the critical section, it releases the semaphore using the `release()` method
- `get()` and `release()` are always atomic, i.e., a task cannot be preempted while executing them

Sharing Resources

- Sharing of resources often lead to what is referred to as "priority inversion"
  - This happens when a lower priority task holds a resource and a higher priority task needing the same resource has to wait and cannot make progress with its computation
  - The higher priority task remains blocked until the lower priority task releases the non-preemptable resource
  - The higher priority task undergoes *priority inversion* on account of the lower priority task for the duration it waits while the lower priority task keeps holding the resource

Sharing Resources

- While such priority inversion does delay a higher priority task, the duration for which a task blocks can be made small if tasks are made to restrict themselves to very brief periods of critical section usage
- This may be achieved through careful programming
- However, a more serious problem is *unbounded priority inversion*

Sharing Resources

- Unbounded priority inversion

- Let T_H be the highest priority task and T_L be the lowest priority task and T_1, T_2, …, T_k be tasks with priorities intermediate between T_H and T_L

- T_H and T_L both need to share a critical resource R, which T_1, T_2, …, T_k does not need

- T_L starts accessing R and is then preempted by T_H. Now T_H needs R too and so it blocks and lets T_L to continue. However, T_L being the lowest priority task is preempted by T_1, T_2, …, T_k and as a result holds on to R for an indefinite amount of time

- Hence, T_H the highest priority task might have to wait for an indefinite amount of time for all lower priority tasks to complete. This results in *unbounded priority inversion*

Priority Inheritance Protocol (PIP)

- **PIP:** Whenever a task suffers priority inversion, the priority of the lower priority task holding the resource is raised through a priority inheritance mechanism
- This enables it to complete its usage of the critical resource as early as possible without having to suffer preemptions from intermediate priority tasks
- Whenever several tasks are waiting for a resource, the task holding the resource inherits the highest priority of all tasks waiting for this resource (if this priority is greater than its own priority)
- As soon as the task releases the resource, it gets back its original priority if it is not holding any other critical resources

Priority Inheritance Protocol (PIP)

- Disadvantages of PIP
  - Deadlock and chain blocking
- Deadlock
  - T1 and T2 need two critical resources CR1 and CR2
  - T1: *Lock CR1*, *Lock CR2*, *Process*, *Unlock CR2*, *Unlock CR1*
  - T2: *Lock CR2*, *Lock CR1*, *Process*, *Unlock CR1*, *Unlock CR2*
  - Assume that T1 has higher priority than T2
  - T2 starts running first and locks CR2 (T1 was not ready until then). Now, T1 arrives, preempts T2 and locks CR1. T1 now blocks on CR2 and T2 inherits T1's priority as a result of PIP. But it cannot lock CR1 because it is being held by T1
  - Now neither T1 or T2 can proceed, which results in a deadlock

Priority Inheritance Protocol (PIP)

- Chain blocking
  - A task is said to undergo chain blocking if each time it needs a resource, it undergoes priority inversion

Highest Locker Protocol (HLP)

- Every critical resource is assigned a *ceiling priority* value
- The ceiling priority of a critical resource is the maximum of the priorities of those tasks which may request this resource

- If FCFS is used among all equal priority tasks then

  Ceil (R) = max{priority(Tj) | Tj needs R}

- If round robin is used among all equal priority tasks then

  Ceil (R) = max{priority(Tj) | Tj needs R} + 1

- Advantage
  - **When HLP is used for resource sharing, once a task gets a resource required by it, it is not blocked any further**

Highest Locker Protocol (HLP)

- Let T1 and T2 needs CR1 and CR2
- T1 acquires CR1 and its priority becomes ceil(CR1)
- Subsequently, T1 also requires CR2. But CR2 is being held by T2. Can this happen?
  - If T2 is holding CR2, then its priority might be ceil(CR2), which might be higher or equal to priority(T1) because T1 also needs CR2
  - Therefore, T2 is currently the higher priority task and T1 shouldn't have gotten a chance to execute
  - This is a contradiction to the assumption that T1 was executing when T2 was holding the resource CR2
  - Therefore, T2 couldn't be holding the resource CR2 when T1 is executing
- ***Corollary***
  - Under HLP, before a task can acquire a resource, all resources that might be required by it must be free

Highest Locker Protocol (HLP)

- Shortcomings of HLP
  - HLP can result in inheritance-related inversion, which occurs when the priority value of a low priority task holding a resource is raised to a high value by the ceiling rule. As a result, intermediate priority tasks not needing the resource cannot execute and are said to undergo inheritance-related priority inversion

- This shortcoming motivates the *Priority Ceiling Protocol* (PCP)

Priority Ceiling Protocol (PCP)

- Difference between PCP and PIP
  - In PIP whenever a request for a resource is made, the resource will be allocated to the requesting task if it is free. However, in PCP a request may not be granted to a requesting task even if the resource is free
- As in HLP, PCP associates a ceiling value Ceil (CR) with every resource CR, that is the maximum of the priority values of all tasks that might use CR
- An operating system variable called CSC (Current System Ceiling) is used to keep track of the maximum ceiling value of all the resources that are in use at any instant of time
  - At system start, CSC is initialized to priority 0 (lower than the priority of the least priority task in the system)

Priority Ceiling Protocol (PCP)

- Resource sharing in PCP happens according to two rules: (i) *the resource grant rule*, and (ii) *the resource release rule*

  - **Resource grant rule**: When a task T_i requests a resource, two clauses are applied

    - ***Resource request clause***:

    1. If the task T_i is holding a resource whose ceiling priority equals CSC then the task is granted access to the resource
    2. Otherwise, T_i will not be granted the resource CR, unless its priority is greater than CSC (i.e., priority(T_i) > CSC)

    - In both the above cases, if T_i is granted access to the resource CR and if CSC < Ceil (CR) , then CSC is set to Ceil (CR)

    - ***Inheritance clause***: When a task is prevented from locking a resource by failing to meet the resource grant clause, it blocks and the task holding the resource inherits the priority of the blocked task if the priority of the task holding the resource is less than the priority of the blocked task

Priority Ceiling Protocol (PCP)

- Resource sharing in PCP happens according to two rules: (i) the resource grant rule, and (ii) the resource release rule
  - **Resource release rule**: If a task releases a critical resource it was holding and if the ceiling priority of this resource equals CSC, then CSC is made equal to the maximum of the ceiling value of all other resources in use; else CSC remains unchanged
    - The task releasing the resource either gets back its original priority or the highest priority of all tasks waiting for any resources which it might still be holding (whichever is higher)

PCP versus HLP

- PCP is similar to HLP except that in PCP a task when granted a resource does not immediately acquire the ceiling priority of a resource
  - In fact, under PCP the priority of a task does not change upon acquiring a resource, only the value of the system variable CSC changes
  - The priority of a task changes by the inheritance clause of PCP only when one or more tasks wait for a resource it is holding

PCP versus HLP

- Tasks requesting a resource, block almost under identical situations under PCP and HLP
  - The only difference with PCP is that a task T_i can also be blocked from entering a critical section if there exist any resource currently held by some other task whose priority ceiling is greater or equal to that of T_i
  - This arrangement prevents the unnecessary inheritance blockings that could be caused due to the priority of a task acquiring a resource being raised to a very high value (the ceiling priority)
  - In PCP, instead of actually raising the priority of the task acquiring a resource, only the value of the system variable CSC is raised to the ceiling value
  - By comparing the value of CSC against the priority of a task requesting a resource, the possibility of deadlocks is avoided
  - If no comparison with CSC would have been made (as in PIP), a higher priority task may later lock some resource required by this task, leading to a potential deadlock situation where each task holds a part of the resources required by the other task

PCP - Example

- Consider a system with four real-time tasks T1, T2, T3 and T4
- They share two non-preemptable resources CR1 and CR2
- CR1 is used by T1, T2 and T3
- CR2 is used by T1 and T4
- The priorities of T1, T2, T3 and T4 are 10, 12, 15 and 20
- FCFS among equal priority tasks and higher priority value indicates higher priority
- Ceiling priorities of the two resources
  - Ceil (CR1) = max { pri(T1), pri(T2), pri(T3)} = 15
  - Ceil (CR2) = max {pri(T1), pri(T4)} = 20
- Consider the case where T1 is executing after acquiring the resource CR1
  - When T1 acquires CR1, CSC is set to Ceil (CR1) = 15

PCP - Example

- Case 1
  - T4 now becomes ready. Being of higher priority, T4 preempts T1 and starts executing. After some time, T4 requests CR2. Since the priority of T4 (equal to 20) is greater than CSC (which is 15), T4 is granted the resource CR2 (follows from the resource request clause) and CSC is set to 20. When T4 completes execution, T1 will get a chance to execute
- Case 2
  - Assume that T3 becomes ready. Being of higher priority, T3 preempts T1 and starts executing. After some time, T3 requests CR1. As the priority of T3 (equal to 15) is not greater than CSC (which is 15), T3 will not be granted CR. T3 would block, and T1 will inherit the priority of T3 (follows from the PCP inheritance clause). Hence, T1's priority will change from 10 to 15
- Exercise (need not be submitted): How is deadlock, unbounded priority inversion, etc. avoided in PCP?

OS Services

- Enable resource sharing (ensure that data is not corrupted, resources are shared in a fair manner, ensure that real-time constraints are satisfied, etc.)
- Facilitate easy implementation of application programs
- Provide appropriate context switching mechanisms for tasks
- Optimize system performance/utilization
- Provide interfaces to other devices

OS Modes

- *User* and *supervisory* mode
- In the user mode, user processes are only allowed to run a subset of functions and instructions in the OS
    - OS functions may be used in the user mode to send messages to waiting processes
- In the supervisory or *kernel* mode, the OS runs privileged functions and instructions
- Example – every clock tick of the system clock results in a system interrupt. This results in system time being updated in the kernel mode, followed by a context switch from the kernel to the user mode

OS Modes

- Typical jobs in the kernel mode
  - Creation and deletion of processes
  - Process structure maintenance
  - Processing resource requests
  - Scheduling processes
  - Inter-process communication
  - File and device management

Process Creation

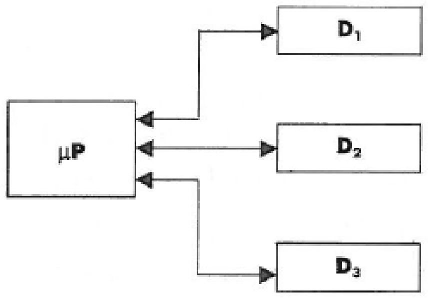
- Define address space for the process (stack, data, heap) and store the initial information for the process in a PCB
- Example
  - OS uses the *OS_Task_Create ( )* function to create a process Task_Send_Card_Info
  - Task_Send_Card_Info creates two processes Task_Send_Port_Output and Task_Read_Port_Input
  - The OS will schedule these tasks and context switch between them

System Timer

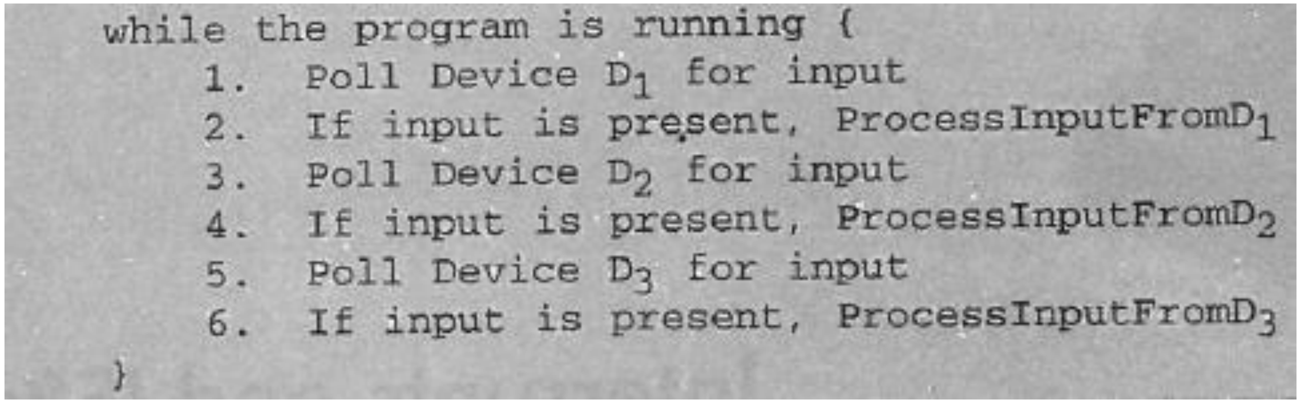
- Each clock tick results in a SysClkIntr interrupt
- There are usually a number of OS functions related to the system clock
  - OS_TICK_PER_SEC sets the SysClkIntr interrupt to occur every second
  - The function OSTickInit ( ) is used to initiate the system clock
  - OSTimeDelay ( ) delays a process making the call by a fixed number of system clock ticks specified in the argument
  - OSTimeSet ( ) sets a counter (to count the number of ticks since the counter was set)
  - OSTimeGet ( ) returns the value of the counter since it was last set

What is an interrupt?

- Most embedded systems need to take inputs from the environment or from an user
- Interrupts are the most common mechanisms for interacting with the environment/user
- The system below shows a microprocessor which processes inputs from three peripheral devices



Polling



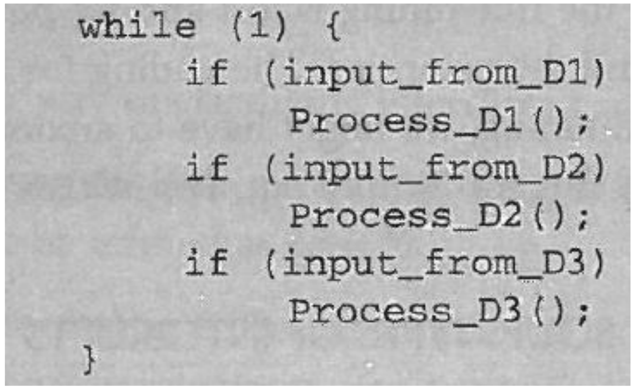
```
while the program is running {
    1.  Poll Device $D_1$ for input
    2.  If input is present, ProcessInputFrom$D_1$
    3.  Poll Device $D_2$ for input
    4.  If input is present, ProcessInputFrom$D_2$
    5.  Poll Device $D_3$ for input
    6.  If input is present, ProcessInputFrom$D_3$
}
```

- One mechanism for servicing these three devices is by *polling*
- Essentially the microprocessor asks each of the devices if there is something to process, and if there is, then the microprocessor processes it
- Advantages: simple to implement
- Disadvantages: it keeps the microprocessor busy forever, even when the devices have nothing to process

Interrupts

- In contrast to polling, a device might as well "inform" or interrupt the microprocessor whenever there is something to process
- Advantages: the processor is free to some other work when the devices have nothing to process
- Properties of interrupts
  - Asynchronous: since the interrupt might come from an external device, this device may not be clocked by the same system clock. Hence, an interrupt might arrive independent of the processor's clock. However, interrupts are *presented* to the microprocessor synchronously
  - Request: an interrupt is just a request. So a processor might be free to ignore it (except for non-maskable interrupts or NMIs)
  - Each interrupt is associated with a specific service (i.e., must be served by an interrupt service routine)

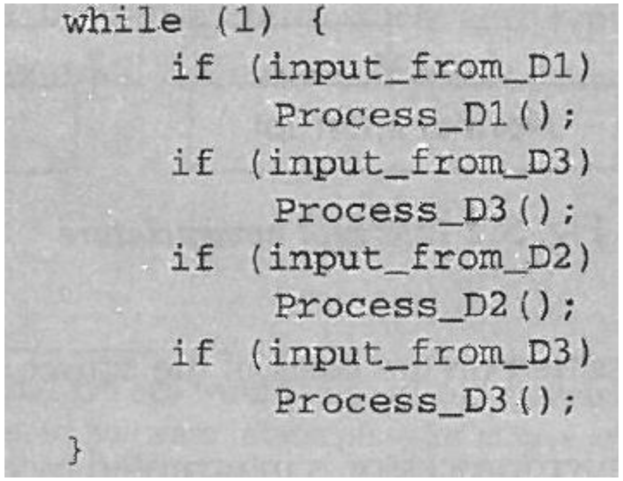
Polling versus Interrupts



```
while (1) {
        if (input_from_D1)
              Process_D1();
        if (input_from_D2)
              Process_D2();
        if (input_from_D3)
              Process_D3();
}
```

- If D1 and D3 have inputs from the processor at the same time, then D3 has to wait
  - for D1 to complete and also query D2
- This wait might, e.g., result in D3’s buffer to overflow
- How can such a situation be prevented?

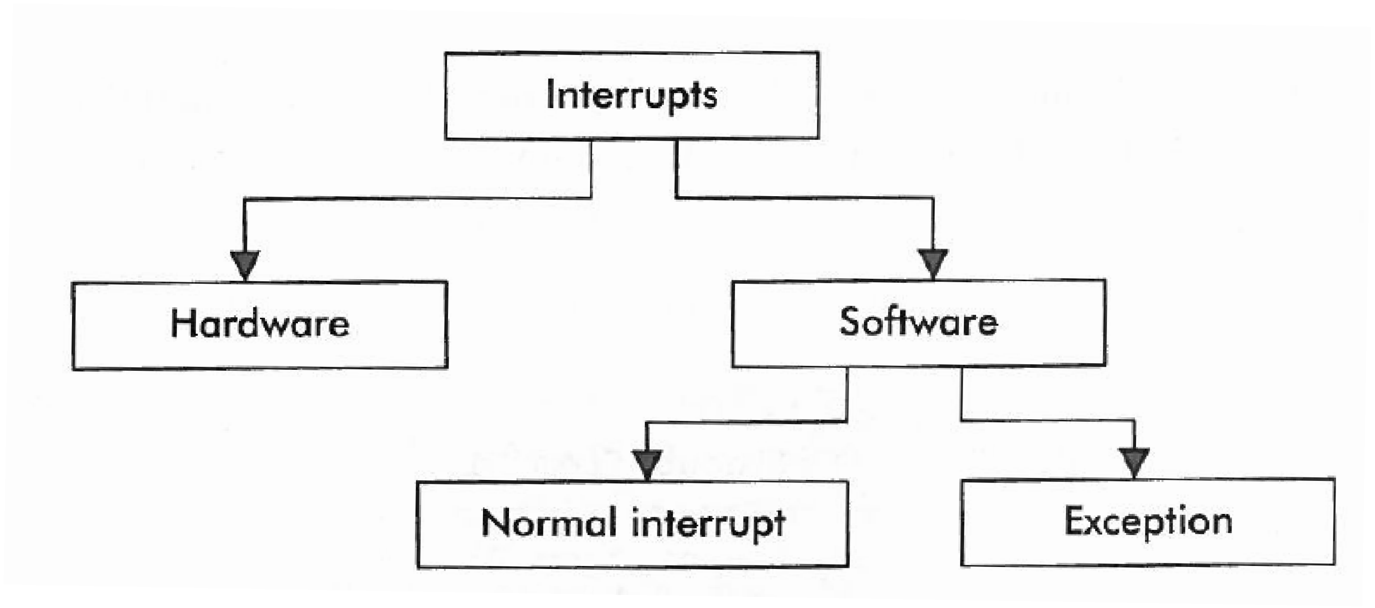
Polling versus Interrupts



```
while (1) {
          if (input_from_D1)
                    Process_D1();
          if (input_from_D3)
                    Process_D3();
          if (input_from_D2)
                    Process_D2();
          if (input_from_D3)
                    Process_D3();
}
```

- This solution will work if the processing of D1 and D2 are within the overflow time of D3
- However, if a new device is now added, this fine-tuning of the code will be lost and the code has to be readjusted or tuned once again
- Hence, polling is also not a clean solution for complex systems

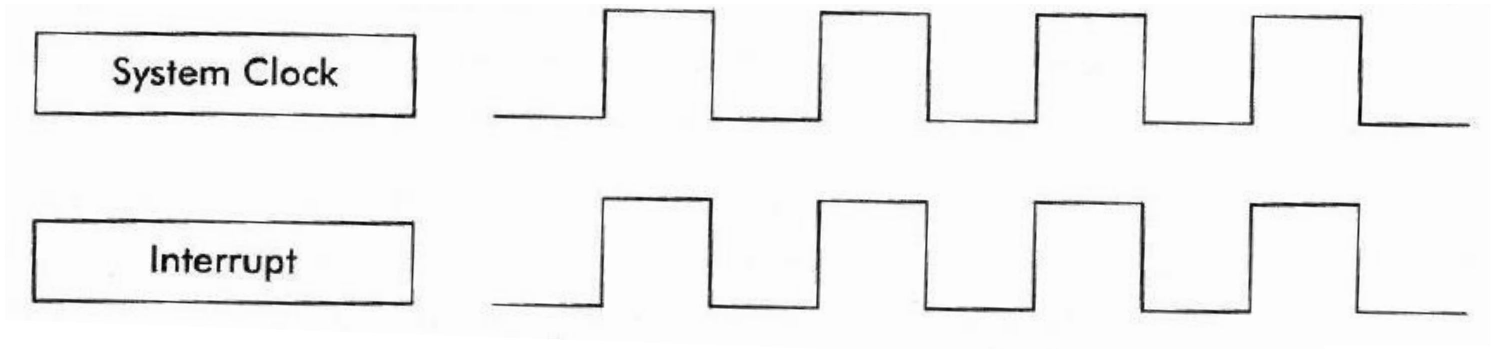
Types of Interrupts



- Hardware interrupts: if the microprocessor is interrupted by external device/hardware
- Software interrupts: those raised by software instructions
  - Exceptions: if it is unplanned (division by zero)
  - Interrupts: raised intentionally and occurs deterministically

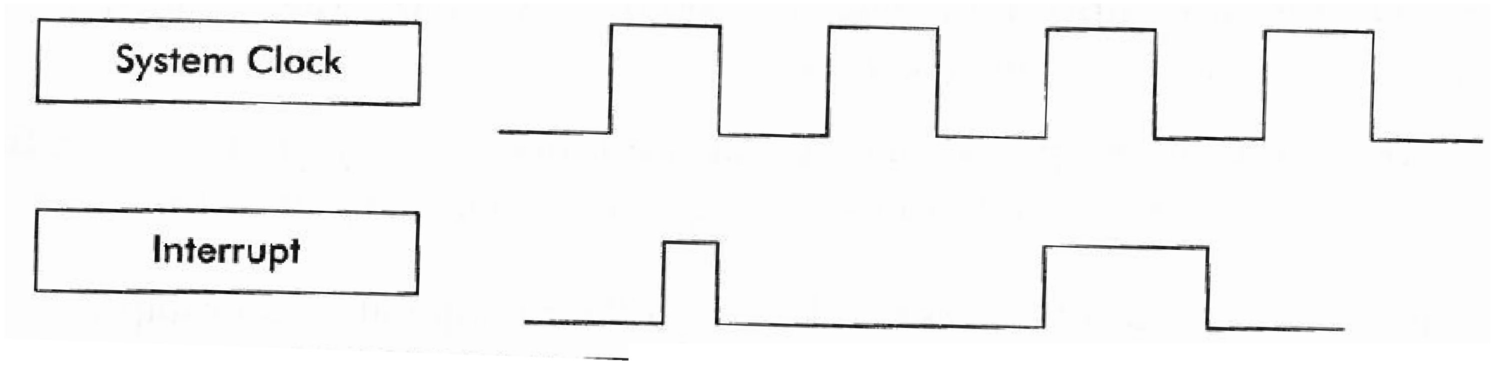
Types of Interrupts

- Interrupts may also be classified as:
  - Periodic interrupts
  - Aperiodic interrupts
  - Synchronous interrupts: those aligned exactly in phase with the system clock
  - Asynchronous interrupts: the interrupt source is not in phase with the system clock



Synchronous interrupt

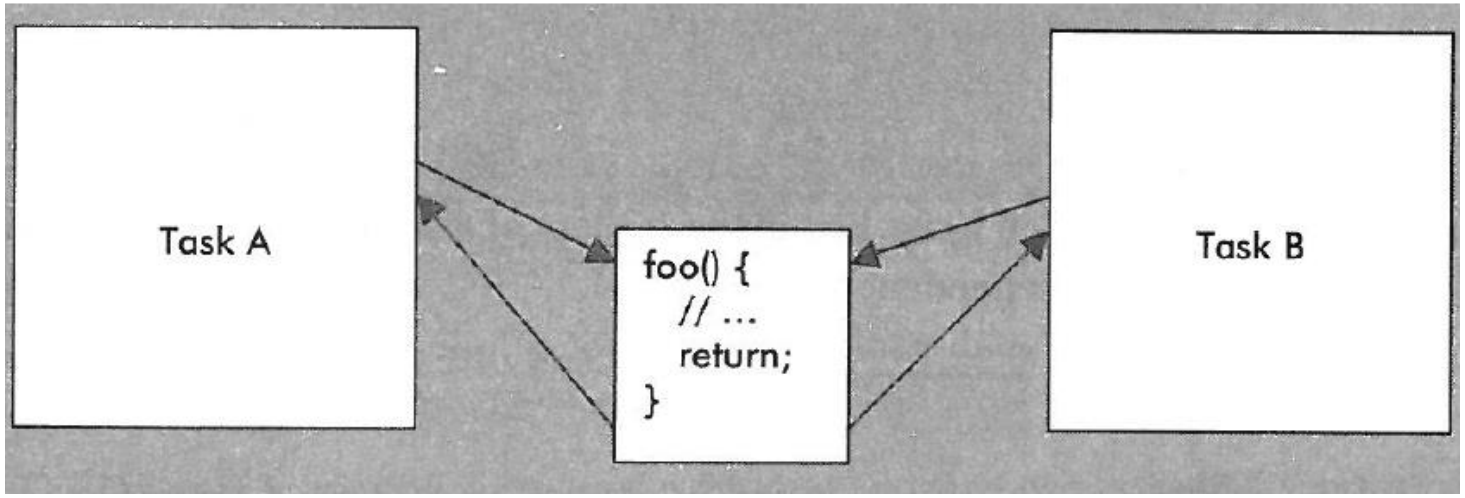
Types of Interrupts



Asynchronous interrupt

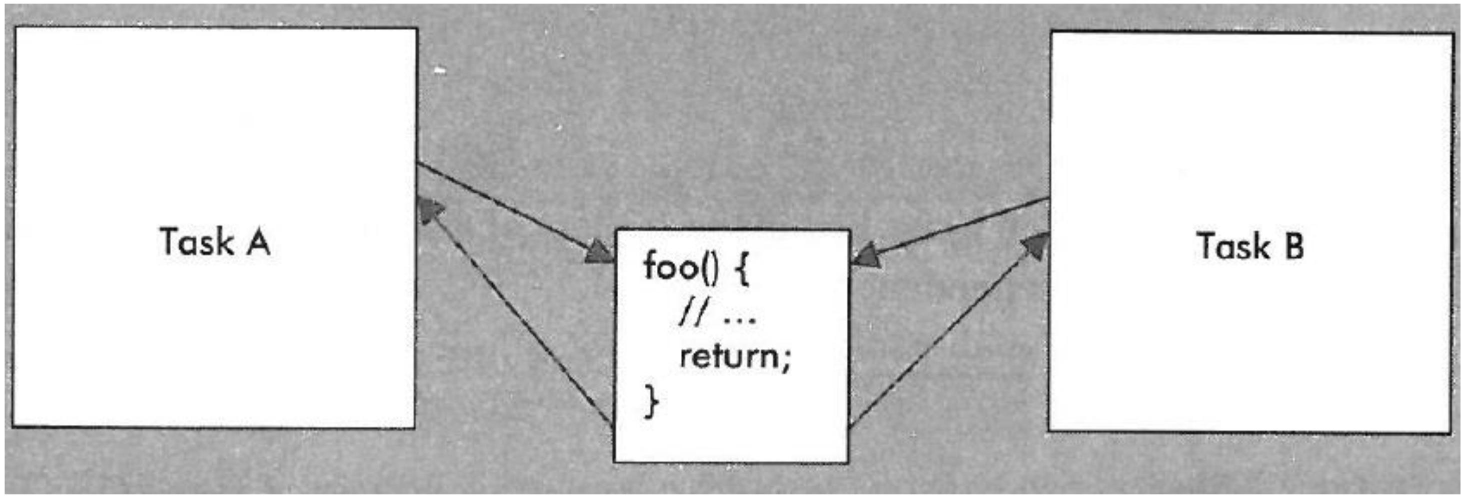
- Asynchronous interrupts are more common than synchronous ones
- Interrupt latency: the time that elapses from the instant the interrupt was raised to the first instruction of the interrupt service routine being executed

Re-entrancy

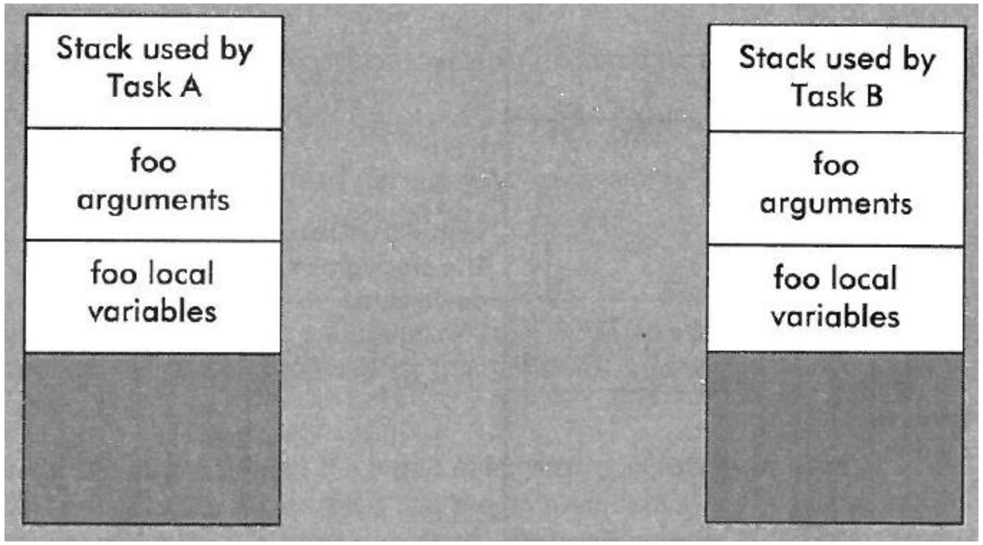


- As discussed before, most embedded systems contain concurrently running applications or tasks
- Often such tasks share common code or functions
- Here, tasks A and B both execute foo()
  - But foo() is executed in the contexts of A and B separately, i.e., in their own stacks

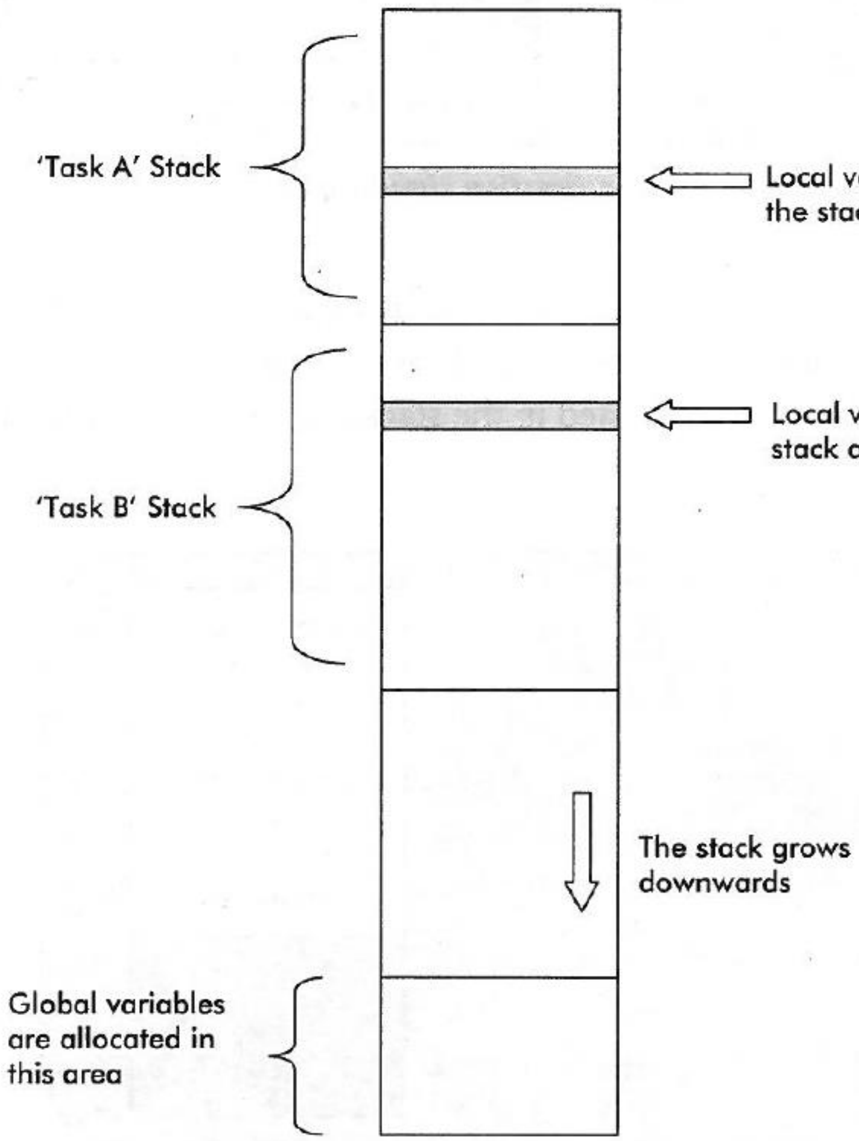
Re-entrancy



- Local variables defined by foo() and the arguments passed to it are allocated to the respective stacks of Tasks A and B

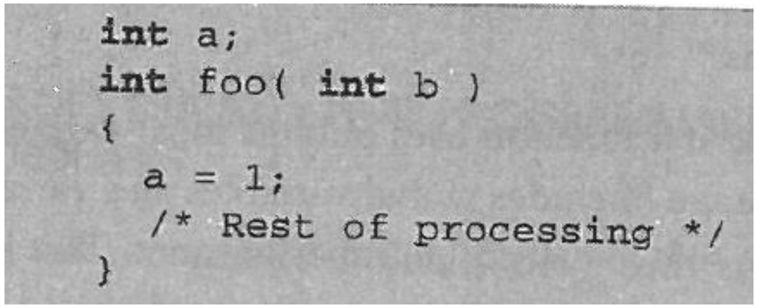


Memory Layout for Reentrant Code


'Task A' Stack
Local variable 'a' allocated in the stack allocated to Task A
Local variable 'a' allocated in the stack allocated to Task B
'Task B' Stack
The stack grows downwards
Global variables are allocated in this area

- When the local variable 'a' of foo() in the context of A is changed, it does not affect the copy of 'a' in the context of B
- The two copies are in different locations of the memory

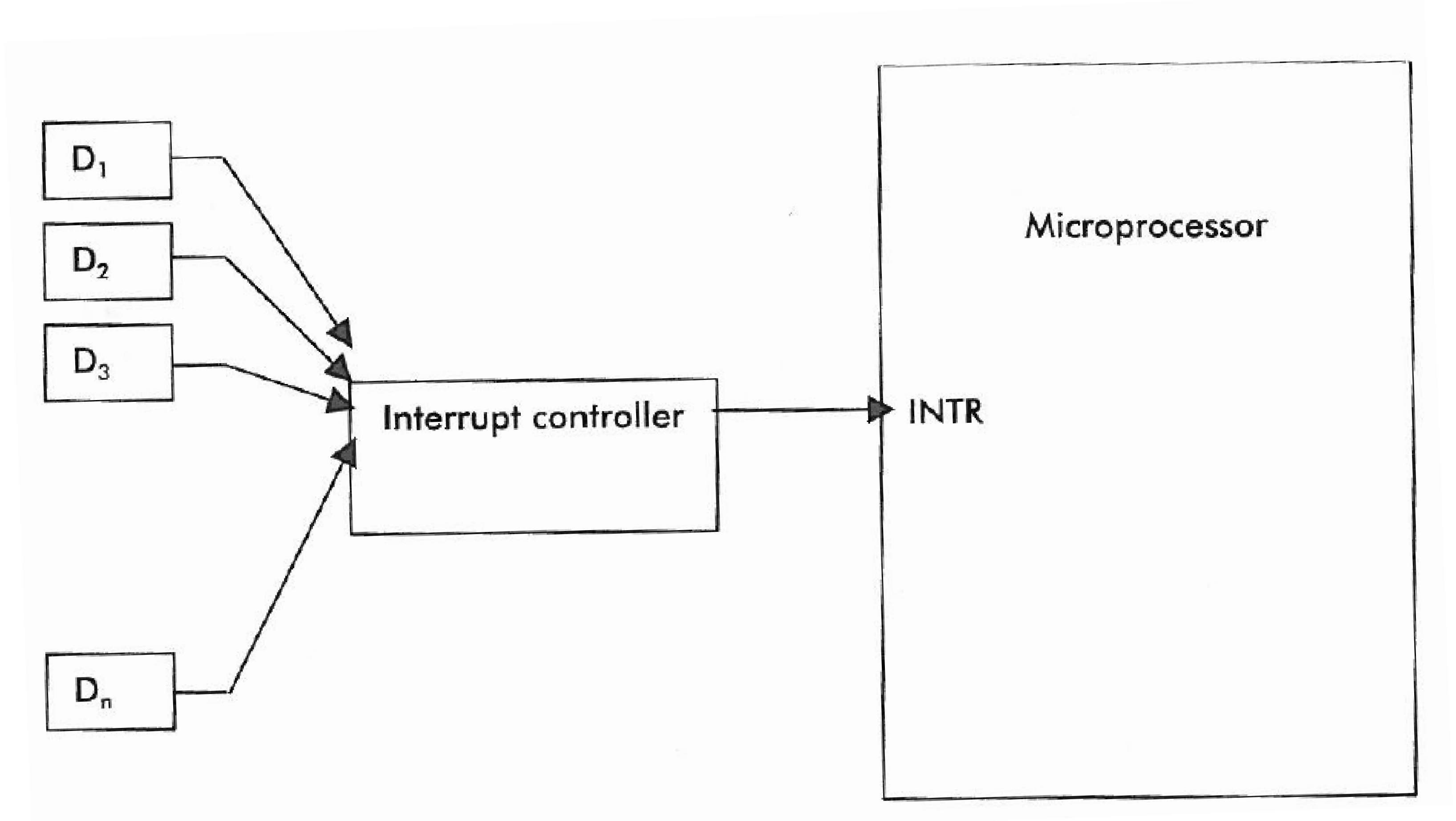
Memory Layout for Reentrant Code



```
int a;
int foo( int b )
{
  a = 1;
  /* Rest of processing */
}
```

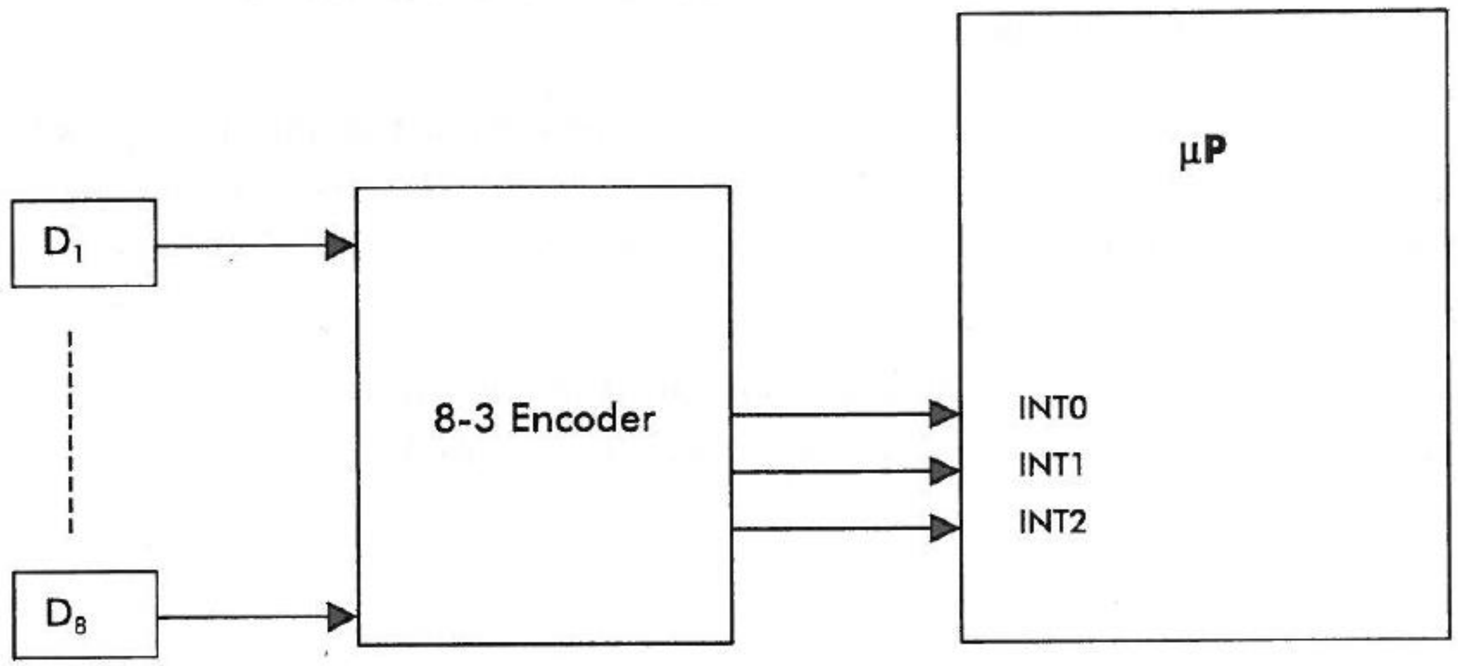
- However, if a shared function has global variables, then two tasks will access the same copy of this variable
- In such cases, some mutual exclusion technique needs to be applied (as discussed previously)
- Hence, functions that contain only local variables and do not use any variables with static storage (includes global variables and variables with static storage specification) are *safe* and are referred to as *reentrant code*

Interrupt Priority and Interrupt Controllers



- Microprocessors and microcontrollers usually have only few interrupt pins
- When more devices need to be connected to them, a *programmable interrupt controller* is typically used

Interrupt Controllers



- In the simplest case, an interrupt controller might be a x-y encoder
  - *The example above shows how 8 devices are connected to a microprocessor with 3 pins*
- More complex interrupt controllers provide features such as assigning different priorities to different interrupts

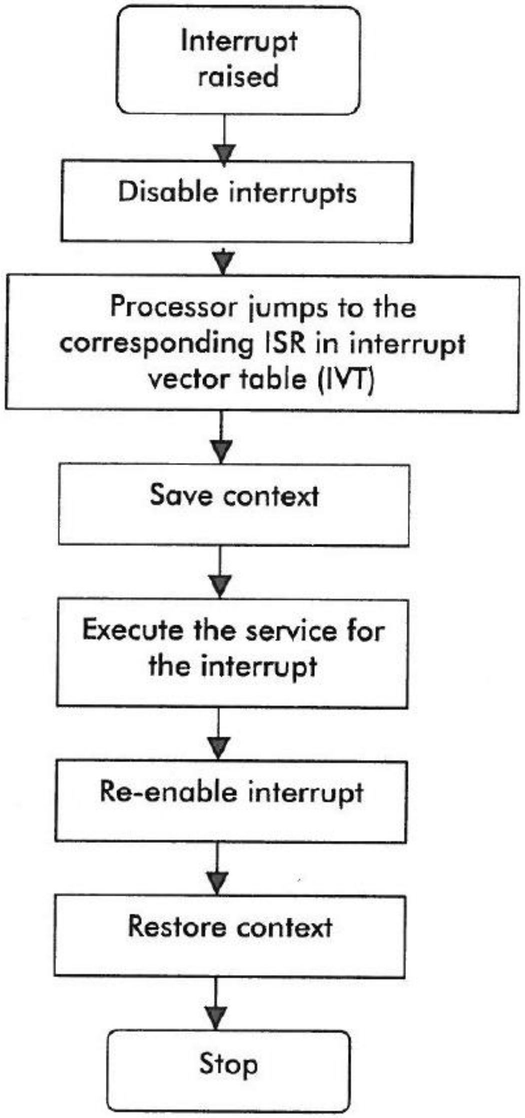
Masking of Interrupts

- Masking an interrupt *disables* it
- When the microprocessor works on a very high-priority interrupt, it is often important that it shouldn't be preempted by other interrupts
- In such cases, lower priority interrupts are masked

Interrupt Service Routines (ISRs)

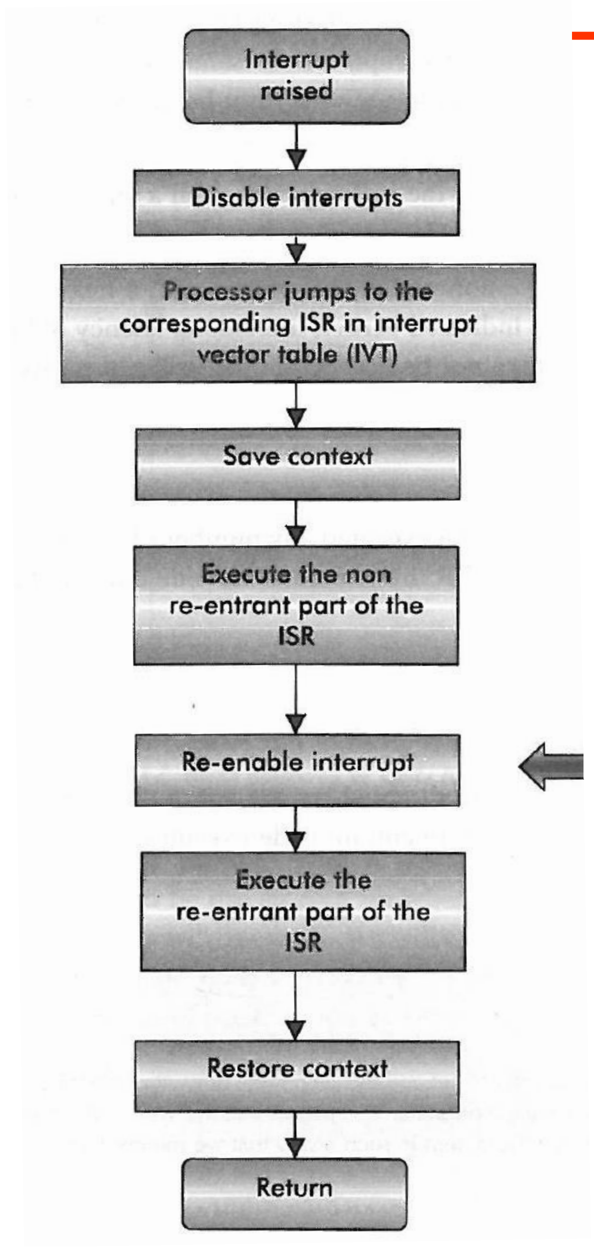
- ISR – the code that gets executed when an interrupt is raised
- Types of ISRs
  - *Non-nested handlers*: when this ISR is executing, it cannot be preempted by another interrupt
  - *Nested handlers*: such handlers may be preempted by the ISRs of other interrupts. Such interrupts should therefore be carefully coded in order to avoid *race* conditions (i.e., where data consistency is lost because of lack of synchronization between multiple tasks which access the same data)

Non-nested Handlers


Interrupt raised
Disable interrupts
Processor jumps to the corresponding ISR in interrupt vector table (IVT)
Save context
Execute the service for the interrupt
Re-enable interrupt
Restore context
Stop

- Such ISRs disable other interrupts (which remain disabled until the ISR enables them)
- The issue of contexts and why/how they are saved was discussed in the first part of the lecture

Nested Interrupt Handlers


Interrupt raised
Disable interrupts
Processor jumps to the corresponding ISR in interrupt vector table (IVT)
Save context
Execute the non re-entrant part of the ISR
Re-enable interrupt
Execute the re-entrant part of the ISR
Restore context
Return

- Here, the code has to be separated into
  - Non-reentrant code
  - Reentrant code
- As a result, interrupts only have to be disabled when the non-reentrant code executes (this reduces interrupt latency)

How interrupts work?

- How does the processor know that an interrupt has occurred?
  - The processor polls for it
  - Here, the processor polls the interrupt pins every system cycle for a fraction of the cycle (this is different from device polling and is far less expensive in terms of overheads)
- Once an interrupt has been raised, how does the processor know which ISR to execute?
  - In simple cases, the address location of the ISR may be hardwired
  - A more widely used scheme is called *vectoring*

Vectoring

- The device or interrupt controller asserts the interrupt (INTR) pin of the microprocessor and waits for an *interrupt acknowledgement* (INTA)
- On receiving the INTA, the device places a 8-bit or 16-bit data on the data bus of the processor
- Each interrupt source (or device) is assigned a unique number
- Hence, this data is used to branch to an appropriate ISR
- The array/vector of interrupt handlers is known as the "interrupt vector table" (IVT)

Passing arguments

- How can arguments be passed to an ISR?
- *Shared memory*: A part of the memory space of the processor is shared with the device. The device can write data at a previously-agreed upon location and in a predefined format and then raise an interrupt
- *Direct memory access (DMA)*: The device writes data directly to the address space of the processor. Once done, it invokes a "DMA Transfer Complete" interrupt

Summary

- In this lecture we have discussed
  - Principles of concurrent systems and resource sharing
  - Protocols to avoid race conditions
  - How to interface embedded systems with external devices
  - How interrupts and interrupt service routines work