Components of RTOS

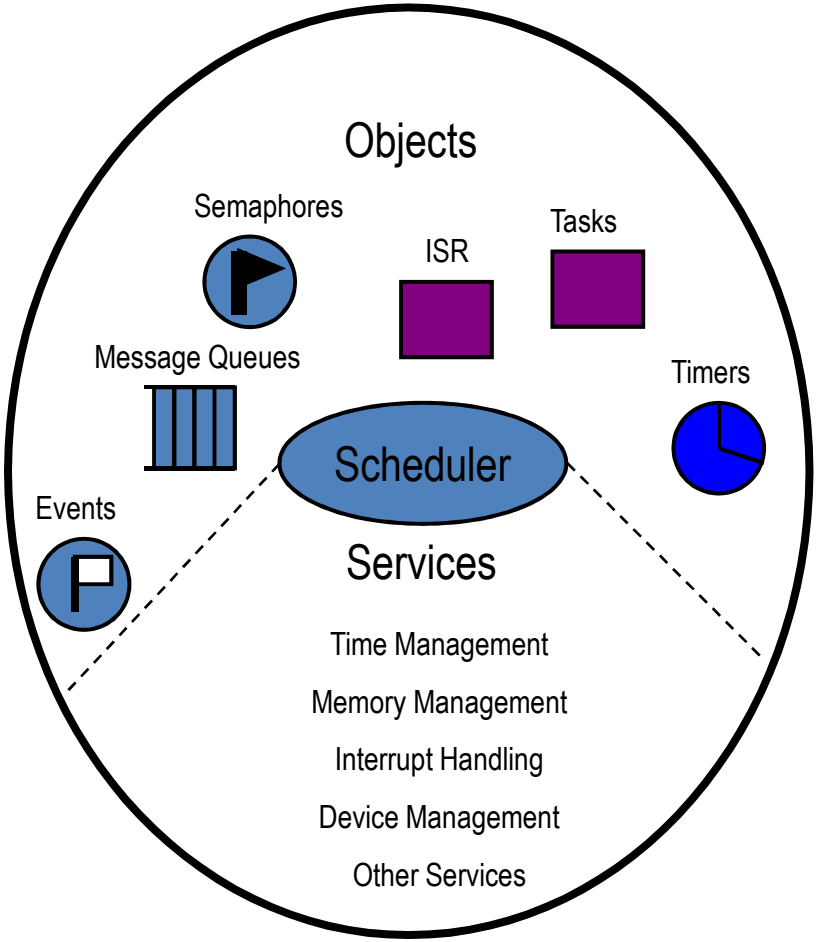
Dr.R.Sundaramurthy

Department of EIE
Pondicherry Engineering College

sundar@pec.edu

Components in an RTOS

- An RTOS has typically the following components:
  - Scheduler
    - Determines which task is running at each time instant
  - Objects
    - Special constructs like tasks, semaphores or message queues
  - Services
    - operations that the kernel performs on objects



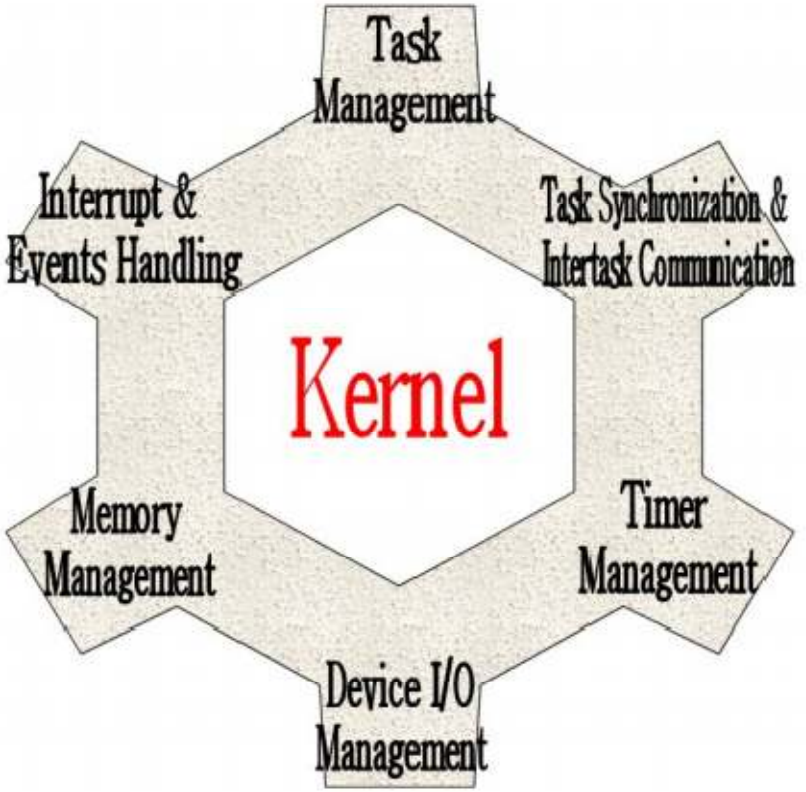
RTOS Kernel Services

Kernel is the vital and central component of an operating system.

It provides the following services

- Task Management
- Task & Resource Synchronization
- InterTask Communication
- Timer Management
- Memory Management
- Interrupts & Events Handling
- Device I/O Management

It provides an interface for software applications to use the services (API)



Part- 1

Task Management

Dr.R.Sundaramurthy.,M.E.,Ph.D.,
sundar@pec.edu

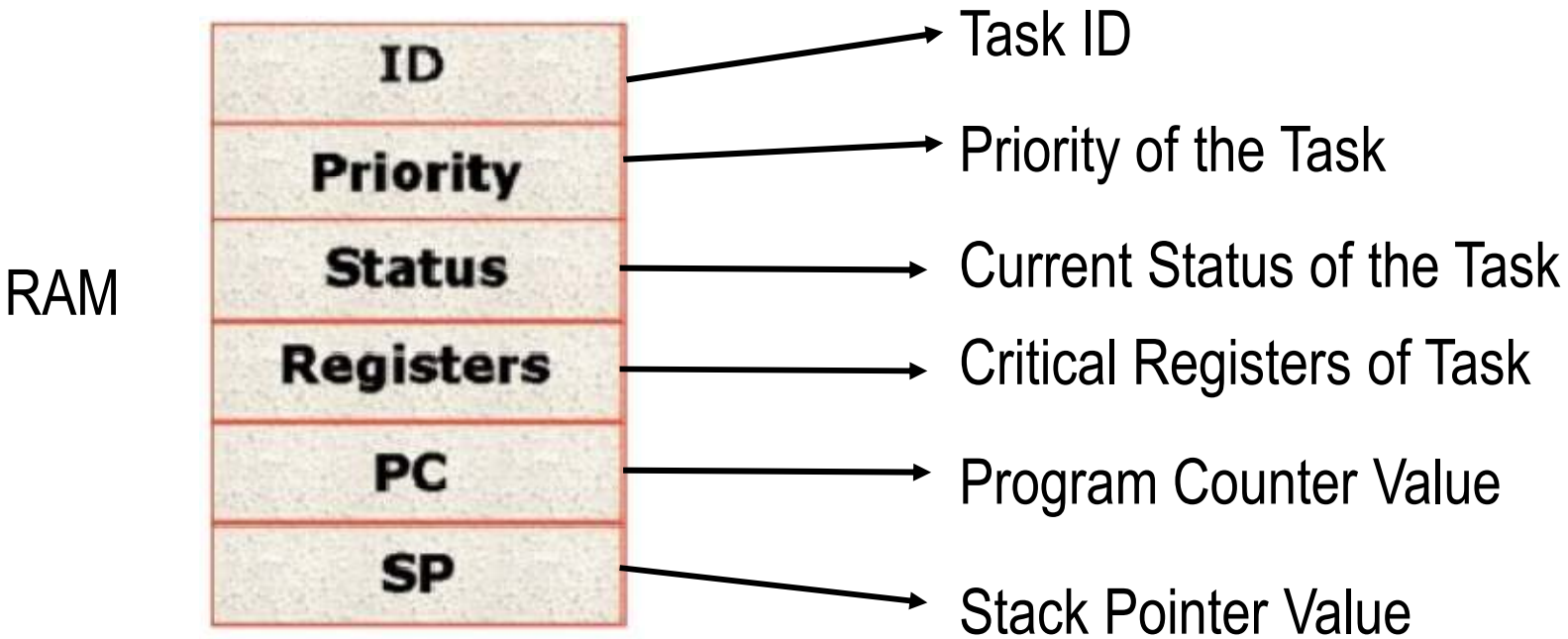
Task Management

- Task management allows programmers to design their software as a number of separate “chunks” of codes with each handling a distinct goal and deadline.
- This service encompasses mechanism such as **scheduler** and **dispatcher** that creates and maintain task objects.

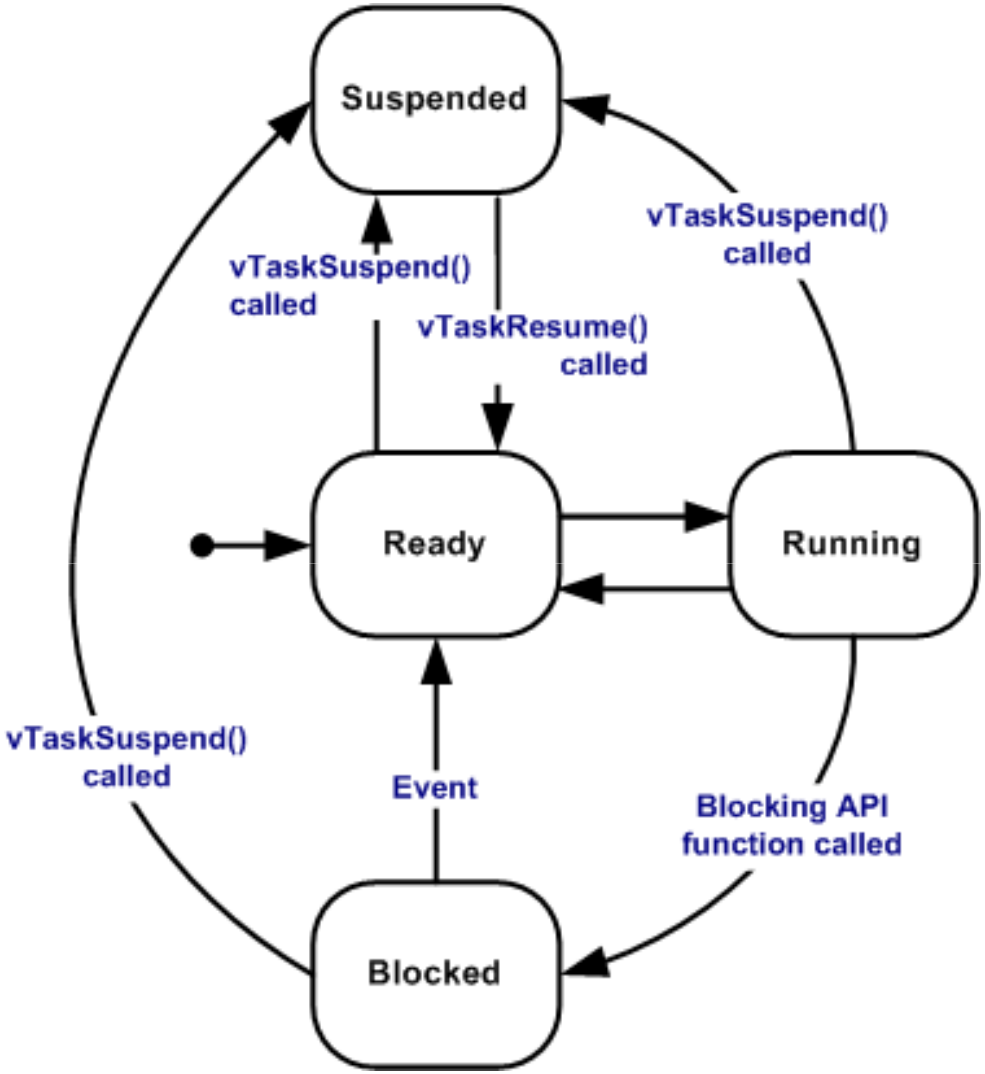
Task

- To achieve concurrency in real-time application program, the application is decomposed into small, schedulable, and sequential program units known as “Task”.
- In real-time context, task is the basic unit of execution and is governed by three time-critical properties; release time, deadline and execution time.
  - **Release time** refers to the point in time from which the task can be executed.
  - **Deadline** is the point in time by which the task must complete.
  - **Execution time** denotes the time the task takes to execute.
- A task object is defined by the following set of components:
  - **Task Control block** - Task data structures residing in RAM and only accessible by RTOS
  - **Task Stack** - Data defined in program residing in RAM and accessible by stack pointer
  - **Task Routine** - Program code residing in ROM

Typical Task Control Block



Task States



1.Running

2.Ready

3.Blocked

4.Suspended

Task States

- A task can exist in one of the following states:

Running

- When a task is actually executing it is said to be in the Running state.
- It is currently utilising the processor.
- If the processor on which the RTOS is running only has a single core then there can only be one task in the Running state at any given time.

Ready

- Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.

Blocked

- A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event.
- For example, if a task calls vTaskDelay() it will block (be placed into the Blocked state) until the delay period has expired - a temporal event.
- Tasks can also block to wait for queue, semaphore, event group, notification or semaphore event.
- Tasks in the Blocked state normally have a 'timeout' period, after which the task will be timeout, and be unblocked, even if the event the task was waiting for has not occurred.
- Tasks in the Blocked state do not use any processing time and cannot be selected to enter the Running state.

Suspended

- Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, but tasks in the Suspended state do not have a time out.
- Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.

Task Management API

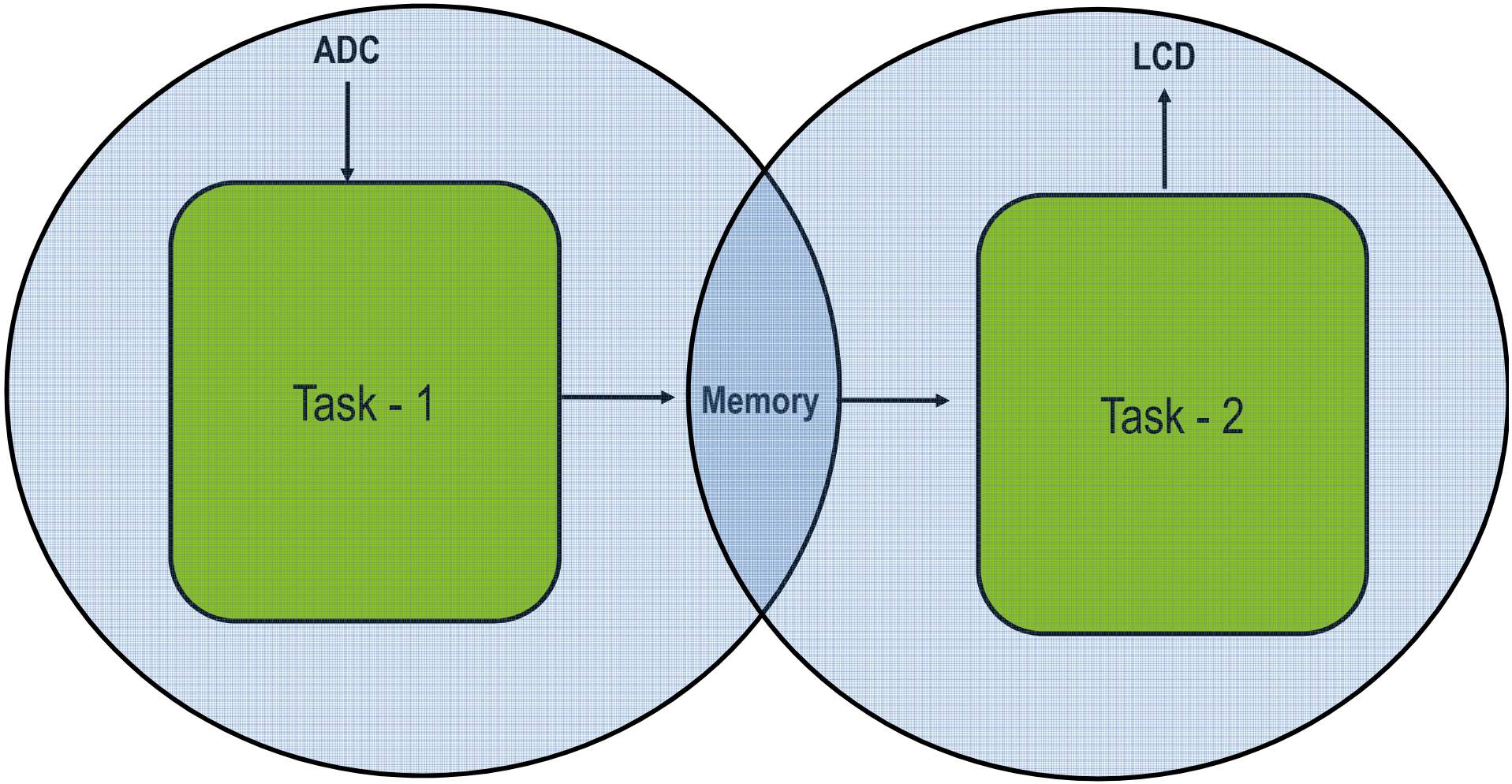
- RTOS offers List of API (Application Programming Interface) which are Nothing but function calls to Manage Tasks.
- Some of the Task Magement API are
  - Create a Task ⇒ xTaskCreate
  - Delete a Task ⇒ vTaskDelete
  - Suspend a Task ⇒ vTaskSuspend
  - Resume a Task ⇒ vTaskResume
  - Change Priority of Task ⇒ vTaskPrioritySet

Part- 2

Task & Resource Synchronization

Dr.R.Sundaramurthy.,M.E.,Ph.D.,
sundar@pec.edu

Task Synchronization



Task 1 = Acquires ADC data and Writes in Memory

Task 2 = Fetch data from Memory and Writes in LCD

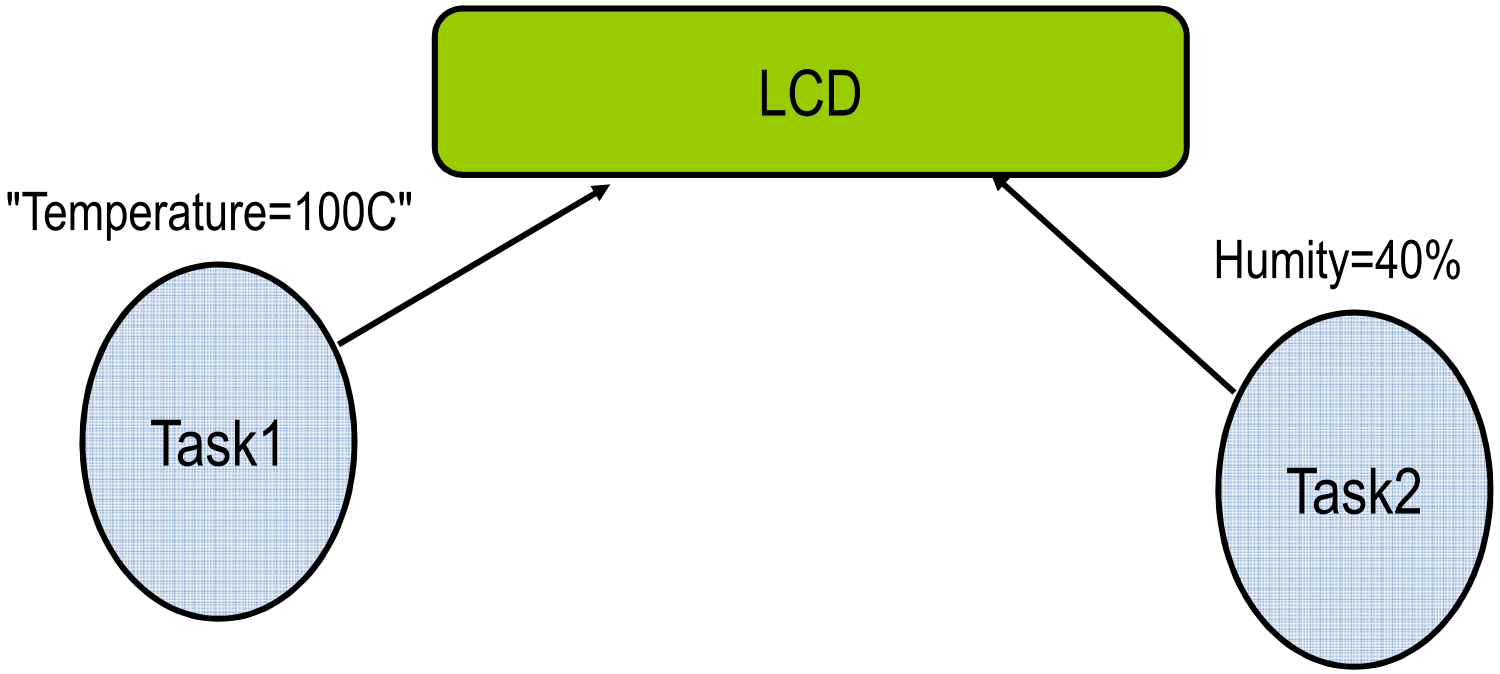
Task Synchronization

Task 1 = Acquires ADC data and Writes in Memory

Task 2 = Fetch data from Memory and Writes in LCD

- In this case Task2 has to wait for Task1 to complete the operation.
- When Task1 informs Task2 that It has acquired ADC data and written to Memory, Task 2 then begins its operation.
- There should be a mechanism for Both Tasks coordinating each other to achieve a common objective.
- This is called **Task Synchronization**

Resource Synchronization



Two Tasks are Trying to Display Text in a Shared Resource (LCD)

Task 1 to Display - "Temperature=100C"

Task 2 to Display - "Humidity=40%"

Resource Synchronization

Two Tasks are Trying to Display Text in a Shared Resource (LCD)

Task 1 to Display - "Temperature=100C"

Task 2 to Display - "Humidity=40%"

•If there is no Synchronization in using the shared resource, the result will be a garbled message - "Temphumieratidirurey is is 100 40C%"

•To Access a Shared resource, there should be a Mechnism so that there is discipline.

•**This is called Resource Synchronization**

Semaphores

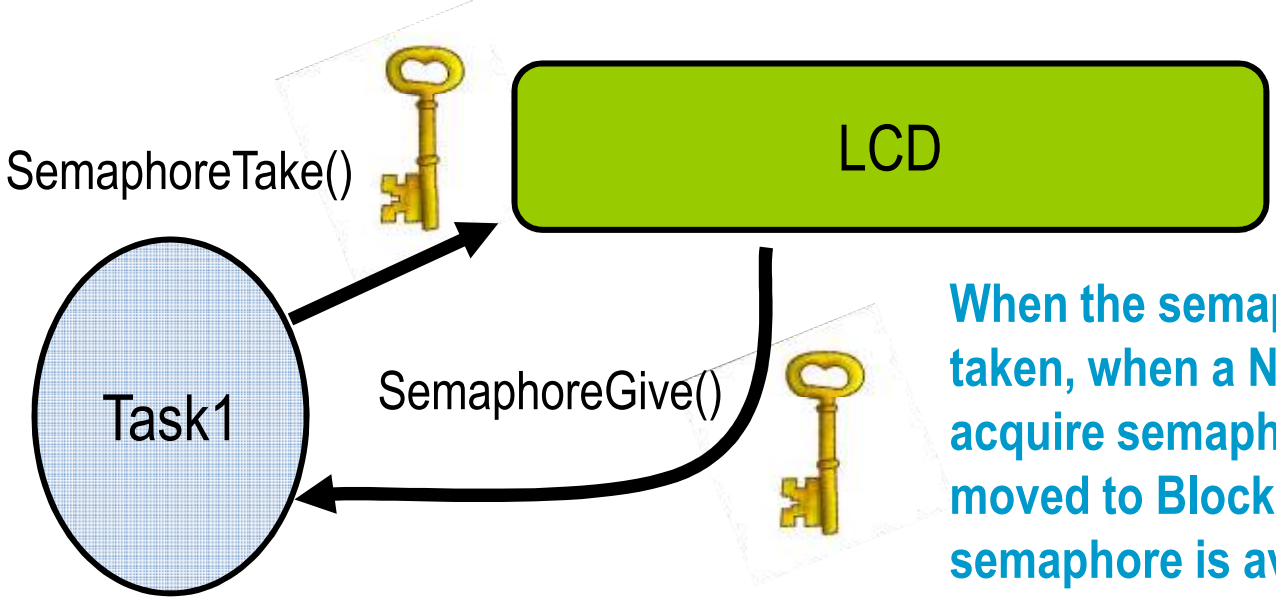
- Semaphores are kernel objects used for both **Resource and Task Synchronization**.
- A *semaphore* is a kernel object that one or more tasks can acquire or release for the purpose of synchronization or mutual exclusion
- Semaphores are of Three types:
  - Binary semaphore
  - Counting semaphore
  - MuTex

Binary Semaphore

- Semaphore is Like a Key to access a shared resource.
- Binary Semaphore can have two Possible Values - 0 or 1
- When a Task Takes a Semaphore, Its value Become 0 (Not Available).
- When the Task Gives the Semaphore, Its value become 1 (Available)

How to Use Binary Semaphore

1. A Task first Takes a Semaphore (Value = 0)
2. Gain access to a Shared Resource
3. Task Uses the shared Resource
4. Task Gives the Semaphore (Value = 1)



When the semaphore is already taken, when a New Task try to acquire semaphore, then It will be moved to Blocked state, till the semaphore is available.

Counting Semaphore

- Semaphore is Like a Key to access a shared resource.
- Counting Semaphore can have Values from

  0 to "User Defined Value(N)"
- When a Task **Takes** a Semaphore, Its value is decremented by 1 (N-1).
- When N= 0, Semaphore is no longer Available.
- When the Task **Gives** the Semaphore, Its value is incremented by 1 (N+1).

Mutex

- MutEx (Mutual Exclusion) is a special Binary Semaphore can have Values from 0 or 1.
- MutEx can have two Possible Values - 0 or 1
- When a Task Takes a MutEx, Its value Become 0 (Not Available).
- When the Task Releases the MutEx, Its value become 1 (Available).
- MutEx can be used for Task & Resource Synchronization like Binary Semaphores.

Binary Semaphore Vs MutEx

- Mutex will have a ownership. The Task which acquires the MutEx becomes the owner.
- Only the Owner of a MutEx can release a MutEx, Not any other Task. Binary Semaphore can be released by any Task that did not originally acquired it.
- Owner can acquire a mutex multiple times in the locked state.If the owner locks it 'n' times , the owner has to release it 'n' times.
- A Task owning a MuTex cannot be deleted.
- A Mutex supports priprity Inheritence or Priority ceiling to avoid priority Inversion problem.

Priority Inversion by Binary Semaphore

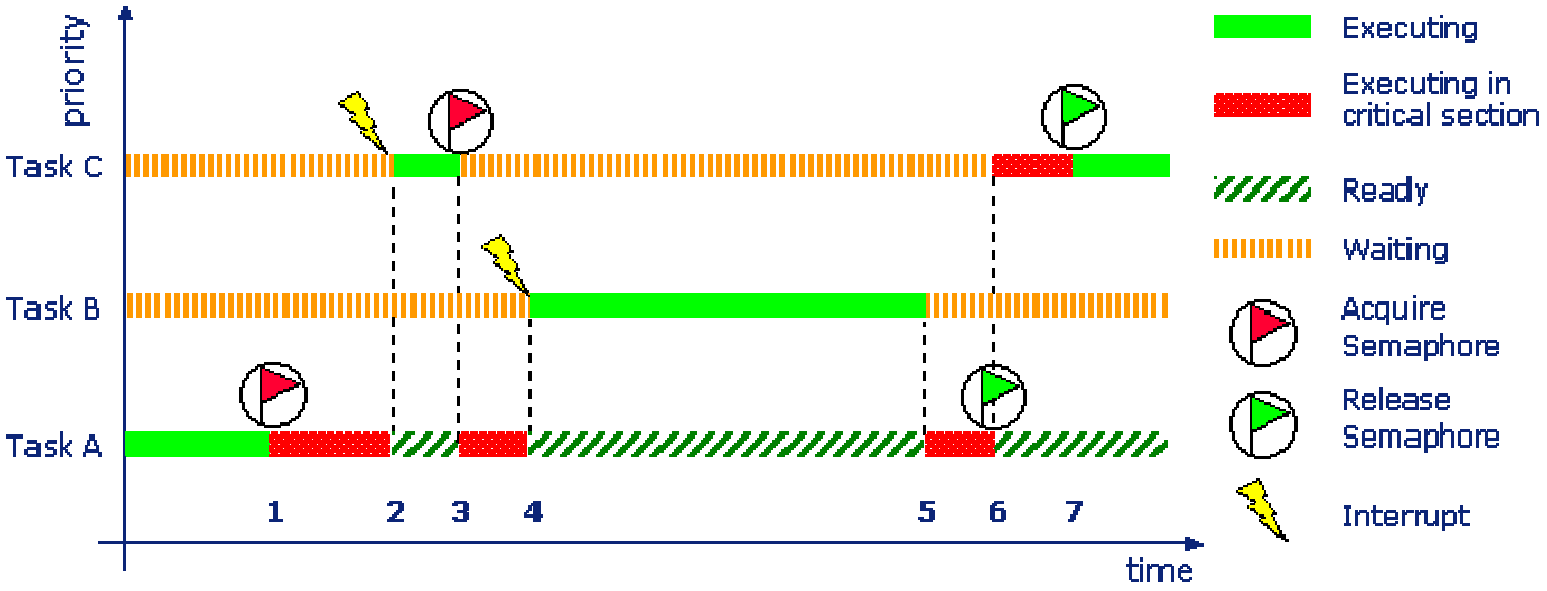


Figure 5. Priority inversion with a binary semaphore

1. Task A is executing and wants to access the resource. To be able to do this Task A must first acquire the semaphore.
2. Task C is made ready and preempts Task A still accessing the resource.
3. Task C also wants to access the resource. So Task C tries to acquire the semaphore, but the semaphore is already busy by Task A, so in this case Task C decides to wait until the semaphore is being released.
4. Task A continues to execute, but is preempted by medium-priority Task B, and Task B continues execute for a relatively long time.
5. Task B stops to execute and Task A can now finish using the shared resource.
6. Task A releases the semaphore, so that Task C can finally acquire the semaphore and get access to the shared resource.
7. Task C releases the semaphore when it has finished using the shared resource

Priority Inheritence

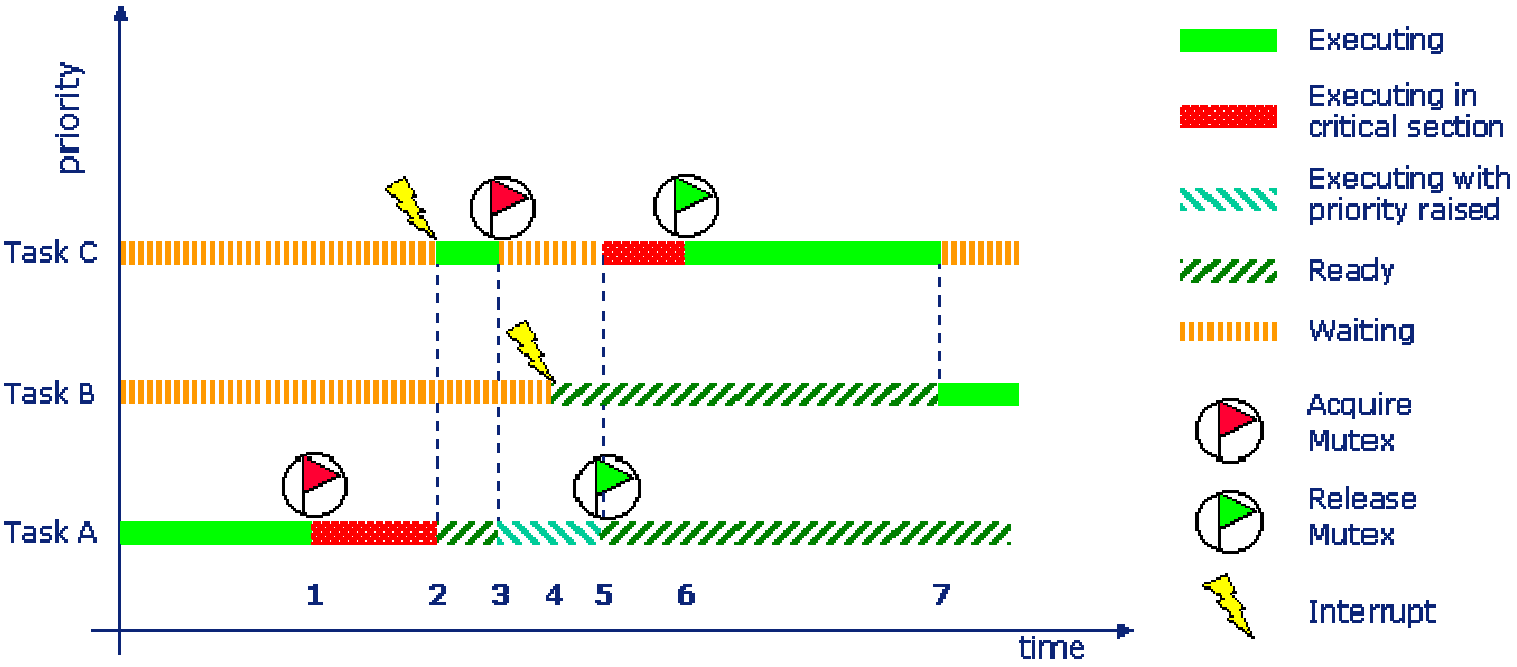


Figure 6. Mutex semaphore, Priority inheritance protocol

1. Task A is executing and wants to access the resource. To be able to do this Task A must first acquire the mutex.
2. Task C is made ready and preempts Task A still accessing the resource.
3. Task C also wants to access the resource. So Task C tries to acquire the mutex, but the mutex is already busy by Task A, so in this case Task C decides to wait until the mutex becomes free again. The priority of Task A will now be raised to the same priority as Task C has. Task A continues to execute.
4. Task B is made ready, but has lower priority than Task A at this moment, so Task A continues to execute.
5. Task A releases the mutex, so that Task C can finally acquire the mutex and get access to the shared resource.
6. Task C releases the mutex when it has finished using the shared resource.
7. Task C finishes to execute and Task B will start to execute.

Priority Ceiling

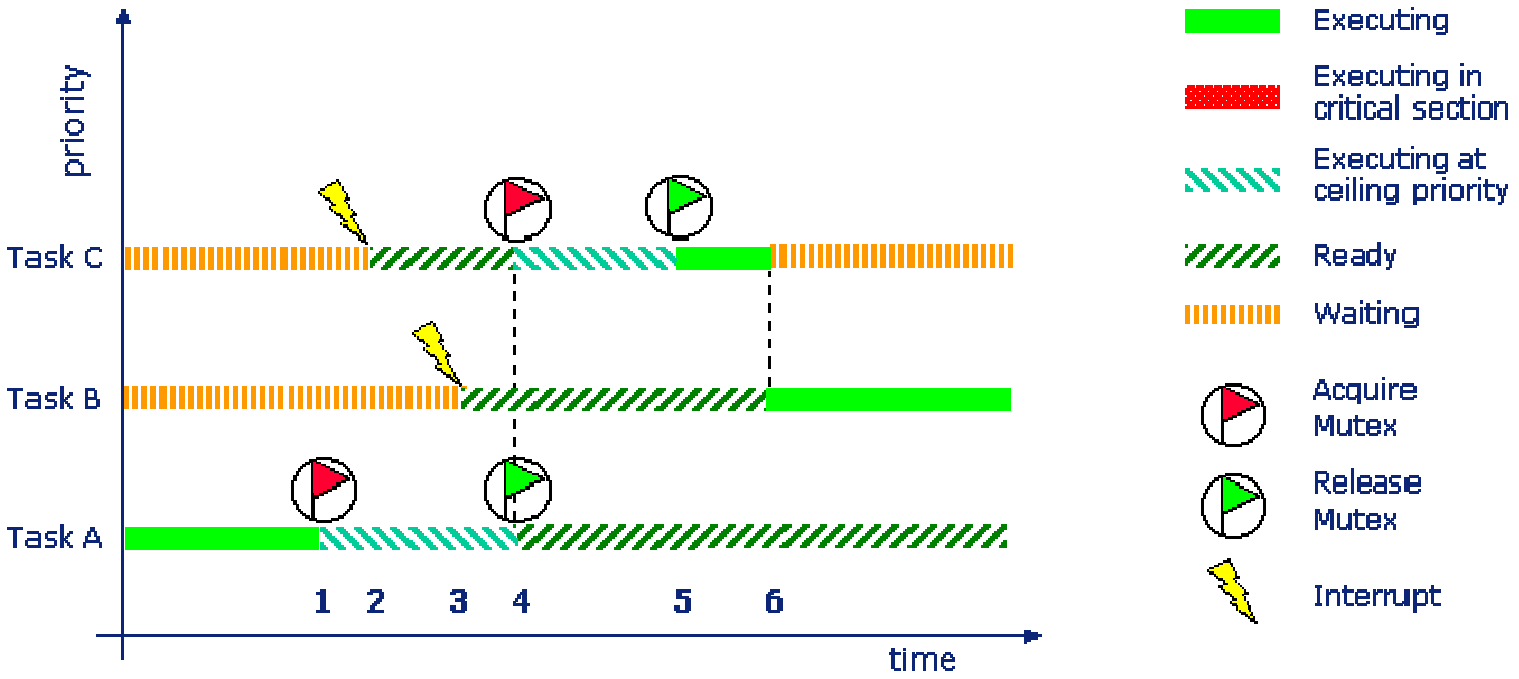


Figure 7. Mutex semaphore, Priority ceiling protocol

Task A is executing and wants to access the resource. To be able to do this Task A must first acquire the mutex. At this moment is also the priority of Task A raised to the ceiling priority, which in this example is higher than the priority of Task C.

1. Task C is made ready but will not preempt Task A as Task A at the moment has higher priority than Task C. Task A will continue to execute.
2. Task B is made ready, but has lower priority than Task A at this moment, so Task A continues to execute.
3. Task A releases the mutex and the priority of Task A is set back to its normal, and then Task A is immediately preempted by Task C, who now will acquire the mutex and the priority of Task C is raised to the ceiling priority.
4. Task C releases the mutex and the priority of Task C is set back to its normal
5. Task C finishes to execute and Task B will start to execute.

Part- 3

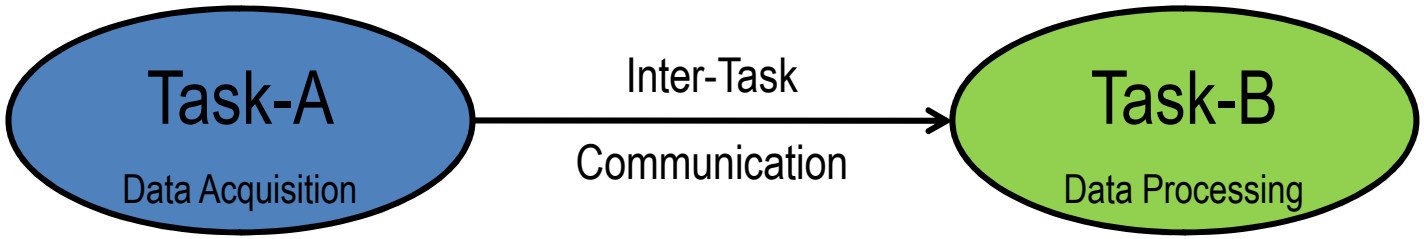
Inter-Task Communication

Inter-Task Communication

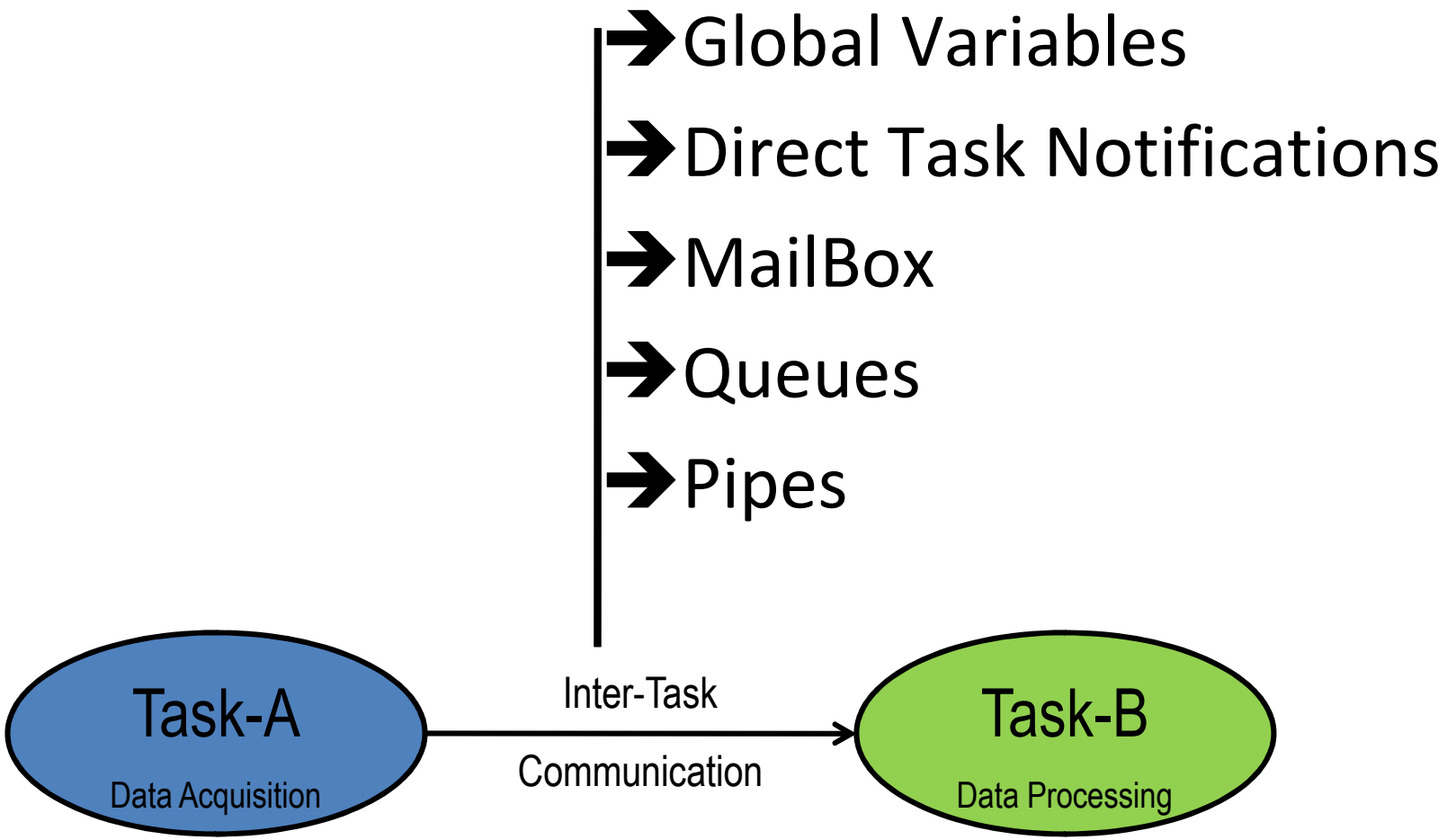
Since only one task can be running at one time, there must be mechanisms for tasks to communicate with one another.

Example :

Task-A is reading data from a sensor at 15 hz. It stores 1024 bytes of data and then needs to **signal** another task (**Task-B**) to take and process the data. Inter-Task communication helps to achieve this



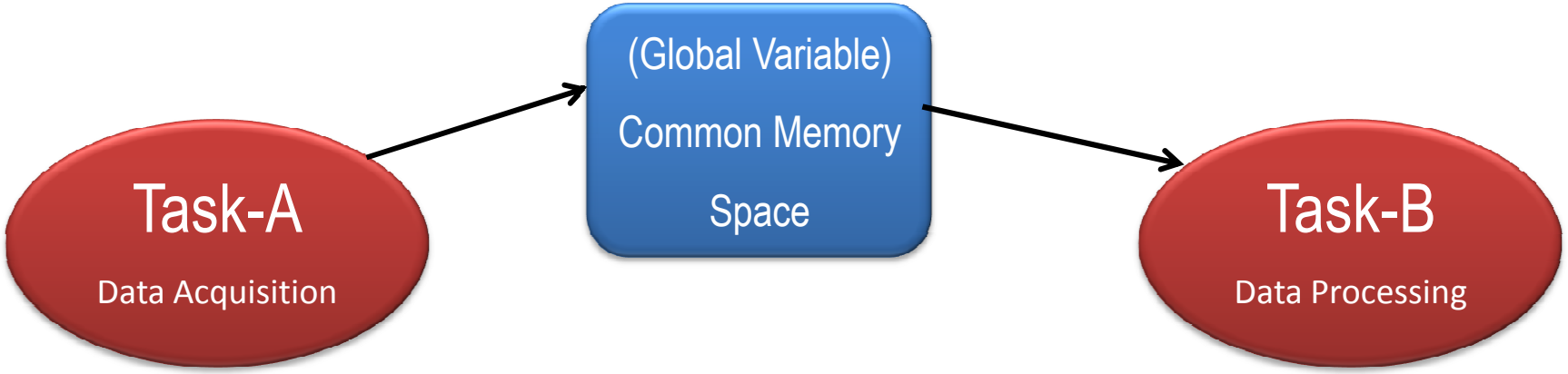
RTOS Objects for Inter-Task Communication



1. Global Variables

- The easiest way to communicate between different Tasks is by using global variables.
- It is the easiest way to communicate between tasks, but most of the time this method has disadvantages.
- For example, if you want to synchronize a task to start when the value of a global variable changes, you must continually poll this variable, wasting precious computation time of the CPU
- The reaction time depends on how often you poll.

Global Variables



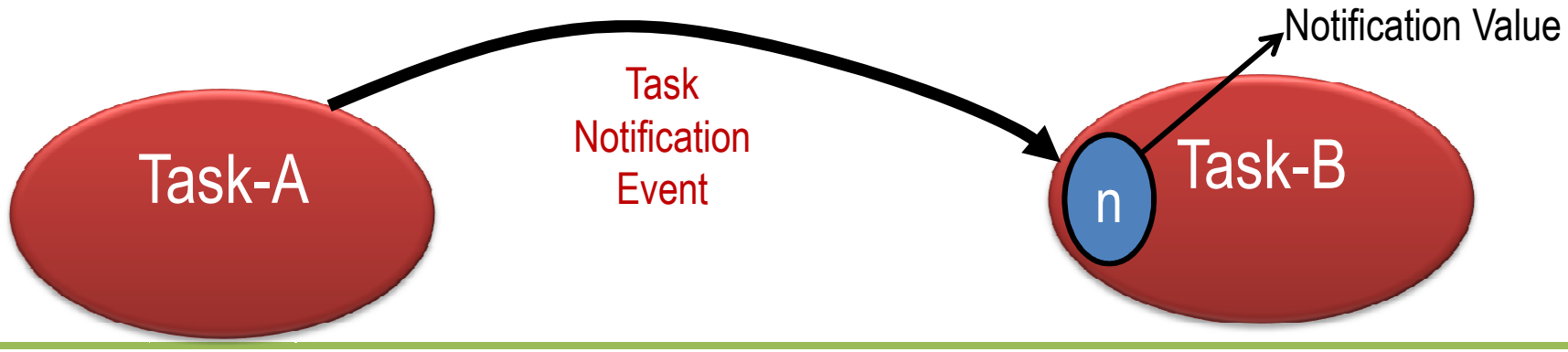
Two Tasks Sharing a common Memory Space.

```
void task1 (void * p)
{
        While(1)
        {
        AcqireData() ;
        finished++;
        }
}
```

```
void task1 (void * p)
{
        While(1)
        {
        while( finished!=1);
        GetBufferData();
        ProcessData();
        }
}
```

2. Direct Task Notifications

- An RTOS task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value.
- Task notifications can update the receiving task's notification value in the following ways:
  - Set the receiving task's notification value without overwriting a previous value
  - Overwrite the receiving task's notification value
  - Set one or more bits in the receiving task's notification value
  - Increment the receiving task's notification value



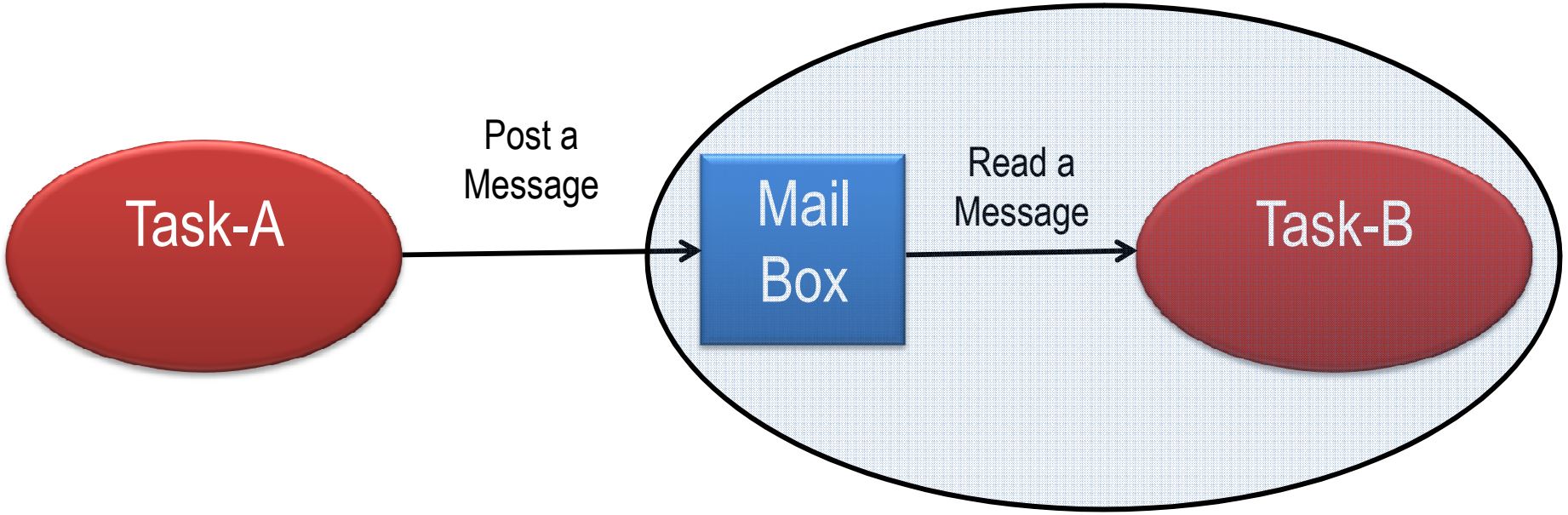
Direct Task Notifications

- Unblocking an RTOS task with a direct notification is 45% faster and uses less RAM than other Inter-task communication objects.
- However RTOS task notifications can only be used when there is only one task that can be the recipient of the event.
- Notifications are sent by Sending Task using
  - xTaskNotify()
  - xTaskNotifyGive()
- Receiving task calls either
  - xTaskNotifyWait()
  - ulTaskNotifyTake()

3. Mailbox

- A mailbox is a data buffer managed by the RTOS and is used for sending a message to a task.
- It works without conflicts even if multiple tasks and interrupts try to access the same mailbox simultaneously.
- RTOS activates any task that is ***waiting for a message in a mailbox,*** the moment mailbox receives new data and, if necessary, switches to this task.

Mailbox



Dr.R.Sundaramurthy.,M.E.,Ph.D.,
sundar@pec.edu

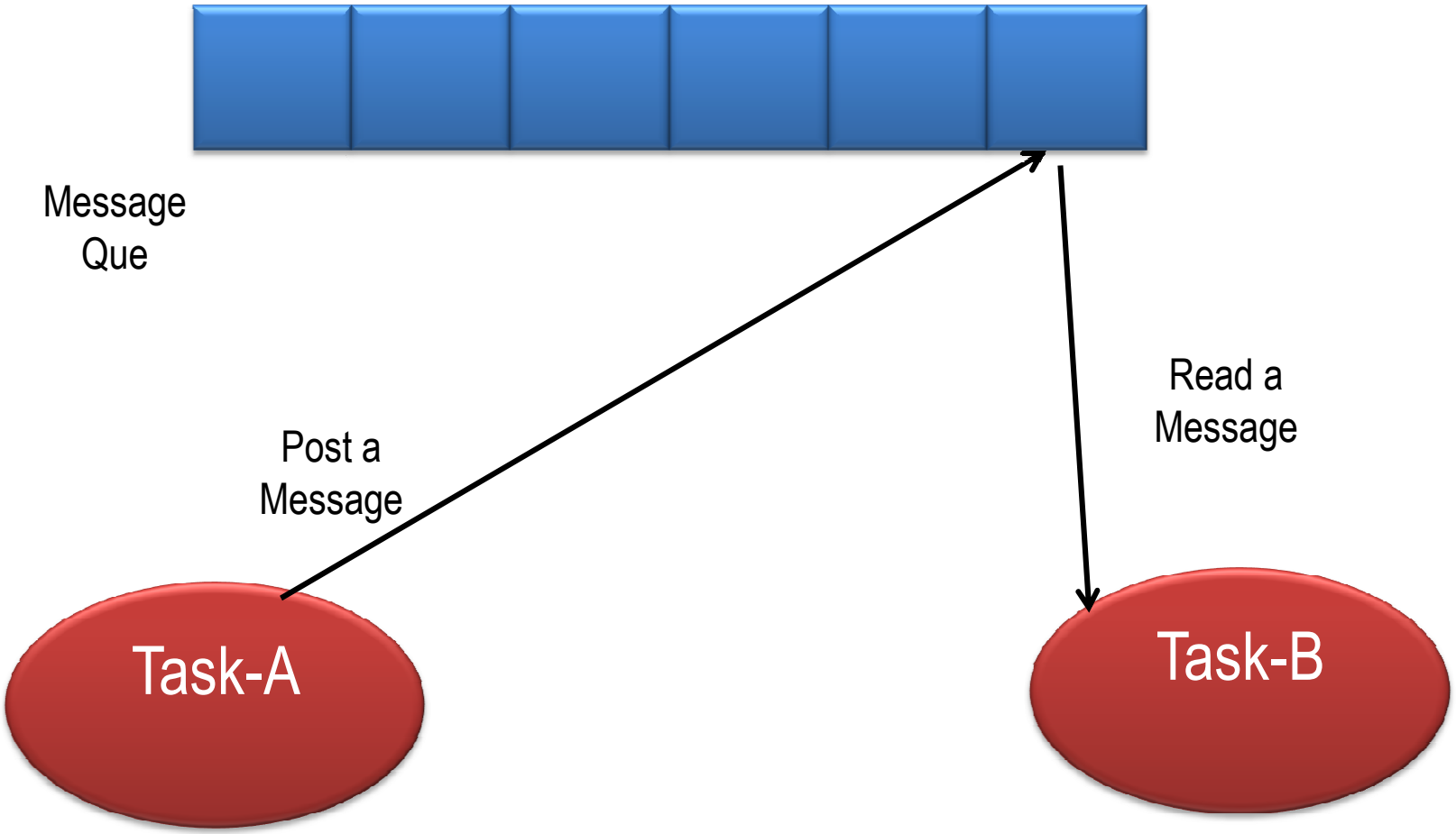
Mailbox

- A mailbox object is just like our postal mail-box
- Someone posts a message in the mailbox and we take out the message.
- A Task can have a mailbox into which others can post a mail.
- Any task or ISR can send the message to mailbox of another Task.

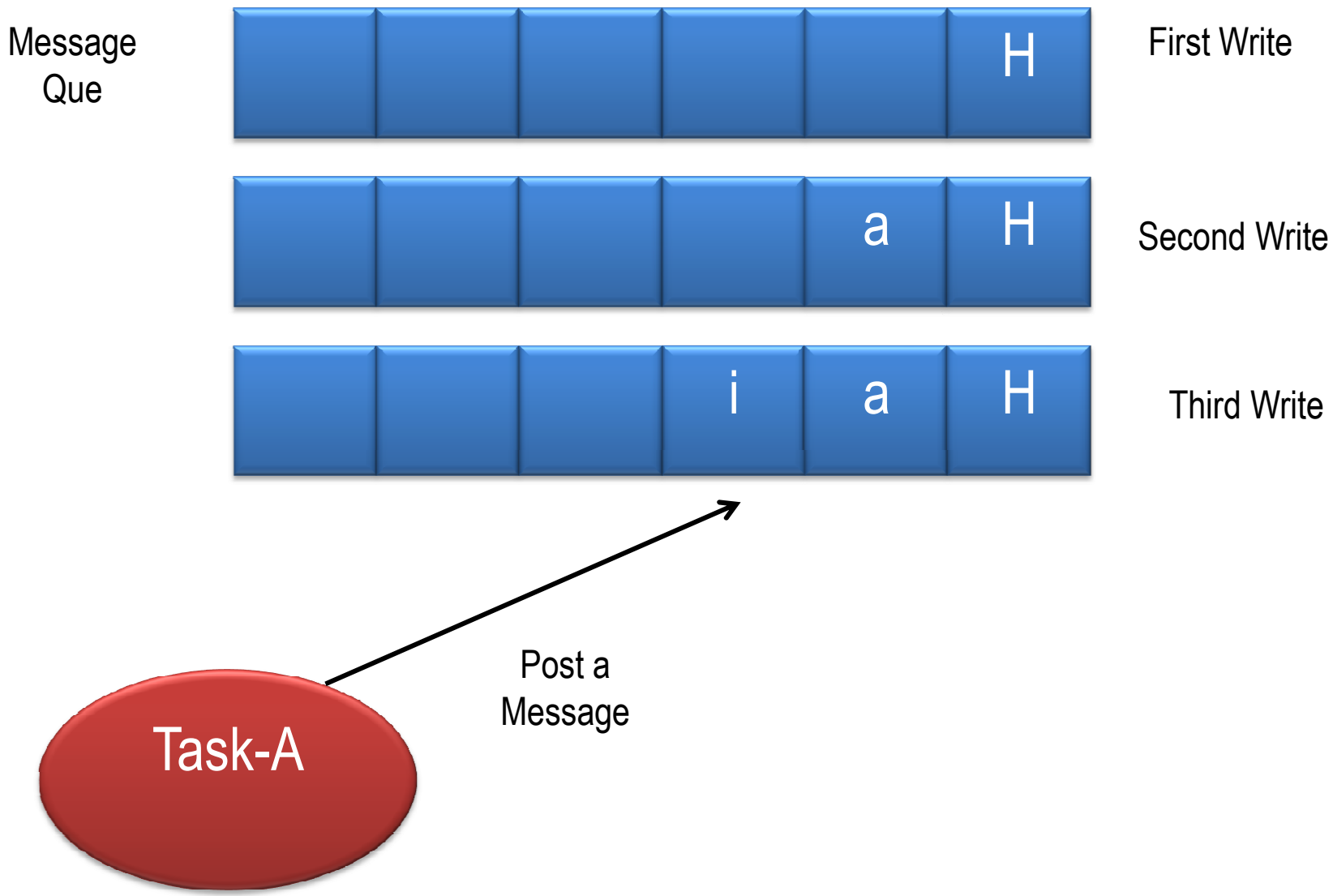
4. Queues

- A queue can be considered as array of mailboxes.
- A task or an ISR deposit the message in the message Queue.
- Other tasks can take the messages.
- At the time of creating the queue, the queue is given the name or ID, queue length , sending task waiting list and receiving task waiting list.

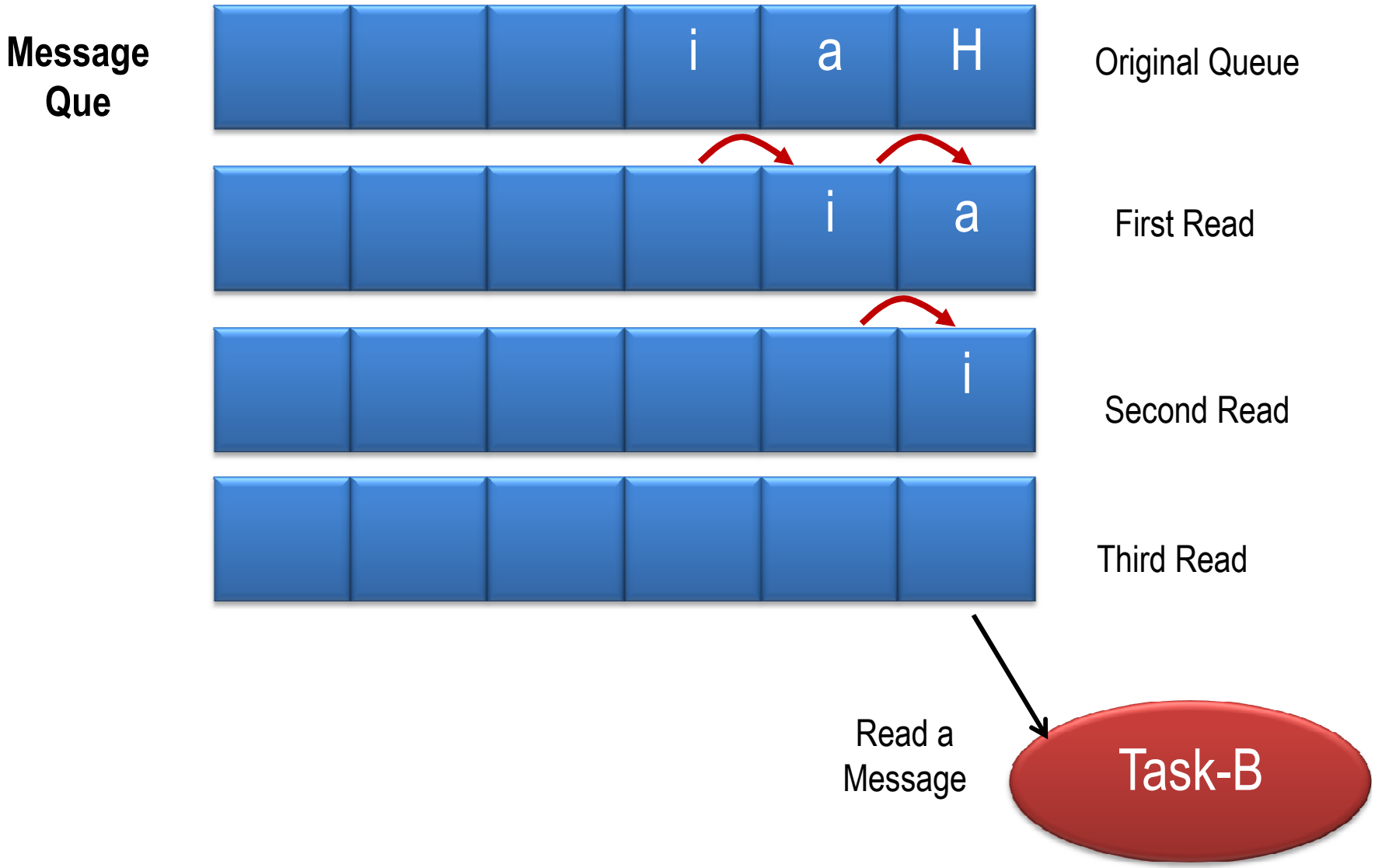
Queues



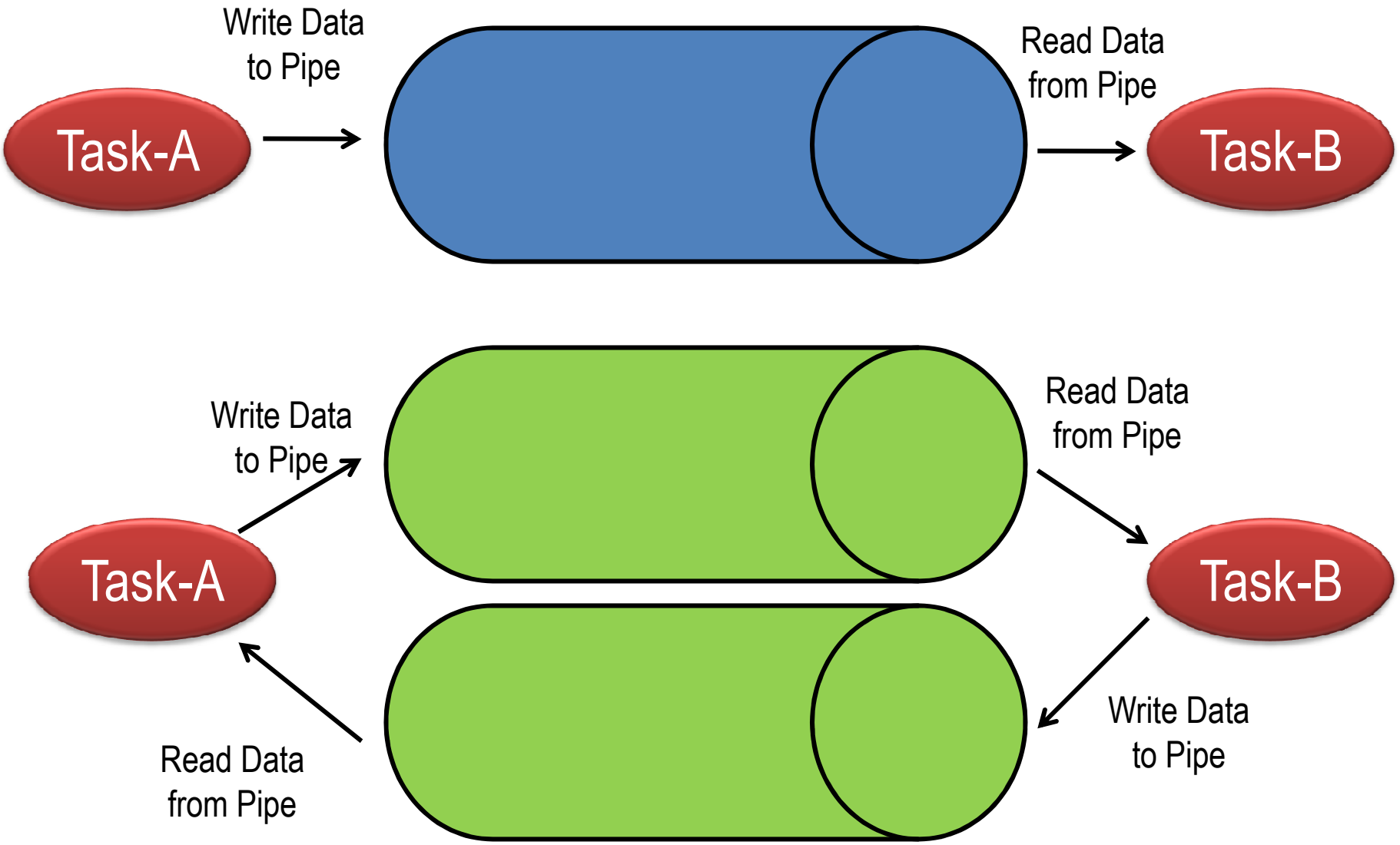
Writing a Queue



Reading a Queue



5. Pipes


Write Data
to Pipe
Task-A
Read Data
from Pipe
Task-B
Write Data
to Pipe
Task-A
Read Data
from Pipe
Task-B
Read Data
from Pipe
Write Data
to Pipe

Pipes

- A pipe is an RTOS object that provide simple communication channel used for **unstructured data exchange** among tasks. A
- Pipes can be opened, closed, written to and read from.
- Traditionally, a pipe is a unidirectional data exchange facility.
- Data is written into the pipe as an unstructured byte stream. Unlike message queue, a pipe does not store multiple messages but stream of bytes.
- In addition, data flow from a pipe cannot be
- prioritized.

Pipes

- A task can write into pipe and another task can read the data from Pipe.
- The output of one task is passed on as input the other task.
- Task to Task or ISR to task data transfer can take place using pipes.

Part- 4

Dynamic Memory Allocation

Dr.R.Sundaramurthy.,M.E.,Ph.D.,
sundar@pec.edu

Dynamic Memory Allocation

- An embedded RTOS usually strive to achieve small footprint by including only the functionality needed for the user's applications.
- There are two types of memory management in RTOSs. They are
  - Stack management.
  - Heap management.

Stack Management

- In a multi-tasking RTOS, each task needs to be allocated with an amount of memory for storing their contexts (i.e. volatile information such as registers contents, program counter, etc) for context switching.
- This allocation of memory is done using task-control block.
- This set of memory is commonly known as kernel stack and the management process termed **Stack Management**.

Heap Management

- Upon the completion of a program initialization, physical memory of the MCU or MPU will usually be occupied with program code, program data and system stack.
- The remaining physical memory is called heap.
- This heap memory is typically used by the kernel for dynamic memory allocation of data space for tasks.
- The memory is divided into fixed size memory blocks, which can be requested by tasks.
- When a task finishes using a memory block it must return it to the pool.
- This process of managing the heap memory is known as **Heap management**

Heap Management

- "malloc" and "free" services are used for heap management.
- Tasks can temporarily borrow some memory from the operating system's heap by calling "malloc" and specifying the size of memory buffer needed.
- When this task (or another task) is finished with this memory buffer it can return the buffer to the operating system by calling "free."
- The operating system will then return the buffer to the heap, where its memory might be used again.

External Memory Fragmentation

- Heaps suffer from a phenomenon called "External Memory Fragmentation" that may cause the heap services to degrade.
- This fragmentation is caused by the fact that when a buffer is returned to the heap, it may in the future be broken into smaller buffers when "malloc" requests for smaller buffer sizes occur.
- After a heap undergoes many cycles of "malloc"s and "free"s, small segments of memory may appear between memory buffers that are being used by tasks.
- These segments are so small that they are useless to tasks. But they are trapped between buffers that are being used by tasks, so they can't be "glued" together into bigger, useful buffer sizes.
- Over time, a heap will have more and more of these segments.
- This will eventually result in situations where tasks will ask for memory buffers ("malloc") of a certain size, and they will be refused by the operating system --- even though the operating system has enough available memory in its heap.
- The problem: That memory is scattered in small Segments distributed in various separate parts of the heap. In operating system terminology, the segments are called "fragments", and this problem is called **"external memory fragmentation."**

Defragmentation

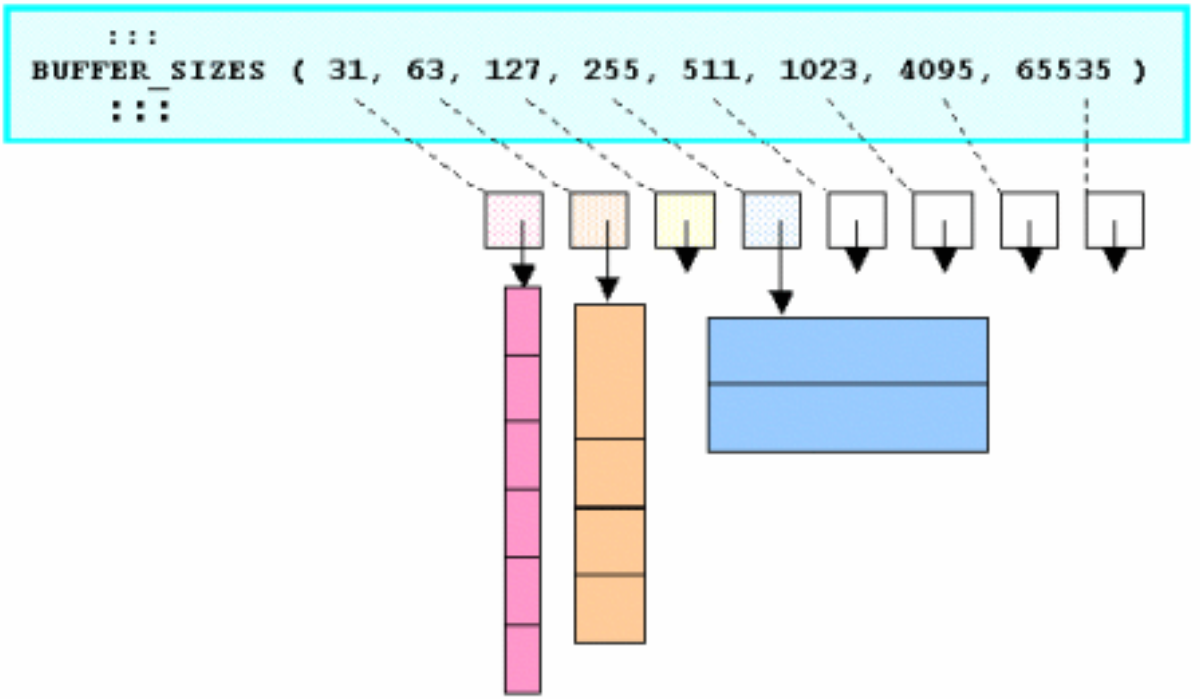
- The fragmentation problem can be solved by so-called "garbage collection" (defragmentation) algorithms.
- Unfortunately, "garbage collection" algorithms are often wildly non-deterministic – injecting randomly-appearing random-duration delays into heap services.
- These are often seen in the memory allocation services of general-computing non-real-time operating systems.
- This puts the embedded system developer who wants to use a general-computing non-real-time operating system into a quandry: Should the embedded system be allowed to suffer occasional randomly-appearing random-duration delays if / when "garbage collection" kicks in?...
- Or, should the embedded system be allowed to fragment its memory until application software "malloc" requests to the heap are refused even though a sufficient total amount of free memory is still available?
- Neither alternative is acceptable for embedded systems that need to provide service continually for long periods of time.

Memory Management in RTOS

- Real-time operating systems, on the other hand, solve this issue by altogether avoiding both memory fragmentation and "garbage collection", and their consequences.
- RTOSs offer non-fragmenting memory allocation techniques instead of heaps.
- They do this by limiting the variety of memory chunk sizes they make available to application software.
- While this approach is less flexible than the approach taken by memory heaps, they do avoid external memory fragmentation and avoid the need for defragmentation.
- For example, the "Pools" memory allocation mechanism allows application software to allocate chunks of memory of perhaps 4 or 8 different buffer sizes per pool.
- Pools totally avoid external memory fragmentation, by not permitting a buffer that is returned to the pool to be broken into smaller buffers in the future.
- Instead, when a buffer is returned the pool, it is put onto a "free buffer list" of buffers of its own size that are available for future re-use at their original buffer size.

Memory Pools

- Memory is allocated and de-allocated from a pool with deterministic, often constant, timing.



Part- 5

Timer Management

Dr.R.Sundaramurthy.,M.E.,Ph.D.,
sundar@pec.edu

Timer Management

- Timers are used to measure the elapsed time of events.
- The kernel has to keep track of different times.
  - A particular task may need to be executed periodically , say, every 10ms. A timer has to keep track this periodicity.
  - A task may be waiting in a queue for an event to occur. If the event does not occur for a specified time , it has to take appropriate action.
  - A task may be waiting in a queue for a shared resource. If the resource . If the resource is not available for a specified time, an appropriate action has to be taken.

Timer Management

- In embedded systems, system and user tasks are often scheduled to perform after a specified duration.
- To provide such scheduling, there is a need for a periodical interrupt to keep track of time delays and timeout.
- Most RTOSs today offer both “relative timers” that work in units of ticks, and “absolute timers” that work with calendar date and time.

Timer Management

- For each kind of timer, RTOSs provide a “task delay” service, and also a “task alert” service based on the signaling mechanism (e.g. event flags).
- Another timer service provided is in meeting task deadline by cooperating with task schedulers to determine whether tasks have met or missed their real-time deadlines.

Timer management Function calls

- The following are the commonly used API calls to manage timers.
  - GetTime
  - SetTime
  - Time Delay(in system clock ticks)
  - Time Delay (in seconds)
  - Reset Timer

Part- 6

Interrupt Handling

Dr.R.Sundaramurthy.,M.E.,Ph.D.,
sundar@pec.edu

Interrupt Handling

- An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred.
- A fundamental challenge in RTOS design is supporting interrupts and thereby allowing asynchronous access to internal RTOS data structures.
- The interrupt and event handling mechanism of an RTOS provides the following functions:
  - Defining interrupt handler
  - Creation and deletion of ISR
  - Referencing the state of an ISR
  - Enabling and disabling of an interrupt
  - Changing and referencing of an interrupt mask

Interrupt Handling

The interrupt and event handling mechanism of an RTOS Help to ensure:

- Data integrity by restricting interrupts from occurring when modifying a data structure
- Minimum interrupt latencies due to disabling of interrupts when RTOS is performing critical operations
- Fastest possible interrupt responses that marked the preemptive performance of an RTOS
- Shortest possible interrupt completion time with minimum overheads

Part- 7

Device I/O management

Dr.R.Sundaramurthy.,M.E.,Ph.D.,
sundar@pec.edu

Device I/O management

- An RTOS kernel is often equipped with a device I/O management service to provide a uniform framework (application programmer's interface-"API") for accessing hardware resources of a processor.
- Supervision facility for an embedded system to organize and access large numbers of diverse hardware device drivers.
- However, most device driver APIs and supervisors are "standard" only within a specific RTOS.

End of Session

sundar@pec.edu