TASK SYNCHRONISATION

If two or more processes are trying to access the same system resources such as display device or memory location, a conflict will occur and gives unexpected result. So each processes should be aware about the shared resources. The act of making processes aware of the access of shared resources to avoid conflicts is known as 'Task/Process Synchronisation'. If processes are not synchronised properly, it causes various issues, such as:

• Racing	:	The situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently
• Deadlock	:	The situation where none of the processes are able to make any progress in their execution. Processes enters in wait state.
• Livelock	:	The situation where two or more processes continuously change their state in response to changes in the state of the other processes, but is unable to make any progress in the execution completion
• Starvation	:	The situation in which a process does not get the resources required to continue its execution for a long time.

Task synchronisation techniques:

Process/Task synchronisation is essential for:

- Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.
- Ensuring proper sequence of operation across processes
- Communicating between processes.

Mutual Exclusion

Mutual exclusion is the mechanism to synchronise access to shared resources. The code memory area which holds the program instructions for accessing a shared resource (like shared memory, shared variables, etc.) is known as 'critical section'. In order to synchronise the access to shared resources, the access to the critical section should be exclusive. Mutual exclusion methods can be classified into two categories:

1. Mutual Exclusion through Busy Waiting/Spin Lock
2. Mutual Exclusion through Sleep & Wakeup

Mutual Exclusion through Busy Waiting/Spin Lock

The 'Busy waiting' technique uses a lock variable for implementing mutual exclusion. Each process/thread checks this lock variable before entering the critical section. The lock is set to '1' by a process/thread if the process/thread is already in its critical section; otherwise the lock is set to '0'. If another process want to enter the critical section, it checks the value of lock variable. If the value is 0, it enters the critical section, otherwise the process waits until the lock value become 0. This keeps the processes/threads always busy and forces the processes/threads to wait for the availability of the lock for proceeding further. Hence this synchronisation mechanism is popularly known as 'Busy waiting'.

Mutual Exclusion through Sleep & Wakeup

The 'Busy waiting' mutual exclusion method causes wastage of CPU time and high power consumption. In 'Sleep & Wakeup' mechanism, when a process is not allowed to access the critical section, the process undergoes 'Sleep' and enters the 'blocked' state. When the current process leaves the critical section, it sends a 'Wake up' message to the blocked process, which was sleeping for want of the critical section and this process can now access the critical section.

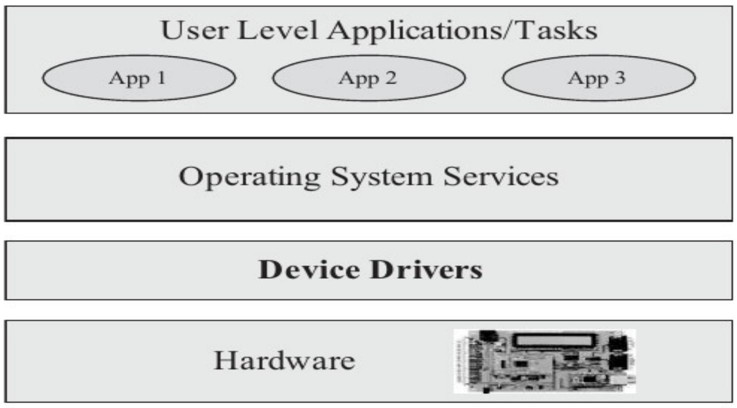
The 'Sleep & Wakeup' policy for mutual exclusion can be implemented using:

- Binary semaphore (Also called mutex)
- Counting semaphore

The binary semaphore provides exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it when it is being owned by a process. The counting semaphore, resources can be accessed by some fixed number of processes.

DEVICE DRIVERS

Device driver is a piece of software that acts as a bridge between the operating system and the hardware. The architecture of the OS kernel will not allow direct device access from the user application. All the device related access should flow through the OS kernel and the OS kernel routes it to the concerned hardware peripheral. OS provides interfaces in the form of Application Programming Interfaces (APIs) for accessing the hardware. Certain drivers come as part of the OS kernel and certain drivers need to be installed.



Device driver is responsible for:

- Initiating and managing the communication with the hardware peripherals
- Establishing the connectivity
- Initialising the hardware
- Transferring of data

To perform the responsibilites, a device driver implements the following functionalities:

1. Device (Hardware) Initialisation and Interrupt configuration
2. Interrupt handling and processing
3. Client interfacing (Interfacing with user applications)

HOW TO CHOOSE AN RTOS

Several factors are to be considered for selecting an RTOS. These factors can be classified as functional or non-functional.

Functional requirements:

- Processor Support
  Ensure the RTOS support the processor architecture
- Memory Requirements
  Ensure the minimum RAM and ROM required by the RTOS
- Real time Capabilities
  Ensure the real time capabilities of the RTOS to meet the embedded application
- Kernel and Interrupt Latency
  For application with high response requirement, the interrupt latency should be minimum
- Inter Process Communication and Task Synchronisation
  Analyse the options available for IPC and Task synchronisation

- Modularisation Support
  If the OS support modularisation, the developer can choose essential modules and recompile the OS to get its functionality.
- Support for Networking and Communication
  Ensure OS provides support for all the interfaces required by the embedded product.
- Development Language Support
  Check the availability of run time libraries required for applications written in certain languages

Non-Functional requirements:

- Custom Developed or Off the Shelf
  Depending on the application requirement, we can go for a complete development of the OS or can use a readily available product
- Cost
  Consider the cost for developing/purchasing and maintenance of the product
- Development and Debugging Tools Availability
  Explore the different tools available with OS for development/debugging
- Ease of Use
- After Sales services