Operating Systems

Introduction to Operating System (OS)

In a nutshell

- Monitors are a programming language component that aids in the regulation of shared data access.
- The Monitor is a package that contains shared data structures, operations, and synchronization between concurrent procedure calls.
- A Monitor is an abstract data type, i.e. a combination of data structures and operations, where at most one operation may be executing at one time. The required mutual exclusions are enforced by the compiler implementation of the monitor.

Characteristics

• **Mutual Exclusion:** Monitors ensure mutual exclusion, which means only one process or thread can be inside the monitor at any given time. This property prevents concurrent processes from accessing shared resources simultaneously and eliminates the risk of data corruption or inconsistent results due to race conditions.

Characteristics (1)

- **Encapsulation:** Monitors encapsulate both the shared resource and the procedures that operate on it. By bundling the resource and the relevant procedures together, monitors provide a clean and organized approach to managing concurrent access. This encapsulation simplifies the design and maintenance of concurrent programs, as the necessary synchronization logic is localized within the monitor.

Characteristics (2)

- **Synchronization Primitives:** Monitors often support synchronization primitives, such as condition variables. Condition variables enable threads within the monitor to wait for specific conditions to become true or to signal other threads when certain conditions are met. These primitives allow for efficient coordination among threads and help avoid busy-waiting, which can waste CPU cycles.

Characteristics (3)

- **Blocking Mechanism:** When a process or thread attempts to enter a monitor that is already in use, it is blocked and put in a queue (entry queue) until the monitor becomes available. This blocking mechanism avoids busy-waiting and allows other processes to proceed while waiting for their turn to access the monitor.

Characteristics (4)

- **Local Data:** Each thread that enters a monitor has its own local data or stack, which means the variables declared within a monitor procedure are unique to each thread's execution. This feature prevents interference between threads and ensures that data accessed within the monitor remains consistent for each thread.

Characteristics (5)

- **Priority Inheritance:** In some advanced implementations of monitors, a priority inheritance mechanism can be used to prevent priority inversion. When a higher-priority thread is waiting for a lower-priority thread to release a resource inside the monitor, the lower-priority thread's priority may be temporarily elevated to avoid unnecessary delays caused by priority inversion scenarios.

Characteristics (6)

- **High-Level Abstraction:** Monitors provide a higher-level abstraction for concurrency management compared to low-level synchronization mechanisms like semaphores or spinlocks. This abstraction reduces the complexity of concurrent programming and makes it easier to write correct and maintainable code.

Components

- **Shared Resource:**
  The shared resource is the data or resource that multiple processes or threads need to access in a mutually exclusive manner. Examples of shared resources can include critical sections of code, global variables, or any data structure that needs to be accessed atomically.

Components (1)

- **Entry Queue:**
  The entry queue is a data structure that holds the processes or threads that are waiting to enter the monitor and access the shared resource. When a process or thread tries to enter the monitor while it is already being used by another process, it is placed in this queue, and its execution is temporarily suspended until the monitor becomes available.

Components (2)

- **Entry Procedures (or Monitor Procedures):** Entry procedures are special procedures that provide access to the shared resource and enforce mutual exclusion. When a process or thread wants to access the shared resource, it must call one of these entry procedures. The monitor's implementation ensures that only one process or thread can execute an entry procedure at a time, thus achieving mutual exclusion.

Components (3)

- **Condition Variables:**
  Condition variables enable communication and synchronization between processes or threads within the monitor. They allow threads to wait until a specific condition is satisfied or to signal other threads when certain conditions become true. Condition variables are crucial for avoiding busy-waiting, which can be inefficient and wasteful of system resources.

Components (4)

- **Local Data (or Local Variables):**
  Each process or thread that enters the monitor has its own set of local data or local variables. These variables are unique to each thread's execution and are not shared between threads. Local data allows each thread to work independently within the monitor without interfering with other threads' data.

Components (5)

- **Condition Variables**

  The condition variables of the monitor can be subjected to two different types of operations:

  – Wait

  – Signal

- Consider a condition variable (y) is declared in the monitor:
- **y.wait():** The activity/process that applies the wait operation on a condition variable will be suspended, and the suspended process is located in the condition variable's block queue.
- **y.signal():** If an activity/process applies the signal action on the condition variable, then one of the blocked activity/processes in the monitor is given a chance to execute.

Example

- // Dining-Philosophers Solution Using Monitors
- monitor DP
- {
- status state[5];
- condition self[5];
-
-

Example ..

```
// Pickup chopsticks
Pickup(int i){
    // indicate that I'm hungry
    state[i] = hungry;
    // set state to eating in test()
    // only if my left and right neighbors
    // are not eating
    test(i);
    // if unable to eat, wait to be signaled
    if (state[i] != eating)
        self[i].wait; }
```

Example ..

```
//Put down chopsticks
    Putdown(int i){
        // indicate that I'm thinking
        state[i] = thinking;
        // if right neighbor R=(i+1)%5 is hungry and
        // both of R's neighbors are not eating,
        // set R's state to eating and wake it up by
        // signaling R's CV
        test((i + 1) % 5);
        test((i + 4) % 5);
    }

```

Example..

```
test(int i){
  if (state[(i + 1) % 5] != eating
      && state[(i + 4) % 5] != eating
      && state[i] == hungry) {
      // indicate that I'm eating
      state[i] = eating;
      // signal() has no effect during Pickup(),
      // but is important to wake up waiting
      // hungry philosophers during Putdown()
      self[i].signal();}
}
```

Example…

```
    init(){
        // Execution of Pickup(), Putdown() and test()
        // are all mutually exclusive,
        // i.e. only one at a time can be executing
For i = 0 to 4
        // Verify that this monitor-based solution is
        // deadlock free and mutually exclusive in that
        // no 2 neighbors can eat simultaneously
        state[i] = thinking;
    }
} // end of monitor
```