**SOFTWARE ARCHITECTURE CONCURRENCY PATTERNS**

1. RENDEZVOUS PATTERN

-This is mostly concerned with modeling the preconditions for synchronization or rendezvous of threats.

-Rendezvous Pattern is a simplified form of the Guarded Call Pattern used to either synchronize a set of threats to permit data sharing among a set of threats.

-It may contain data to be shared as the threats synchronize, it provides a means for synchronizing an arbitrary number of threats on a point with some synchronization Policy or precondition before allowing them all to continue independently.

**How it works:**

Each thread becomes ready to rendezvous, it registers with the Rendezvous class and then blocks until the Rendezvous class releases it to run.Once the set of preconditions is met,the registered tasks are released to run using the scheduling policy is currently in force.

**Rendezvous Pattern**

 Synch Policy

 Client Thread

 Notify(Void): Void

 Rendezvous

 Reset (Void): Void

 Register (callback: address

 Release(Void); Void

(mapped)

Callback

**Advantages of Rendezvous Pattern**

-It is a general pattern and it is very easy to apply to ensure that arbitrarily complex sets of preconditions can be met at run time.

-It scales up well to arbitrary numbers of threads and to arbitrarily complex synchronization policies.

**Disadvantages of Rendezvous Pattern**

-In this pattern the problem is to codify a collaboration structure that allows any arbitrary set of precondition in variations to be met for thread synchronization, independence of task phasing, scheduling policies and priorities.

1. **Round Robin Pattern**

-It employs a ‘’fairness’’ scheduling doctrine that may be appropriate for systems in which it is more important for all tasks to progress than it is for specific deadline to be met.

-Round Robin Pattern has the ability to prompt running taks and does so when it receives a tick message from its associated timer.

**How it works:**

When the Scheduler is run, it configures the Timer with the proper period and then executes each task in turn for the time slice. Initially, it uses the default run()operation on the tasks ,subsequently, it will merely jump directly to where that task left off. Prior to executing the task, the relevant task’s stack is set active, and the stack pointer is set with the current top of stack.

**Simplified Round Robin Pattern Structure**

 Timer

Time slice source

 Abstract Thread

 run (void): Void

 Scheduler

{ordered}

 Concrete Thread

**Advantages of Round Robin Pattern**

-A misbehaving task won’t stop the entire system from running because the Timer will interrupt each task when it is time to perform a task switch.

-Round Robin Pattern is fair as all the tasks get a chance to run.

**Disadvantages of Round Robin Pattern**

-For tasks to get a chance to run, it must use critical sections.

-This pattern is not at the best possible level in terms of response incoming events and it is unstable in the sense that you can’t predict which task will fail in an overload situation.

1. **Static Priority Pattern**

It is called Static because it is assigned during design and can’t be changed during execution of the system.

-It is the most common approach to scheduling.

**How it works:**

Each <<active>>object registers with the *scheduler* object in the operating system by calling ***create thread*** operation and passing to it, the address of a method defined.

Each ***concrete thread*** executes until it completes (which it signals to the OS by calling ***scheduler:: return()***), it is preempted by a higher-priority task being inserted into the ***Ready Queue***, or it is blocked in an attempt to access a ***Shared Resource*** that has a locked ***Mutex*** semaphore.

**Simplified Static Priority Pattern Structure**

 Abstract Thread

 Run(Void):Void

|  |
| --- |
| Scheduler |
|  CreateThread (StartAddr: address; Priority: int); destroy Thread (TCBAddr: address); block Thread (mutexID: int; EntryPoint: address)’ UnblockThread (mutexID: int); Return(TCBAddr); |

 Shared Resource

 Mutex

 Concrete Thread

**Advantages of Static Priority Pattern**

* It is simple and scales fairly well to large numbers of tasks.
* Static Priority Pattern is simple to analyze for schedulability, using standard rate monotonic analysis methods.
* In this Pattern you can predict which tasks will fail to meet their deadlines.
* Static Priorities may be applied to all sizes of systems where the environment and desired system response is highly predictable.
* The Pattern may be adjusted to use various different scheduling approaches such as Round Robin Policies.

**Disadvantages of static Priority Pattern**

* It is not suited for highly complex situations.
* Works well for less dynamic situations.
1. **DYNAMIC PRIORITY PATTERN**

In this pattern the priority of a task is set at run time based solely on the urgency of the task.

It explicitly emphasizes urgency over ***criticality.***

Dynamic Priority Pattern sets the priority of each task as a function of the time remaining until its deadline, the closer, the higher the priority.

**How it works**

Abstract Thread class also contains an attitude called ***Deadline***. This is normally the duration of time from the invocation of the task until the point in time at which the task becomes late. It is specified as a duration, but the scheduler will compute an absolute deadline first. When a new task becomes ready to run, it is inserted in the ready queue based on its next deadline.

 Task Control Block

 AbsoluteDeadline: TimeValue

 StartAddr: address

 EntryPont: address

 Stack

BaseAddr: address

 Top: int

**Dynamic Priority Pattern Structure**

|  |
| --- |
| Ready Queue: Priority Queue |

 Abstract Thread

 Deadline: Duration

 Run(Void):Void

|  |
| --- |
|  Scheduler |
|  |
| Create Thread(StartAddr: address; priority: int) destroyThread (TCBAddr: address); blockThread(mutevID: int; EntryPoint: address); UnblockThread(mutexID: int); Return (TCBAddr); |

(mapped)

Shared Resource

Concrete Thread

Mutex

|  |
| --- |
| Blocked Queue: Priority Queue |

**Advantage of Dynamic Priority Pattern**

* It is well suited for highly complex situations.
* Dynamic Priority Pattern is best suited for task sets that are at least of approximately equal criticality so that urgency is the overriding concern.
* Dynamic Priority Pattern scales well to larger Numbers of Threads.

**Disadvantages of Dynamic Priority Pattern**

* Dynamic Priority Pattern in complex situations, it is difficult or impossible to construct optimal static priorities for the tasks.
* Dynamic Priority is not stable, hence it is impossible to predict at design time which tasks will fail in an overload situation.