Objective: Selecting Thread to Run

- Scheduler selects a thread to run
- Processor relinquished

Multiple Waiting Threads → Context Switch

A Single Running Thread

Thread States: Initial View

- Ready: Thread waiting for scheduler to give it a turn.
- Running: Thread unknown to scheduler.
- Inactive: Thread waiting for scheduler to give it a turn.

Time-Sliced Scheduling

- Each thread runs for a fixed amount of time.
- Threads are run in a round-robin sequence.
- Appropriate for regular multi-programming environments.
- Poor response time performance.
- Need better strategy for real-time systems.
Pending Threads

- Threads must often wait for an event to occur or some shared resource to become available.

- Simply moving the task back to the ready state isn’t appropriate because it may run again later only to discover that it must continue to wait, thus adding unnecessary task switching overhead.
Preemptive Context Switch

- State of Thread A:
  - Running
  - Interrupted
  - Ready

- State of Thread B:
  - Pending
  - Ready
  - Interrupted
  - Running

Interrupt Routine → Run-Time Kernel → Context Switch → Post

Preemptive Thread States

- Pending → Running
- Interrupted → Ready → Inactive

Priority-Based Scheduling

- Each thread is assigned a priority number.
- Static priorities never change.
- Dynamic priorities can vary during execution.
  - Required to avoid Priority Inversion Problem.

Priority Inversion

- Resource Owned by High-Priority Thread
  - Running
  - Pending
  - Running

- Request resource
- Run-Time Kernel
- Release Resource

- Resource Owned by Low-Priority Thread
  - Ready
  - Running
  - Ready
Priority Inversion

• **Bounded Priority Inversion**
  – Duration is no longer than that of the critical section where the lower-priority thread owns the resource.

• **Unbounded Priority Inversion**
  – Occurs when a *third* (medium-priority) thread preempts the low-priority thread during the inversion for an indefinite amount of time.

Mars Pathfinder Suffered Unbounded Priority Inversion

**Low-priority meteorological thread**

• Acquired the (shared) bus.

**Medium-priority, long-running, communications thread**

• Woke up and preempted the meteorological thread.

**High-priority bus management thread**

• Woke up and was blocked because it couldn't acquire the bus; when it couldn't meet its deadline it reinitialized the computer via a hardware reset.

Limiting the Duration of an Unbounded Priority Inversion

• **Objective:**
  – To prevent low-priority thread from being preempted by medium-priority thread during the priority inversion.

• **Strategies:**
  – Priority Inheritance Protocol
  – Priority Ceiling Protocol

• **Technique:**
  – Manipulate thread priorities at run-time.

Priority Inheritance Protocol

• When a high-priority thread attempts to lock a mutex already locked by a lower-priority thread, the Priority Inheritance Protocol (PIP) temporarily *raises the priority of the low-priority thread to match that of the blocked thread* until the low-priority thread unlocks the mutex.

• **Advantage:** It is transparent to the application.

• **Disadvantage:** Adds complexity to the kernel.
Priority Ceiling Protocol

• Priority of the low-priority thread is raised immediately when it locks the mutex rather than waiting for a subsequent lock attempt by a higher-priority thread.

• Advantage: Easy to implement.
• Disadvantage: The priority ceiling value must be predetermined for use with the mutex; this value must be the highest among all the threads that attempt to lock the same mutex.

Assigning Priorities

• Scheduling: Threads with higher priority are scheduled to run first.

• Objective: Assign priorities in such a way that all outputs are computed before their deadlines.

Assignment Strategies

• Deadline-Driven Assignment
  – Assign highest priorities to threads with shortest deadlines.

• Rate Monotonic Assignment
  – Assign highest priorities to threads that run most frequently without regard to deadlines.
Deadlock

- **Definition:** Two or more threads are blocked waiting on each other's resources.

- **Necessary Condition:** Threads require exclusive access to two or more shared resources simultaneously.

**Entering Deadlock**

| Thread T1: | Running | Ready | Running | Pending on M2 |
| Mutex M1:  | Locked by T1 |       |         |              |
| Mutex M2:  | Locked by T2 |       |         |              |
| Thread T2: | Ready    | Running | Pending on M1 |

**Preventing Deadlock**

1. Require all threads to acquire resources in the same order.
   - Often locks resources longer than necessary.
   - Occasionally there is no order that is convenient for all threads.

2. Require threads to release all resources and start over if any one resource can't be acquired.