**Background**

Real-time computing is becoming an increasingly important discipline. The operating

system, and particularly the scheduler, is perhaps the most important component.

of a real-time system. Examples of current applications of real-time systems include.

control of laboratory experiments, process control in industrial plants, robotics, air

traffic control, telecommunications, and military command and control systems. Next generation systems will include the autonomous land rover, controllers of robots.

with elastic joints, systems found in intelligent manufacturing, the space station, and

undersea exploration.

The real-time tasks have a certain degree of urgency. Such tasks are attempting to control or react. Thus, it is usually possible to associate a deadline with a particular task, where the deadline specifies either a start time or a completion time. Such a task may be classified as hard or soft.

**Hard real-time task**

It must meet a deadline; otherwise, it will cause unacceptable damage or a fatal error to the system.

**Soft real-time task** has an associated deadline that is desirable but not mandatory; it still makes sense to schedule and complete the task even if it has passed its deadline.

**Aperiodic task**

Has a deadline by which it must finish or start, or it may have a constraint on both start and finish time.

**Periodic task**

The requirement may be stated as “once per period T” or “exactly T units apart.”

**Characteristics of Real-Time Operating Systems**

Real-time operating systems can be characterized as having unique requirements in

five general areas:

1. Determinism

2. Responsiveness

3. User control

4. Reliability

5. Fail-soft operation

**Deterministic**

The extent to which an operating system can deterministically satisfy requests depends first on the speed with which it can respond to interrupts and, second, on whether the system has sufficient capacity to handle all requests within the required time. One useful measure of the ability of an operating system to function deterministically is the maximum delay from the arrival of a high-priority device interrupt to when servicing begins.

**Responsiveness.**

Responsiveness is concerned with how long, after acknowledgment, it takes an operating system to service the interrupt. Aspects of responsiveness include the following:

1. The amount of time required to initially handle the interrupt and begin execution of the interrupt service routine (ISR). If execution of the ISR requires a process switch, then the delay will be longer than if the ISR can be executed within the context of the current process.

2. The amount of time required to perform the ISR. This generally is dependent on the hardware platform.

3. The effect of interrupt nesting. If an ISR can be interrupted by the arrival of

another interrupt, then the service will be delayed.

Response time requirements are critical for real-time systems, because such

systems must meet timing requirements imposed by individuals, devices, and data

flows external to the system.

**User control**

The user should be able to distinguish between hard and soft tasks and to specify relative priorities within each class. A real-time system may also allow the user to specify such characteristics as the use of paging or process swapping, what processes must always be resident in main memory, what disk transfer algorithms are to be used, what rights the processes in various priority bands have, and so on.

**Reliability**

Is important for real-time systems than non-realtime systems. A transient failure in a non-real-time system may be solved by simply rebooting the system. A processor failure in a multiprocessor non-real-time system may result in a reduced level of service until the failed processor is repaired or replaced. But a real-time system is responding to and controlling events in real time. Loss or degradation of performance may have catastrophic consequences, ranging. from financial loss to major equipment damage and even loss of life.

**Fail-soft operation**

This characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible. Fail-soft operation is referred to as stability. A real-time system is stable if, in cases where it is impossible to meet all task deadlines, the system will meet the deadlines of its most critical, highest-priority tasks, even if some less critical task deadlines are not always met.

Although there is a wide variety of real-time OS designs to meet the wide variety of real-time applications, the following features are common to most real-time OSs:

• A stricter use of priorities than in an ordinary OS, with preemptive scheduling

that is designed to meet real-time requirements.

• Interrupt latency (the amount of time between when a device generates an

interrupt and when that device is serviced) is bounded and relatively short.

• More precise and predictable timing characteristics than general purpose OSs

The heart of a real-time system is the short-term task scheduler. In designing

such a scheduler, fairness and minimizing average response time are not paramount.

Most contemporary real-time operating systems are unable to deal directly with deadlines. Instead, they are designed to be as responsive as possible to real-time.

In a preemptive scheduler that uses simple round-robin scheduling, a real-time task would be added to the ready queue to await its next time slice.

In a no preemptive scheduler, we could use a priority scheduling mechanism, giving real-time tasks higher priority. This could lead to a delay of several seconds if a slow, low-priority task were executing at a critical time. Again, this approach is not acceptable. A more promising approach is to combine priorities with clock-based interrupts. Preemption points occur at regular intervals.

While this last approach may be adequate for some real-time applications, it will not suffice for more demanding applications. In those cases, the approach that has been taken is sometimes referred to as immediate preemption.

**Real-Time Scheduling**

Real-time scheduling is one of the most active areas of research in computer science. In a survey of real-time scheduling algorithms, [RAMA94] observes that the various scheduling approaches depend on

* Whether a system performs schedulability analysis if it does, whether it is done statically or dynamically and whether the result of the analysis itself produces a schedule or plan according to which tasks are dispatched at run time.

**Static table-driven approaches**: these perform a static analysis of feasible schedules of dispatching. The result of the analysis is a schedule that determines, at run time, when a task must begin execution.

**Static priority-driven preemptive approaches**: Again, a static analysis is performed, but no schedule is drawn up. Rather, the analysis is used to assign priorities to tasks.

**Dynamic planning-based approaches**: Feasibility is determined at run time

(dynamically) rather than offline prior to the start of execution (statically).

An arriving task is accepted for execution only if it is feasible to meet its time

constraints.

**Dynamic best effort approaches**: The system tries to meet all deadlines and aborts any started process whose deadline is missed. Static table-driven scheduling is applicable to tasks that are periodic. Input to the analysis consists of the periodic arrival time, execution time, periodic ending deadline, and relative priority of each task. The scheduler attempts to develop a schedule that enables it to meet the requirements of all periodic tasks. With dynamic planning-based scheduling, after a task arrives, but before its execution begins, an attempt is made to create a schedule that contains the previously scheduled tasks as well as the new arrival. Dynamic best effort scheduling is the approach used by many real-time systems that are currently commercially available. When a task arrives, the system assigns a priority based on the characteristics of the task. Some form of deadline scheduling, such as earliest-deadline scheduling, is typically used. Typically, the tasks are aperiodic, so no static scheduling analysis is possible. With this type of scheduling, until a deadline arrives or until the task completes, we do not know whether a timing constraint will be met. This is the major disadvantage of this form of scheduling. Its advantage is that it is easy to implement.

**Deadline Scheduling**

Most contemporary real-time operating systems are designed with the objective of starting real-time tasks as rapidly as possible, and hence emphasize rapid interrupt handling and task dispatching. Real-time applications are generally not concerned with sheer speed but rather with completing (or starting) tasks at the most valuable times, neither too early nor too late, despite dynamic resource demands and conflicts, processing overloads, and hardware or software faults

There have been several proposals for more powerful and appropriate approaches to real-time task scheduling. All of these are based on having additional information about each task. In its most general form, the following information about each task might be used:

• Ready time: Time at which task becomes ready for execution. In the case of a repetitive or periodic task, this is a sequence of times that is known in advance. In the case of an aperiodic task, this time may be known in advance, or the operating system may only be aware when the task is ready.

• Starting deadline: Time by which a task must begin

• Completion deadline: Time by which a task must be completed. The typical

real-time application will either have starting deadlines or completion deadlines, but not both.

• Processing time: Time required to execute the task to completion. In some cases, this is supplied. In others, the operating system measures an exponential average.

• Resource requirements: Set of resources (other than the processor) required by the task while it is executing

• Priority: Measures relative importance of the task. Hard real-time tasks may have an “absolute” priority, with the system failing if a deadline is missed. If the system is to continue to run no matter what, then both hard and soft real-time tasks may be assigned relative priorities as a guide to the scheduler.

• Subtask structure: A task may be decomposed into a mandatory subtask and an optional subtask. Only the mandatory subtask possesses a hard deadline.

**Rate Monotonic Scheduling**

One of the more promising methods of resolving multitask scheduling conflicts for periodic tasks is rate monotonic scheduling (RMS) [LIU73, BRIA99, SHA94]. RMS assigns priorities to tasks based on their periods. For RMS, the highest-priority task is the one with the shortest period, the second highest-priority task is the one with the second shortest period, and so on. When more than one task is available for execution, the one with the shortest period is serviced first. If we plot the priority of tasks as a function of their rate, the result is a monotonically increasing function, hence the name “rate monotonic scheduling.

One measure of the effectiveness of a periodic scheduling algorithm is whether or not it guarantees that all hard deadlines are met. Suppose we have n tasks, each with a fixed period and execution time. Then for it to be possible to meet all deadlines. Nevertheless, RMS has been widely adopted for use in industrial applications. [SHA91] offers the following explanation:

1. The performance difference is small in practice.

2. Most hard real-time systems also have soft real-time components, such as certain noncritical displays and built-in self-tests that can execute at lower-priority levels to absorb the processor time that is not used with RMS scheduling of hard real-time tasks.

3. Stability is easier to achieve with RMS. When a system cannot meet all deadlines because of overload or transient errors, the deadlines of essential tasks need to be guaranteed provided that this subset of tasks is schedulable. In a static priority assignment approach, one only needs to ensure that essential tasks have relatively high priorities. This can be done in RMS by structuring essential tasks to have short periods or by modifying the RMS priorities to account for essential tasks.

**Priority Inversion**

Priority inversion is a phenomenon that can occur in any priority-based preemptive scheduling scheme but is particularly relevant in the context of real-time scheduling. The best-known instance of priority inversion involved the Mars Pathfinder mission. This rover robot landed on Mars on July 4, 1997, and began gathering and transmitting voluminous data back to Earth. But a few days into the mission, the lander software began experiencing total system resets, each resulting in losses of data. After much effort by the Jet Propulsion Laboratory (JPL) team that built the Pathfinder, the problem was traced to priority inversion [JONE97]. In any priority scheduling scheme, the system should always be executing the task with the highest priority. Priority inversion occurs when circumstances within the system force a higher-priority task to wait for a lower-priority task. A simple example of priority inversion occurs if a lower-priority task has locked a resource (such as a device or a binary semaphore) and a higher-priority task attempts to lock that same resource. The higher-priority task will be put in a blocked state until the resource is available. If the lower-priority task soon finishes with the resource and releases

it, the higher-priority task may quickly resume, and it is possible that no real-time constraints are violated. A more serious condition is referred to as an unbounded priority inversion, in which the duration of a priority inversion depends not only on the time required to handle a shared resource but also on the unpredictable actions of other unrelated tasks. The priority inversion experienced in the Pathfinder software was unbounded and serves as a good example of the phenomenon. Our discussion follows that of