## Timing Services

In embedded systems, system tasks and user tasks often schedule and perform activities after some time has elapsed. For example, a RTOS scheduler must perform a context switch of a preset time interval periodically - among tasks of equal-priorities - to ensure execution fairness when conducting a round-robin scheduling algorithm. A software-based memory refresh mechanism must refresh the dynamic memory every so often or data loss will occur. In embedded network devices, various communication protocols schedule activities for data retransmission and protocol recovery. The target monitor software sends system information to the host-based analysis tool periodically to provide system-timing diagrams for visualization and debugging.

In any case, embedded applications need to schedule future events. Scheduling future activities is accomplished through timers using timer services.

Timers are an integral part of many real-time embedded systems. A timer is the scheduling of an event according to a predefined time value in the future, similar to setting an alarm clock.

A complex embedded is comprised of many different software modules and components, each requiring timers of varying.

timeout values. Most embedded systems use two different forms of timers to drive time-sensitive activities: hard timers and soft timers. Hard timers are derived from programmable physical timer chips that directly interrupt the processor when they expire. Operations with demanding requirements for precision or latency need the predictable performance of a hard timer. Soft timers are software events that are scheduled through a software facility.

A soft timer facility allows for efficiently scheduling of non-high-precision software events. A practical design for the soft-timer handling facility should have the following properties:

- efficient timer maintenance i.e., counting down a timer,
- efficient timer installation i.e., starting a timer,
- efficient timer removal i.e., stopping a timer.

While an application might require several high-precision timers with resolutions on the order of microseconds or even nanoseconds, not all of the time requirements have to be high precision. Even demanding applications also have some timing functions for which resolutions on the order of milliseconds or even of hundreds of milliseconds, are sufficient. Aspects

## The Dispatcher:

Dispatcher module gives control of CPU to the process selected by the short-term scheduler; this involves:

→ switching context
→ switching to user mode
→ jumping to proper location in the user program to restart that program

Dispatch latency:
time it takes for the dispatcher to stop one process and start another running.

## Scheduling Algorithms

- FIRST COME, FIRST SERVED:
  → It is same as First in first out (FIFO)
  → Simple, fair, but poor performance. Average queuing time may be long
  → what are the average queueing and residence times for this scenario?
  → Average queuing and residence times depend on ordering of these processes in the queue.

- SHORTEST JOB FIRST
  → Optimal for minimizing queueing time, but impossible to implement. Tries to predict the process to schedule based on previous history
  → Predicting the time process will use on its next schedule:

$$t(n+1) = \omega * t(n) + (1-\omega) * T(n)$$

of applications requiring timeouts with course granularity (for example, with tolerance for bounded inaccuracy) should use soft timers. Examples include the Transmission Control Protocol module, the Real-time Transport Protocol module, and the Address Resolution Protocol module.

Another reason for using soft timers is to reduce system-interrupt overhead. The physical timer chip rate is usually set so that the interval between consecutive timer interrupts is within tens of milliseconds or even within tens of microseconds. The interrupt latency and overhead can be substantial and can grow with the increasing number of outstanding timers. This issue particularly occurs when each timer is implemented by being directly interfaced with the physical timer hardware

## The Scheduler

- Selects from among the processes in memory that are ready to execute and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process
  - → Switches from running to waiting state
  - → Switches from running to ready state
  - → Switches from waiting to ready state.
  - → Terminates.
- All above mentioned scheduling are non preemptive while all the other scheduling is preemptive.

Here
$t(n+1)$ : is time of next burst
$t(n)$ : is time of current burst
$T(n)$ : is average of all previous bursts
$w$ : is a weighting factor emphasizing current or previous bursts

PREEMPTIVE ALGORITHMS:

→ Yank the CPU away from the currently executing process when a higher priority process is ready
→ Can be applied to both Shortest Job first or Priority scheduling
→ Avoid "hogging" of the CPU
→ On time sharing machines, this type of scheme is required because the CPU must be protected from a run-away low priority process.
→ Give short job higher priority - pre perceived response time is thus better.

PRIORITY BASED SCHEDULING

→ Assign each process a priority. Schedule highest priority first. All processes within same priority are FCFS
→ Priority may be determined by user or by some default mechanism. The system may determine the priority based on memory requirements, time limits, or other resource usage
→ Starvation occurs if a low priority process never runs. Solution: build aging into a variable priority
→ Delicate balance between giving favorable response for interactive jobs, but not starving batch jobs.

ROUND ROBIN

→ Use a timer to cause an interrupt after a predetermined time

Preempts if task exceeds its quantum.

→ Train of events

- Dispatch
- Time slice occurs or process suspends on event
- Put process on some queue and dispatch next.

→ Definitions

- Context Switch - changing the processor from running one task to another. Implies changing memory.
- Processor sharing - use a small quantum such that each process runs frequently at speed 1/n.
- Reschedule latency - How long it takes from when a process requests to run, until it finally gets control of CPU.

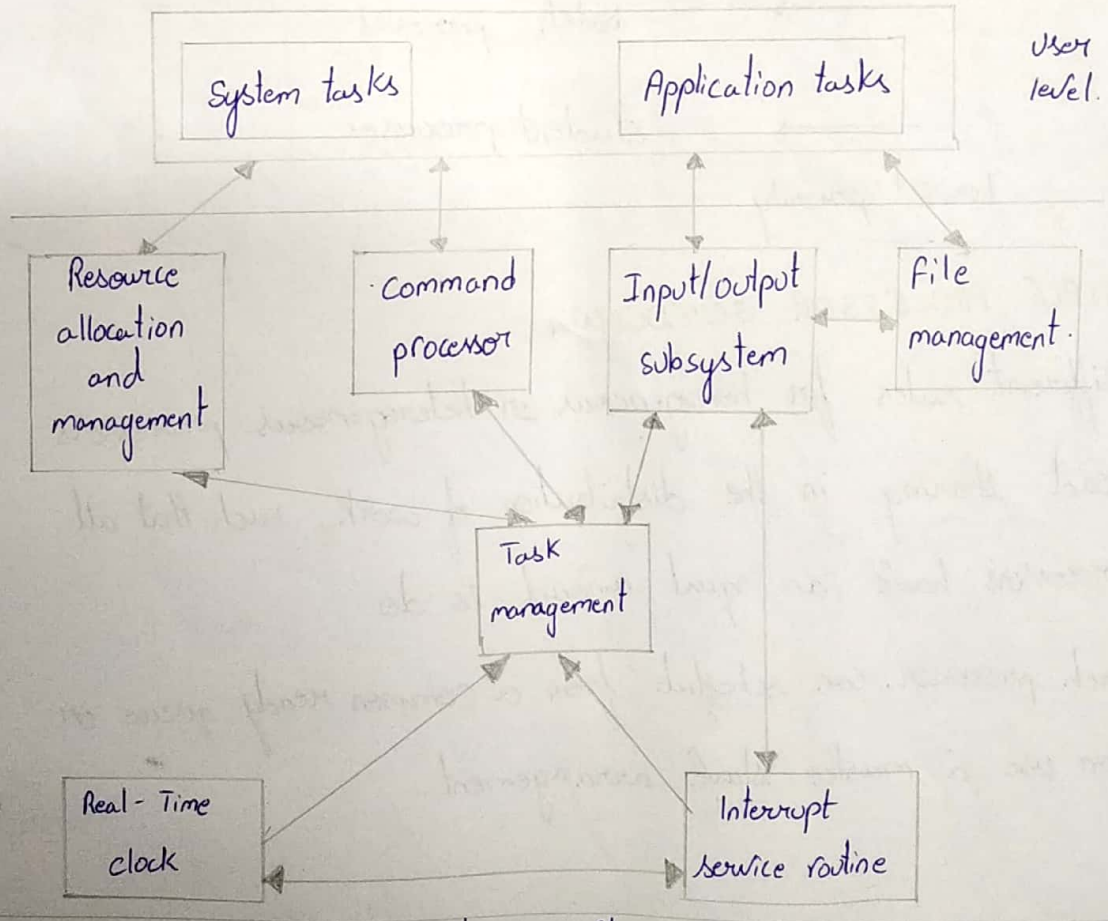
→ Choosing a time quantum.

- Too short - inordinate fraction of the time is spent in context switches.
- Too long - reschedule latency is too great. If many processes want the CPU, then it's a long time before a particular process can get the CPU. This then acts like FCFS
- Adjust so most processes won't use their slice. As processors have become faster, this is less of an issue

## BASIC OPERATING SYSTEM FUNCTIONS

A real time multi-tasking operating system has to support the resource sharing and the time requirement of the tasks and the functions can be divided as follows.

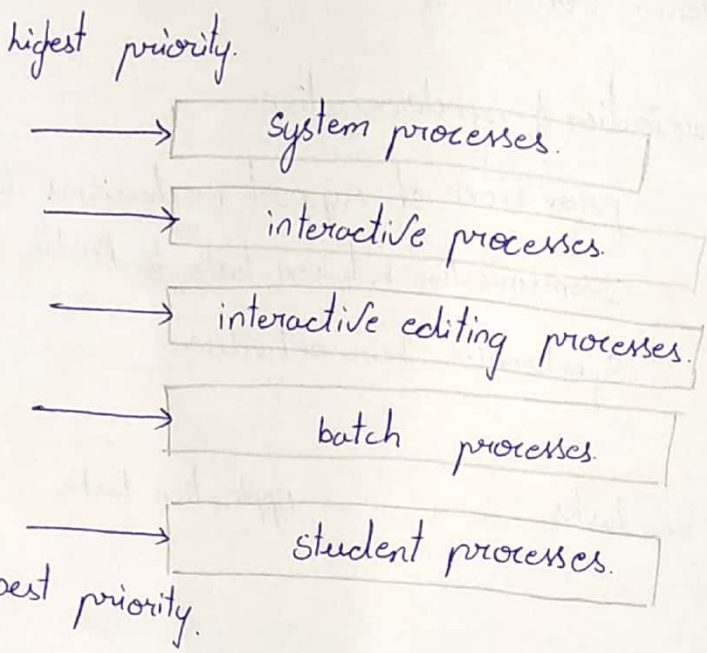
- Task Management: the allocation of memory and processor time to tasks
- Memory Management: Control of memory allocation.
- Intertask Communication & Synchronization:
  provision of support mechanisms to provide safe communication between tasks and to enable tasks to synchronies their activities.




Typical structure of a real-time operating system.

## MULTI - LEVEL QUEUES

- Each queue has its scheduling algorithm
- Then some other algorithm arbitrates between queues.
- Can use feedback to move between queues.
- Method is complex but flexible
- For example, could separate system processes, interactive, batch, favored, unfavored processes.




## MULTIPLE PROCESSOR SCHEDULING

- Different rules for homogeneous or heterogeneous processors.
- Load sharing in the distribution of work, such that all processors have an equal amount to do
- Each processor can schedule from a common ready queue or can use a master slave arrangement.

## Processor reserves

Processor capacity reserves represent reserved time on a processor. Reserve requests specify capacity in terms of time that will be reserved on the processor, rather than in terms of instructions that will be executed or any similar measure of processor usage. The requests may specify other information, depending on the admission control and scheduling policy in effect. The discussion in the following sections assumes a deadline monotonic scheduling framework where the reservation request specifies amount of time to reserve on the processor, a period at which the allocation will be replenished, and a delay bound. ~~These sections~~

pen system architecture

Vendor-independent, non-proprietary, computer system (or) device design based on official / or popular standards. It allows all Vendors (in competition with one another) to create add on products that increase a system's (or device's) flexibility, functionality, interoperatability, potential use, and useful life. And enables the users to customize and extend a System's (or device's) capabilities to suit individual requirements. Also called open Architecture.

Open System architecture, in telecommunication is a standard that describes the layered hierarchical structure, configuration (or) model of a comonunications (or) distributed data processing system that:

→ Enables system description, design, development, installation, operation, improvement and maintenance to be performed at a given layer (or) layers in the hierarchical structure

→ Allows each layer to provide a set of accessible functions that can be controlled and used by the functions in the layer above it.

→ Enables each layer to be implemented without affecting the implementation of other layers

→ Allows the alteration of system performance by modification of one (or) more layers without altering existing equipment, procedures, protocols at the remaining layers.

Open system architecture Motivation and benefits:

→ General purpose distributed computing environments are evolving towards real time systems. There is a great demand to provide real time functionality as normal system services, rather than as special add on features.

→ Real time applications are evolving towards large distributed systems. There is an increasing need to adopt an open and architectural approach so that real time software engineering can be augmented with other technique to address evolution, scale, distribution and other issue.

benefits:

→ Service providers will be able to specify and implement services that give required performance and real-time gurantees

→ Except in the most demanding of situations, real-time and performance management will cease to be special cases, decreasing the costs and time to deployment for systems

→ open distributed processing technique will be able to

support performance and real-time guarantees, increasing the range of applications and services it can support and increasing the market for the technology

Desirable features for real-time systems

1. predictability
2. programmer control
3. Timelines
4. Mission orientation
5. performance.

Predictability for real time systems:

Real time systems span a broad spectrum of complexity from very simple microcontrollers to highly sophisticated, complex and distributed systems. some future systems will be even more complex. These complex future systems include the space station, integrated vision/robotics/AI systems, collections of human/robots coordinating to achieve common objectives, and various command & control applications. To further complicate the problem there are many dimensions along with real-time systems can be categorized. The main ones include:

* the granularity of deadlines and laxities for tasks.
* the strictness of the deadlines

→ reliability requirements of system
→ the size of the system and degree of interaction among components and
→ the characteristics of environment in which system operates.

The characteristics of the environment, in turn, seem to give rise to how static (or) dynamic the system has to be. As one can be imagine, depending on the above considerations many different system design occur. However, one common denominator seems to be that all designers want their real-time system to be predictable.

Predictability: It means that it should be possible to show, demonstrate or prove that requirements are met subjects to any assumptions made, for example, concerning failures and workloads. In other words, predicitability is always subject to underlying assumptions being made.

Categorizing real-time systems:

Real-time systems are those systems in which the correctness of the system depends not only on the logical results of computations, but also on the time at which the results are produced

granularity of the deadline and laxity of the tasks

In a real-time system some of the tasks have deadlines and/or periodic timing constraints. If the time between when a task is activated and when it must complete execution is short, then the deadline is tight. This implies that the operating system reaction has to be short, and the scheduling algorithm to be executed must be fast and very simple. Tight time constraints may also arise when the deadline granularity is large, but the amount of computation required is also great. In other words even large granularity deadlines can be tight when the laxity is small. In many real-time systems tight timing constraints predominate and consequently designers focus on developing very fast and simple technique to react to this type of task activation.

* strictness of deadline

The strictness of the deadline refers to the value of executing a task after the deadline has passed. For a hard real-time task there is no value to executing the task after the deadline has passed. A soft real-time task retains some diminished value after its deadline so it should still be executed. Very different techniques are usually used for hard and soft real-time tasks. In many systems hard real-time tasks are preallocated

and prescheduled resulting in 100% of them making their deadlines, soft real-time tasks are often scheduled either with non-real time scheduling algorithms, (or) with algorithms that explicitly address the timing constraints but aim only at good average case performance, (or) with algorithms that combine importance and timing requirements.

*Reliability

Many real-time systems operate under severe reliability requirements. That is, if certain tasks, called critical tasks, miss their deadline then a catastrophe may occur. These tasks are usually guaranteed to make their deadlines by an off line analysis and by schemes that reserve resources for these tasks even if it means that those resources are idle.

pabilities of commercial real time systems:

The real time version of commercial operating systems are generally slower and less predictable than the proprietary Kernels, but have greater functionality and better software development environments - Very important considerations in many large (or) complex applications.

Another significant advantage is that they are based on a set of familiar interfaces that facilitate portability. Real-time capabilities can be added to operating systems in multiple ways. The real-time versions of linux that have been created and commercialized in recent years can be grouped into following categories.

* compliant Kernels:

In this approach, an existing real-time operating is modified such that linux binaries can be run without any modification. Essentially, the functionality and semantics of linux system calls need to be appropriately emulated under the native operating system.

For example, LynxOs from lynux works adpts this approach.

*Dual Kernals:

In this approach, a hard but thin real-time kernel sits below the native operating system and traps all accesses to and interrupts from the underlying hardware. The thin kernel schedules several hard real-time tasks co-located with it and runs the native OS as its lowest priority task. As a result, native applications can be run without change, while hard real-time tasks can get excellent performance and predictability. A means of communication is also provided between the thin real-time kernel and the native non real-time kernel for data exchange purposes. The downside of this approach is that there is no memory protection between the real-time tasks and the native/thin kernels. As a result, the failure of any real-time task can lead to a complete system crash. The thin real-time kernel also needs to have its own set of device drivers for real time functionality. RT-Linux is an example of this approach.

*core kernel modifications:

In this approach, changes are made to the core of a non real-time kernel in order to

ake it predictability and deterministic enough so
s to behave as a real-time OS. using fixed priority
scheduling with a O(1) scheduler, employing high
resolution timers, making Kernel preemptive,
Support for priority inheritance protocols to
minimize priority inversion, making interrupt
handlers schedulable using Kernel threads, the use of
periodic processes, replacing FIFO queues with
priority queues and optimizing long paths through
the Kernel are typical means of accomplishing this
goal.

*Resource Kernel approach:

In this approach, the Kernel is extended to
provide support for resource reservations in
addition to the traditional fixed priority
preemptive scheduling approach. The latter approach
can run into problems when a relatively high-
priority process overruns its expected execution time
(or) even goes into an infinite loop. Resource Kernel
support and enforce resource reservation, such that
no misbehaving task can directly impact the
timing behaviour of another task. CMU's Linux/RK
and its commercial cousin, TimeSys Linux/RT all into
this category.