RESOURCE SHARING FOR REAL TIME TASKS

Resource sharing among tasks- Priority inversion Problem - Priority inheritance and Priority ceiling Protocols – Features of commercial and open source real time operating systems: Vxworks, QNX, Micrium OS, RT Linux and Free RTOS

Sharing of critical resources among tasks requires a different set of rules, compared to the rules used for sharing resources such as a CPU among tasks. We have in the last Chapter discussed how resources such as CPU can be shared among tasks. Priority inversion is a operating system scenario in which a higher priority process is preempted by a lower priority process. This implies the inversion of the priorities of the two processes.

Problems due to Priority Inversion

Some of the problems that occur due to priority inversion are given as follows −

- **A system malfunction may occur if a high priority process is not provided the required resources.**
- **Priority inversion may also lead to implementation of corrective measures. These may include the resetting of the entire system.**
- **The performance of the system can be reduces due to priority inversion. This may happen because it is imperative for higher priority tasks to execute promptly.**
- **System responsiveness decreases as high priority tasks may have strict time constraints or real time response guarantees.**
- **Sometimes there is no harm caused by priority inversion as the late execution of the high priority process is not noticed by the system.**

Solutions of Priority Inversion

Some of the solutions to handle priority inversion are given as follows −

- **Priority Ceiling**

  All of the resources are assigned a priority that is equal to the highest priority of any task that may attempt to claim them. This helps in avoiding priority inversion

- **Disabling Interrupts**

  There are only two priorities in this case i.e. interrupts disabled and preemptible. So priority inversion is impossible as there is no third option.

- **Priority Inheritance**

  This solution temporarily elevates the priority of the low priority task that is executing to the highest priority task that needs the resource. This means that medium priority tasks cannot intervene and lead to priority inversion.

- **No blocking**

  Priority inversion can be avoided by avoiding blocking as the low priority task

blocks the high priority task.

- **Random boosting**

  The priority of the ready tasks can be randomly boosted until they exit the critical section.

Difference between Priority Inversion and Priority Inheritance

Both of these concepts come under Priority scheduling in Operating System. In one line, Priority Inversion is a problem while *Priority Inheritance* is a solution. Literally, Priority Inversion means that priority of tasks get inverted and *Priority Inheritance* means that priority of tasks get inherited. Both of these phenomena happen in priority scheduling. Basically, in *Priority Inversion*, higher priority task (H) ends up waiting for middle priority task (M) when H is sharing critical section with lower priority task (L) and L is already in critical section. Effectively, H waiting for M results in inverted priority i.e. Priority Inversion. One of the solution for this problem is Priority Inheritance.

In Priority Inheritance, when L is in critical section, L inherits priority of H at the time when H starts pending for critical section. By doing so, M doesn't interrupt L and H doesn't wait for M to finish. Please note that inheriting of priority is done temporarily i.e. L goes back to its old priority when L comes out of critical section.

Priority Inheritance Protocol (PIP)

Priority Inheritance Protocol (PIP) is a technique which is used for sharing critical resources among different tasks. This allows the sharing of critical resources among different without the occurrence of unbounded priority inversions.

Basic Concept of PIP :

The basic concept of PIP is that when a task goes through priority inversion, the priority of the lower priority task which has the critical resource is increased by the priority inheritance mechanism. It allows this task to use the critical resource as early as possible without going through the preemption. It avoids the unbounded priority inversion.

Working of PIP :

When several tasks are waiting for the same critical resource, the task which is currently holding this critical resource is given the highest priority among all the tasks which are waiting for the same critical resource. Now after the lower priority task having the critical resource is given the highest priority then the intermediate priority tasks cannot preempt this task. This helps in avoiding the unbounded priority inversion. When the task which is given the highest priority among all tasks, finishes the job and releases the critical resource then it gets back to its original priority value (which may be less or equal). If a task is holding multiple critical resources then after releasing one critical resource it cannot go back to it original priority value. In this case it inherits the highest priority among all tasks waiting for the same critical resource.

Advantages of PIP :

Priority Inheritance protocol has the following advantages:

- **It allows the different priority tasks to share the critical resources.**
- **The most prominent advantage with Priority Inheritance Protocol is that it avoids the unbounded priority inversion.**

Disadvantages of PIP :

Priority Inheritance Protocol has two major problems which may occur:

Deadlock :

There is possibility of deadlock in the priority inheritance protocol. For example, there are two tasks T_1 and T_2. Suppose T_1 has the higher priority than T_2. T_2 starts running first and holds the critical resource CR_2. After that, T_1 arrives and preempts T_2. T_1 holds critical resource CR_1 and also tries to hold CR_2 which is held by T_2. Now T_1 blocks and T_2 inherits the priority of T_1 according to PIP. T_2 starts execution and now T_2 tries to hold CR_1 which is held by T_1. Thus, both T_1 and T_2 are deadlocked.

Chain Blocking :

When a task goes through priority inversion each time it needs a resource then this process is called chain blocking. For example, there are two tasks T_1 and T_2. Suppose T_1 has the higher priority than T_2. T_2 holds the critical resource CR_1 and CR_2. T_1 arrives and requests for CR_1. T_2 undergoes the priority inversion according to PIP. Now, T_1 request CR_2, again T_2 goes for priority inversion according to PIP. Hence, multiple priority inversion to hold the critical resource leads to chain blocking.

Priority Ceiling Protocol

In real-time computing, the priority ceiling protocol is a synchronization protocol for shared resources to avoid unbounded priority inversion and mutual deadlock due to wrong nesting of critical sections. In this protocol each resource is assigned a priority ceiling, which is a priority equal to the highest priority of any task which may lock the resource. The protocol works by temporarily raising the priorities of tasks in certain situations, thus it requires a scheduler that supports dynamic priority scheduling. It is a job task synchronization protocol in a real-time system that is better than Priority inheritance protocol in many ways. Real-Time Systems are multitasking systems that involve the use of semaphore variables, signals, and events for job synchronization. In Priority ceiling protocol an assumption is made that all the jobs in the system have a fixed priority. It does not fall into a deadlock state.

The chained blocking problem of the Priority Inheritance Protocol is resolved in the Priority Ceiling Protocol.

The basic properties of Priority Ceiling Protocols are:

1. **Each of the resources in the system is assigned a priority ceiling.**
2. **The assigned priority ceiling is determined by the highest priority among all the jobs which may acquire the resource.**
3. **It makes use of more than one resource or semaphore variable, thus eliminating chain blocking.**
4. **A job is assigned a lock on a resource if no other job has acquired lock on that resource.**
5. **A job J, can acquire a lock only if the job's priority is strictly greater than the priority ceilings of all the locks held by other jobs.**
6. **If a high priority job has been blocked by a resource, then the job holding that resource gets the priority of the high priority task.**
7. **Once the resource is released, the priority is reset back to the original.**
8. **In the worst case, the highest priority job J_1 can be blocked by T lower priority tasks in the system when J_1 has to access T semaphores to finish its execution.**

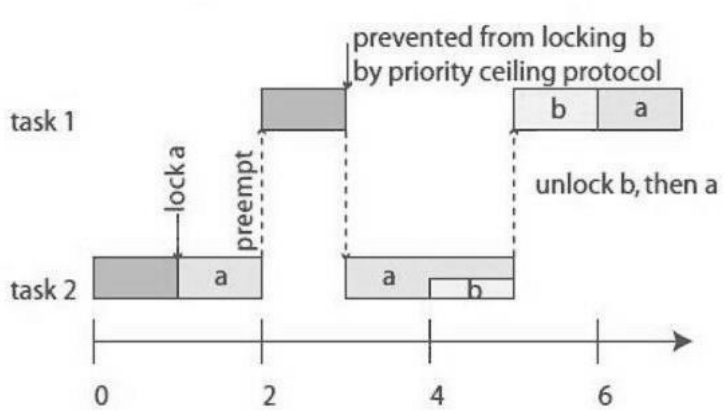

Fig:3.1: Priority Ceiling Protocol

Priority Scheduling Protocol can be used to tackle the problem of the priority inversion problem unlike that of Priority Inheritance Protocol. It makes use of semaphores to share the resources with the jobs in a real-time system.

Features of firmware and commercial real time operating systems:

Vx Works features:

- **High –performance**
- **Unix performance**
- **Unix -like, multitasking**
- **Environment scalable and hierarchical RTOS**
- **Hierarchical RTOS**
- **Host and target based development approach Supports**
- **Device Software Optimization ─ a new methodology that enables development and running of device software faster, better and more reliably Vx works RTOS Kernel**
- **VxWorks 6.x processor abstraction layer**

- **The layer enables application design for new versions later by just changing the layer hardware**
- **interface**
- **Supports advanced processor architectures ─ ARM, Cold Fire, MIPS, Intel, SuperH.**
- **Hard real time applications**
- **Supports kernel mode execution of Supports kernel mode execution of tasks**
- **Supports open source Linux and TIPC protocol**
- **Provides for the preemption points at kernel**
- **Provides preemptive as well as round robin scheduling**
- **Support POSIX standard asynchronous IOs**
- **Support UNIX standard buffered I/Os**
- **PTTS 1.1 (Since Dec. 2007)**
- **IPCs in TIPC for network and clustered system environment**
- **POSIX 1003.1b standard IPCs and interfaces additional availability**
- **Separate context for tasks and ISRs**

Micrium OS:

µC/OS is a full-featured embedded operating system. Features support for TCP/IP, USB, CAN bus, and Modbus. Includes a robust file system, and graphical user interface

Reliable: Micrium software includes comprehensive documentation, full source code, powerful debugging features, and support for a huge range of CPU architectures. Micrium software offers unprecedented ease-of-use, a small memory footprint, remarkable energy efficiency, and all with a full suite of protocol stacks.Real-Time Kernels

Portable. Offering unprecedented ease-of-use, µC/OS kernels are delivered with complete source code and in-depth documentation. The µC/OS kernels run on huge number of processor architectures

Scalable. The µC/OS kernels allow for unlimited tasks and kernel objects. The kernels' memory footprint can be scaled down to contain only the features required for your application, typically 6–24 KBytes of code space and 1 KByte of data space.

Efficient. Micrium's kernels also include valuable runtime statistics, making the internals of your application observable. Identify performance bottlenecks, and optimize power usage, early in your development cycle.
The features of the µC/OS kernels include:

- **Preemptive multitasking real-time kernel with optional round robin scheduling**
- **Delivered with complete, clean, consistent source code with in-depth documentation.**

- **Highly scalable: Unlimited number of tasks, priorities and kernel objects**
- **Resource-efficient: 6K to 24K bytes code space, 1K+ bytes data space)**
- **Very low interrupt disable time**
- **Extensive performance measurement metrics (configurable)**
- **Certifiable for safety-critical applications**
- **Micrium provides two extensions to the µC/OS-II kernel that provide memory protection and greater stability and safety for the applications.**

µC/OS-MPU is an extension for Micrium's µC/OS-II kernel that provides memory protection. The µC/OS-MPU extension prevents applications from accessing forbidden locations, thereby protecting against damage to safety-critical applications, such as medical devices and avionics systems. µC/OS-MPU builds a system with MPU processes, with each containing one or more tasks or threads. Each process has an individual read, write, and execution right. Exchanging data between threads is accomplished in the same manner using µC/OS-II tasks, however handling across different processes is achieved by the core operating system.

µC/OS-MPU includes the following features:

- **Prevents access of forbidden locations**
- **Appropriate for safety-critical applications**
- **Easy integration of protocol stacks, graphical modules, FS libraries**
- **Simplified debugging and error diagnosis**
- **Available for any microcontroller equipped with a hardware Memory Protection Unit (MPU) or Memory Management Unit (MMU).**
- **Third-party certification support available**

Debugging and error diagnosis is simplified as an error management system provides information on the different processes. The hardware protection mechanism cannot be bypassed by software. Existing µC/OS-applications can be adapted with minimal effort. µC/OS-MPU is available for any microcontroller containing a Memory Protection Unit (MPU) or Memory Management Unit (MMU). Third-party certification support is also available. µC/TimeSpaceOS is an extension for Micrium's µC/OS-II kernel that manages both memory and time allocated to diverse types of applications. With µC/TimeSpaceOS you can certify complex systems for safety-critical applications cost-effectively.

Its features and benefits include:

- **Memory protection so that multiple applications cannot influence, disturb or interact with each other**
- **Deterministic and run-time guaranteed**
- **Configurable so that virtual Dynamic Random Access Memory (DRAM) is optionally available**
- **A small footprint for use in a wide-range of applications**
- **Compatibility: can be used within the protected segment in existing µC/OS-II applications**
- **Certification according to DO178B and IEC61508**
- **Available for a large number of microcontrollers; contact us for details**

µC/TimeSpaceOS makes it possible for several independent applications (with or without real-time kernels) within one environment to be executed on one target hardware platform. It guarantees that the applications will not influence or interfere with each other. Each application is developed in a protected memory area (partition). The application is independent with respect to other partitions. This makes it easier for multiple developers to develop complex control devices. Each partition can be considered its own virtual CPU.

Features of RT Linux:

- **Multi-tasking**
- **Priority-based scheduling**
- **Application tasks should be programmed to suit**
- **Ability to quickly respond to external interrupts**
- **Basic mechanisms for process communication and synchronization**
- **Small kernel and fast context switch**

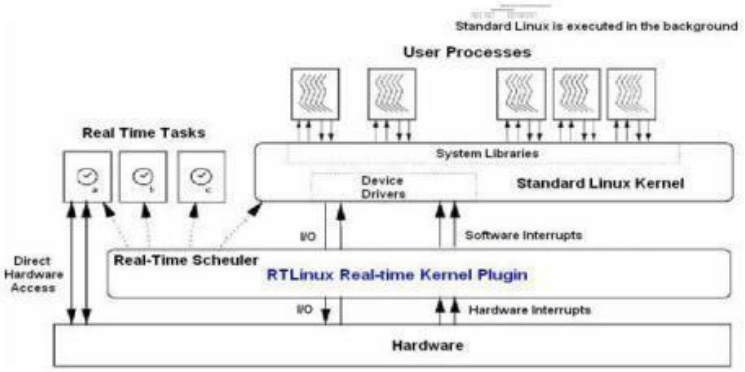

Fig:3.2: Features of RT Linux

Priority based kernel for embedded applications e.g. OSE, Vx Works, QNX, VRTX32, pSOS . Many of them are commercial kernels . Applications should be designed and programmed to suite priority-based scheduling e.g deadlines as priority etc . Real Time Extensions of existing time-sharing OS. e.g. Real time Linux, Real time NT by e.g locking RT tasks in main memory, assigning highest priorities etc .Research RT Kernels e.g. SHARK, TinyOS . Run-time systems for RT programming languages e.g. Ada, Erlang, Real-Time Java

Free RTOS and C Executive:

Free RTOS is a popular real-time operating system kernel for embedded devices, which has been ported to 35 microcontrollers. It is distributed under the GPL with an additional restriction and optional exception. The restriction forbids benchmarking while the exception permits users' proprietary code to remain closed source while maintaining the kernel itself as open source, thereby facilitating the use of Free RTOS in proprietary applications.

Free RTOS is designed to be small and simple. The kernel itself consists of only three or four C files. To make the code readable, easy to port, and maintainable, it is written mostly in C, but there are a few assembly functions included where needed (mostly in architecture-specific scheduler routines). Thread priorities are supported. In addition there are four schemes of memory allocation provided:

Allocate only;

- **Allocate and free with a very simple, fast, algorithm;**
- **A more complex but fast allocate and free algorithm with memory coalescence;**
- **C library allocate and free with some mutual exclusion protection**

Key features:

- **Very small memory footprint, low overhead, and very fast execution.**
- **Tick-less option for low power applications.**
- **Equally good for hobbyists who are new to OSes, and professional developers working on commercial products.**
- **Scheduler can be configured for either preemptive or cooperative operation.**
- **Co routine support (Co routine in Free RTOS is a very simple and lightweight task that has very limited use of stack)**
- **Trace support through generic trace macros. Tools such as Trace alyzer (a.k.a. Free RTOS+Trace, provided by the Free RTOS partner Percepio) can thereby record and visualize the runtime behavior of Free RTOS-based systems. This includes task scheduling and kernel calls for semaphore and queue operations. Trace analyzer is a commercial tool, but also available in a feature-limited free version.**

Features of a RTOS:

- **Allows multi-tasking**
- **Scheduling of the tasks with priorities**

- **Synchronization of the resource access**
- **Inter-task communication**
- **Time predictable**
- **Interrupt handling**

Predictability of timing

- **The timing behavior of the OS must be predictable**
- **For all services of the OS, there is an upper bound on the execution time**
- **Scheduling policy must be deterministic**
- **The period during which interrupts are disabled must be short (to avoid unpredictable delays in the processing of critical events)**

QNX RTOS v6.1

The QNX RTOS v6.1 has a client-server based architecture. QNX adopts the approach of implementing an OS with a 10 Kbytes micro-kernel surrounded by a team of optional processes that provide higher-level OS services .Every process including the device driver has its own virtual memory space. The system can be distributed over several nodes, and is network transparent. The system performance is fast and predictable and is robust. It supports Intel x86family of processors, MIPS, PowerPC, and Strong ARM .

QNX has successfully been used in tiny ROM-based embedded systems and in several hundred node distributed systems.

VxWorks (Wind River Systems)

VxWorks is the premier development and execution environment for complex real-time and embedded applications on a wide variety of target processors. Three highly integrated components are included with Vxworks: a high performance scalable real-time operating system which executes on a target processor; a set of powerful cross-development tools; and a full range of communications software options such as Ethernet or serial line for the target connection to the host. The heart of the OS is the Wind microkernel which supports multitasking, scheduling, inter task management and memory management. All other functionalities are through processes. There is no privilege protection between system and application and also the support for communication between processes on different processors is poor.

TEXT / REFERENCE BOOKS

1. Jane W. S Liu, "Real Time Systems" Pearson Higher Education, 3rd Edition, 2000.
2. Raj Kamal, "Embedded Systems- Architecture, Programming and Design" Tata McGraw
Hill, 2nd Edition, 2014.
3. Jean J. Labrosse, "Micro C/OS-I : The real time kernel" CMP Books, 2nd Edition,2015.
5. Richard Barry, "Mastering the Free RTOS: Real Time Kernel", Real Time Engineers Ltd,

1st Edition, 2016.
6. David E. Simon, " An Embedded Software Primer", Pearson Education Asia Publication