ISSN: 2277-3754
ISO 9001:2008 Certified
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 4, Issue 4, October 2014

An Overview: Real-Time Task Scheduling Using Preprocess Scheduler

Dr. Sajidullah S. Khan, R. N. Khobragade, Dr. N. A. Koli
Assistant Professor, ICSR, VMV, JMT and J J P Science College, Nagpur.
Researcher, Dept. of Computer Science and Engg. Sant Gadge Baba Amravati University, Amravati.
Head, Computer Center, Sant Gadge Baba Amravati University, Amravati.

Abstract - A Real Time System is a system where the time at which events occurs is important. Real Time Scheduling is fundamentally concerned with satisfying application time constraints. Task scheduling is the main activity in the design of Real-Time System. It assures both functionality and safety of such systems. RTS can be modeled as a set of periodic tasks that must be completed before specific deadlines. Adaptive Real Time Systems are designed to handle the undesirable effects such as overload and deadline misses dynamically by softly degrading performances. In adaptive soft real time system an acceptable deadline misses and delays are tolerable. The main objective of this work is to design and development of efficient preprocess scheduler that will select the algorithm which is best suited for the particular problem in the real time environment.

***Keywords*-Real Time System, Task Scheduling.**

I. INTRODUCTION

A Real Time System is a system where the time at which events occur is important. Real Time Scheduling is fundamentally concerned with satisfying application time constraints. Such as overload and deadline misses Adaptive Real Time Systems are designed to handle the undesirable effects dynamically by softly degrading performances. In adaptive soft real time system an acceptable deadline misses and delays are tolerable [Litoiu & Tadei, 2001]. Task scheduling is the main activity in the design of Real-Time System (RTS). It assures both functionality and safety of such systems. RTS can be modeled as a set of periodic tasks that must be completed before specific deadlines. The scheduling algorithms used in a particular application can have a significant impact on the functionality of the real-time system. One effect is to accumulate the aperiodic tasks at a point in time in an overloaded system. In this situation the scheduler may not be able to meet all of the aperiodic and periodic tasks deadlines [Churnetski, 2003]. Every algorithm has a specific set of type of input and produces the output corresponding to the input pattern. In this paper the objective is to find such the undesirable effects dynamically by softly degrading performances. In adaptive soft real time system an acceptable deadline misses and delays are tolerable [Litoiu & Tadei, 2001]. Task scheduling is the main activity in the design of Real-Time System (RTS). It assures both functionality and safety of such systems. RTS can be modeled as a set of periodic tasks that must be completed before specific deadlines. The scheduling algorithms used in a particular application can have a significant impact on the functionality of the real-time system. One effect is to accumulate the aperiodic tasks at a point in time in an overloaded system. In this situation the scheduler may not be able to meet all of the aperiodic and periodic tasks deadlines [Churnetski, 2003]. Every algorithm has a specific set of type of input and produces the output corresponding to the input pattern. In this paper the objective is to find such algorithms and then design a scheduler that will select the particular algorithm depending upon the given input pattern.

II. SCHEDULING

A. Scheduling Mechanisms

A multiprogramming operating system allows more than one process to be loaded into the executable memory at a time and for the loaded process to share the CPU using time-multiplexing. Part of the reason for using multiprogramming is that the *operating system itself is* maintaining the Integrity of the Specifications *implemented as one or more processes*, so there must be a way for the operating system and application processes to share the CPU. Another main reason is the need for processes to perform *I/O operations* in the normal course of computation. Since I/O operations ordinarily require orders of magnitude more time to complete than do CPU instructions, multiprogramming systems allocate the CPU to another process whenever a process invokes an I/O operation. Here we are discussing the scheduling to schedule the process for execution [David Kalinsky, 2004].

1. Context Switching

Typically there are several tasks to perform in a computer system. So if one task requires some I/O operation, you want to initiate the I/O operation and go on to the next task. You will come back to it later. This act of switching from one process to another is called a "Context Switch". When you return back to a process, you should resume where you left off. For all practical purposes, this process should never know there was a switch, and it should look like this was the only process in the system [David Kalinsky, 2004]. To implement this, on a context switch, you have to save the context of the current process select the next process to run restore the context of this new process.

2. What is the context of a process?

Program Counter, Stack Pointer, Registers Code + Data + Stack (also called Address Space). Other state

ISSN: 2277-3754
ISO 9001:2008 Certified
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 4, Issue 4, October 2014

information maintained by the OS for the process (open files, scheduling info, I/O devices being used etc.) All this information is usually stored in a structure called Process Control Block (PCB). All the above has to be saved and restored.

3. What does a context_switch() routine look like?

```
context_switch() {
Push registers onto stack
Save ptrs to code and data.
Save stack pointer
Pick next process to execute
Restore stack ptr of that process
/* You have now switched the stack */
Restore ptrs to code and data.
Pop registers
Return    }
```

B. Non-Preemptive Vs Preemptive Scheduling

Non-Preemptive: Algorithms are designed so that once a process enters the running state (is allowed a process), it is not removed from the processor until it has completed its service time (or it explicitly yields the processor). context_switch () is called only when the process terminates or blocks.

Preemptive: Preemptive algorithms are driven by the notion of prioritized computation. The process with the highest priority should always be the one currently using the processor. If a process is currently using the processor and a new process with a higher priority enters, the ready list, the process on the processor should be removed and returned to the ready list until it is once again the highest-priority process in the system. context_switch() is called even when the process is running usually done via a timer interrupt.

III. TASK

A Task T_i is a sequential program that is activated multiple times by external or internal events. At each activation (also called *job*) a piece of code is executed, and at the end of the allocated quantum task is blocked waiting for the next activation.

A. Task Types

A task is a sequential program that is invoked for execution by the occurrence of a particular event. Tasks may be periodic, aperiodic or sporadic in nature. A periodic task is characterized by a release time, a deadline, and a period. The release time is the time at which the task is ready to execute, the deadline is the time by which the task must complete execution, and the period is the exact spacing between successive invocations of the task. When the release time of the task is specified before it is scheduled, the task is called a concrete periodic task. When release times are arbitrary, a task is invoked periodically after its first release. Aperiodic tasks have soft or no deadlines. Sporadic tasks are tasks that may enter and leave the system at any time. Sporadic tasks are characterized by a release time, a deadline, and a period. For a sporadic task, the period represents the minimum time after which the invocation of the next task occurs. When release times are specified in advance, scheduling decisions can be made off-line or statically. When release times are arbitrary, scheduling decisions are made on-line. Figure. 2 illustrate this classification [Litoiu & Tadei, 2001].

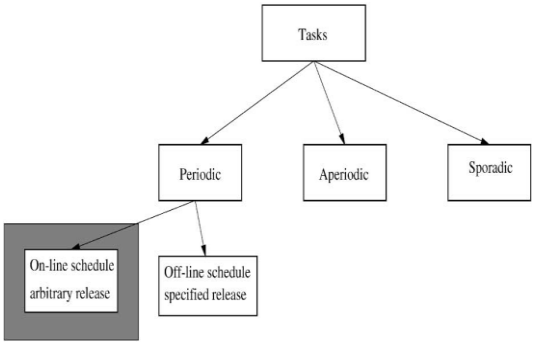

Fig 1. Classification of the different types of tasks

1. Periodic

Scheduling of the tasks that needs to run periodically with the fixed periods can be *periodic* and can be done with a CPU load very close to 1. An example of a periodic task is as follows. There may be inputs at a port with predetermine periods, and the inputs are in succession without any time-gap [Raj Kamal, 2003].

2. Aperiodic

When a task needs to run only once, then it is aperiodic (one shot) in an application [Raj Kamal, 2003].

3. Sporadic

When a task cannot be scheduled at fixed periods, its schedule is called sporadic. For example, if a task is expected to receive inputs at variable time gaps, then the task schedule is sporadic. An example is the packets from the routers in a network. The variable time gaps must be within defined limits [Raj Kamal, 2003].

B. Task scheduling

Most RTOSs do their scheduling of tasks using a scheme called "priority-based preemptive scheduling." Each task in a software application must be assigned a priority, with higher priority values representing the need for quicker responsiveness. Very quick responsiveness is made possible by the "preemptive" nature of the task scheduling. "Preemptive" means that the scheduler is allowed to stop any task at any point in its execution, if it determines that another task needs to run immediately [Liu, 2000]. The basic rule that governs priority-based preemptive scheduling is that at every moment in time, "The Highest Priority Task that is ready to Run will be the Task that must be running." In other words, if both a low-priority task and a higher-priority task are ready to run, the scheduler will allow the higher-priority task to run first. The low-priority task will only get to run after the higher-priority task has finished with its current work [David

ISSN: 2277-3754
ISO 9001:2008 Certified
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 4, Issue 4, October 2014

Kalinsky, 2004]. What if a low-priority task has already begun to run, and then a higher-priority task becomes ready? This might occur because of an external world trigger such as a switch closing. A priority-based preemptive scheduler will behave as follows: It will allow the low-priority task to complete the current assembly-language instruction that it is executing. (But it won't allow it to complete an entire line of high-level language code; nor will it allows it to continue running until the next clock tick.) It will then immediately stop the execution of the low-priority task, and allow the higher-priority task to run. After the higher-priority task has finished its current work, the low-priority task will be allowed to continue running. This is shown in *Figure 1*, where the higher-priority task is called "*Mid-Priority Task*." Of course, while the mid-priority task is running, an even higher-priority task might become ready. This is represented in *Figure 1* by "*Trigger_2*" causing the "High-Priority Task" to become ready. In that case, the running task ("*Mid-Priority Task*") would be preempted to allow the high-priority task to run. When the high-priority task has finished its current work, the mid-priority task would be allowed to continue. And after both the high-priority task and the mid-priority task complete their work, the low-priority task would be allowed to continue running. This situation might be called "nested preemption" [David Kalinsky, 2004].

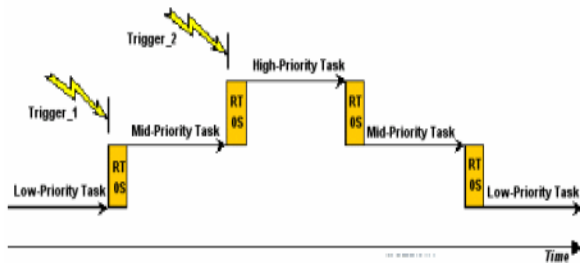

Fig 2: Timeline for Priority-based Preemptive Scheduling Examples

Each time the priority-based preemptive scheduler is alerted by an external world trigger (such as a switch closing) or a software trigger (such as a message arrival); it must go through the following *5 steps:* called as "***task switching***."

1. Determine whether the currently running task should continue to run.
2. If not Determine which task should run next?
3. Save the environment of the task that was stopped (so it can continue later).
4. Set up the running environment of the task that will run next.
5. Allow this task to run.

3.3. Real Time System

Real-time Systems are computer systems that require responses within specified time limits or constraints. Many Real-time Systems are digital control systems comprised entirely of binary logic or a microprocessor dedicated to one software application that is own operating system. In recent years, the reliability of general purpose real time operating system (RTOS) consisting of a scheduler and system resource management has improved [Churnetski, 2003].

1. Hard Real Time Task

Hard Real-time system are required to complete a critical task within a guaranteed amount of time generally a process is submitted along with a statement of the amount of time in which it needs to complete or perform I/O. The scheduler then either admits the process, guaranteeing that the process will complete on time or rejects the requests as impossible. This is known as resource reservation such a guarantee required that the scheduler know exactly how long each type of operating system function takes to perform, and therefore each operation must be guaranteed to take a maximum amount of time. Such a guarantee is impossible in a system with secondary storage or virtual memory, because these subsystems cause unavoidable and unforeseen variation in amount of time to execute a particular process. Therefore Hard Real-time system are compose of special purpose software running on hardware dedicated to their critical process, and lack the full functionality of modern computer and operating system [Churnetski, 2003].

2. Soft Real Time Task

Soft Real-time computing is less restrictive. It requires that critical process receive priority over less fortunate ones. Although adding soft real time functionality to a time sharing system may cause an unfair allocation of resources and may result in longer delays, or even starvation, for some processes, it is at least possible to achieve. The result is general purpose systems that can also multimedia, high speed interactive graphics, and a variety of task that would not function acceptably in an environment that does not support soft real time computing. Implementing soft real time functionality requires carefully design of the scheduler and related aspect of the operating system. First, the system must have priority scheduling, and real time processes must not degrade over time, even though the priority of non real time processes may. Second, the dispatch latency must be small. The smaller the latency, the faster a real time processes can start executing ones it is run-able [Churnetski, 2003].

3 Adaptive

Adaptive Real-Time System is designed to handle the undesirable effects such as overload and deadline misses, dynamically by softly degrading performances. In Adaptive Real-Time System, an acceptable deadline misses and delays are tolerable [Ahmad et.al. 2003].

4. Scheduling Algorithm

Many dynamic scheduling algorithms are available as follows:

1. Cyclic Scheduling
2. Deterministic Scheduling
3. Capacity Base Scheduling

ISSN: 2277-3754
ISO 9001:2008 Certified
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 4, Issue 4, October 2014

4. Dynamic Priority Scheduling
5. Earliest Deadline First
6. Least Slack Time
7. Value Function Scheduling.[Raj Kamal,2003],

IV. TASK SCHEDULING REQUIREMENTS

The following are the basic requirements of an RTOS:

1. **Multi-tasking and preemptable:** To support multiple tasks in real-time applications, an RTOS must be multi-tasking and preemptable. The scheduler should be able to preempt any task in the system and give the resource to the task that needs it most. An RTOS should also handle multiple levels of interrupts to handle multiple priority levels.

2. **Dynamic deadline identification*:*** In order to achieve preemption, an RTOS should be able to dynamically identify the task with the earliest deadline. To handle deadlines, deadline information may be converted to priority levels that are used for resource allocation. Although such an approach is error prone, nonetheless it is employed for lack of a better solution.

3. **Predictable synchronization:** For multiple threads to communicate among themselves in a timely fashion, predictable inter-task communication and synchronization mechanisms are required. Semantic integrity as well as timeliness constitutes predictability. Predictable synchronization requires compromises. Ability to lock/unlock resources is one of the ways to achieve data integrity.

4. **Sufficient Priority Levels:** When using prioritized task scheduling, the RTOS must have a sufficient number of priority levels, for effective implementation. Priority inversion occurs when a higher priority task must wait on a lower priority task to release a resource and in turn the lower priority task is waiting upon a medium priority task. Two workarounds in dealing with priority inversion, namely priority inheritance and priority ceiling protocols (PCP), need sufficient priority levels. In a priority inheritance mechanism, a task blocking a higher priority task inherits the higher priority for the duration of the blocked task. In PCP a priority is associated with each resource which is one more than the priority of its highest priority user. The scheduler makes the priority of the accessing task equal to that of the resource. After a task releases a resource, its priority is returned to its original value. However, when a task's priority is increased to access a resource it should not have been waiting on another resource.

5. **Predefined latencies:** The timing of system calls must be defined using the following specifications:
   - ***Task switching latency*** or the time to save the context of a currently executing task and switch to another.
   - ***Interrupt latency*** or the time elapsed between the execution of the last instruction of the interrupted task and the first instruction of the *interrupt handler*.
   - ***Interrupt dispatch latency*** or the time to switch from the last instruction in the *interrupt handler* to the next task scheduled to run [Rivas & Harbour, 2003].

V. REAL-TIME TASK MODEL

A real-time application is specified by means of a set of tasks. Real-time tasks are the basic executable entities that are scheduled; they may be periodic or aperiodic, and have hard (late data are bad data) or soft (late data may still be good data) real-time constraints. The quality of scheduling depends on the exactness of these parameters, so their determination is an important aspect of real-time design.

- **r**, task release time, i.e. the triggering time of the task execution request.
- **C**, task worst-case computation time, when the processor is fully allocated to it.
- **D**, task relative deadline, i.e. the maximum acceptable delay for its processing.
- **T**, task period (valid only for periodic tasks). When the task has hard real-time constraints, the relative deadline allows computation of the absolute deadline d = r + D. Transgression of the absolute deadline causes a timing fault. Also, when tasks are allowed to access shared resources, their access needs to be controlled in order to maintain data consistency [Rivas & Harbour, 2003].

The Algorithms consider in this paper are as follows:

1) First Come First Serve (FCFS)
2) Shortest Job First (SJF), or Earliest
3) Deadline First (EDF)
4) Priority Scheduling Algorithm
5) Round Robin Scheduling Algorithm

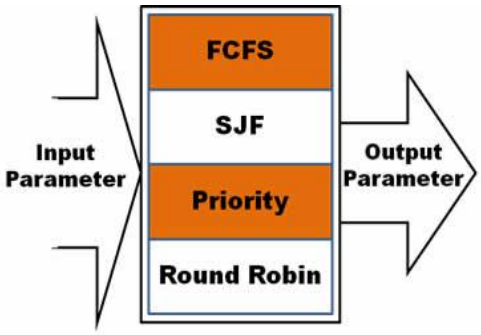

Figure 3. Block diagram for Pre-process scheduler

The Input Parameters for the above algorithms are as follows:

1. First Come First Serve (FCFS): Number of processes & Deadline.
2. Shortest Job First (SJF): Number of processes, Arrival time, Deadline.
3. Priority Scheduling Algorithm: Number of processes, Priority, Deadline.
4. Round Robin Scheduling Algorithm: Number of processes, Arrival time, Time quantum, Deadline.

ISSN: 2277-3754
ISO 9001:2008 Certified
International Journal of Engineering and Innovative Technology (IJEIT)
Volume 4, Issue 4, October 2014

The Output Parameters for the above algorithms are as follows: Waiting Time, Average Waiting Time, and Completion Time.

VI. CONCLUSION

A Real Time System is a system where the time at which events occur is important. Real Time Scheduling is fundamentally concerned with satisfying application time constraints. The main objective of this paper is to design and development of efficient preprocess scheduler that will select the algorithm which is best suited for the particular problem in the real time environment called as preprocess scheduler.

REFERENCES

[1] [Ahmad et.al., 2003] Idawaty Ahmad, S. Shamala, M. Othman and Muhammad Fauzan Othman "A Preemptive Utility Accrual Scheduling Algorithm for Adaptive Real Time System".

[2] [Churnetski, 2003] Kevin Churnetski "A comparison of real-time scheduling algorithms using visualization of tasks and evaluation of real-time extensions to Linux "Computer Science-RIT in 2003.

[3] [David Kalinsky, 2004] David Kalinsky "Basic concepts of real-time operating systems" www.linuxdevice.com.in 2004.

[4] [Litoiu & Tadei, 2001] Marin Litoiu, Roberto Tadei, "Fuzzy scheduling with application to Real-time system", Elsevier Science B. V. in 2001.

[5] [Rivas & Harbour, 2003] Mario Aldea Rivas and Michael González Harbour "MaRTE OS: An Ada Kernel for Real-Time Embedded Applications" Department of Electronics Compotators Universidad de Cantabria, 39005-Santander, SPAIN.

[6] [Raj Kamal, 2003] Raj Kamal, "Embedded Systems Architecture Programming and Design", Tata McGraw-Hill, 2003.