Embedded Operating Systems - Part 2: Process scheduling

Tammy Noergaard - April 08, 2013

Editor's Note: Embedded Systems Architecture, 2nd Edition, is a practical and technical guide to understanding the components that make up an embedded system’s architecture. Offering detailed explanations and numerous code examples, the book provides a comprehensive get-up-and-running reference for those new to the field and those updating their skills. In Part 1, the author defined the unique characteristics of an embedded OS and describes how an embedded OS works with processes. In this installment, the author reviews the different types of process schedulers found in embedded OSs.

Adapted from "Embedded Systems Architecture, 2nd Edition" by Tammy Noergaard (Newnes)

9.2.2 Process Scheduling

In a multitasking system, a mechanism within an OS, called a scheduler (shown in Figure 9-18), is responsible for determining the order and the duration of tasks to run on the CPU. The scheduler selects which tasks will be in what states (READY, RUNNING, or BLOCKED), as well as loading and saving the TCB information for each task. On some OSs the same scheduler allocates the CPU to a process that is loaded into memory and ready to run, while in other OSs a dispatcher (a separate scheduler) is responsible for the actual allocation of the CPU to the process.

![Figure 9-18. OS Block diagram and the scheduler.][3]
There are many scheduling algorithms implemented in embedded OSs, and every design has its strengths and tradeoffs. The key factors that impact the effectiveness and performance of a scheduling algorithm include its response time (time for scheduler to make the context switch to a ready task and includes waiting time of task in ready queue), turnaround time (the time it takes for a process to complete running), overhead (the time and data needed to determine which tasks will run next), and fairness (what are the determining factors as to which processes get to run). A scheduler needs to balance utilizing the system’s resources, keeping the CPU, I/O, as busy as possible, with task throughput, processing as many tasks as possible in a given amount of time. Especially in the case of fairness, the scheduler has to ensure that task starvation, where a task never gets to run, doesn’t occur when trying to achieve a maximum task throughput.

In the embedded OS market, scheduling algorithms implemented in embedded OSs typically fall under two approaches: non-pre-emptive and pre-emptive scheduling. Under non-pre-emptive scheduling, tasks are given control of the master CPU until they have finished execution, regardless of the length of time or the importance of the other tasks that are waiting.

**Title-1**

Scheduling algorithms based upon the non-pre-emptive approach include:

*First Come First Served (FCFS)/Run-To-Completion*, where tasks in the READY queue are executed in the order they entered the queue and where these tasks are run until completion when they are READY to be run (see Figure 9-19). Here, nonpre-emptive means there is no BLOCKED queue in an FCFS scheduling design. The response time of a FCFS algorithm is typically slower than other algorithms (i.e., especially if longer processes are in front of the queue requiring that other processes wait their turn), which then becomes a fairness issue since short processes at the end of the queue get penalized for the longer ones in front. With this design, however, starvation is not possible.

![Figure 9-19. FCFS scheduling.](image)

*Shortest Process Next (SPN)/Run-To-Completion*, where tasks in the READY queue are executed in the order in which the tasks with the shortest execution time are executed first (see Figure 9-20). The SPN algorithm has faster response times for shorter processes. However, then the longer processes are penalized by having to wait until all the shorter processes in the queue have run. In this scenario, starvation can occur to longer processes if the ready queue is continually filled with shorter processes. The overhead is higher than that of FCFS, since the calculation and storing of run times for the processes in the ready queue must occur.
Co-operative, where the tasks themselves run until they tell the OS when they can be context switched (for I/O, etc.). This algorithm can be implemented with the FCFS or SPN algorithms, rather than the run-to-completion scenario, but starvation could still occur with SPN if shorter processes were designed not to “cooperate,” for example (see Figure 9-21).

Non-pre-emptive algorithms can be riskier to support since an assumption must be made that no one task will execute in an infinite loop, shutting out all other tasks from the master CPU. However, OSs that support non-pre-emptive algorithms don’t force a context-switch before a task is ready, and the overhead of saving and restoration of accurate task information when switching between tasks that have not finished execution is only an issue if the non-pre-emptive scheduler implements a cooperative scheduling mechanism. In pre-emptive scheduling, on the other hand, the OS forces a context-switch on a task, whether or not a running task has completed executing or is cooperating with the context switch. Common scheduling algorithms based upon the pre-emptive approach include: Round Robin/FIFO (First In First Out) scheduling, priority (pre-emptive) scheduling, and EDF (Earliest Deadline First)/Clock Driven scheduling.

Round Robin/FIFO Scheduling. The Round Robin/FIFO algorithm implements a FIFO queue that stores ready processes (processes ready to be executed). Processes are added to the queue at the end of the queue and are retrieved to be run from the start of the queue. In the FIFO system, all processes are treated equally regardless of their workload or interactivity. This is mainly due to the possibility of a single process maintaining control of the processor, never blocking to allow other processes to execute.
running isn’t finished executing by the end of its allocated time slice, it is pre-empted and returned to the back of the queue to complete executing the next time its turn comes around. If a process finishes running before the end of its allocated time slice, the process voluntarily releases the processor, and the scheduler then assigns the next process of the FIFO queue to the processor (see Figure 9-22).

![Figure 9-22. Round Robin/FIFO scheduling.][7]

While Round Robin/FIFO scheduling ensures the equal treatment of processes, drawbacks surface when various processes have heavier workloads and are constantly pre-empted, thus creating more context switching overhead. Another issue occurs when processes in the queue are interacting with other processes (such as when waiting for the completion of another process for data) and are continuously pre-empted from completing any work until the other process of the queue has finished its run. The throughput depends on the time slice. If the time slice is too small, then there are many context switches, while too large a time slice isn’t much different from a non-pre-emptive approach, like FCFS. Starvation is not possible with the round-robin implementation.

**Priority (Pre-Emptive) Scheduling.** The priority pre-emptive scheduling algorithm differentiates between processes based upon their relative importance to each other and the system. Every process is assigned a priority, which acts as an indicator of orders of precedence within the system. The processes with the highest priority always pre-empt lower priority processes when they want to run, meaning a running task can be forced to block by the scheduler if a higher priority task becomes ready to run. Figure 9-23 shows three tasks (1, 2, and 3, where task 1 is the lowest priority task and task 3 is the highest), and task 3 pre-empts task 2 and task 2 pre-empts task 1.
Pre-emptive priority scheduling resolves some of the problems associated with round-robin/FIFO scheduling in dealing with processes that interact or have varying workloads, new problems can arise in priority scheduling including:

**Process starvation:** A continuous stream of high priority processes keep lower priority processes from ever running. Typically resolved by aging lower priority processes (as these processes spend more time on queue, increase their priority levels).

**Priority inversion:** Higher priority processes may be blocked waiting for lower priority processes to execute, and processes with priorities in between have a higher priority in running, thus the lower priority as well as higher priority processes don’t run (see Figure 9-24).

How to determine the priorities of various processes. Typically, the more important the task, the higher the priority it should be assigned. For tasks that are equally important, one technique that can be used to assign task priorities is the Rate Monotonic Scheduling (RMS) scheme, in which tasks are assigned a priority based upon how often they execute within the system. The premise behind this model is that, given a pre-emptive scheduler and a set of tasks that are completely independent (no shared data or resources) and are run periodically (meaning run at regular time intervals), the more often a task is executed within this set, the higher its priority should be. The RMS Theorem says that if the above assumptions are met for a scheduler and a set of “n” tasks, all timing deadlines will be met if the inequality \( \sum \frac{E_i}{T_i} \leq n (2^{1/n} - 1) \) is verified, where \( i \) is the Periodic task, \( n \) is the number of Periodic tasks, \( T_i \) is the execution period of task \( i \), \( E_i \) is the worst-case execution time of task \( i \), and \( E_i/T_i \) is the fraction of CPU time required to execute task \( i \). So, given two tasks that have been prioritized according to their periods, where the shortest period task has been assigned the highest priority, the “\( n(2^{1/n} - 1) \)” portion of the inequality would equal approximately 0.828, meaning the CPU utilization of these tasks should not exceed about 82.8% in order to meet all hard deadlines. For 100 tasks that have been prioritized according to their periods, where the shorter period tasks have been assigned the higher priorities, CPU utilization of these tasks should not exceed approximately 69.6% \((100 \times (21/100 - 1))\) in order to meet all deadlines.

**EDF/Clock Driven Scheduling.** As shown in Figure 9-25, the EDF/Clock Driven algorithm schedules priorities to processes according to three parameters: frequency (number of times a process is run), deadline (when processes execution needs to be completed), and duration (time it takes to execute the process). While the EDF algorithm allows for timing constraints to be verified and enforced
(basically guaranteed deadlines for all tasks), the difficulty is defining an exact duration for various processes. Usually, an average estimate is the best that can be done for each process.

![Figure 9-25. EDF scheduling.[2]](image)

### Real-World Advice

*To Benefit Most from a Fixed-Priority Pre-Emptive OS*

Algorithms for assigning priorities to OS tasks are typically classified as fixed-priority where tasks are assigned priorities at design time and do not change through the lifecycle of the task, dynamic-priority where priorities are assigned to tasks at runtime, or some combination of both algorithms. Many commercial OSs typically support only the fixed-priority algorithms, since it is the least complex scheme to implement. The key to utilizing the fixed-priority scheme is:

To assign the priorities of tasks according to their periods, so that the shorter the periods, the higher the priorities.

To assign priorities using a fixed-priority algorithm (like the Rate Monotonic Algorithm, the basis of RMS) to assign fixed priorities to tasks and as a tool to quickly to determine if a set of tasks is schedulable.

To understand that in the case when the inequality of a fixed-priority algorithm, like RMS, is not met, an analysis of the specific task set is required. RMS is a tool that allows for assuming that deadlines would be met in most cases if the total CPU utilization is below the limit (“most” cases meaning there are tasks that are not schedulable via any fixed-priority scheme). It is possible for a set of tasks to still be schedulable in spite of having a total CPU utilization above the limit given by the inequality. Thus, an analysis of each task’s period and execution time needs to be done in order to determine if the set can meet required deadlines.

To realize that a major constraint of fixed-priority scheduling is that it is not always possible to completely utilize the master CPU 100%. If the goal is 100% utilization of the CPU when using fixed priorities, then tasks should be assigned harmonic periods, meaning a task’s period should be an exact multiple of all other tasks with shorter periods.
Pre-Emptive Scheduling and the Real-Time Operating System (RTOS)

One of the biggest differentiators between the scheduling algorithms implemented within embedded OSs is whether the algorithm guarantees its tasks will meet execution time deadlines. If tasks always meet their deadlines (as shown in the first two graphs in Figure 9-26) and related execution times are predictable (deterministic), the OS is referred to as an RTOS.

![Figure 9-26. OSs and deadlines](image)

Pre-emptive scheduling must be one of the algorithms implemented within RTOS schedulers, since tasks with real-time requirements have to be allowed to pre-empt other tasks. RTOS schedulers also make use of their own array of timers, ultimately based upon the system clock, to manage and meet their hard deadlines.

Whether an RTOS or a non-RTOS in terms of scheduling, all will vary in their implemented scheduling schemes. For example, VxWorks (Wind River) is a priority-based and round-robin scheme, Jbed (Esmertec) is an EDF scheme, and Linux (Timesys) is a priority-based scheme. Examples 7, 8, and 9 examine further the scheduling algorithms incorporated into these embedded off-the-shelf OSs.

**Example 7: VxWorks Scheduling**

The Wind scheduler is based upon both pre-emptive priority and round-robin real-time scheduling algorithms. As shown in Figure 9-27a1, round-robin scheduling can be teamed with pre-emptive priority scheduling to allow for tasks of the same priority to share the master processor, as well as allow higher priority tasks to pre-empt for the CPU.
Without round-robin scheduling, tasks of equal priority in VxWorks would never pre-empt each other, which can be a problem if a programmer designs one of these tasks to run in an infinite loop. However, the pre-emptive priority scheduling allows VxWorks its real-time capabilities, since tasks can be programmed never to miss a deadline by giving them the higher priorities to pre-empt all other tasks. Tasks are assigned priorities via the “taskSpawn” command at the time of task creation:

```c
int taskSpawn(
    {Task Name},
    {Task Priority 0-255, related to scheduling and will be discussed in the next section},
    {Task Options - VX_FP_TASK, execute with floating point coprocessor,
    VX_PRIVATE_ENV, execute task with private environment
    VX_UNBREAKABLE, disable breakpoints for task
    VX_NO_STACK_FILL, do not fill task stack with 0x0E}
    {Task address of entry point of program in memory - initial PC value}
    {Up to 10 arguments for task program entry routine})
```

**Example 8: Jbed and EDF Scheduling**
Under the Jbed RTOS, all six types of tasks have the three variables—“duration,” “allowance,” and “deadline”—when the task is created for the EDF scheduler to schedule all tasks, as shown in the method (Java subroutine) calls below.
Example 9: TimeSys Embedded Linux Priority-Based Scheduling

As shown in Figure 9-27b1, the embedded Linux kernel has a scheduler that is made up of four modules:[9]

- **System call interface module**: acts as the interface between user processes and any functionality explicitly exported by the kernel.
- **Scheduling policy module**: determines which processes have access to the CPU.
- **Architecture specific scheduler module**: an abstraction layer that interfaces with the hardware (i.e., communicating with CPU and the memory manager to suspend or resume processes).
- **Architecture independent scheduler module**: an abstraction layer that interfaces between the scheduling policy module and the architecture specific module.

![Figure 9-27b1. Embedded Linux block diagram.][9]
The scheduling policy module implements a “priority-based” scheduling algorithm. While most Linux kernels and their derivatives are non-pre-emptable, have no rescheduling, and are not real-time, Timesys’ Linux scheduler is priority-based, but has been modified to allow for real-time capabilities. Timesys has modified the traditional Linux’s standard software timers, which are too coarsely grained to be suitable for use in most real-time applications because they rely on the kernel’s jiffy timer, and implements high-resolution clocks and timers based on a hardware timer. The scheduler maintains a table listing all of the tasks within the entire system and any state information associated with the tasks. Under Linux, the total number of tasks allowed is only limited to the size of physical memory available. A dynamically allocated linked list of a task structure, whose fields that are relevant to scheduling are highlighted in Figure 9-27b2, represents all tasks in this table.

![Figure 9-27b2. Task structure.][15]

After a process has been created in Linux, through the fork or fork/exec commands, for instance, its
priority is set via the setpriority command.

```c
int setpriority(int which, int who, int prio);
  which = PRIO_PROCESS, PRIO_PGRP, or PRIO_USER_
  who = interpreted relative to which
  prio = priority value in the range -20 to 20
```