

* Generalized Deterministic Task Scheduling Algorithm for Embedded Real-Time Operating Systems

Myoung-Jo Jung, Moon-Haeng Cho, Yong-Hee Kim, Cheol-Hoon Lee

Dept. of Computer Engineering, Chungnam National Univ.,
Yuseong-Gu, Daejeon, 305-764, Korea.
{mjjung, root4567, yonghee, clee}@cnu.ac.kr

Abstract - In recent years, there has been a rapid and wide spread proliferation of non-traditional embedded computing platforms such as digital camcorders, cellular phones, and portable medical devices. As applications become increasingly sophisticated and processing power increases, the application designer has to rely on the services provided by the real-time operating systems (RTOSs). These RTOSs must not only provide predictable services but must also be efficient and small in size. Kernel services should also be deterministic by specifying how long each service call will take to execute. Having this information allows the application designers to better plan their real-time application software so as not to miss the deadline of each task. In earlier work, we proposed a deterministic scheduling algorithm which can be generalized such that it makes the task scheduling time constant irrespective of the number of tasks created in an application. In this paper, we present the complete generalized algorithm to determine the highest priority in the ready list with 2^r levels of priorities for an arbitrary integer number of r . The proposed algorithm eliminates the restriction on the maximum number of task priorities imposed on the existing ones, without additional memory overhead.

Key words: *real-time task scheduling, real-time operating system.*

1. Introduction

Real-time computing systems must behave predictably even in unpredictable environments [1]. This predictability is ensured by system-level services, most important among them being the real-time operating system (RTOS). The RTOS must ensure that all real-time tasks complete by their deadlines and that no low-priority execution or communication activities are able to block higher-priority tasks for an extended period [2]. The wide variety of real-time applications and the variety of hardware used in these

systems have resulted in dozens of RTOSs being designed to support these applications [3-6].

The RTOS's real-time scheduler and task management service should guarantee real-time deadlines. The task scheduler's overhead (Δt) is the time consumed by the execution of the scheduler code. This has to do with managing the lists of tasks and selecting the highest-priority task to execute whenever some task blocks or unblocks. When a running task blocks, the RTOS must update some data structures to identify the task as being blocked and then pick a new task for execution. We call the overheads associated with these two steps the *blocking overhead* Δt_b and the *selection overhead* Δt_s , respectively. Similarly, when a blocked task unblocks, the RTOS must again update some internal data structures, incurring the *unblocking overhead* Δt_u . The RTOS must also pick a task to execute (since the newly-unblocked task may have higher priority than the previously-executing one), so the selection overhead is incurred as well. The typical implementation for the most commercial RTOSs is to have a queue of ready tasks sorted by task priorities. Tasks are blocked and unblocked by changing one variable in the appropriate task control block (TCB). All blocked and unblocked tasks are in a single queue sorted by priority, highest-priority task first. A single pointer **highestP** points to the highest-priority ready task, so Δt_s is $O(1)$ because **highestP** is the task which should execute next. To block a task, one variable is updated in the TCB (*i.e.*, $\Delta t_b = O(1)$), and now **highestP** is set to point to the next task in the ready queue (*i.e.*, $\Delta t_s = O(1)$). However, to unblock a task, the scheduler parses down the queue till it finds the appropriate place for the task in the sorted queue. This is why Δt_u takes $O(n)$ time, where n is the number of tasks. This means that the worst-case scheduling overhead increases as n increases. It has a significantly bad impact on the performance of a real-time kernel, since worst-case overheads should be taken into account so as to guarantee real-time deadlines [7,8]. Real-time kernel services should be deterministic by specifying how long each service call will take to execute. Having this information allows the

* This work was supported in part by the Leading R&D Support Project of MIC, Korea

application designers to better plan their application software.

The $\mu\text{C}/\text{OS}$ real-time kernel [6] suggested a deterministic scheduler using novel data structures. Its task scheduler's overhead, Δt , is constant irrespective of the number of tasks created in an application. Since $\mu\text{C}/\text{OS}$ was originally targeted for an 8-bit microcontroller, it can handle only up to 64 tasks each with a unique priority. However, considering the fact that applications become increasingly sophisticated and processing power increases, this restriction on the maximum number of tasks (*i.e.*, priorities) is becoming rapidly unacceptable in many applications. In earlier work [9], we proposed a deterministic scheduling algorithm which can be generalized such that it eliminates the restriction on the maximum number of task priorities imposed on $\mu\text{C}/\text{OS}$, without additional memory overhead. In this paper, we present the complete generalized algorithm to determine the highest priority in the ready list with 2^{2r} levels of priorities for an arbitrary integer number of r .

In the next section, we describe briefly the deterministic task scheduler of $\mu\text{C}/\text{OS}$. Section 3 presents our scheduling algorithm that eliminates the restriction on the maximum number of task priorities imposed on $\mu\text{C}/\text{OS}$. In this section, we present the complete generalized algorithm to determine the highest priority in the ready list with 2^{2r} levels of priorities. This paper concludes with Section 4.

2. The Deterministic Task Scheduler

This section presents how $\mu\text{C}/\text{OS}$ schedules the tasks in a deterministic manner. $\mu\text{C}/\text{OS}$ always executes the highest priority task ready to run. Each task is assigned a unique priority level between 0 and 63. The lowest task priority 63 is always assigned to $\mu\text{C}/\text{OS}$'s idle task when $\mu\text{C}/\text{OS}$ is initialized. Each task that is ready to run is placed in a ready list consisting of two variables, $OSRdyGrp$ and $OSRdyTbl[8]$. Task priorities are grouped (8 tasks per group) in $OSRdyGrp$. Each bit in $OSRdyGrp$ is used to indicate whenever any task in a group is ready to run. When a task is ready to run it also sets its corresponding bit in the ready table, $OSRdyTbl[8]$. To determine which priority (and thus which task) will run next, the scheduler determines the lowest priority number that has its bit set in $OSRdyTbl[8]$. When a blocked task unblocks, the task is placed in the ready list by the following code:

```
OSRdyGrp    /= OSMaPtbl[p >> 3];
OSRdyTbl[p >> 3] /= OSMaPtbl[p & 0x07];
```

where p is the task's priority. This means that the unblock operation can be done by a fixed number of machine instructions. Therefore, Δt_u is $O(1)$. As you can see in Fig. 1, the lower 3 bits of the task's priority are used to determine the bit position in $OSRdyTbl[8]$, while the next three most significant bits are used to determine the index into $OSRdyTbl[8]$. Note that $OSMaPtbl[8]$ (see Fig. 2) is a table used to equate an index from 0 to 7 to a bit mask. When a running task blocks, it is removed from the ready list by reversing the process. The following code is executed in this case:

```
if ((OSRdyTbl[p >> 3] &= ~OSMaPtbl[p & 0x07]) == 0)
    OSRdyGrp &= ~OSMaPtbl[p >> 3];
```

This code clears the ready bit of the task in $OSRdyTbl[8]$ and clears the bit in $OSRdyGrp$ only if all the tasks in a group are not ready to run, *i.e.*, all bits in $OSRdyTbl[p >> 3]$ are 0. Therefore, the blocking overhead Δt_b is also $O(1)$.

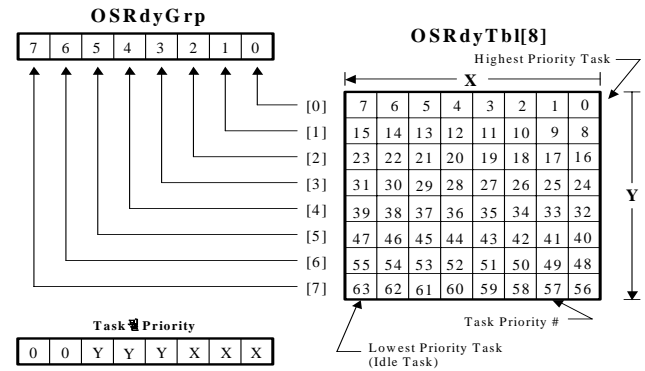


Figure 1. Ready list.

Another table lookup is performed, rather than scanning through the table starting with $OSRdyTbl[0]$ to find the highest priority task ready to run. $OSUnMapTbl[256]$ is a priority resolution table (see Fig. 2). Eight bits are used to represent when tasks are ready in a group. The least significant bit has the highest priority.

```
/* Mapping Table to Map Bit Position to Bit Mask */
UBYTE const OSMaPtbl[] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20,
0x40, 0x80};

/* Priority Resolution Table */
UBYTE const OSUnMapTbl[] = {
0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
```

```

4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};

```

Figure 2. Listing of map and unmap tables.

Using this byte to index the table returns the bit position of the highest priority bit set, a number between 0 and 7. Determining the priority of the highest priority task ready to run is accomplished with the following section of code:

```

y = OSUnMapTbl[OSRdyGrp];
x = OSUnMapTbl[OSRdyTbl[y]];
p = (y << 3) + x;

```

where p is the highest priority in the ready queue, and getting the pointer to the TCB of the corresponding task (*i.e.*, **highestP**) is trivial since each priority is mapped to a unique task. Therefore, we can select the highest-priority task ready to run by executing a fixed number of machine instructions, *i.e.*, Δt_s is $O(1)$. Since, $\Delta t_u = \Delta t_b = \Delta t_s = O(1)$, we can conclude that the task scheduling time of $\mu C/OS$ is constant irrespective of the number of tasks created in an application (*i.e.*, the scheduler is deterministic). For the detail of the source code of the task scheduler, refer to [6]. However, as stated in the above, the maximum number of tasks is restricted to 64. In the next section, we propose a generalized deterministic scheduling algorithm that eliminates the restriction on the maximum number of task priorities imposed on $\mu C/OS$.

3. Generalized Deterministic Scheduling Algorithm

In this section, we first describe how the scheduling method is extended to 256 levels of priorities (tasks) using the same data structure $OSUnMapTbl[256]$ in Fig. 2. In this case, task priorities are grouped (16 tasks per group) in $OSRdyGrp$ which is a 16 bit variable. Also the ready table should be $OSRdyTbl[16]$, each entry is 16-bit-wide. For a bit mask, we use $OSMapTbl[16]$ to equate an index from 0 to 15. Then, to unblock a task, the following code is used to place the task in the ready list:

```

OSRdyGrp   |= OSMapTbl[p >> 4];
OSRdyTbl[p >> 4] |= OSMapTbl[p & 0x0F];

```

where p is the task's priority. In the same manner, a task is removed from the ready list by reversing the process as follows:

```

if ((OSRdyTbl[p >> 4] &= ~OSMapTbl[p & 0x0F]) == 0)
    OSRdyGrp &= ~OSMapTbl[p >> 4];

```

Therefore, by the same reason described in the previous section, $\Delta t_u = \Delta t_b = O(1)$. Determining the highest priority in the ready list is accomplished with the following section of code:

```

if ((OSRdyGrp & 0x00FF) == 0)
    y = OSUnMapTbl[OSRdyGrp >> 8] + 8;
else y = OSUnMapTbl[OSRdyGrp & 0x00FF];

```

```

if ((OSRdyTbl[y] & 0xFF) == 0)
    x = OSUnMapTbl[OSRdyTbl[y] >> 8] + 8;
else x = OSUnMapTbl[OSRdyTbl[y] & 0x00FF];

```

```

p = (y << 4) + x;

```

Therefore, we can select the highest-priority task ready to run by executing a fixed number of machine instructions, *i.e.*, Δt_s is $O(1)$. Since, $\Delta t_u = \Delta t_b = \Delta t_s = O(1)$, it can be concluded that the task scheduling method of $\mu C/OS$ is extended to 256 levels of priorities without any sacrifice of determinism. In this way, we can extend the scheduling algorithm to any number of priorities levels by modifying the code for placing a task in the ready list and the code for selecting the highest priority in the ready list appropriately. For example, the following code is the complete generalized algorithm to determine the highest priority in the ready list with 2^{2r} levels of priorities. In the algorithm, $mask[i]$ means that the lower i bits are 1 and the other bits are 0.

```

int i, j, a, b, c;
i = 0, a = r - 1, b = OSRdyGrp;
loop1: if ((b & mask[2a]) == 0)
    i = i + 2a, b = b >> 2a;
    else b = b & mask[2a];
    a = a - 1;
    if (a > 0) goto loop1;
    y = i + OSUnMapTbl[b];
    j = 0, a = r - 1, c = OSRdyTbl[y];
loop2: if ((c & mask[2a]) == 0)
    j = j + 2a, c = c >> 2a;
    else c = c & mask[2a];
    a = a - 1;
    if (a > 0) goto loop2;
    x = j + OSUnMapTbl[c];
    p = (y << r) + x;

```

It is also shown that we can reduce the size of the resolution table, *OSUnMapTbl[]*, which is the most memory-consuming data structure. In that case, the selection overhead increases, but is still deterministic in time.

4. Conclusion

With continued miniaturization and increasing computing power, we see ever growing use of powerful microprocessors running sophisticated, intelligent control software in a vast number of devices including digital camcorders, cellular phones, and portable medical devices. These embedded systems require that kernel services should be deterministic by specifying how long each service call will take to execute. As applications become increasingly sophisticated, this requirement is becoming inevitable so as not to miss the deadline of each task comprising the application. In this paper, we suggested a generalized deterministic scheduling algorithm of which the scheduling overhead is constant. The proposed algorithm eliminates the restriction on the maximum number of tasks imposed on the existing ones without additional memory overhead.

References

- [1] K. G. Shin and P. Ramanathan, "Real-time computing: a new discipline of computer science and engineering," *Proc. of the IEEE*, vol. 82, no. 1, pp. 6-24, Jan. 1994.
- [2] K. M. Zuberi, P. Pillai, and K. G. Shin, "EMERALDS: A small-memory real-time microkernel," in *Proc. 17th ACM Symposium on Operating Systems Principles*, 1999.
- [3] L. M. Thompson, "Using pSOS+ for embedded real-time computing," in *COMPCON*, pp. 282-288, 1990.
- [4] D. Hildebrand, "An architectural overview of QNX," in *Proc. Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, Apr. 1992.
- [5] *VxWorks Programmer's Guide*, 5.1, Wind River Systems, 1993.
- [6] Jean J. Labrosse, *μC/OS: The Real-Time Kernel*, R&D Publications, Lawrence, 1993.
- [7] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating system support for real-time systems," *Proc. of IEEE*, vol. 82, no. 1, pp. 55-67, Jan. 1996.
- [8] C. M. Krishna and K. G. Shin, *Real-Time Systems*, McGraw-Hill, 1997.
- [9] S.-J. Oh, *et al.*, "Deterministic Task Scheduling for Embedded Real-Time Operating Systems," *IEICE Trans. Inf. & Syst.*, Vol. E87-D, No. 2, pp. 123-126, Feb. 2004.