An Improved Multi-Priority Preemptive Scheduler for Transputer-Based Real-Time Systems*

Tae-Young Choe†, Chan-Ik Park†, Chan Mo Park†, and Byung-Seop Kim‡

†Department of Computer Science and Engineering
POSTECH
San 31, HyoJa Dong
Pohang, KOREA 790-784

‡Team 5-3-4
Agency for Defense Development
P.O. Box 35-5, Yusung
Daejeon, KOREA 305-600

Abstract

Real-time applications require an efficient scheduler supporting multiple priority levels and fast preemption. In this paper, we propose a scheduler based on the hardware-supported scheduler of transputers. Though the hardware-supported scheduler of transputers is very efficient in terms of scheduling overhead, it should be extended to support multiple priority levels and fast preemption in order to be used in real-time applications. Many schedulers have been proposed. However, they have several drawbacks in terms of scheduling overhead, preemption latency, and portability. In reference [3], we have proposed a scheduler featuring low scheduling overhead and portability while suffering from a long preemption delay. In this paper, we propose an improved scheduler which greatly reduces preemption delay by using ISL (Interrupt Save Location) in transputers. Experimental results show that the improved scheduler overhead is about 13.54μsec and its preemption delay is well below 42μsec.

Key words : multi-priority scheduler, real-time system, transputer

1 Introduction

Supporting multi-priority scheduling is very critical in real-time systems which have to produce results subject to time limit. Since the hardware scheduler of transputers supports only 2-level priority, multi-priority scheduling must be supported by other means in order to use transputers in a real-time system. When we implement a multi-priority scheduler in transputer-based real-time systems, it should have low scheduling overhead as well as fast preemption time, and provide transparency to a programmer.

Many schedulers based on transputer hardware-supported scheduler (from now on, we call it hardware scheduler) have been proposed, but they have several drawbacks in terms of scheduling overhead, preemption latency, and portability. In [8], an efficient scheduler is proposed, but it is designed only for a computer-controlled system. Since the schedulers in [1, 10, 11] are written in OCCAM, they cannot manipulate the hardware process-queue directly, and therefore a new process is always attached to the end of the process queue regardless of its priority. This makes the preemption delay time longer depending on the number of processes in the queue. To reduce this delay time, descheduling codes have to be inserted in application programs. In [9], the overhead of scheduler and preemption delay time are greatly reduced by direct manipulation of the hardware process queue. However, it should be executed periodically even when all processes have equal priorities. Moreover, whenever the scheduler is called, it resets the hardware process queue, selects a high priority process, and inserts the process into the hardware process queue. In [3], we have proposed a general purpose multi-priority scheduler. It reduces the scheduler overhead by manipulating the hardware process queue directly, and invoking the scheduler call using the event manager when necessary. However, it has a signifi-

*This research has been supported in part by ADD, in part by KOSEF, and in part by MOST, 1995.

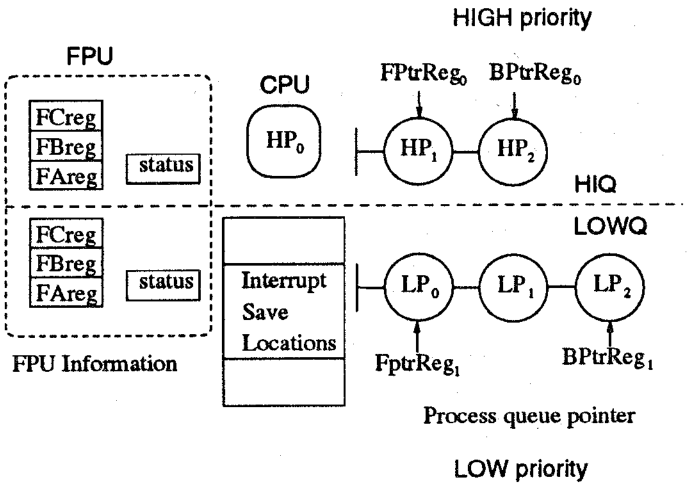


Figure 1. Information to be saved at preemption: when a HIGH priority process HP_0 preempts a currently running LOW priority process LP_3, the context information of LP_3 is saved in the Interrupt Save Location and FPU information is saved in FPU_{LOW} if any.

cant drawback that preemption delay time can exceed 2048μsec since it is based on the round robin scheme of the transputer hardware scheduler. In [2] and [8], a preemption mechanism which can significantly reduce the maximum preemption delay is presented, independently.

In this paper, we improve the scheduler in [3] by adapting the preemption technique presented in [2] and [8]. The preemption technique is based on the information that has to be saved when a process is preempted and restored later when the process is resumed. The information includes work space pointer, instruction pointer, and general/status registers which will be automatically saved in a special memory location called Interrupt Save Location (ISL). In addition, FPU information such as floating point registers and status register also has to be saved.

2 The structure of transputer and scheduler

2.1 Transputer hardware scheduler

The scheduling mechanism of the transputer hardware scheduler is shown in Figure 1. The hardware scheduler manages two hardware process queues denoted as HIQ and LOWQ, i.e., supports 2-level hardware priority. In T800 transputers, each hardware process queue is handled differently by the scheduler: the HIQ is scheduled nonpreemptively while the LOWQ is scheduled preemptively, i.e., a HIGH priority process occupies CPU until it voluntarily releases CPU while a LOW priority process occupies CPU by the round-robin policy with 2048μsec timeslice. Note that each hardware queue has its own head and tail pointer: $FPtrReg_0$ and $BPtrReg_0$ for HIQ, and $FPtrReg_1$ and $BPtrReg_1$ for LOWQ. FPU information is also separately managed for each hardware queue. FPU_{HIGH} and FPU_{LOW} are used for clear differentiation as shown in Figure 1.

Assume that a LOW priority process LP_3 is currently running. When a new HIGH priority process HP_0 is ready to run, it will preempt LP_3 and get the control of CPU as shown in Figure 1. At this time, the context of LP_3 is automatically saved in a special memory area called Interrupt Save Location (ISL). Process context information includes work space pointer (Wptr), instruction pointer (Iptr), and general/status registers. FPU information of LP_3 (if any) such as floating point registers and status register is also saved in FPU_{LOW}. In addition to process context information, FPU information of LP_3 (if any) is essential for resuming LP_3 later. So, we have to save FPU information as well as context information somewhere else for resuming the preempted process LP_3 correctly later when needed. Remember that FPU information is separately managed for each hardware process queue: one for HIQ and the other for LOWQ. However, HP_0 cannot directly access the FPU information of LP_3 since processes of the same hardware priority level are allowed to access the corresponding FPU information in transputers, i.e., the processes in LOWQ can access FPU_{LOW} whereas the processes in HIQ can access FPU_{HIGH}. In the coming section, we will describe how to save FPU_{LOW} by our software scheduler of HIGH priority.

2.2 Overview of the proposed software scheduler

Our scheduler consists of two modules : one is event manager which handles blocked processes and the other is multi-priority scheduler. Event manager handles interprocess communication and process management which are currently handled by language runtime library, and then informs an event occurrance of the multi-priority scheduler if the event handling requires process scheduling function. In transputer programming environments like INMOS ANSI C, language runtime library acts as kernel which handles interprocess communication, process management, and memory management for applications. Process scheduling is

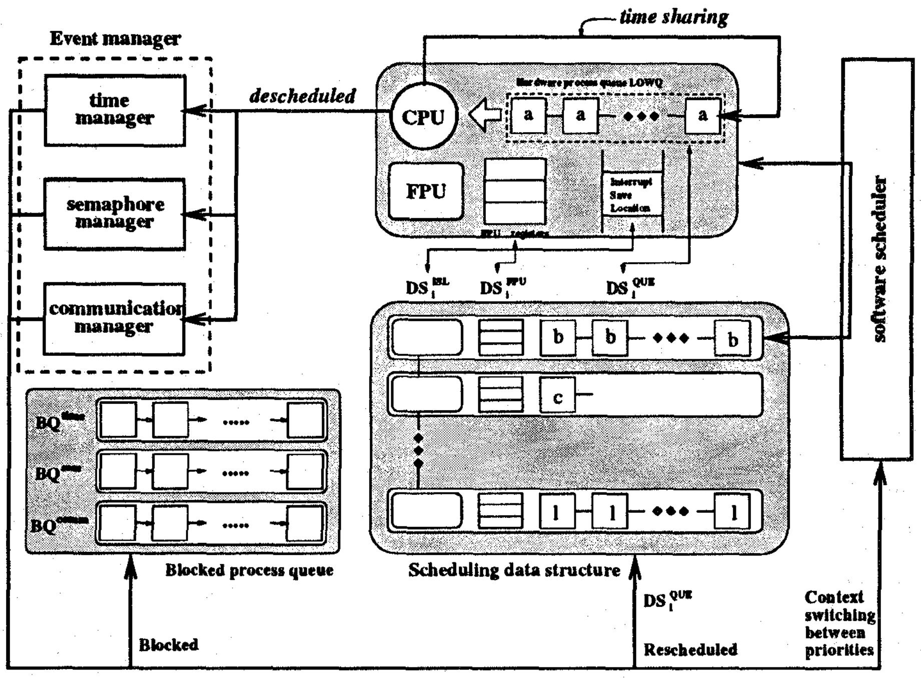


Figure 2. Overview of the scheduler operation assuming n priority levels: The scheduler maintains one data structure for each priority level which stores interrupted process context information, FPU information, and software process queue.

usually handled by the hardware scheduler. So, when a new process is created in INMOS ANSI C, a programmer has to choose one of two hardware priority levels for the process. Although our scheduler is proposed as a process scheduler, it provides all the functions of language runtime library to applications, keeping application interfaces unchanged. In other words, our software scheduler acts as kernel supporting multiple priority levels which can substitute language runtime library. Our software scheduler consisting of the multi-priority scheduler and the event manager is assigned to have HIGH priority and all other user processes are assigned to have LOW priority.

Figure 2 shows an overview of the operation of our software scheduler. Note that the scheduler maintains one scheduling data structure for each priority level as well as three kinds of blocked process queues used by event manager. The scheduling data structure consists of three kind of areas: one to store ISL information, one to store FPU information, and software process queue of the corresponding priority. Given a priority level as i, we denote its corresponding data structure consisting of three kinds of areas as DS_i^{ISL}, DS_i^{FPU}, and DS_i^{QUE}, respectively. Additional links are used to construct a global list connecting n software process queues for easy management. Blocked process queues are used by event manager to store process information for resuming later. According to blocking reasons, there are three kinds of queues: wait-time, semaphore, and communication. We denote these blocked process queues as BQ^{time}, BQ^{sem}, and BQ^{comm}, respectively.

Event manager manages three blocked process queues, BQ^{time}, BQ^{sem}, and BQ^{comm}, storing all processes to be blocked due to wait-time, semaphore, or communication operation. When a process is to be blocked due to some reason, event manager inserts the process into a corresponding blocked process queue. And, when a process is to be ready-to-run from blocked with the priority level of i, event manager removes the process from a blocked process queue and then inserts it into DS_i^{QUE}. Note that all ready-to-run processes are stored in the scheduling data structure and the multi-priority scheduler keeps processes of the highest priority level running in LOWQ.

3 Scheduling Algorithm

Figure 3 shows an overview of our scheduling algorithm. The scheduling algorithm checks the event type and operates for each type of event. The scheduler handles three cases: a process is to be ready-to-run, blocked, or terminated. In the case that a process is to

```
1. if a process is to be blocked
    a. if blocking reason == wait-time
        (1) insert the process into wait-time blocked
            process queue.
    b. if blocking reason == semaphore
        (1) insert the process into semaphore
            blocked process queue.
    c. if blocking reason == communication
        (1) insert the process into communication
            blocked process queue.
    d. if the number of processes in LOWQ is zero
        (1) inserts all processes of the software pro-
            cess queue of the highest priority level
            into LOWQ
        (2) if there is a preempted process for the
            same priority level
              • restore FPU registers and status
              • restore Interrupt Save Location
2. if a process is to be ready-to-run from blocked or
   a new process is to be created
    a. remove the process from the correspond-
       ing blocked process queue if the process is
       blocked.
       /* the priority of the process = Pnew */
       /* the priority of currently running process
       = Pcur */
    b. if Pnew < Pcur
        (1) insert the process into the software pro-
            cess queue of Pnew.
    c. else if Pnew = Pcur
        (1) attach the process to the end of LOWQ
    d. else if Pnew > Pcur
        (1) insert all processes in LOWQ into the
            software process queue of Pcur and clear
            LOWQ.
        (2) save the content of Interrupt Save Loca-
            tion to the corresponding save area for
            interrupt process context information.
        (3) clear Interrupt Save Location memory
            area.
        (4) save FPU information such as FPU reg-
            isters and status.
        (5) insert the process to the end of LOWQ.
3. if a process is to be terminated
    a. do the same execution steps in 1-d.
```

Figure 3. The scheduling algorithm of our scheduler

be ready-to-run, we have to compare the priority of the newly ready-to-run process with that of the currently running process. And then, scheduling policy may differ according to comparison results. In the case that a process is to be blocked, our scheduler inserts the process into the corresponding blocked process queue. When a process is to be terminated, we have to check if there are any runnable processes of the current priority level in LOWQ. If there are any, then our scheduler need not do anything. But if there are no runnable processes, our scheduler has to set up all necessary information to make the processes of the next highest priority level running.

In [3], preemption is implemented approximately by using two dummy processes and a round-robin time interval of LOWQ. For example, though a new process has higher priority than currently running process, it must wait for its time quantum in order to preempt currently running process. By doing this, we can reduce the amount of context information to be saved for preemption. However, this method inevitably increases the maximum preemption delay. If `2d-blockmove` instruction is executed, the maximum preemption delay may be much longer. To fit this scheduler to real-time systems, we have to reduce the preemption delay of the scheduler. Now we will explain more in detail about how to reduce maximum preemption delay while keeping scheduling overhead as low as in [3].

Here is how to implement preemption mechanism. Let us assume that a new process P_{new} with its priority level i is to be ready-to-run and a currently running process P_{cur} has the priority j. A condition which preemption should occur is that $i > j$. This condition is notified to event manager in a form of event, making our scheduler ready-to-run in HIQ. Then, our scheduler with HIGH priority will get control of CPU, preempting the currently running process P_{cur} of LOW priority since all user processes have LOW priority even though their software priority levels are defined to have more than two by software. However, before our scheduler is running, i.e., preempting P_{cur}, the context information of P_{cur} is automatically saved in a special memory location called ISL (Interrupt Save Location). Now our scheduler saves necessary information into DS_j^{ISL}, DS_j^{FPU}, and DS_j^{QUE}. First, our scheduler saves information such as Wptr (workspace pointer) on all ready-to-run processes residing at LOWQ into DS_j^{QUE} and clears LOWQ. Second, it saves current ISL information into DS_j^{ISL} and clears all information in ISL as if there is no interrupted process. Third, it saves any FPU information related to the currently interrupted P_{cur} into DS_j^{FPU}. However, we have some difficulty to do the third saving operation since a process of HIGH priority cannot access FPU information related to a process of LOW priority, i.e., the scheduler having HIGH priority cannot access FPU information of P_{cur} having LOW priority. So, in order to overcome this difficulty, we use two special processes having LOW priority executing some function of our scheduler accessing FPU information : *save_fpr* and *restore_fpr*. *Save_fpr* saves FPU information in DS_j^{FPU}, and *restore_fpr* restores DS_j^{FPU} to floating point registers including status register. Since two special processes have LOW priority, they can access FPU information of interrupted process of LOW priority no matter how high the software priority level of the process is. After saving operations are completed, it inserts the new process P_{new} into LOWQ. Then the new process starts execution as if it preempts lower priority processes.

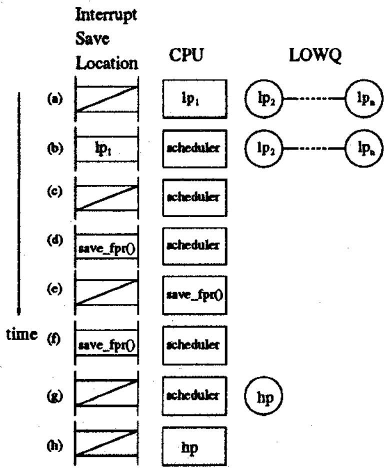


Figure 4. Illustrative operations of preemption which move low priority processes to the corresponding process queue in the data structure

Figure 4 shows an illustrative example how a new process preempts currently running process if it has higher priority.

If there remain no ready-to-run or running processes for a given software priority level i, then the scheduler searches for a priority level lower than i at which there exist some ready-to-run or interrupted processes. Assume the priority level found by this operation is j. The scheduler copies DS_j^{QUE} and DS_j^{ISL} into LOWQ and ISL, respectively. Note that copying DS_j^{FPU} into FPU will be done with the help of the special process called *restore_fpr* in the same way as *save_fpr* for saving FPU information.

Note that our scheduler has little scheduling overhead when all user (application) processes are defined to have equal priority. We usually come up with this situation in general computing. Our scheduler can substitute any language run-time library without any problems due to this effect.

4 Experimental Results

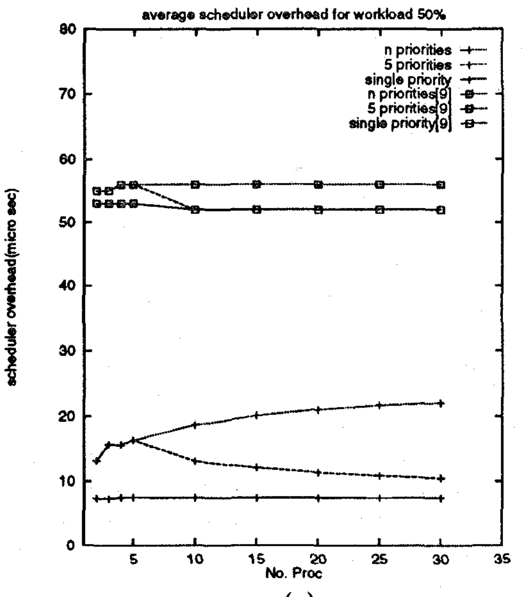
The scheduler has been implemented in INMOS ANSI C [6, 5] and transputer assembly language [4] on T805 25MHz. Experimental results were obtained by executing the scheduler 10 times and averaging the results. Though running time is small, deviation is below 1μsec. For comparison, we also implemented Shea et al.'s scheduler [9] which is regarded as general and efficient one. Application processes are assumed to be periodic and priorities are assigned according to the rate monotonic algorithm [7]. Computation loads are 50% and 90%, which correspond to the percentage of computation time required for a fixed period. Three cases of priority levels are considered: single priority level, 5 priority levels, and the case each process has its own priority which we call 'n-priority levels'.

	CPU load			
	50%		90%	
	min	max	min	max
5-priority	29.5	33.24	33.76	41.57
n-prioirty	29.5	30.31	31.3	34.75

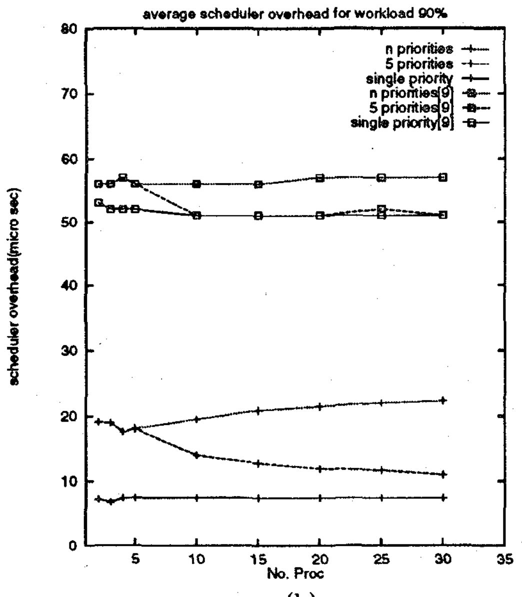
Table 1. The scheduler preemption delay (μsec)

Table 1 shows the preemption delay for each experiment. The preemption delay is zero in 1-priority level since preemption does not occur. Experimental results show that the maximum preemption delay is below 42μsec. In general, preemption delay is larger in 5-priority than in n-priority, since only one process is preempted in n-priority while more than one process is preempted in 5-priority.

Figure 5 shows the average scheduler overhead. When the number of application processes is less than five, we cannot differentiate 5-priority level from n-priority level. As we can imagine, two extreme results come from single priority and n-priority levels, and 5-priority level in between. When the number of application processes is large, 5-priority level behaves like single priority level since there exist many processes in one priority level. So, as the number of application processes increases, we can see from Figure 6 that the



(a)



(b)

Figure 5. The scheduler overhead vs the number of processes (a) in case of 50% load (b) in case of 90% load

scheduling overhead of 5-priority level approaches to that of single-priority level.

5 Conclusion

We have improved the scheduler of [3] in terms of maximum preemption delay by manipulating directly Interrupt Save Location of transputer memory. This technique is very complicated and difficult since very few information are available. We could see from experimental results that the maximum preemption delay is greatly reduced from 2048μsec to 42μsec while keeping the scheduling overhead as low as about 13.54μsec. Therefore, we think this scheduler is suitable for transputer-based embedded real-time applications. The scheduler does not incur any scheduling overhead if all processes are of the same priority. With the help of this feature, our scheduler is easily integrated and used in any transputer applications. We are currently trying to reduce maximum preemption delay further. For example, this delay can be reduced if scheduler save process queue pointer instead of process queue.

References

[1] O. Caprani, J. Kristensen, C. Mork, and H. Pedersen. Implementation of real-time scheduling algorithms in a transputer environment. In *Proceedings of the 13th Occam User Group Technical Meeting*, pages 186–197, September 1990.

[2] M. Cheung, K. Shea, and F. C. Lau. A technique for process preemption in the transputer. *Microprocessors and Microsystems*, 19(1), February 1995.

[3] T.-Y. Choe, C.-I. Park, C. M. Park, and B.-S. Kim. A multi-priority scheduler for transputer-based real-time systems. In *Proceedings of World Transputer Congress'95, Harrogate, UK*, September 1990.

[4] INMOS. *Transputer Instruction Set*, 1988.

[5] INMOS. *ANSI C toolset reference manual*, 1990.

[6] INMOS. *ANSI C toolset user manual*, 1990.

[7] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1(1), January 1973.

[8] E. Ploeg, J. Sunter, A. Bakkers, and H. Roebbers. Dedicated multi-priority scheduling. In *Proceedings of the 17th World OCCAM and Transputer User Group Technical Meeting, Bristol, U.K.*, April 1994.

[9] K. Shea, M. Cheung, and F. Lau. *An Efficient Multi-Priority Scheduler for the Transputer*, pages 139–153. IOS Press, 1992.

[10] J. Sunter, K. Wijbrans, and A. Bakkers. Cooperative priority scheduling in occam. In *Proceedings of the 13th Occam User Group Technical Meeting*, September 1990.

[11] P. Welch. Multi-priority scheduling for transputer-based real-time control. In *Proceedings of the 13th Occam User Group Technical Meeting*, September 1990.