

Resource Reservation for Adaptive QOS Mapping in Real-Time Mach

Tatsuo Nakajima

Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Ishikawa, 923-12, JAPAN

Abstract. In this paper, we describe an adaptive QOS mapping scheme where the QOS parameters of applications are mapped into resource requirements dynamically, and the resources for the applications are reserved based on the measured resource utilization. Real-Time Mach provides resource reservation facilities that are suitable for supporting the adaptive QOS mapping scheme, and we show the effectiveness of the resource reservation facilities when the adaptive QOS mapping scheme is adopted.

1 Introduction

Resource reservation such as processor reservation and memory reservation is a promising approach for supporting timing critical applications such as continuous media applications[BKTZ94, MST94, Nak96]. However, it is difficult to predict how much resource is consumed for respective applications before executing the applications. In real-time communities, the schedulability analysis can be used to map QOS values of continuous media applications to their resource utilization. The approach is realistic in embedded systems, but it is not suitable when various types of computers may execute the applications since we cannot predict the characteristics of the computers when the applications are designed and implemented.

In the other approach, continuous media applications independently adjust their qualities of media according to the capacity of allocated resources for respective applications[CT94]. The approach does not require resource reservation mechanisms, then the approach is suitable for executing continuous media applications on traditional operating systems that support the best effort resource management. However, the approach does not guarantee the QOS requirements of the applications.

Some researchers supporting the first approach claim that the second approach does not guarantee the QOS values of applications, and the approach is not suitable for future advanced continuous media applications. On the other hand, some researchers supporting the second approach claim that the assumption required by the first approach is too strong for executing continuous media applications in traditional environments. However, we believe that the two approaches should not refuse each other's, rather, the two approaches should compensate each other's.

In this paper, we describe an adaptive QOS mapping scheme where the QOS parameters of applications are mapped into resource requirements dynamically, and the resources necessary for the applications are reserved based on the measured resource utilization. The approach combines the merits of the both approaches described in the previous paragraph. Real-Time Mach provides resource reservation facilities that are suitable for supporting the adaptive QOS mapping scheme. The first resource reservation mechanism is the processor reservation mechanism that reserves CPU resources for respective applications. The second resource reservation mechanism is the memory reservation that reserves free

pages for respective applications. We describe these resource reservation mechanisms, and show the effectiveness of the mechanisms.

2 Adaptive QOS Mapping

2.1 Overview of System Components

Figure 1 shows three components that are required for realizing our QOS mapping scheme. A processor reservation system and a memory reservation system reside in a kernel. They monitor the CPU and memory capacities for respective applications. The second component is the QOS server which reserves CPU and memory resources to respective applications. The third component is the adaptive QOS mapping system which is linked by respective applications, and adjusts each application's high level QOS value according to reserved resource capacity by the QOS server.

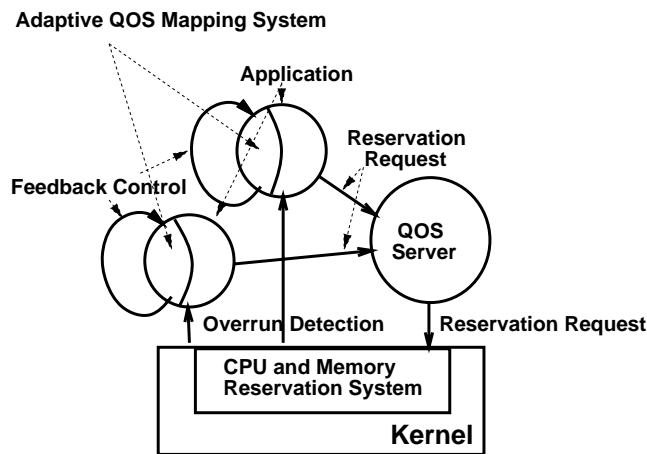


Fig. 1. Components for Our QOS Mapping Scheme

2.2 Adaptive QOS Mapping System

The adaptive QOS mapping system resides in each continuous media application in our system, and adjusts the quality of media according to reserved resource capacity by the QOS server. In the section, we describe how the adaptive QOS mapping system adjusts the quality of media using a feedback control scheme.

In our scheme, the adaptive QOS mapping system makes resource utilization higher for consuming that an application can utilize reserved resource capacity as much as possible if the controlling application does not causes an overrun of resources. The adaptive QOS mapping system scales the quality of media up for increasing and decreasing the resource utilization of an application if the amount of actual resource utilization is less than the amount of reserved resource capacity within a constant time. The condition can be detected by checking that there is no overrun notification within the constant time.

Also, the application scales the quality of media down if it receives a number of successive overrun notifications from a kernel. As described in the above

paragraph, our scheme uses a feedback control for controlling the reserved resource capacity of applications. When using a feedback control like our scheme, the important issue is how to make the behavior of an application stable. In our approach, the quality of media is scaled down rapidly, but the quality of media is scaled up very slowly for making a system stable.

3 Processor Reservation

3.1 Processor Reservation System

A processor reservation system is originally developed in CMU[MST94] for protecting processor resource reserved for an application from other applications on Real-Time Mach[NTAT93].

The central abstraction of the system is "reserve" which manages an amount of processor resource reserved for each application. A thread which needs to reserve processor resource passes two parameters to kernel when a new reserve is created. The first parameter specifies a reserve period, and the second parameter specifies computation time, which means the worst case execution time in a period. The CMU processor reservation system schedules threads under the rate monotonic scheduling. Thus, a thread which has a shorter period is scheduled ahead of others.

The processor reservation system allocates CPU capacities to the threads reserving processor resource. Each thread is scheduled using the reserve period as a scheduling priority if the entire capacity is not consumed. The thread consumes the entire capacity when the execution time of the thread exceeds over the specified computation time in each reserve period. In this case, the thread is executed at a background priority.

Our processor reservation system improves the original CMU processor reservation system in the following ways.

- *The admission control policy is too pessimistic for continuous media applications.*

In the CMU reservation system, processor resource is reserved based on the worst case CPU usage. However, when a dynamic QOS control scheme is used, a system allows us to do an admission test using their average CPU usage, since system can decrease CPU usages of less important applications when it is overloaded. This means that admission control policies should be changed according to the characteristics of applications.

- *An application can specify a rate for reserved processor capacity in a reserve attribute.*

In the CMU reserve system, the attribute for a reserve can contains the period and the computation time. However, the computation time should be recalculated in an application when the period of a reserver is changed. The rate of processor usage is more suitable for the adaptive QOS mapping scheme.

- *A processor reservation system should be used under any scheduling policies.*

The CMU processor reservation system assumes to use the rate-monotonic scheduling algorithm. However, it may be suitable for using other scheduling policies such as the early deadline first and simpler fixed priority scheduling algorithms for continuous media applications[TN95]. Processor reservation systems should be used under any scheduling policies.

- *There is no mechanism notifying to an application when an application overruns the time allowed it.*

If an application uses much processor resource than it reserves, we say that the application causes an *overrun*. In order to make the behavior of an

application stable, the application must change its execution period when an overrun occurs. However, an application can indirectly know that an overrun occurs by detecting missed deadlines, which may make a system unstable for a long time.

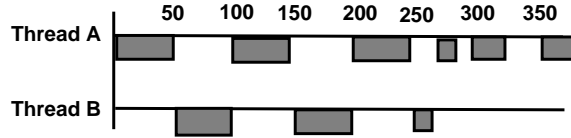


Fig. 2. Execution of Thread under Processor Reservation System

Figure 2 shows how threads are executed under our processor reservation system. Thread A is a periodic thread whose period is 100 ms, and the thread reserves 50% of the total CPU capacity. Thus, the thread can execute 50 ms at the higher priority level within every period. However, when the utilization of the thread exceeds 50%, the priority of the thread is degraded. For example, in Figure 2, thread A causes an overrun at 250. Therefore, the priority of the thread is degraded, so thread B can preempt thread A since the priority of thread B is higher than the priority of thread A while thread A causes an overrun. After thread B is terminated, the remaining computation of thread A is resumed.

When thread A changes its period to 50 ms, the thread can execute 25 ms within every period at a higher priority. In our reservation system, the time is recalculated automatically from the rate parameter in the reserve attribute. However, in the CMU reservation system, the computation time should be recalculated by an application, and it should call a kernel with the recalculated parameter.

3.2 Interface

The main functions of our kernel support are to deliver a notification message when an application causes an overrun, and to control the priority of an application reserving processor resource. Table 1 shows the interface of the kernel support.

The following steps show the behavior of our kernel support when processor resource is reserved.

1. The QOS server sets a priority, a port used to deliver overrun notifications, a reserve period, and the rate of CPU usage within a period using `reserve.attribute_init()` when a new request is received from an application. Then, `reserve.create()` including above parameters is called for creating a new reserve.
2. When processor resource reserved for a thread is not consumed, the thread is scheduled using a priority specified in the reserve of the thread. The thread overruns, and the priority of the thread specified at the creation time is used as its scheduling priority if the reserved processor resource is entirely consumed for a reserve period. Also, an overrun notification message is delivered to the thread. Also, a thread waiting the message using `reserve.overrun_notify_wait()` is resumed and recovers its overrun.

3. The thread receiving overrun notification messages decreases its execution period until the overruns are disappeared. Also, the execution period is increased when no overrun is detected within a constant time.

reserve_create (pset, &reserve, reserve_attr)
<i>Creates a new reserve with a CPU reservation attribute.</i>
reserve_request (thread, reserve)
<i>Sets a reserve attribute to a reserve.</i>
reserve_attribute_init (priority, period_secs, period_nsecs, rate, overrun_port, overrun_action, &reserve_attr)
<i>Sets an attribute of CPU reservation.</i>
thread_get_reserve (thread, &reserve)
<i>Gets a current reserve of the specified thread.</i>
thread_set_reserve (thread, reserve)
<i>Set a reserve to the specified thread.</i>
reserve_get_attribute (reserve, flavor, &new_attr, &new_attr_count)
<i>Gets a CPU reservation attribute.</i>
reserve_set_attribute (reserve, flavor, new_attr, new_attr_count)
<i>Changes a CPU reservation attribute.</i>
reserve_overrun_notify_wait (overrun_port, &thread)
<i>Waits for notification of an overrun.</i>

Table 1. Kernel Interface

A reserved period is synchronized with the period of an periodic thread. Thus, when the period of the thread is changed, its reserved period is also automatically changed. The scheme does not require for a user to change the attribute of its reserve whenever the period of a thread is changed.

Under our reservation system, the QOS server controls an amount of reserved processor resource of each application according to its importance and system load. Continuous media applications adapt their execution periods using overrun notification messages using feedback schemes as shown in above step 3. The approach enables the processor reservation system not to manage the mapping between an application's QOS value and its CPU usage directly, and makes the system much simpler.

3.3 Evaluation

The benchmark that the adaptive QOS mapping scheme can adjust the periods of movie players when the reserved CPU capacity of the players is renegotiated by the QOS server. In the policy of the QOS server, the reserved CPU capacity of the least important player is decreased first if total CPU utilization exceeds a threshold by adding a new movie player.

In the test, three movie players are executed. The maximum CPU capacity of movie player 1 is 20%, the minimum CPU capacity is 10%, and its importance is 16. Also, the player is started at 0 sec. Movie player 2 is started at 50 sec, and the importance is 14. The maximum and the minimum CPU capacity is 20% and 10%. Also, movie player 3 is started at 100 sec, and its importance is 12 which is the highest importance. Also, the maximum and the minimum CPU capacity of player 3 is 30% and 15%. In the test, the QOS server allocates 60% CPU capacity to continuous media applications. Thus, all three players are executed, the CPU resources should be negotiated.

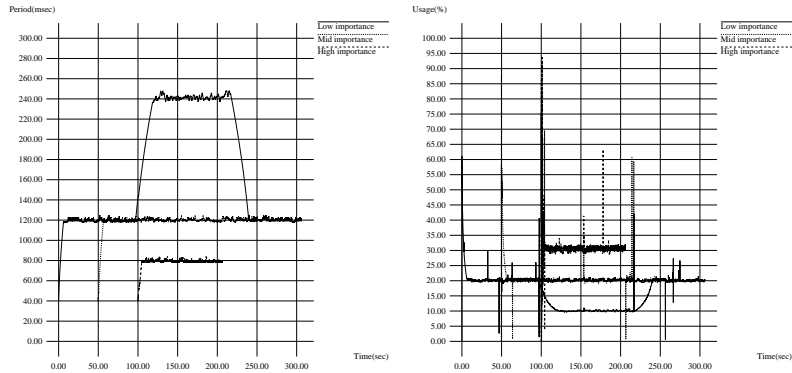


Fig. 3. Evaluation of CPU Reservation

Figure 3 shows results of the benchmark. The period of movie player 1 is decreased when movie player 3 is executed.

The policies for dynamic QOS control schemes should be chosen according to the characteristics of applications. Our system easily changes the QOS server since the server is implemented as a user-level server.

4 Memory Reservation

4.1 Memory Reservation System

The memory reservation system enables continuous media applications to reserve the number of wired pages in physical memory. There are two functionalities in the memory reservation system. The first functionality reserves the necessary number of wired pages by applications. When pages are reserved for an applications, a kernel fetches the specified number of pages from a free page list, and inserted in the reserved page pool for the application. If enough pages are not found in the free list, physical pages allocated for non timing-critical applications are reclaimed, and inserted in the reserved page pool of the continuous media application. If the pages are dirty, the contents of pages may be written back in secondary storages before inserting them in the reserved page pool.

The next functionality provided by the memory reservation system is a notification mechanism. If the number of wired pages in physical memory exceeds the number of reserved pages, a notification message is delivered to an application. When the application receives the notification message, it decreases the number of wired pages or increases the number of reserved pages. In this case, the pages may be allocated for more important continuous media applications.

The advantage of our memory reservation system is that applications can wire pages in physical memory in a secure way since the applications are not allowed to wire more pages than it reserves. When the application calls a memory reservation request, the request may be rejected if the total number of reserved pages exceeds a threshold. In this case, the application must issue the reservation request later after other applications release their reserved pages.

The virtual address space of a continuous media application contains several segments: code, static data, dynamic data, and stack segments. It is not easy to decrease the number of wired pages of these segments since it is difficult to predict which pages will be accessed in future for processing media data. However,

dynamic data segments may be decreased by reducing the quality of media by using media scaling techniques and dynamic QOS control schemes[NT94, Nak96], and the size of a buffer in a dynamic data segment can be decreased. For example, decreasing the rate of a media stream or the size of respective video frames reduces the total amount of data that must be stored in the buffer in every period.

4.2 Incremental Memory Wiring

Our memory reservation system allows applications not to specify which pages should be wired explicitly in physical memory. The requirement is achieved by wiring pages when a page fault occurs. This means that a page is not swapped out to secondary storages once the page is fetched from a file in which a code and a data segments are contained. Our memory reservation system requires to specify whether threads are used for processing media data or not. The threads processing media data are called *real-time threads*, and the remaining threads are called *non real-time threads*. The separation makes the number of wired pages small, and allows programmers not to specify pages that should be wired in physical memory since only pages that are actually touched by real-time threads are wired.

Our memory reservation system makes all pages of an application in physical memory invalid before starting incremental memory wiring, since respective pages touched by real-time threads should cause page faults for wiring pages in physical memory. On the other hand, a page fault caused by non real-time threads is processed in a traditional way. The pages may be reclaimed in order to allocate them for other applications. However, a page is allocated from a reserved page pool for a continuous media application, and the page is wired in physical memory when a page fault is caused by a real-time thread.

The strategy ensures that less pages are wired in physical memory than the traditional memory wiring primitives wire since real-time threads do not require to wire the entire memory segments of applications.

4.3 Interface

The following three primitives are added in Real-Time Mach for supporting our memory reservation system.

```
ret = vm_reserve(priv_port, task, reserve_size, notify_port, exceed_policy)
ret = vm_thread_wire_policy(thread, policy)
ret = memory_exceed_wait(notify_port)
```

Vm_reserve() is used to reserve pages for a task specified in arguments. The first argument *priv_port* allows the primitives to be used by the QOS server which is one of privileged applications. *Reserve_size* specifies the number of pages reserving for an applications. *Notify_port* is a port for receiving messages to notify when a real-time thread causes a page fault, but there is no reserved pages for the application. *Exceed_policy* specifies policies when a page is wired when the number of wired pages exceeds the number of reserved pages. Currently, two policies are supported. The first policy is *WIRE_WHEN_EXCEED*. The policy continues to wire pages in physical memory even when the number of wired pages is more than the number of reserved pages. The second policy is *DONT_WIRE_WHEN_EXCEED*. In this case, a page is not wired in physical memory under the policy, when the number of wired page exceeds the number of reserved pages.

The second primitive is *vm_thread_wire_policy()*. The primitive specifies a memory wiring policy as an argument. Currently, there are three policies: *WIRE_POLICY_NONE*, *WIRE_POLICY_ALL*, *WIRE_POLICY_SCHED*.

WIRE_POLICY_NONE makes all threads of a continuous media application non real-time threads, and *WIRE_POLICY_ALL* makes all threads of a continuous media application real-time threads. *WIRE_POLICY_SCHED* determines whether threads are real-time threads or non real-time threads according to the current scheduling policy. For example, a real-time thread does not wire touched pages in physical memory under the round-robin policy, and all periodic threads wires touched pages under the rate monotonic scheduling policy. Also, threads with real-time priorities are real-time threads, and other threads are non real-time threads under the fixed priority + timeshare scheduling [TN96]. Since the kernel changes a memory wiring strategy according to the scheduling policies, the same program can be used under different scheduling policies without modifying programs.

The third primitive is *memory_exceed_wait()*. The primitive waits for a notification message to a port specified as an argument. The primitive is used to wait for a notification message from the kernel when the number of wired pages exceeds the number of reserved pages.

4.4 Evaluation

In the evaluation, we uses QtPlay that is a simple video player running on Real-Time Mach. QtPlay retrieves video frames and audio samples from a QuickTime file stored in Unix file system using CRAS [TN96] which is a continuous media storage server on Real-Time Mach. Threads in QtPlay are classified into three categories. A thread belonging in the first category executes an initialization code and enters in an event loop for processing input events such as mouse and keyboard input events from X server.

Our memory reservation system wires only pages accessed by threads processing media data in physical memory. On the other hand, in traditional approaches, all pages in code, data, stack segments are wired for avoiding page faults. In this section, we show how our approach is effective and reduces the number of wired pages.

Table 2 shows the sizes of the respective memory segments and the entire address space of QtPlay and X server. As shown in the table, the size of QtPlay is very big since QtPlay links the Motif toolkit library that contains a very large code and data segment. However, most of these segments are not touched by continuous media applications.

	Code	Static Data	Dynamic Data	Address Space
QtPlay	1004KB	151KB	592KB	6.38MB
X Server	780KB	52KB	82KB	8.29MB

Table 2. Size of Memory for Code, Data Segments and Entire Address Space

Table 3 shows the number of actual wired pages and resident pages in physical memory under five wiring strategies. In the first strategy, no page is wired in physical memory. In the second strategy, only code segments are wired, and code and static data segments are wired in the third strategy. In the fourth strategy, all memory segments are wired. Lastly, the fifth strategy is our approach, and

pages are wired incrementally. In the evaluation, we run only QtPlay during the evaluation.

<i>Wiring Strategy(QtPlay)</i>	Resident	Wired
No Wiring	1.93MB	0.00MB
Code	2.25MB	1.02MB
Code + Static Data	4.44MB	3.85MB
Code + Static Data + Dynamic Data	4.88MB	4.88MB
Our Approach	1.94MB	0.71MB
<i>Wiring Strategy(X Server)</i>	Resident	Wired
No Wiring	2.04MB	0MB
Code	2.73MB	0.91MB
Code + Static Data	7.79MB	7.78MB
Code + Static Data + Dynamic Data	8.11MB	8.09MB
Our Approach	2.51MB	0.07MB

Table 3. Wiring Pages under Different Wiring Strategies

As shown in the table, our approach can reduce the number of wired pages dramatically. Also, the result shows that traditional approaches increase the number of resident pages since the approaches wire pages that are not necessary.

5 Discussion

In this paper, we describe the adaptive QOS mapping scheme and resource reservation mechanisms supporting the scheme. This section describes four experiences with the current design and implementation of our approach.

The first experience is about scheduling policies for real-time threads. In the current implementation, Real-Time Mach supports several real-time scheduling policies such as the rate monotonic scheduling and the fixed priority scheduling. One of the serious problems of the real-time scheduling is that the policies may delay the response time of the user interface. Some researchers developed proportional fair scheduling policies for continuous media applications[?, NL97], but it is difficult to control overload conditions under the scheduling policies since the overloaded capacity is also distributed fairly, and all applications may violate their timing constraints.

Our second experience is about the memory reservation system. In our system, a page is wired incrementally when the page is touched at the first time. However, an application may know how to use a memory region allocated by the application. Also, it is hard to use our scheme for a large memory region that is accessed sequentially. In the former case, the memory region should be wired eagerly and manually. In the latter case, the wired pages should be unwired after the pages are accessed. This experience shows that only one scheme does not support all requirements of various continuous media applications. In the next version, we make it possible to specify the memory management policies for respective memory regions.

In the third experience, we focus the adaptability mechanism used in our scheme. Our scheme uses a feedback mechanism for adjusting QOS values. When the utilization of a resource usage exceeds a specified value, an application decreases the utilization by degrading some QOS values. A serious problem of the approach is how to control a feedback loop in a stable fashion. An essential difficult problem is that an application may change the resource usage at every moment, and it is difficult to predict when the changes occur. Especially, if an

application processes a media stream with the variable bit rate, the problem becomes very serious, and it is difficult to make its feedback loop stable. We need to investigate more sophisticated control scheme that ensures the stability of the adaptation.

The fourth experience shows a difference between the characteristics of between processor reservation and memory reservation. In the processor reservation, unused processor resource can be used by non timing critical applications since the resource can be preempted easily¹. On the other hand, a page is reserved for timing critical applications cannot be used for non timing critical applications. If a non timing critical application modifies the page, the page should be swapped out for allowing the page to be used for the timing critical application. The first solution for solving the problem makes a swapping device fast, but the solution is very expensive. The second solution does not allow to modify a page reserved for timing critical applications. If the page is modified, another page that is not reserved should be allocated, and the former page is copied to the latter page. However, the solution requires to modify a large part of the virtual memory system.

6 Conclusion

In this paper, we described an adaptive QOS mapping scheme, and resource reservation mechanisms supporting the adaptive QOS mapping scheme. The reservation mechanisms described in the paper are implemented on Real-Time Mach, and we described the effectiveness of our approach.

However, complete continuous media applications may require other resource reservation mechanisms such as a device reservation and a network reservation. In our project, we have been working on these mechanisms[TN96], but the current mechanisms are not suitable for the adaptive QOS mapping scheme. We need to investigate on these reservation mechanisms that can be used in the adaptive QOS mapping scheme.

References

- [BKTZ94] A.Banerjea, E.Knightly, F.Templin, and H.Zang, "Experiments with the Tenet Real-Time Protocol Suite on the Sequoia 2000 Wide Area Network", In Proceedings of ACM Multimedia'94, ACM, 1994.
- [CT94] C.L.Compton, and D.L.Tennenhouse, "Collaborative Load Shedding for Media-Based Application", In Proceedings of International Conference on Multimedia Computing and Systems, 1994.
- [MST94] C.W.Mercer, S.Savage, and H.Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications", In Proceedings of the First International Conference on Multimedia Computing and Systems, IEEE, 1994.
- [NTAT93] T.Nakajima, T.Kitayama, H.Arakawa, and H.Tokuda, "Integrated Management of Priority Inversion in Real-Time Mach", In Proceedings of the Real-Time System Symposium' 93, IEEE, 1993.
- [NT94] T.Nakajima and H.Tezuka, "A Continuous Media Application supporting Dynamic QOS Control on Real-Time Mach", In Proceedings of the ACM Multimedia '94, 1994.
- [Nak96] T.Nakajima, "A Dynamic QOS Control based on Optimistic Processor Reservation", In Proceedings of the 3rd International Conference on Multimedia Computing and Systems, IEEE, 1996.
- [NL97] J. Nieh, and M. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications", In Proceeding of the 16th ACM SIGOPS, 1997.
- [TN95] H.Tezuka, and T.Nakajima, "Experiences with building a Continuous Media Application on Real-Time Mach", In Proceedings of the 2nd International Workshop on Real-Time Computing, Systems, and Applications, IEEE, 1995.
- [TN96] H. Tezuka, and T. Nakajima "Simple Continuous Media Storage Server on Real-Time Mach", In Proceedings of USENIX Technical Conference, 1996.

This article was processed using the \LaTeX macro package with LLNCS style

¹ Actually, it is difficult to preempt a currently executed thread at any time. For example, if a kernel routine is executed while any interrupts are masked, the preemption does not occur until a system enables interrupts.