システムソフトウェアと　77－10
オペレーティング・システム
マルチメディア通信と分散処理　87－10
（1998．2．26）

Memory Reservation System in Real-Time Mach

Tatsuo Nakajima
tatsuo@jaist.ac.jp
http://mmmc.jaist.ac.jp:8000/
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Ishikawa, 923-12, JAPAN

Abstract

This paper proposes a virtual memory management system suitable for interactive continuous media applications. Our virtual memory management system enables continuous media applications to reserve physical memory for allocating pages as soon as possible when the applications require the pages. The system implicitly and incrementally allocates and wires pages used for processing timing critical media data. Also, our system supports applications that adapt the amount of wired memory to the memory usages of other continuous media applications.

1 Introduction

Operating system supports for continuous media applications are one of the most exciting topics in operating system researches. Continuous media applications such as multimedia conference systems and video on-demand systems that play audio and video data must satisfy their timing constraint requirements. Real-time techniques are attractive since the correctness of continuous media processing depends on whether timing constraints of respective media data are satisfied or not. Actually, some researchers have reported the effectiveness of the real-time technologies for supporting continuous media applications[2, 4, 7, 9, 12].

Continuous media applications usually contain some codes for interacting with users[5, 11]. We call the applications ***interactive continuous media applications***. For example, movie player applications have several buttons such as ***play***, ***stop***, and ***pause*** for providing VCR capability. Most interactive continuous media applications adopt standard user interface toolkit libraries such as the Motif toolkit. Usually, such libraries make the code and data segments of the applications very big, and they cause new serious problems for supporting continuous media applications.

In traditional operating systems, demand paging makes it possible to execute such large applications by storing a large part of pages in secondary storages. However, continuous media applications should avoid page faults for ensuring timing constraints of continuous media since it takes a long time to swap pages between physical memory and secondary storages, and this makes it difficult to satisfy timing constraints of continuous media. Therefore, some operating systems provide memory wiring primitives that enable applications to wire pages in physical memory by specifying the range of virtual address spaces explicitly.

However, the traditional wiring primitives cause the following two serious problems that are not preferable for general operating systems.

- Memory wiring primitives are not secure since malicious applications may monopolize physical memory by wiring memory unlimitedly. Thus, the wiring primitives should be called by only privileged users.
- It is difficult to predict which pages are used for processing media data since memory wiring primitives require to specify the range of virtual address spaces explicitly for wiring pages.

This paper proposes a virtual memory management system for interactive continuous media applications. Our virtual memory management system enables continuous media applications to reserve physical memory for allocating pages as soon as possible when the application requires pages. The system implicitly and incrementally allocates and wires pages used for processing timing critical media data. The approach solves the above problems caused in traditional memory wiring primitives. Also, we implemented a prototype system, and show the effectiveness of our approach in this paper.

2 Issues of Virtual Memory Management for Interactive Continuous Media Applications

2.1 Memory Resource Reservation

Recently, several operating system supports adopting real-time technologies are proposed for ensuring timing constraints of continuous media applications. For example, processor reservation systems reserve processor cycles for continuous media applications[4, 7], and several network systems that enable applications to reserve network bandwidth have been developed[2]. Also, real-time synchronization and IPC make the blocking time of threads small, and the real-time server model is proposed for improving the preemptability of servers[8].

However, these real-time resource management techniques cannot ensure to satisfy timing constraints of continuous media if page faults occur. Thus, traditional operating systems provide some primitives to wire pages in physical memory. For example, *plock()*, *mlock()*, *mlockall()* are provided in the Unix[6]. The primitives are very dangerous since there is no limit to wire pages in physical memory, and physical memory may be monopolized by an application. Thus, these primitives can be used from only privileged users. For instance, database servers that are started by only privileged users use the primitives to wire buffer caches in physical memory.

Continuous media applications on Real-Time Mach use memory wiring primitives for avoiding page faults in continuous media applications. *Vm_wire()* is used to wire pages of a specified range of an application's virtual address space. Also, *task_wire_code()* is used for wiring code segments, and *task_wire_data()* is used for wiring data segments. For wiring all pages of applications, *task_wire_future_data()* may be used[1].

The primitives enable applications to avoid page faults. However, operating systems cannot protect physical memory from malicious uses if the primitives are used by usual application programmers freely. Thus, continuous media applications require secure operating system primitives for avoiding page faults in continuous media applications.

2.2 Memory Management for Interactive Continuous Media Applications

One of hot topics in operating system researches is building extensible operating system kernels for customizing physical resource managements using applications specific policies. Actually, several systems implemented for supporting application specific memory management policies. However, a virtual memory management policy suitable for interactive continuous media applications has not been reported yet. In this section, we analyze how memory is used in interactive applications in detail, and describe which virtual memory supports are required for supporting interactive continuous media applications.

Figure 1 shows a typical virtual address space layout for a continuous media application. Let us assume that the application contains two threads so that two stack segments are contained in the address space. The code segment is shared by the two threads, and the static data segment is also shared by the threads, and contains global variables used in the application. Lastly, the dynamic data segments are allocated when a new buffer is required. The buffer is used for storing media data for removing jitters as described in Section 2.1.

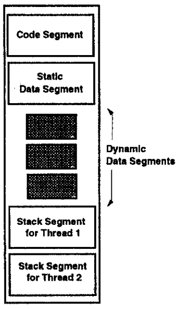

Figure 1: Address Space of Interactive Continuous Media Applications

The dynamic data segments have different characteristics from other memory segments. Programmers know the exact size and position of the segments in the address space since the dynamic segments are explicitly allocated by the programmers. Thus, the size of the segments can be explicitly increased or decreased by changing the quality of a media stream since the necessary sizes of the dynamic data segments can be exactly calculated from the sizes of media elements. The issue will be described in the next section in detail. In the section, we focus on issues caused in a code segment. We omit issues in the management of a static data segment and stack segments since their managements are similar to the management of the code segment.

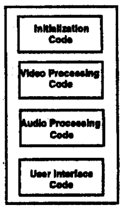

Figure 2: Layout of Code Segment

Figure 2 illustrates a memory layout of a code segment of a typical interactive continuous media application. The application contains an initialization code, a user interface management code, a video processing code, and an audio processing code. The execution of the initialization code and the user interface management code is not timing critical. Therefore, the number of wired pages is decreased dramatically by wiring only pages containing the video processing code and the audio processing code in physical memory while processing media data, since the video processing code and the audio processing code are not usually bigger than the initialization code and the user interface management code.

However, it is difficult to know which pages contain procedures called by timing critical threads since a linker that is available on a traditional workstation does not combine the procedures by taking into account the relationship between the procedures, and page faults cannot be controlled without knowing which pages containing the procedures called by timing critical threads in traditional operating systems.

3 Memory Reservation System for Continuous Media Applications

3.1 Design Goals

The following design goals were considered in the design of our virtual memory management system.

- The virtual memory management system should be simple and easily implemented.
- The virtual memory management system should be secure. This means that applications cannot wire pages in physical memory without reserving pages that will be wired for applications.
- The system should enable applications to negotiate the amount of wired pages in physical memory.
- The virtual memory management system avoids page faults even if the virtual address of code and data segments used for processing media data are not known in advance[1].

The first goal can be realized by reducing the modification of an existing virtual memory system to the minimum. The strategy makes our virtual memory system to port to other operating systems easy.

Our virtual memory management system is divided into two parts. The first part is a memory reservation system and the second part is an incremental memory wiring system. The second and third goals can be achieved by the memory reservation system, and the fourth goal can be realized by the incremental memory wiring system. In the following sections, we describe these two systems in detail.

3.2 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.

[1]Our approach presented in this paper does not avoid all page faults since it requires one page fault during startup for each page.

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.

As described in Section 2.1, 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[3, 9, 10], 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.

Figure 3 illustrates how applications change the size of dynamic data segments. Let us assume that application 1 causes a page fault, but there is no reserved free page in the reserved page pool for the application. In this case, a message notifying that there is no reserved page in the pool is delivered to the application. Then, the application changes the quality of media, and reduces the size of its dynamic data segment. On the other hand, application 2 calls a memory reservation request for increasing the number of reserved pages for the application periodically. If the reservation request is succeeded, the application may increase the size of its dynamic data segment, and may upgrade the quality of media.

3.3 Incremental Memory Wiring

Our memory reservation system allows applications not to specify which pages should be wired explicitly in

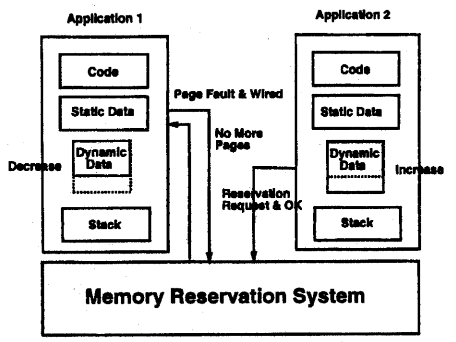

Figure 3: Renegotiation of Wired Pages between Applications

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 Prototype Implementation

We implemented a prototype memory reservation system described in the previous section on Real-Time Mach. This section presents how Real-Time Mach kernel is modified for supporting our memory reservation system. Also, we describe additional system primitives for the memory reservation system, and a sample program using the memory reservation system.

4.1 Structure

Figure 4 shows the structure of our prototype implementation. An application sends a request to the admission server for reserving pages for the application. The admission server determines whether the request can be accepted or rejected. When the request is accepted, the admission server calls a memory reservation primitive to a kernel.

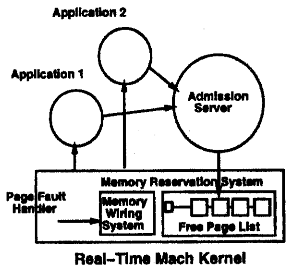

Figure 4: Structure of Prototype Implementation

In the current implementation, a traditional free list and reserved page pools are combined in a unique page list for making the implementation simple. We call the page list *free page list* in this paper. When the memory reservation primitive is called, the memory reservation system checks the number of pages in the free page list. In the prototype, the reserved pages are classified into wired pages and reserved free pages, and the reserved free pages are kept in the free page list. Thus, the kernel ensures that the number of pages in the free list is more than the number of reserved free pages. If the number of pages in the free list is equal to the number of reserved free pages, pages allocated for other non real-time applications are reclaimed and cleaned until the number of pages in the free list exceeds the total number of reserved free pages required for respective continuous media applications.

When a real-time thread causes a page fault, the incremental memory wiring system allocates a reserved free page from the free page list and wires it in physical memory. Also, the incremental memory wiring system monitors the number of the pages for respective tasks in physical memory, and if the number of wired pages exceeds the number of reserved pages for the task, it delivers a notification message to the task.

In Real-Time Mach, many OS functionalities are implemented as OS servers, then it is important how to manage page faults in the servers since timing constraints of media data may be violated due to the server's page fault. In our prototype, we assume that servers receive requests from application tasks using RT-IPC[8]. When a server receives a request using RT-IPC, the attribute of an application thread is inherited by the thread processing the request in the server[2]. Thus, if a thread sending a request from an application task is a real-time thread, the thread receiving the request also becomes a real-time thread, and pages touched by the thread are wired in physical memory.

Actually, X server on Real-Time Mach receives requests from an application using RT-IPC. When video frames are displayed by X server, the code and data segments used for displaying the video frames are wired in physical memory for avoid extra page faults.

4.2 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 admission 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[13]. 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.

[2]The attribute is cleared when the thread waits for other requests from clients.

5 Evaluation

In this section, we show the evaluation of our prototype implementation using QtPlay movie player[12]. The evaluation used Gateway2000/P5-66 which has a 66MHz Intel Pentium processor, 16 MB of memory and 1 GB SCSI disk.

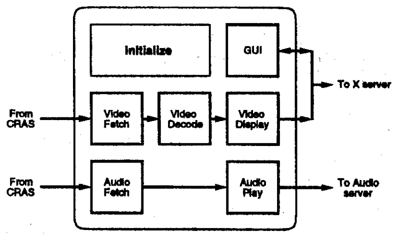

Figure 5: Movie Player

Figure 5 shows the structure of QtPlay movie player. QtPlay retrieves video frames and audio samples from a QuickTime file stored in Unix file system using CRAS[13] 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.

The two threads belonging in the second category process an audio stream. One thread fetches audio samples from a storage server, and another thread sends them to the audio server. The third category contains three threads for processing a video stream. The first thread retrieves video frames from the storage server, the second thread decompresses the video frames, and the last and third thread sends them to X server.

In QtPlay, threads in the second and third categories that process video frames and audio samples are real-time threads. Thus, memory pages accessed by the threads are wired in physical memory. Also, when the threads send requests to the X server and the audio server, pages touched by the threads receiving the requests in the servers are also wired. In the evaluation described in the section, stack segments are explicitly wired when threads are created since Real-Time Mach allocates stack segments and wires them in physical memory inside a primitive initializing a thread attribute.

5.1 The Number of Wired Pages

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 1 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 1: Size of Memory for Code, Data Segments and Entire Address Space

Table 2 shows the number of actual wired pages and resident pages[3] 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.

Wiring Strategy(QtPlay)	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
Wiring Strategy(X Server)	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 2: 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.

[3]Physical pages which are allocated for an application.

6 Conclusion

In this paper, we proposed a virtual memory management for interactive continuous media applications. Our approach solves several problems in traditional virtual memory management systems when interactive continuous media applications are executed.

In our approach, the necessary number of physical pages is reserved before starting to process timing critical continuous media streams, and it is automatically determined which pages are touched by threads processing media data. The approach wires pages in physical memory, and it does not require to specify the range of the address space for wiring pages since pages are wired in physical memory when they are touched for the first time.

References

[1] CMU ART Group. *"Real-Time Mach 3.0 User Reference Manual"*. School of Computer Science, Carnegie Mellon University, 1994.

[2] 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.

[3] L.Delgrossi, C.Halstrick, D.Hehmann, R.Herrtwich, O.Krone, J.Sandvoss, and C.Vogt, "Media Scaling in a Multimedia Communication System", Multimedia Systems, Vol.2, No.4, Springer-Verlag, 1994.

[4] H.Fujita, H.Tezuka, and T.Nakajima, "A Processor Reservation System supporting Dynamic QOS control", In Proceedings of the 2nd International Workshop on Real-Time Computing, Systems, and Applications, IEEE, 1995.

[5] V.Jacobson, "Multimedia Conferencing on the Internet", SIGCOMM'94 Tutorial, 1994.

[6] S.J.Leffler, M.K.McKusick, M.J.Karels. *"The Design and Implementation of the 4.3BSD UNIX Operating System"*, Chapter 5: Memory Management. Addison-Wesley, 1989.

[7] 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.

[8] 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

[9] T.Nakajima and H.Tezuka, "A Continuous Media Application supporting Dynamic QOS Control on Real-Time Mach", In Proceedings of ACM Multimedia'94, ACM, 1994.

[10] 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.

[11] L.A.Rowe, and B.C.Smith, "A Continuous Media Player, In Proceedings of the 3th International Workshop on Network and Operating System Support for Digital Audio and Video, 1992.

[12] 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.

[13] H.Tezuka, and T.Nakajima, "Simple Continuous Media Storage Server on Real-Time Mach", In Proceedings of the USENIX 1996 Technical Conference, USENIX, 1996.