Virtual memory management for interactive continuous mediaapplications

Conference Paper · July 1997

DOI: 10.1109/MMCS.1997.609752 · Source: IEEE Xplore

2 authors, including:

Tatsuo Nakajima
Waseda University
416 PUBLICATIONS **4,094** CITATIONS

SEE PROFILE

Virtual Memory Management for Interactive Continuous Media Applications

Tatsuo Nakajima Hiroshi Tezuka
{tatsuo,tezuka}@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. Interactive continuous media applications usually require a large amount of memory for storing their code, data, and stack segments. In traditional operating systems, demand paging makes it possible to execute such large applications by storing most pages in secondary disks. 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. Thus, it is 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. On the other hand, 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, 7, 12, 14, 17].

Continuous media applications usually contain some codes for interacting with users[8, 16]. 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.

The remainder of this paper is structured as follows. In Section 2, we present some issues of the virtual memory management system for supporting timing-critical applications. Section 3 describes our approach of virtual memory management system for supporting interactive continuous media applications. In Section 4. we present a prototype implementation of our proposed virtual memory system on Real-Time Mach[13, 19]. Section 5 shows the evaluation of the prototype implementation, and Section 6 presents several experiments with our virtual memory management system.

2 Issues of Virtual Memory Management for Interactive Continuous Media Applications

This section describes several problems in traditional virtual memory managements when they are used for interactive continuous media applications. First, we present the structure of interactive continuous media applications that this paper assumes, then describe problems of traditional memory wiring primitives.

2.1 Interactive Continuous Media Applications

The structure of interactive continuous media applications has a great impact on the timing constraints of processing continuous media. In a traditional process model that provides only one activity in each process[1], processing events from users such as mouse inputs and processing media data should be executed in the same loop statement as shown below.

```
1  while(TRUE) {
2      ev_name = event_wait(33 ms);
3      switch (ev_name) {
4      case USER_EVENT:
5          process an event from a user;
6          break;
7      case TIME_OUT:
8          process a media element;
9          break;
10     }
11 }
```

The program processes a media stream whose frame rate is 30 frames per second. While processing the media stream, the program needs to process user events. When a user issues a keyboard or a mouse input, the name of an event is returned, and the event is executed in line 5. However, If no event occurs within 33 ms, a timeout event is returned, and the program processes a video frame in line 8.

The approach that is adopted in many interactive continuous media applications on MBONE[8] is difficult to ensure timing constraints of media data since the timing constraints are easily violated if the time spent for processing an user event exceeds 33 ms. Also, adding a new media stream in the loop is not easy since it may require to change the structure of the loop. Also, it is difficult to write a program that detects timing errors and recovers from the errors.

In an alternative approach, different threads for processing events from users and for processing media data are created. The application structure is illustrated in Figure 1. Also, in the approach, respective threads are created for processing respective media streams.

In the figure, circles indicate respective threads. Also, one thread executes a procedure containing a traditional event loop that processes events from users. In the figure, two streams are created: one stream processes video frames and another stream processes audio samples. Each media stream is processed by respective two threads. For the video stream, one thread retrieves video frames from a continuous media storage system, and another thread draws the video frames to a video window. A buffer between the two threads is used for removing jitters by keeping video frames in the buffer until the timestamp of each video frame is greater than the current time.

[1] A typical example of the model is UNIX process model.

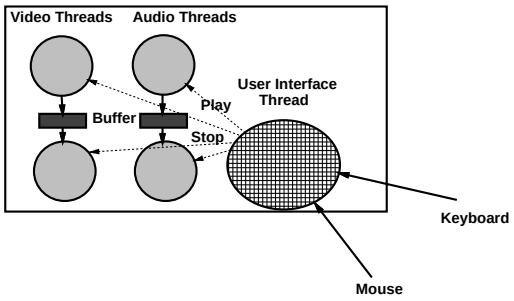

Figure 1: Structure of Interactive Continuous Media Application

The application structure is very suitable for ensuring timing constraints of continuous media data since the threads in the applications are clearly partitioned into real-time threads and non real-time threads. The non real-time threads execute an initialization code, an error recovery code, and an user interface management code. However real-time threads process continuous media streams, and these threads are scheduled in a timely fashion using the real-time scheduling.

In this paper, we assume that interactive continuous media applications adopt the application structure. Our virtual memory management system relies on the separation of threads executing a real-time part from those executing a non real-time part.

2.2 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[7, 12], 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[13].

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[11]. 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.

2.3 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[3, 4, 6]. Actually, several systems implemented for supporting application specific memory management policies[9, 10]. 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.

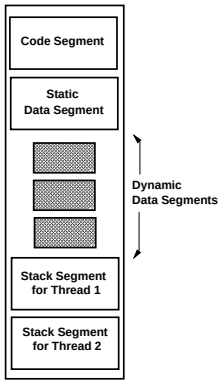

Figure 2: Address Space of Interactive Continuous Media Applications

Figure 2 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.

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 3 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

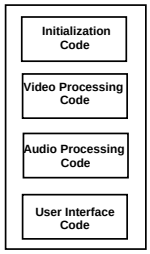

Figure 3: Layout of Code Segment

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[2].

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.

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

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.

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[5, 14, 15], 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 4 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 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.

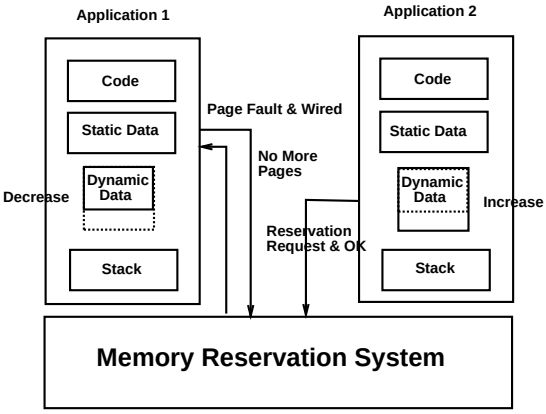

Figure 4: Renegotiation of Wired Pages between Applications

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 5 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.

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

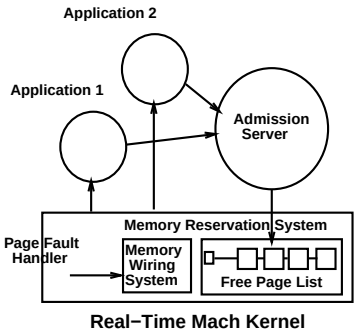

Figure 5: Structure of Prototype Implementation

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[13]. 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[3]. 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)

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

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[18]. 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.

4.3 Dynamic Adjustment of Reserved Pages

In our prototype implementation, applications need to specify the number of reserved pages for the applications. However, it is usually difficult to predict how many pages are accessed by real-time threads. Our prototype system provides the following optimistic strategy for reserving pages.

- An application specifies the number of reserves pages in the initialization step.
- The number of reserved pages is chosen by programmers, but the number is changes by monitoring the actual memory usage during the execution of an application.
- If the application receives a message notifying that the number of pages accessed by real-time threads exceeds the number of reserved pages, it increases the number of reserved pages.
- The application monitors its memory usage periodically, and it decreases the number of reserved pages

if the current number of reserved pages is more than the number of actually wired pages.

The scheme makes the memory utilization high since the feedback control ensures that the number of reserved pages are adjusted to the number of actual wired pages in physical memory by monitoring the current memory usage.

4.4 Sample Scenario

In this section, we present a sample program showing how to use virtual memory primitives described in Section 4.2. The program contains two threads. The first thread executes the initialization code and waits for notification messages from a kernel. The second thread processes media data, and pages accessed by the thread are wired in physical memory incrementally.

```
1  main()
2  {
3      initialize_task();
4
5      ret = vm_reserve_request(my_task, size, port);
6      if(ret != SUCCESS) {
7          exit(1);
8      }
9
10     thread = create_realtime_thread();
11     vm_thread_wire_policy(thread, WIRE_POLICY_SCHED);
12
13     sleep(few seconds);
14
15     actual_size = vm_reserve_usage();
16     vm_reserve_request(my_task, actual_size, port);
17
18     while (!finish) {
19         if (memory_exceed_wait(port) == SUCCESS) {
20                 vm_reserve_request(my_task, actual_size);
21         }
22     }
23
24     vm_reserve_request(my_task, NULL, NULL);
25 }
```

In line 5, the program sends a memory reservation request to the admission server. The admission server calls *vm_reserve* to reserve physical memory for the application that sends the request. If the request is failed, the program is terminated.

In line 10, a new real-time thread is created, and the pages that are accessed by the thread will be incrementally wired after calling *vm_thread_wire_policy* in line 11. In line 15, the actual number of wired pages is monitored by using *vm_reserve_usage* that is a library function provided by Real-Time Mach, and the program releases extra pages in line 16 by sending a memory reserve request to the admission server again.

In line 19, the program waits for messages notifying that the number of wired pages exceeds the number of reserved pages. If the primitive is returned, the number of reserved pages is increased in line 20 by sending a request to the admission server. In line 24, all reserved pages are released before the application is terminated.

5 Evaluation

In this section, we show the evaluation of our prototype implementation using QtPlay movie player[17]. The evaluation used Gateway2000/P5-66 which has a 66MHz Intel Pentium processor, 16 MB of memory and 1 GB SCSI disk.

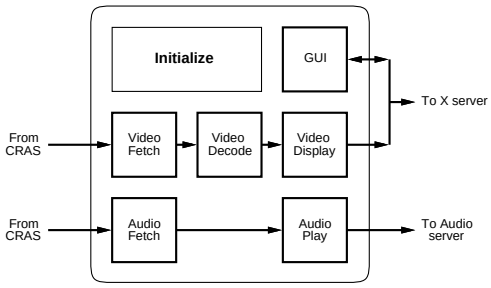

Figure 6: Movie Player

Figure 6 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[18] 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[4] 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.

5.2 The Number of Page Faults

In this section, we show how many times page faults occur, and how long page faults are processed while executing QtPlay. In the evaluation, QtPlay draws 30 frames per second, where each frame has 320 × 240 and each pixel has 8 bit depth. We run QtPlay for 5 seconds, and measure the time spent for processing the page faults in Figure 7, 8, 9, and 10 under four wiring strategies. In the figure, the horizontal lines indicate the time when page faults occur, and the vertical lines indicate the time spent for processing the page faults.

Wiring Strategy	The Number of Page Faults
No Wiring	322
Code	357
Code + Static Data	326
Code + Static Data + Dynamic Data	0
Our Approach	181

Table 3: The Number of Page Faults

Also, we show the number of page faults under respective wiring policies in Table 3. In the evaluation, we run an application that uses a large amount of memory with QtPlay simultaneously.

The result shows that wiring code segments and data segments does not reduce the number of page faults, but our approach reduces the number of page faults significantly.

Figure 7 shows the result when no memory is wired, and Figure 8 shows the result when the code segments are wired. Since most of pages in the code segments are frequently accessed, a few pages are reclaimed even when the code segments are not wired, and the memory utilization

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

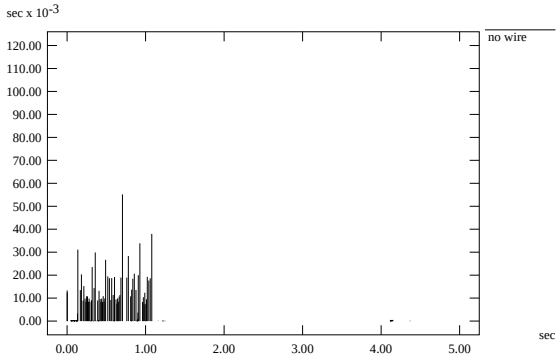

Figure 7: The time spent for page faults without memory wiring

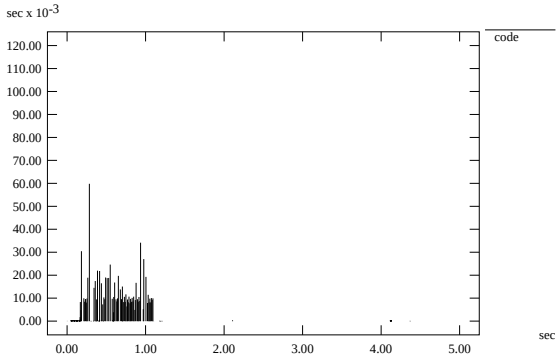

Figure 8: The time spent for page faults with wired code segments

is very high[5]. Thus, the result when the code segments are wired is similar to the result when no memory is wired.

Figure 9 shows the result when both the code and static data segments are wired. The result shows that it takes a long time to process page faults since many pages are wired in physical memory, and some pages may be swapped out in a disk. In this strategy, dynamic data segments are not wired[6]. Thus, it needs to take a long time since the pages for the dynamic data segment are required to be transferred from a swap disk. Figure 10 shows the result when using our approach. The result shows that a few unwired pages may become the bottleneck of a continuous media application under high memory utilization so that our approach reduces the number of page faults dramatically although a few pages are wired.

5.3 Dynamic Adjustment of Memory Reservation

In Section 4.3 and 4.4, we describe a feedback scheme for adjusting the number of reserved pages for a continuous media application. In this section, we show how our memory reservation system adjusts the number of pages using the feedback scheme.

[5] In this case, most of reclaimed pages will cause page faults before they are swapped out in a disk.

[6] If all segments are wired, no page fault occurs.

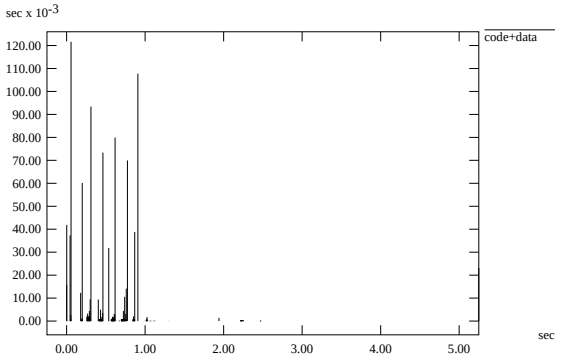

Figure 9: The time spent for page faults with wired code and static data segments

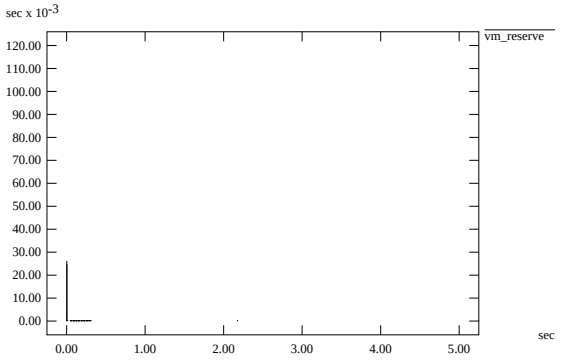

Figure 10: The time spent for page faults under our approach

Figure 11 shows that our reservation system can adjust the amount of reserved memory according to the number of actual wired pages in physical memory. In the figure, the vertical line indicates the amount of the resident, reserved, and wired memory of the test application. The horizontal line indicates the time while the test program runs.

In the test, the application decreases the number of reserved pages when the number of wired pages is not changed for a second. Also, when a message, which notifies that the number of wired pages exceeds the number of reserved pages, is received by an application, the application increases the number of reserved pages.

In the evaluation, the application decreases the number of wired pages from 22 sec. to 32 sec. since the application reduces the frame rate of a video stream. The result shows that our approach makes memory utilization high by adjusting the number of reserved pages according to the number of actual wired pages.

6 Discussion

In our prototype implementation, programmers need to specify the number of pages in order to reserve the pages touched by timing critical threads before starting the threads. However, it is difficult to predict how many pages are used for continuous media applications. There

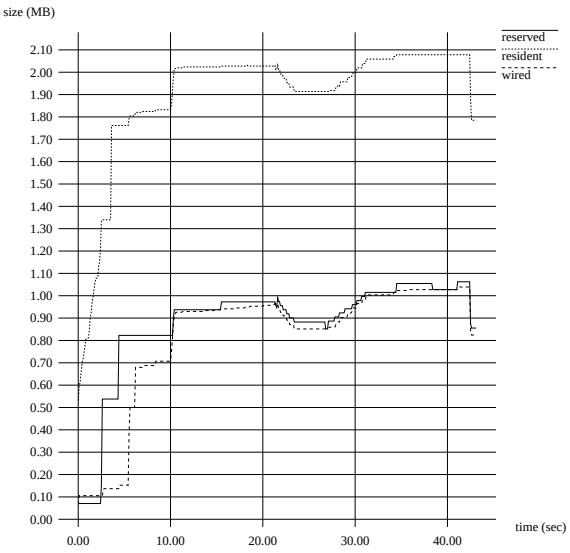

Figure 11: Negotiation

are several strategies for solving the problem. In the first strategy, programmers reserve a large number of pages for the applications, and release unused pages by monitoring actual memory usages. If many applications run concurrently, these applications need to reserve a very large number of pages until they can know the actual memory usages. Thus, a system may reject the memory reservation requests of some applications. In the second strategy, an application reserves the minimum number of pages, and increases the number of reserved pages when it receives messages notifying that all reserved pages are wired. In this case, the application may violate its timing constraints until enough pages are reserved. The strategy is not appropriate if it is important to ensure the quality of media data at every moment. In the third approach, an application reserves pages whose number is the same as the number of pages when the application run previously. However, the number of necessary wired pages may be different if media streams may have different media formats.

The interaction between continuous media applications and interactive applications causes many serious problems that must be solved. In our current implementation, unwired pages in physical memory may be swapped out in LRU order when a memory overload occurs. However, if a page fault frequently occurs, and interactive applications access code and data segments, a response time may be significantly degraded. Especially, the problem is very serious when many pages are reserved for continuous media applications, and a small amount of memory can be remained for other applications. In this case, resident pages should be carefully controlled for maintaining a good response time for interactive applications.

However, the problem is very difficult to solve in Real-Time Mach. In Real-Time Mach, real-time applications are carefully designed so that it is easy to control their resources by communicating with operating system servers through RT-IPC[13], and real-time thread packages[19]. However, threads processing events from users are executed as non real-time threads in Real-Time Mach, and they do not reserve CPU and memory. There are three reasons for making it difficult to solve the problem. The

first reason is that it is difficult to schedule aperiodic events without disturbing threads that process continuous media data. If many events occur and they are scheduled at a high priority, processing continuous media may violate the timing constraints. However, processing events at a low priority may also cause a serious problem for continuous media applications. For example, if an event for focusing a window to playback a video stream is delayed to be processed, the application may not control the video stream in a timely fashion. The second reason is caused by the structure of Unix server. The Unix server cannot distinguish requests from applications so that an important request may be blocked until another unimportant request is terminated. Thus, it is difficult to receive keyboard and mouse inputs from Unix server in a timely fashion for achieving a good response time. The last reason is caused by the current implementation of X server. The playback of video frames to a window may be delayed by a previous long processing event since the X server is single threaded. Also, a window manager should respond to events from the X server as soon as possible if the events(e.x. focus management) may be related to interactive continuous media applications. This requires that the X server and window managers distinguish events related to continuous media applications from those of other applications.

In the current implementation, we assume that the incremental memory wiring described in the paper is used for wiring all pages. However, the virtual address regions for dynamic data segments can be explicitly allocated by programs. This means that the virtual address space of the segments can be known by the programs. In this case, the applications can wire pages in physical memory for the segments before the segments are used, and release the pages after the segments are not necessary[7]. Therefore, for dynamic data segments, it is better to control physical memory explicitly by using vm_wire and vm_discard primitives[8]. On the other hand, the incremental memory wiring is used for code, static data, and stack segments in order to wire pages in physical memory implicitly.

One of the limitation of the current implementation is that all pages wired by the incremental memory wiring reside in physical memory until an application that uses the pages is terminated. Therefore, the memory utilization becomes worse since rarely used pages may be wired in physical memory. The problem can be solved, if pages wired by the incremental memory wiring are unwired periodically. The pages accessed by the applications cause page faults, and they are wired in physical memory again. The strategy ensure to wire only pages that are used frequently in physical memory.

7 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.

[7] When the amount of wired pages exceeds the amount of reserved pages, a notification is delivered to an application for wiring pages in physical memory in a secure fashion.

[8] Vm_discard is a primitive provided by Real-Time Mach, and makes pages in a specified virtual address space free if the pages are in physical memory.

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] B.N.Bershad, S.Savage, P.Pardyak, E.G.Sirer, M.Fiuczynski, D.Becker, S.Eggers, and C.Chambers, "Extensibility, Safety, and Performance in the SPIN Operating Systems", In Proceedings of the 15th ACM Symposium on Operating Systems Principles, ACM, 1995.

[4] D.R.Cheriton, K.J.Duda, "A Cashing Model of Operating System Kernel Functionality", In Proceedings of the First Symposium on Operating System Design and Implementation, USENIX, 1994.

[5] 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.

[6] D.R.Engler,M.F.Kaashoek, and O'Toole, " Exokernel: An Operating System Architecture for Application-Level Resource Management", In Proceesings of the 15th Symposium on Operating Systems Principles, ACM, 1995.

[7] 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.

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

[9] K.Harty, and D.R.Cheriton, "Application-Controlled Physical Memory using External Page Cache Management", In Proceedings of the 5th International Conference on Architecture Support for Programming Language and Operating Systems, ACM, 1992.

[10] C.Lee, M.Chang and R.C. Chang, "HiPEC: High Performance External Virtual Memory Cashing", In Proceedings of the First Symposium on Operating System Design and Implementation, USENIX, 1994.

[11] 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.

[12] 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.

[13] 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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] H.Tokuda, T.Nakajima, and P.Rao, "Real-Time Mach: Towards a Predictable Real-Time System", In Proceedings of the USENIX First Mach Symposium, USENIX, 1990.