

Real-Time Mach RK98 User Reference Manual

Resource Kernel Project
Real-time and Multimedia Systems Lab
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

April 1998

Acknowledgements

Contributors to this document include Takuro Kitayama, Clifford W. Mercer, Tatsuo Nakajima, Stefan Savage, Hideyuki Tokuda, and Jim Zelenka. Jim Zelenka was the primary author of the original revision. The members of the project also include Ragunathan Rajkumar, Chen Lee and Julie Landers. The Real-Time Mach project would like to thank the many people who contributed to the implementation of Real-Time Mach including Stephen T.-C. Chou, Hiroshi Arakawa and Noritake Ashida. Members of the MKng project at Keio University, JAIST and member companies are also gratefully acknowledged.

This work was supported in part by the Defense Advanced Research Projects Agency in part under agreement E30602-97-2-0287 and in part by the Office of Naval Research and in part by the Office of Naval Research under agreement N00014-92-J-1524, a National Science Foundation Graduate Fellowship and by Bellcore. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of NSF, Bellcore, NOSC, or the U.S. Government.

Point of Contact:

Ragunathan (Raj) Rajkumar
Phone: 412 268-8707 (or) 412 268-7668
Fax: (412) 621-5473
Email: raj+@cs.cmu.edu

Contents

1	Introduction	1
2	Programming Environment	3
2.1	<i>Scheduler 1-2-3</i>	3
2.2	<i>ARM</i> , Advanced Real-time Monitor	4
3	Creating and Executing a Real-Time Program	7
3.1	UX Environment	7
3.2	RTS Environment	7
3.3	Standalone Environment	9
4	Realtime Scheduling and Thread Creation	10
4.1	Retrieving the current scheduling policy	10
4.2	Creating and Running Realtime Threads	12
4.2.1	Creating Aperiodic Threads	12
4.2.2	Using rate-monotonic scheduling	13
5	Clock and Timer Management	18
5.1	Realtime Clocks and Timers	18
5.2	Using a Realtime Clock	18
5.2.1	Reading the clock	18
5.2.2	Memory-mapping the clock	19
5.2.3	Clock Resolution	21
5.2.4	Identifying clocks	23
5.3	Alarmclock Timer	25
5.3.1	Sleeping on a timer	25
6	Processor Reservations	28
6.1	Creating and Destroying Reservations	28
6.2	Reservation status	31
7	Hierarchical Reservation Scheduling	34
7.0.1	Hierarchical Reservation Example 1	34
7.0.2	Hierarchical Reservation Example 2	43
7.0.3	Hierarchical Reservation Example 3	46
7.0.4	Hierarchical Reservation Example 4	50
7.0.5	Hierarchical Reservation Example 5	55
8	Virtual Memory Managment	64
8.1	Allocating and Wiring	64
8.2	Allocating and Wiring Program Components	64
8.3	Allocating and Wiring Program Text and Data	64
8.4	Sharing	65

8.5	Sharing Program Components	65
8.6	Sharing Program Text	66
9	Real-Time Synchronization	68
9.1	Real-Time Mutex	68
9.2	Condition Variable	70
10	Real-Time IPC	74
10.1	Server Program	74
10.2	Client Program	76
10.3	MIG Definition File	78
11	Device Management	79
11.1	Device	79
11.1.1	Console	79
11.1.2	Floppy Disk	80
11.2	Name Space for Device	82
12	Pro Audio Spectrum 16 and Sound Blaster	84
12.0.1	The PAS-16	84
12.0.2	Running the PAS-16 utilities	84
13	Display Screen (DS) Library	86
13.1	Program Overview	86
13.2	Program Components	86
13.3	Program Text	87
14	Timeline and Fault-Tolerant Scheduling	92
14.0.1	Timeline Scheduling Example 1	92
14.0.2	Timeline Scheduling Example 2	94
14.0.3	Timeline Scheduling Example 3	98
14.0.4	Timeline Scheduling Example 4	102
14.0.5	Timeline Scheduling Example 5	106
14.0.6	Fault-Tolerant Timeline Scheduling Example 1	109
14.0.7	Fault-Tolerant Timeline Scheduling Example 2	111
14.0.8	Fault-Tolerant Timeline Scheduling Example 3	116

List of Figures

2.1	<i>Scheduler 1-2-3</i>	5
2.2	<i>ARM</i>	6
3.1	<i>RT-Mach Programming Environment</i>	8

Chapter 1

Introduction

This Real-Time Mach User Reference Manual describes how to create, execute and debug a distributed real-time program on the Real-Time Mach operating system. In particular, this manual contains example programs to demonstrate novel features in Real-Time Mach. If you need to understand the details of Real-Time Mach kernel primitives, please consult the man pages.

Several papers which describe various aspects of the design and implementation of Real-Time Mach have been published.

Please refer to the publications available online at <http://www.cs.cmu.edu/~rtmach> for the most recent list of published papers. The following is a very outdated list of some of these papers.

Tokuda, H., Nakajima, T. and Prithvi Rao, "Real-Time Mach: Towards Predictable Real-Time Systems", in *Proceedings of USENIX 1990 Mach Workshop*, October 1990.

Tokuda, Hideyuki and Nakajima, Tatsuo, "Evaluation of Real-Time Synchronization in Real-Time Mach", in *Proceedings of USENIX 2nd Mach Symposium*, October 1991.

Nakajima, T., and Tokuda, H., "Implementation of Scheduling Policies in Real-Time Mach", *Proceedings of IWOOS*, September 24-25, 1992.

Savage, S. and Tokuda, H., "RT-Mach Timers: Exporting Time to the User", *Proceedings of 3rd USENIX Mach Symposium*, April 1993.

Nakajima, T., Kitayama, T. and Tokuda H., "Experiments with Real-Time Servers in Real-Time Mach", *Proceedings of 3rd USENIX Mach Symposium*, April 1993.

In the user reference manual, we assume that the reader is familiar with the C and C++ programming languages, object-oriented programming, and real-time programming. However, if you are not familiar with these terms and concepts, you may wish to consult the following papers and books before you read the manual.

Tokuda, H., and Mercer, C. W., "ARTS: A Distributed Real-Time Kernel", *ACM Operating Systems Review*, Vol. 23, No. 3, July, 1989.

Tokuda, H. and Kotera, M., "A Real-Time Tool Set for the ARTS Kernel", *Proceedings of the 9th Real-Time Systems Symposium*, December 1988.

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd Ed., Prentice-Hall, Englewood Cliffs, NJ, 1988.

S.B. Lippman, *C++ Primer*, Addison-Wesley, Reading, MA 1989.

W. Kim and F. H. Lochovsky, Ed., *Object-Oriented Concepts, Databases, and Applications*, ACM Press, New York, NY 1989.

J. A. Stankovic, Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems, *IEEE Computer*, Vol. 21, No. 10, Oct. 1988.

ACM Operating Systems Review Special Issue on Real-Time Operating Systems, Vol. 23, No. 3, July 1989.

Note that this is the fourth version of the user reference manual for Real-Time Mach, and we are working on future versions of this document.

Chapter 2

Programming Environment

Unlike traditional real-time operating systems, we have attempted to provide a good programming environment for application designers. Because we support the C language, many of the tools from the Unix programming environment may be inherited. For instance, we are able to use `gnu-emacs`, `make`, `rCS`, and `adb` for the development of distributed real-time programs under Real-Time Mach.

In addition to the existing tools, we also have developed and modified the ARTS Real-Time Toolset for analyzing the runtime behavior of distributed Real-Time Mach programs. The goal of the toolset is to incorporate system-wide scheduling analysis which includes communication and synchronization among real-time tasks. It can predict and analyze the schedulability of the application task set and perform runtime monitoring and debugging [?]. Currently, the toolset consists of *Scheduler 1-2-3* [?] and *ARM* [?].

2.1 *Scheduler 1-2-3*

Scheduler 1-2-3 is an X11-window based interactive schedulability analyzer for creating, manipulating, and analyzing real-time task sets. It employs analysis methods ranging from closed form analysis to simulation to determine whether a feasible schedule exists for a given task set and under what conditions deadlines cannot be met. *Scheduler 1-2-3* can be also used as a synthetic workload generator, which can be integrated with the other tool, the advanced real-time monitor (*ARM*) to generate the execution history diagram of the task set under a specified scheduling policy.

The basic functions of *Scheduler 1-2-3* are summarized as follows.

Schedulability Analysis The schedulability is verified for the given hard deadline task sets under one of the pre-programmed scheduling algorithm sets. The algorithm set contains the rate monotonic (RM) [?], the first-in first-out (FIFO), the round robin (RR), the earliest deadline first (DL), the shortest processing time first (SF), the minimum slack time first (MS), and the fixed priority preemptive (FP) algorithms.

Schedulability analysis can be performed by selecting a specific context overhead (μsec) and processor speed which is relative to a 16.67 MHz MC68020 processor. If the task set does not contain any aperiodic tasks and the scheduling policy is RM, closed form analysis can be chosen. Otherwise, analysis is done based on the simulation. For closed form analysis, we perform the utilization bound check, harmonic task check, and critical zone check [?]. In addition to the schedulability analysis, *Scheduler 1-2-3* also verifies whether or not the runtime system behavior can be monitored by *ARM*.

Response Time Analysis for Aperiodic Tasks The performance of a given set of aperiodic tasks with soft deadlines can be analyzed using specific scheduling algorithms, such as Polling, Background, and Deferrable Servers [?]. *Scheduler 1-2-3* will simulate the aperiodic task activities and estimates the minimum, average, and maximum response times of the given aperiodic task set. The aperiodic task set is defined by priority, the mean interarrival time, standard deviation, and distribution.

X11 based User Interface An interactive user interface is provided on an X11 window system with bitmap display for ease of use. As shown in Figure 2.1, there are four major sections. From the top, the first section consists of various action menus and two sliding bars which allow the user to set some parameters. The third and fourth

sections are used to specify timing attributes of the real-time task set such as its priority, worst case execution time, period, etc. The last section is an output window which displays the results of the schedulability analysis.

Synthetic Workload Generator *Scheduler 1-2-3* outputs a workload table. To confirm the schedulability of the given task set on a practical environment, a synthetic task set is generated based on the workload table written in C. By using the workload table and a preprogrammed workload program, a user can run the synthetic workload under the target execution environment.

The implementation of *Scheduler 1-2-3* consists of three major modules: the Analysis Module, the File Interface Module, and the User Interface Module. The current X11 version of *Scheduler 1-2-3* was derived by simply rewriting the User Interface Module from the SunView version.

Scheduler 1-2-3 is a very useful tool to analyze the schedulability of the given task set and to validate a particular implementation of a scheduling policy by comparing the execution history of the simulation results and the actual run. However, the current version is limited to a single target node environment. Extensions are being made to cope with 1) communication scheduling analysis and 2) end-to-end analysis where a task set is defined by end-to-end time constraints in a distributed environment. The current task descriptions also do not yet allow a user to specify any precedence relationship among tasks.

2.2 *ARM*, Advanced Real-time Monitor

ARM is also an X11 window based "visualizer" for the real-time tasks' runtime behavior. It allows us to verify the timing correctness of each real-time task and support real-time debugging and monitoring. *ARM* also analyzes the timing aspects of the time window (specified by using the mouse to define start and end times) by showing the number of periodic and aperiodic tasks, total CPU utilization, the number of met deadlines, the number of missed deadlines, and the number of scheduling events. Since *ARM*'s input is a sequence of event packets, it can depict the event stream either from the actual target system or from *Scheduler 1-2-3*'s simulation output.

The basic functions of *ARM* are summarized as follows.

Visualization *ARM* can visualize the system's scheduling decisions, namely context switches among periodic and aperiodic tasks by means of an execution history diagram. Figure 2.2 shows an example of the history execution diagram and its zooming function. The top six boxes indicate the action menus. The top half portion of the tasks indicated by bars 1 through 8 corresponds to the periodic activities defined in the target task set and the bottom half corresponds to the aperiodic activities. Character 'R' shown in the execution history diagram indicates that a periodic task becomes runnable and 'E' indicates that it terminates with its deadline being met; otherwise, 'A' or 'C' indicate the task is aborted or canceled due to a missed deadline. The bottom window shows various statistical information.

Event Dump and Analysis After *ARM* has completed its event recording, it can also play back and generate a sequence of scheduling events and information on the timing aspects of the real-time tasks.

The implementation of *ARM* is divided into three functional units; the Event Tap, the Reporter, and the Visualizer module. The Event Tap is a probe embedded inside the target operating system to pick up the raw data on scheduling events. The Reporter is in charge of sending the raw data to the Visualizer on a remote host which analyzes and visualizes the events.

ARM is a very useful monitor which allows us to debug the real-time scheduler as well as application tasks. Since the Visualizer is independent from the target system, *ARM* can be ported easily into different real-time operating system environments. The current limitation is that we cannot monitor the real-time task's activity at an arbitrary level of abstraction. This requires complex monitorability analysis if *Scheduler 1-2-3* must estimate the worst case interference caused by the monitoring activities.

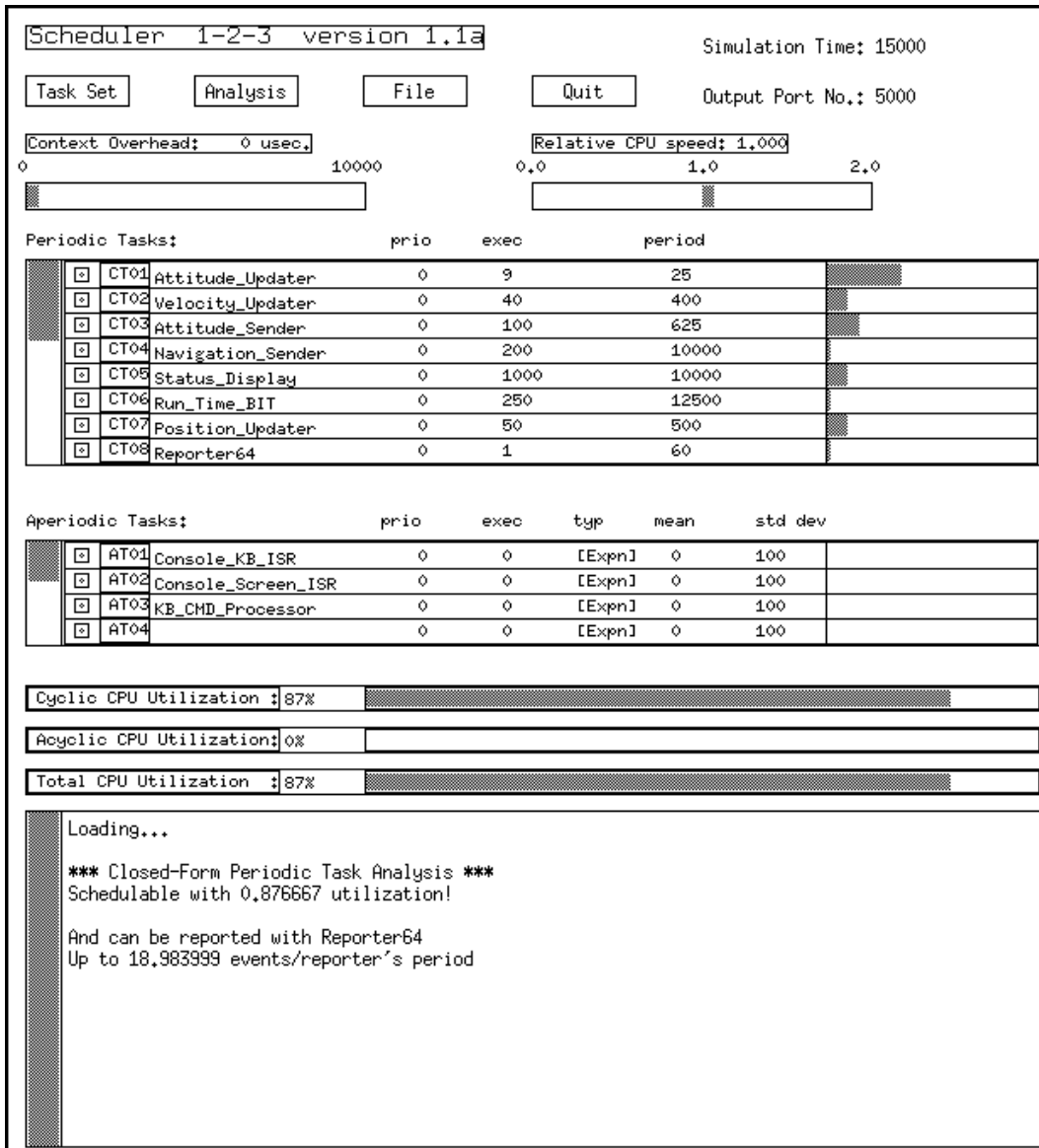


Figure 2.1: Scheduler 1-2-3

Schedulability Analysis by Scheduler 1-2-3: The bottom output window indicates that the above task set was schedulable under the given condition with about 87% of CPU utilization. If the given task set is not schedulable, then it will indicate when one of tasks will miss its deadline.

This figure shows an example of the history execution diagram. The top six boxes indicate the action menus. The top half of threads correspond to the periodic threads and the bottom half correspond to the aperiodic threads. Character 'R' shown in the execution history diagram indicates a periodic thread which becomes runnable and 'E' indicates that it terminates with its deadline being met; 'A' or 'C' indicate that the thread is aborted or canceled due to a missed deadline. 'B' indicates that the thread is blocked waiting for some event, or for preemption. The bottom window shows the various statistical information.

Figure 2.2: *ARM*

Chapter 3

Creating and Executing a Real-Time Program

This section describes how to create an executable image file from source files and link it with the Real-Time Mach system call library to create an executable program.

RT-Mach provides three environments for running applications (Figure 3.1). The first environment is Unix environment. Applications can use Unix primitives such as files and sockets. The second environment is the RTS environment, which provides a simple file system, process management, and tty interface. The environment is suitable for developing embedded applications. The last environment is a standalone environment. The application programmers who use the environment may write their own file systems, and process management.

3.1 UX Environment

A source file must be written in C. This source file must be compiled by using the native Unix C compiler which produces a ".o" file from the code. Then, this .o file is linked with the Real-Time Mach library called `libmach_rt.a`. For example, using a hypothetical example of a Real-Time Mach source program called `example.c`, we create a Real-Time Mach executable program as follows:

```
cc -o example example.c -lmach_rt
```

This example program may be executed simply by typing its name at the Unix shell prompt as follows:

```
example
```

Several libraries may be used by programmers to create advanced applications in Unix Environment. One such library is the display (DS) library. When programmers use this library, for instance, `libds.a` should be linked their applications.

```
cc -o example example.c -lds -lmach_rt
```

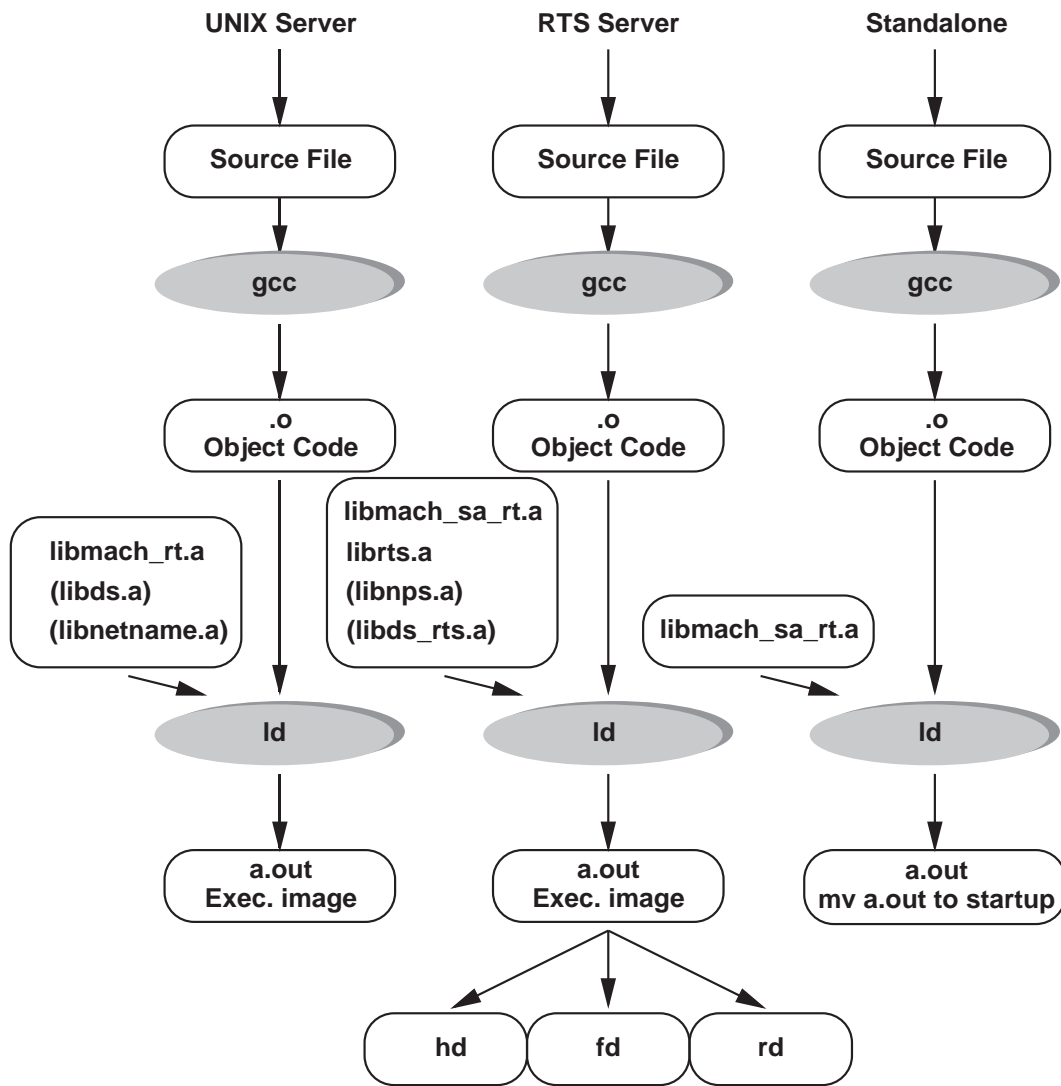
Another such second library is the name server library. Name servers must be used when programmers create servers in the Unix environment. Before using name services, `snames` should be run. `libnetname.a` should be linked when applications which access the name server. The library includes the following functions for registering and importing ports.

```
netname_look_up(name_server_port, "*", "server_name", &serv_port);  
netname_check_in(name_server_port, "server_name", MACH_PORT_NULL, serv_port);
```

```
cc -o example example.c -lnetname -lmach_rt
```

3.2 RTS Environment

When programmers create programs which run on RTS environment, `libmach_rt.a` and `librts.a` should be linked in the following way.



././mach_servers/startup must be rts

Figure 3.1: *RT-Mach Programming Environment*

```
cc -o example example.c -lrt -lmach_rt_sa
```

The display library(DS) can be used by linking `libds_rts.a` in RTS environment. NPS can be used by linking `libnps.a`.

When the programs running in Unix environment is ported in RTS environment, programmers need to pay attention to the following two points.

- Unix primitives should be removed or replaced to RTS primitives.
- `rts_init()` should be called before calling other RTS primitives.
- `libmach_rt_sa.a` should be linked instead of `libmach_rt.a`.

3.3 Standalone Environment

If the example program is to be run under the stand alone environment, then it should be linked with the “stand-alone” version of the Real-Time Mach library called `libmach_rt_sa.a`.

```
cc -o example example.c -lmach_rt_sa
```

Chapter 4

Realtime Scheduling and Thread Creation

4.1 Retrieving the current scheduling policy

Program Overview

This program retrieves and outputs the current scheduling policy. `get_scheduling_policy()` does the actual work.

Program Components

lines 14-44 Retrieve the current scheduling policy

lines 26-28 Get the default processor set

lines 31-33 Get the control port for this default processor set

lines 36-40 Using this control port, get the current scheduling policy for the default processor set

Program Text

```
1 /*
2  * getpolicy.c
3  */
4 /*
5  * RTMach example program
6  * get the current scheduling policy
7  */
8
9 #include <stdio.h>
10 #include <mach.h>
11 #include <rt/pset_attribute.h>
12 #include <rt/sched_policy.h>
13
14 /*
15  * get the scheduling policy for the default
16  * processor set, store it where policy points
17  */
18 kern_return_t get_scheduling_policy(int *policy)
19 {
20     struct pset_sched_policy_attr pset_attr;
21     processor_set_name_t psetname, pset;
22     unsigned int count,i;
```

```

23     kern_return_t ret;
24
25     /* get the default processor set */
26     ret = processor_set_default(mach_host_self(), &psetname);
27     if (ret != KERN_SUCCESS)
28         return(ret);
29
30     /* get the control port for this processor set */
31     ret = host_processor_set_priv(mach_host_priv_self(), psetname, &pset);
32     if (ret != KERN_SUCCESS)
33         return(ret);
34
35     /* get the scheduling policy for this processor set */
36     count = PSET_SCHED_POLICY_ATTR_COUNT;
37     ret = processor_set_get_attribute(pset,
38                                     PSET_SCHED_POLICY_ATTR,
39                                     (processor_set_attr_t)&pset_attr,
40                                     &count);
41
42     *policy = pset_attr.sched_policy;
43     return(ret);
44 }
45
46 main(int argc, char **argv)
47 {
48     kern_return_t ret;
49     int policy;
50
51     /* get the scheduling policy */
52     ret = get_scheduling_policy(&policy);
53     if (ret != KERN_SUCCESS) {
54         printf("ERROR: got %s getting the scheduling policy\n",
55              mach_error_string(ret));
56         exit(-3);
57     }
58
59     printf("The scheduling policy is: ");
60     switch(policy) {
61         case SCHED_POLICY_MKTIMESHARE:
62             printf("mach timesharing");
63             break;
64         case SCHED_POLICY_FIXEDPRI_RR:
65             printf("round-robin fixed priority");
66             break;
67         case SCHED_POLICY_FIXEDPRI_FIFO:
68             printf("fifo fixed priority");
69             break;
70         case SCHED_POLICY_RATE_MONOTONIC:
71             printf("rate monotonic");
72             break;
73         case SCHED_POLICY_DEADLINE_MONOTONIC:
74             printf("deadline monotonic");
75             break;
76         case SCHED_POLICY_EARLIEST_DEADLINE_FIRST:
77             printf("earliest deadline first");
78             break;
79         case SCHED_POLICY_RESERVES:
80             printf("reserves");

```

```

81             break;
82         default:
83             printf("unknown");
84     }
85
86     printf( ".\n" );
87     exit(0);
88 }

```

4.2 Creating and Running Realtime Threads

4.2.1 Creating Aperiodic Threads

Program Overview

This program creates and runs three aperiodic threads with fixed priority preemptive scheduling policy. The threads are created and launched in `rt_simple_rtthread_fork()`. Threads will run immediately upon creation if their priority is higher than the creating thread. (In this program, the creating thread has the default priority of 12.)

Program Components

line 13 initialize `thread_count` to 3

line 44 set the scheduling policy on the default processor set (we request FIFO fixed priority scheduling)

lines 27-37 The function `athread()` is the code executed by each of the three additional threads we create. Each thread prints its identification number (which is passed to it as an argument, and comes from the call to `rt_simple_rtthread_fork()` which created that thread). `thread_count` was initialized to 3 earlier; each thread decrements it. When a thread is done, it does a `thread_terminate(mach_thread_self())` to destroy itself, unless it is the last thread running in the task, in which case it calls `exit()` to also clean up its parent task.

line 52 create and launch a fixed priority thread (#1) with priority 13

line 53 create and launch a fixed priority thread (#2) with priority 5

line 54 create and launch a fixed priority thread (#3) with priority 12

line 56 terminate the main thread

Program Text

```

1 /*
2  * fixedpri_thread.c
3  */
4 /*
5  * RTMach example program
6  * create fixed-priority real-time threads
7  */
8
9 #include <stdio.h>
10 #include <mach.h>
11 #include <rt/sched_policy.h>
12
13 int thread_count=3;
14

```

```

15 /*
16  * a thread can end its execution by saying
17  * END_THREAD()
18  * the last thread to finish running should
19  * call exit() instead, so it can also clean
20  * up the parent task
21  */
22 #define END_THREAD() thread_terminate(mach_thread_self())
23
24 /*
25  * a generic thread
26  */
27 void athread(int *arg)
28 {
29     int threadnum = (int)arg;
30
31     printf("athread running, num=%d\n", threadnum);
32     thread_count--;
33     if (thread_count)
34         END_THREAD();
35     else
36         exit(0);
37 }
38
39 main(int argc, char **argv)
40 {
41     kern_return_t ret;
42
43     /* set the scheduling policy to fixed priority */
44     ret = rt_set_scheduling_policy(SCHED_POLICY_FIXEDPRI_FIFO);
45     if (ret != KERN_SUCCESS) {
46         printf("ERROR: got %s setting the scheduling policy\n",
47             mach_error_string(ret));
48         exit(-3);
49     }
50
51     /* run some threads */
52     rt_simple_thread_fork(athread, (int *)1, 13);
53     rt_simple_thread_fork(athread, (int *)2, 5);
54     rt_simple_thread_fork(athread, (int *)3, 12);
55
56     END_THREAD();
57 }

```

4.2.2 Using rate-monotonic scheduling

Program Overview

This program creates a periodic thread which initially runs with a period of .2 seconds. Each time it runs, it increases its workload. The program sets a deadline handler (`deadlinefunc()`) for this thread. When the thread can no longer complete its workload within its period, this deadline handler will be notified, and the periodic thread will be suspended. The handler will then adjust the period of the thread (it increases it by .08 seconds) and restart it.

Program Components

line 16 declare a handle on a thread; the deadline handler will need this

line 18 declare a port that will be used as our deadline port

lines 26-48 create our rate monotonic thread, with the given period, store the deadline port, and return a handle on the created thread

lines 50-58 `rtthread()` is the function that is run by our rate monotonic thread. Each time, it does a little more work. Eventually, it will try to do too much and will miss its deadline (that is, it won't be able to do all its work before the next time that it's supposed to run).

lines 65-113 `deadlinefun()` is our deadline handler

lines 80-87 retrieve the current period of the rate monotonic thread

lines 89-104 increase this period by .08 seconds

line 112 resume the rate monotonic thread, which was automatically suspended when it missed a deadline

lines 145-155 create a deadline handling thread which will call `deadlinefun()` when thread misses its deadline

Program Text

```
1 /*
2  * ratemono_thread.c
3  */
4 /*
5  * RTMach example program
6  * create a rate-monotonic realtime thread
7  */
8
9 #include <stdio.h>
10 #include <mach.h>
11 #include <rt/sched_policy.h>
12 #include <rt/rt_thread.h>
13 #include <rt/thread_attribute.h>
14
15 int iterations=0;
16 thread_t thread;
17 rt_thread_attr_data_t thread_attr;
18 mach_port_t deadline_port=MACH_PORT_NULL;
19
20 /*
21  * take a function, an argument for it (int *),
22  * and a rate, and runs that function with that
23  * argument in a rate monotonic thread with the
24  * given rate
25  */
26 thread_t ratemono_rtthread_fork(rt_thread_attr_data_t *thread_attr,
27                               void (*func)(), int *arg,
28                               int period_secs, int period_nsecs,
29                               mach_port_t *portp)
30 {
31     thread_t new_thread;
32     kern_return_t ret;
33
34     ret = rt_thread_attribute_init(
35         12, /* priority */
36         period_secs, /* period and deadline secs */
37         period_nsecs, /* period and deadline nsecs */
38         func, /* entry point */
```

```

39             arg,          /* argument */
40             portp,       /* port */
41             thread_attr);
42     if (ret != KERN_SUCCESS)
43         return(ret);
44
45     /* create and run the thread */
46     ret = rt_thread_create(mach_task_self(), &new_thread, thread_attr);
47     return(new_thread);
48 }
49
50 void rthread(int *arg)
51 {
52     int x, i;
53
54     iterations += 75000;
55     printf("thread counting to %d\n", iterations);
56     for(x=0;x<iterations;x++); /* busy loop */
57     printf("thread counted to %d\n", iterations);
58 }
59
60 /*
61  * called when the thread misses a deadline
62  * see the call to rt_thread_deadline_handler
63  * too see where call is generated
64  */
65 void deadlinefunc(timespec_t time, thread_t thread, int *arg)
66 {
67     static int deadlines_missed=0;
68     rt_thread_attr_data_t attr;
69     unsigned int count;
70     kern_return_t ret;
71
72     printf("thread missed deadline %d times\n", ++deadlines_missed);
73     printf("time = %d %d\n", time.seconds, time.nanoseconds);
74
75     /*
76      * adjust the deadline and period of the thread to give it more time
77      * next time
78      */
79
80     /* get current attribute */
81     count = THREAD_SCHED_BASIC_ATTR_COUNT;
82     ret = thread_get_attribute(thread, THREAD_SCHED_BASIC_ATTR, &attr, &count);
83     if (ret != KERN_SUCCESS) {
84         printf("ERROR: got %s from thread_get_attribute\n",
85             mach_error_string(ret));
86         exit(-29);
87     }
88
89     /* adjust period & deadline */
90     attr.deadline.nanoseconds += 80000000;
91     if (attr.deadline.nanoseconds >= NANOSEC_PER_SEC) {
92         attr.deadline.nanoseconds -= NANOSEC_PER_SEC;
93         attr.deadline.seconds++;
94     }
95     attr.period = attr.deadline;
96

```

```

97     /* set attributes */
98     ret = thread_set_attribute(thread, THREAD_SCHED_BASIC_ATTR,
99                               &attr, THREAD_SCHED_BASIC_ATTR_COUNT);
100    if (ret != KERN_SUCCESS) {
101        printf("ERROR: got %s from thread_get_attribute\n",
102              mach_error_string(ret));
103        exit(-29);
104    }
105
106
107    /*
108     * the thread is no longer scheduled,
109     * so we must explicitly say it's OK to
110     * run it
111     */
112    thread_resume((mach_port_t)thread);
113 }
114
115 main(int argc, char **argv)
116 {
117     kern_return_t ret;
118     int x;
119
120     /* set the scheduling policy to rate monotonic */
121     ret = rt_set_scheduling_policy(SCHED_POLICY_RATE_MONOTONIC);
122     if (ret != KERN_SUCCESS) {
123         printf("ERROR: got %s setting the scheduling policy\n",
124               mach_error_string(ret));
125         exit(-3);
126     }
127
128     /*
129     * run a rate-monotonic thread
130     */
131
132     /* thread runs every .2 seconds */
133     thread = ratemono_rtthread_fork(
134         &thread_attr,
135         rthread, (int *)1,
136         0, 200000000,
137         &deadline_port);
138
139     /*
140     * rt_thread_deadline_handler blocks waiting for deadline
141     * messages for the given thread; when they come in, it will
142     * call the specified function as:
143     *   func(timespec_t *time, thread_t thread, arg)
144     */
145     ret = rt_thread_deadline_handler(
146         &thread, /* thread */
147         &thread_attr, /* thread attributes */
148         deadlinefunc, /* function */
149         0 /* arg */
150     );
151     if (ret != KERN_SUCCESS) {
152         printf("Got %s from rt_thread_deadline_handler\n",
153               mach_error_string(ret));
154         exit(-23);

```

```
155     }
156
157     sleep(3600);
158     exit(0);
159 }
```

Chapter 5

Clock and Timer Management

5.1 Realtime Clocks and Timers

A clock is an abstraction for a timing mechanism. A timer is a mechanism for reporting when a particular clock reaches a particular time. Depending on the configuration of your system, you may have many clocks configured. Clocks are created and configured at boot time, and are persistent throughout system execution. Timers are dynamically created and destroyed throughout. There are two important clocks to be aware of: the *realtime* clock and the *update* clock. The realtime clock is the highest-precision clock that will support the use of timers. If the realtime clock is not periodic, the update clock is the highest-precision periodic clock. The resolution of the update clock is the resolution at which the memory-mapped time for the realtime clock will be updated. If the realtime clock is periodic, then the realtime clock and the update clock are one and the same.

5.2 Using a Realtime Clock

5.2.1 Reading the clock

Program Overview

This program demonstrates how to read the realtime clock using the `clock_gettime()` primitive. This primitive will force the clock to return its time at its current precision. This incurs substantially more overhead than memory-mapping the clock's time, but is potentially much more accurate (see above).

Program Components

lines 22-26 `clock_get_port()` returns a port that represents the realtime system clock.

lines 28-33,35-40 `clock_gettime()` reads the current time off the realtime system clock.

line 45 `timespec_sub()` subtracts `t1` from `diff` and stores the result in `diff`. This will ensure that `diff` is a valid `timespec_t`, but assumes that `t2` is less than `t1`.

Program Text

```
1 /*
2  * gettimeofday.c
3  */
4 /*
5  * RTMach example program
6  * read the real-time clock
7  */
```

```

8
9 #include <stdio.h>
10 #include <mach.h>
11 #include <rt/mach_clock.h>
12 #include <rt/timespec.h>
13
14 main(int argc, char **argv)
15 {
16     timespec_t t1, t2, diff;
17     mach_clock_t clock_port;
18     kern_return_t ret;
19     int x;
20
21     /* get a port on the default clock */
22     ret = clock_get_port(mach_host_self(), &clock_port);
23     if (ret != KERN_SUCCESS) {
24         printf("ERROR: could not get clock port.\n");
25         exit(-3);
26     }
27
28     ret = clock_get_time(clock_port, &t1);
29     if (ret != KERN_SUCCESS) {
30         printf("ERROR: got %s from clock_get_time\n",
31             mach_error_string(ret));
32         exit(-5);
33     }
34     for(x=0;x<120000;x++);
35     clock_get_time(clock_port, &t2);
36     if (ret != KERN_SUCCESS) {
37         printf("ERROR: got %s from clock_get_time\n",
38             mach_error_string(ret));
39         exit(-7);
40     }
41
42     printf("t1 = %d seconds, %d nanoseconds\n", t1.seconds, t1.nanoseconds);
43     printf("t2 = %d seconds, %d nanoseconds\n", t2.seconds, t2.nanoseconds);
44     diff = t2;
45     timespec_sub(diff, t1);
46     printf("diff = %d seconds, %d nanoseconds\n", diff.seconds, diff.nanoseconds);
47
48     exit(0);
49 }

```

5.2.2 Memory-mapping the clock

Program Overview

Reading the clock in the manner illustrated above invokes a certain overhead, namely, that of the `clock_get_time()` primitive. In this example, the `clock_map()` library call is used to memory map the realtime clock, so that reading its current value does not invoke this extra overhead.

Program Components

line 16 Declare a pointer to a `timespec_t` which will become our pointer to the memory-mapped time from the realtime clock. This *must* be declared as a `volatile` pointer to avoid having the compiler incorrectly optimize accesses to this time.

lines 22-26 `clock_get_port()` returns a port that represents the realtime system clock.

lines 34-39 Memory-map the realtime clock. At this point, figure references to mapped_clock will be to the current time; mapped_clock acts as a pointer to the realtime clock.

Program Text

```
1 /*
2  * maptime.c
3  */
4 /*
5  * RTMach example program
6  * memory-map and read the realtime clock
7  */
8
9 #include <stdio.h>
10 #include <mach.h>
11 #include <rt/mach_clock.h>
12 #include <rt/timespec.h>
13
14 main(int argc, char **argv)
15 {
16     volatile timespec_t *mapped_clock=NULL;
17     mach_clock_t clock;
18     kern_return_t ret;
19     int x;
20
21     /* get a port on the default clock */
22     ret = clock_get_port(mach_host_self(), &clock);
23     if (ret != KERN_SUCCESS) {
24         printf("ERROR: could not get clock port.\n");
25         exit(-3);
26     }
27
28     /*
29     * memory-map the realtime clock
30     * now, references to the contents of mapped_clock
31     * will give you the time the realtime clock reads
32     * without the overhead of a kernel trap
33     */
34     ret = clock_map(clock, &mapped_clock);
35     if (ret != KERN_SUCCESS) {
36         printf("ERROR: got %s from clock_map\n",
37             mach_error_string(ret));
38         exit(-5);
39     }
40
41     printf("t1 = %d seconds, %d nanoseconds\n", mapped_clock->seconds,
42         mapped_clock->nanoseconds);
43     for(x=0;x<120000;x++);
44     printf("t2 = %d seconds, %d nanoseconds\n", mapped_clock->seconds,
45         mapped_clock->nanoseconds);
46
47     exit(0);
48 }
```

5.2.3 Clock Resolution

Getting the clock resolution

Program Overview

This program illustrates how to retrieve the resolution of a clock. In this case, we retrieve the resolution of the update clock, which is the resolution at which the memory-mapped time of the realtime clock will be updated. The resolution of a particular clock also sets the limit on the accuracy of that clock's timers.

Program Components

lines 20-25 `clock_update_get_port()` returns a port that represents the update clock.

lines 46-54 Retrieve the current clock resolution and store it in `resolution`. This resolution is in nanoseconds.

Program Text

```
1 /*
2  * getres.c
3  */
4 /*
5  * RTMach example program
6  * get the update clock's resolution
7  */
8
9 #include <stdio.h>
10 #include <mach.h>
11 #include <rt/mach_clock.h>
12 #include <rt/timespec.h>
13
14 main(int argc, char **argv)
15 {
16     unsigned long resolution, skew;
17     mach_clock_t update_clock;
18     kern_return_t ret;
19
20     /* get a port on the update clock */
21     ret = clock_update_get_port(mach_host_self(), &update_clock);
22     if (ret != KERN_SUCCESS) {
23         printf("ERROR: could not get update clock port.\n");
24         exit(-3);
25     }
26
27     /* read the resolution status of the clock device */
28     ret = clock_get_resolution(update_clock, &resolution, &skew);
29     if (ret != KERN_SUCCESS) {
30         printf("ERROR: got %s getting clock resolution\n",
31               mach_error_string(ret));
32         exit(-5);
33     }
34
35     printf("resolution=%d\n", resolution);
36     exit(0);
37 }
```

Setting the clock resolution

Program Overview

This program illustrates how to set a clock's resolution. The `clock_set_resolution()` primitive takes two parameters: a desired resolution (in nanoseconds), and an acceptable skew (again, in nanoseconds). Since most hardware clocks are restricted to multiples of particular values for their resolution, arbitrary desired resolutions may not be available. The skew specifies an acceptable amount of rounding (in either direction) from the requested resolution that `clock_set_resolution` may perform. If the desired resolution is not achievable with an acceptable (\neq skew) error margin, then `KERN_INVALID_VALUE` is returned.

Program Components

lines 20-25 `clock_update_get_port()` returns a port that represents the update clock.

lines 48-53 Set the desired clock resolution and maximum allowable skew. The new resolution and its skew from the requested resolution are returned in `resolution` and `skew` by `clock_set_resolution()`.

Program Text

```
1 /*
2  * setres.c
3  */
4 /*
5  * RTMach example program
6  * set the real-time clock's resolution
7  */
8
9 #include <stdio.h>
10 #include <mach.h>
11 #include <rt/mach_clock.h>
12 #include <rt/timespec.h>
13
14 main(int argc, char **argv)
15 {
16     unsigned long resolution, skew;
17     mach_clock_t update_clock;
18     kern_return_t ret;
19
20     /* get a port on the default clock */
21     ret = clock_update_get_port(mach_host_self(), &update_clock);
22     if (ret != KERN_SUCCESS) {
23         printf("ERROR: could not get update clock.\n");
24         exit(-3);
25     }
26
27     /*
28      * how many nanoseconds should elapse between
29      * system clock ticks
30      * the resolution is the maximum granularity
31      * you can get with timers
32      *
33      * in this case, we're setting it to half a millisecond
34      */
35     resolution = 500000;
36     /*
37      * when we ask for a clock resolution, we're actually
```

```

38     * saying to our timer hardware how often we want
39     * timer interrupts
40     * because timer hardware often has bizarre linear
41     * or exponential scaling, we specify a skew, which
42     * is how far off we'll allow the actual resolution to
43     * get from what we set above
44     */
45     skew = 250000;
46
47     /* actually set in that resolution */
48     ret = clock_set_resolution(update_clock, &resolution, &skew);
49     if (ret != KERN_SUCCESS) {
50         printf("ERROR: got %s setting clock resolution\n",
51             mach_error_string(ret));
52         exit(-5);
53     }
54
55     printf("We got %d with skew %d.\n", resolution, skew);
56
57     exit(0);
58 }

```

5.2.4 Identifying clocks

Program Overview

This program identifies the realtime clock and the update clock (if it is different from the realtime clock). It uses the `clock_get_name` primitive to retrieve the names of these clocks. Next, it uses the `host_clocks` primitive to obtain an array of ports representing all the clocks in the system, and shows their names and their current times.

Program Components

lines 15-38 `show_clock()` prints the name and current time of a clock.

lines 22-26 `clock_get_name()` retrieves the name of a realtime clock.

lines 28-32 `clock_get_time()` retrieves the current time of a realtime clock.

lines 49-54 `clock_get_port()` returns a port that represents the update clock.

lines 56-61 `clock_update_get_port()` returns a port that represents the update clock.

lines 85-89 `host_clocks` returns an array of ports representing all the clocks on a particular host in `clocks`, and the number of items in this array in `count`. The memory pointed to by `clocks` is automatically allocated by `host_clocks()`. To destroy this array, `mach_port_destroy()` should be called on each port within it, and the memory it occupies should be released using `vm_deallocate()`.

Program Text

```

1  /*
2  * listclocks.c
3  */
4  /*
5  * RTMach example program
6  * show the names of all the clocks
7  * currently available
8  */
9

```

```

10 #include <stdio.h>
11 #include <mach.h>
12 #include <rt/mach_clock.h>
13 #include <rt/timespec.h>
14
15 void show_clock(mach_clock_t clock)
16 {
17     mach_clock_name_t clock_name;
18     kern_return_t ret;
19     timespec_t ts;
20     int i;
21
22     ret = clock_get_name(clock, clock_name);
23     if (ret != KERN_SUCCESS) {
24         printf("failed getting clock name, %s\n", mach_error_string(ret));
25         return;
26     }
27
28     ret = clock_get_time(clock, &ts);
29     if (ret != KERN_SUCCESS) {
30         printf("failed getting time on clock %s, %s\n", clock_name, mach_error_string(ret));
31         return;
32     }
33
34     printf("%s", clock_name);
35     for(i=strlen(clock_name);i<=16;i++)
36         printf(" ");
37     printf("%d:%09d\n", ts.seconds, ts.nanoseconds);
38 }
39
40 main(int argc, char **argv)
41 {
42     mach_clock_name_t clock_name, update_clock_name;
43     mach_clock_t clock, update_clock;
44     mach_clock_array_t clocks;
45     unsigned int count;
46     kern_return_t ret;
47     int i;
48
49     ret = clock_get_port(mach_host_self(), &clock);
50     if (ret != KERN_SUCCESS) {
51         printf("ERROR: could not get clock port (%s).\n",
52             mach_error_string(ret));
53         exit(-3);
54     }
55
56     ret = clock_update_get_port(mach_host_self(), &update_clock);
57     if (ret != KERN_SUCCESS) {
58         printf("ERROR: could not get update clock port (%s).\n",
59             mach_error_string(ret));
60         exit(-5);
61     }
62
63     ret = clock_get_name(clock, clock_name);
64     if (ret != KERN_SUCCESS) {
65         printf("ERROR: could not get clock name (%s).\n",
66             mach_error_string(ret));
67         exit(-7);

```

```

68     }
69
70     ret = clock_get_name(update_clock, update_clock_name);
71     if (ret != KERN_SUCCESS) {
72         printf("ERROR: could not get update clock name (%s).\n",
73             mach_error_string(ret));
74         exit(-11);
75     }
76
77     printf("The realtime clock is %s.\n", clock_name);
78     if (strcmp(clock_name, update_clock_name)) {
79         printf("The realtime clock's updates are driven by %s.\n",
80             update_clock_name);
81     }
82
83     printf("\n");
84
85     ret = host_clocks(mach_host_self(), &clocks, &count);
86     if (ret != KERN_SUCCESS) {
87         printf("failed getting host clock list, %s\n", mach_error_string(ret));
88         exit(-1);
89     }
90
91     for(i=0;i<count;i++) {
92         show_clock(clocks[i]);
93     }
94 }

```

5.3 Alarmclock Timer

5.3.1 Sleeping on a timer

Program Overview

This program creates a timer, and sleeps on it twice. The first time, it uses the timer in “relative” mode. In this mode, it specifies how long the calling thread should be suspended before the alarm goes off and awakens the thread (in this case, 1.5 seconds). In the second example, the timer is set in “absolute” mode. In this mode, we specify a time at which the alarm should go off. When the realtime clock reaches this time, the alarm will go off.

Program Components

lines 25-30 `clock_get_port()` returns a port that represents the realtime system clock.

lines 33-38 Memory-map the realtime clock.

lines 41-46 Create a timer running off the realtime clock.

line 61 Use this timer to sleep for a preset interval (in this case, 1.5 seconds, as set in lines 51-52).

line 82 Use the timer to sleep until the system clock reaches a preset time (set at lines 68-69).

Program Text

```

1 /*
2  * sleepclock.c
3  */
4 /*

```

```

5  * RTMach example program
6  * sleep on a realtime clock, waiting for it to expire
7  */
8
9  #include <stdio.h>
10 #include <mach.h>
11 #include <rt/mach_clock.h>
12 #include <rt/timespec.h>
13 #include <rt/mach_timer.h>
14
15 main(int argc, char **argv)
16 {
17     volatile timespec_t *mapped_clock=NULL;
18     timespec_t sleep_time;
19     mach_clock_t clock;
20     mach_timer_t timer;
21     kern_return_t ret;
22     int x;
23
24     /* get a port on the default clock */
25     ret = clock_get_port(mach_host_self(), &clock);
26     if (ret != KERN_SUCCESS) {
27         printf("ERROR: could not get clock port (%s).\n",
28             mach_error_string(ret));
29         exit(-3);
30     }
31
32     /* memory-map the realtime clock */
33     ret = clock_map(clock, &mapped_clock);
34     if (ret != KERN_SUCCESS) {
35         printf("ERROR: got %s from clock_map\n",
36             mach_error_string(ret));
37         exit(-5);
38     }
39
40     /* create a timer */
41     ret = timer_create(current_task(), &timer, clock);
42     if (ret != KERN_SUCCESS) {
43         printf("ERROR: got %s creating timer\n",
44             mach_error_string(ret));
45         exit(-7);
46     }
47
48     /*
49     * sleep for 1.5 seconds
50     */
51     sleep_time.seconds = 1;
52     sleep_time.nanoseconds = 500000000;
53
54     printf("going to sleep, time = %d seconds %d nanoseconds\n",
55         mapped_clock->seconds, mapped_clock->nanoseconds);
56     /*
57     * actually go to sleep
58     * the 0 indicates that this is a relative sleep
59     * that is, sleep until current_time+sleep_time
60     */
61     timer_sleep(timer, sleep_time, 0);
62     printf("awake,         time = %d seconds %d nanoseconds\n",

```

```

63         mapped_clock->seconds, mapped_clock->nanoseconds);
64
65     /*
66     * sleep until 4 seconds after current time
67     */
68     sleep_time.seconds = mapped_clock->seconds + 4;
69     sleep_time.nanoseconds = mapped_clock->nanoseconds;
70
71     printf("time1 = %d seconds %d nanoseconds\n",
72           mapped_clock->seconds, mapped_clock->nanoseconds);
73     for(x=0;x<5555555;x++);
74     printf("time2 = %d seconds %d nanoseconds\n",
75           mapped_clock->seconds, mapped_clock->nanoseconds);
76
77     /*
78     * sleep again
79     * the TIMER_ABSOLUTE indicates that the time is absolute;
80     * that is, sleep until current_time=sleep_time
81     */
82     timer_sleep(timer, sleep_time, TIMER_ABSOLUTE);
83
84     printf("time3 = %d seconds %d nanoseconds\n",
85           mapped_clock->seconds, mapped_clock->nanoseconds);
86
87     exit(0);
88 }

```

Chapter 6

Processor Reservations

6.1 Creating and Destroying Reservations

Program Overview

This program demonstrates how to create and terminate reserves, bind them to threads, and determine what reserve is bound to a particular thread. When run, the program creates a reserve, sets its name to “Example,” and requests that it have a reserve of 10 milliseconds out of every 50 milliseconds. Next it shows the reserve that it is currently running against, binds itself to run against this new reserve, and again shows the name of the reserve that it’s running against. Finally, it destroys the reserve, and one last time, shows the name of the reserve that it’s running against (demonstrating that a thread will be re-bound to the default reserve automatically when its reserve is terminated).

Program Components

lines 14-42 show the name of the reserve the given thread is bound to

lines 24-29 get the reserve the given thread is bound to

lines 34-39 retrieve the name of this reserve

lines 53-59 set the scheduling policy to `SCHED_POLICY_RESERVES`

lines 62-67 retrieve the default processor set

lines 72-73 a start time of zero represents an immediate start

lines 78-79 we will request 10 milliseconds of compute time

lines 84-85 we will request a 50 millisecond period

lines 87-92 create a processor reserve. Note that when a reserve is first created, it does not actually have a reservation until we call `reserve_request()`.

lines 97-103 associate a logical name with our new reserve

lines 108-113 actually request a reservation of 10 milliseconds out of every 50 milliseconds starting immediately

line 115 get our thread

line 118 show the name of the reserve our thread is running against

lines 123-128 bind our thread to the newly created reserve

line 131 show the name of the reserve our thread is running against

lines 136-141 terminate (destroy) this new reserve

line 144 show the name of the reserve our thread is running against

Program Text

```
1 /*
2  * RTMach example program
3  * create and bind a thread to a reserve
4  */
5
6 #include <stdio.h>
7 #include <mach.h>
8 #include <rt/mach_reserves.h>
9 #include <rt/sched_policy.h>
10
11 void show_my_reserve(thread)
12     thread_t thread;
13 {
14     mach_reserve_name_t myreserve_name;
15     mach_reserve_t myreserve;
16     kern_return_t ret;
17
18     /*
19      * What reserve are we running against?
20      */
21     ret = thread_get_reserve(thread, &myreserve);
22     if (ret != KERN_SUCCESS) {
23         printf("thread_get_reserve failed with %s\n",
24             mach_error_string(ret));
25         exit(1);
26     }
27
28     /*
29      * Get the name of this reserve
30      */
31     ret = reserve_name(myreserve, myreserve_name);
32     if (ret != KERN_SUCCESS) {
33         printf("reserve_name failed with %s\n",
34             mach_error_string(ret));
35         exit(1);
36     }
37
38     printf("Now running against reserve %s\n", myreserve_name);
39 }
40
41 main()
42 {
43     mach_reserve_name_t reserve_name;
44     timespec_t compute, period, start;
45     processor_set_name_t psetname;
46     mach_reserve_t reserve;
47     mach_reserve_t another_reserve;
48     kern_return_t ret;
49     thread_t thread;
50     int res;
51
52     /* use reservation scheduling */
53     ret = rt_set_scheduling_policy(SCHED_POLICY_HIER_RESERVES_FP);
54     if (ret != KERN_SUCCESS) {
55         printf("ERROR: got %s setting the scheduling policy\n",
```

```

56                                     mach_error_string(ret));
57         exit(-3);
58     }
59
60     /* get the default processor set */
61     ret = processor_set_default(mach_host_self(), &psetname);
62     if (ret != KERN_SUCCESS) {
63         printf("processor_set_default failed with %s\n",
64               mach_error_string(ret));
65         exit(1);
66     }
67
68     /*
69      * set start time to zero (immediate)
70      */
71     start.seconds = 0;
72     start.nanoseconds = 0;
73
74
75     /*
76      * computation time of reserve
77      */
78     compute.seconds = 0;
79     compute.nanoseconds = 10000000; /* 10 ms */
80
81     /*
82      * period of reserve
83      */
84     period.seconds = 0;
85     period.nanoseconds = 50000000; /* 50 ms */
86
87     ret = reserve_create(psetname, &reserve);
88     if (ret != KERN_SUCCESS) {
89         printf("reserve_create failed with %s\n",
90               mach_error_string(ret));
91         exit(1);
92     }
93
94     ret = reserve_create(psetname, &another_reserve);
95     if (ret != KERN_SUCCESS) {
96         printf("reserve_create failed with %s\n",
97               mach_error_string(ret));
98         exit(1);
99     }
100
101     /*
102      * Set a name for our new reserve
103      */
104     /* strcpy(reserve_name, "Example");
105     ret = reserve_set_name(reserve, reserve_name);
106     if (ret != KERN_SUCCESS) {
107         printf("reserve_set_name failed with %s\n",
108               mach_error_string(ret));
109         exit(1);
110     }
111     */
112     /*
113      * Actually make our reservation

```

```

114     */
115     ret = reserve_request(reserve, compute, period, start);
116     if (ret != KERN_SUCCESS) {
117         printf("reserve_request failed with %s\n",
118             mach_error_string(ret));
119         exit(1);
120     }
121
122     thread = mach_thread_self();
123
124     /* show our reserve */
125     show_my_reserve(thread);
126
127     /*
128      * Bind our thread to this new reserve
129      */
130     ret = thread_set_reserve(thread, reserve);
131     if (ret != KERN_SUCCESS) {
132         printf("thread_set_reserve failed with %s\n",
133             mach_error_string(ret));
134         exit(1);
135     }
136
137     /* show our reserve */
138     show_my_reserve(thread);
139
140     /*
141      * Destroy the reserve
142      */
143     ret = reserve_terminate(reserve);
144     if (ret != KERN_SUCCESS) {
145         printf("reserve_terminate failed with %s\n",
146             mach_error_string(ret));
147         exit(1);
148     }
149
150     /* show our reserve */
151     show_my_reserve(thread);
152 }

```

6.2 Reservation status

Program Overview

This program demonstrates how to retrieve the list of reservations made against a processor set, and retrieve vital statistics about these reservations. When run, the program uses `processor_set_reserves()` to obtain the list of reservations associated with the default processor set. `reserve_name()` and `reserve_get_attribute()` are used to retrieve the name, accumulated computation time, reserved computation time, and reservation period of each processor reservation. The accumulated total computation time is the amount of cpu time consumed collectively by all the threads bound to a processor reserve since the creation of that reservation. The computation time and period are the actual processor reservation of each reserve.

Program Components

lines 9-18 use `reserve_get_attribute()` to retrieve a given timing attribute of a reserve

lines 20-59 show the name, accumulated cpu time, reserved computation time, and period of a processor reserve

lines 27-32 retrieve the logical name of this processor reserve

lines 34-39 get the accumulated cpu time of this processor reserve

lines 41-46 get the computation time Ci of this processor reserve

lines 34-39 get the period Ti of this processor reserve

lines 70-75 get the default processor set

lines 78-82 get a list of reserves on this processor set. This list is in the form of an array- the size of the array is returned in count. The memory for this array is allocated by `processor_set_reserves()` and a pointer to it is returned in `reserves`. To properly destroy all the resources allocated by `processor_set_reserves()`, `mach_port_destroy()` should be called on each element of the array at `reserves`, and memory occupied by the array itself should be returned to the system by `vm_deallocate()`.

Program Text

```
1 /*
2  * name_res.c
3  */
4
5 #include <stdio.h>
6 #include <mach.h>
7 #include <rt/mach_reserves.h>
8
9 kern_return_t reserve_read_time(reserve, attr, tp)
10     mach_reserve_t reserve;
11     int attr;
12     timespec_t *tp;
13 {
14     unsigned int count;
15
16     count = RESERVE_TIME_COUNT;
17     return(reserve_get_attribute(reserve, attr, (mach_reserve_attr_t)tp, &count));
18 }
19
20 void show_reserve(res)
21     mach_reserve_t res;
22 {
23     timespec_t total, comptime, period;
24     mach_reserve_name_t name;
25     kern_return_t ret;
26
27     ret = reserve_name(res, name);
28     if (ret != KERN_SUCCESS) {
29         printf("reserve port: %x\n", res);
30         printf("failed reserve_get_name (%s)\n", mach_error_string(ret));
31         exit(1);
32     }
33
34     ret = reserve_read_time(res, RESERVE_ACCUM_TOTAL, &total);
35     if (ret != KERN_SUCCESS) {
36         printf("failed reserve_get_attribute RESERVE_ACCUM_TOTAL (%s)\n",
37             mach_error_string(ret));
38         exit(1);
39     }
40 }
```

```

41     ret = reserve_read_time(res, RESERVE_COMPUTATION, &comptime);
42     if (ret != KERN_SUCCESS) {
43         printf("failed reserve_get_attribute RESERVE_COMPUTATION (%s)\n",
44             mach_error_string(ret));
45         exit(1);
46     }
47
48     ret = reserve_read_time(res, RESERVE_PERIOD, &period);
49     if (ret != KERN_SUCCESS) {
50         printf("failed reserve_get_attribute RESERVE_PERIOD (%s)\n",
51             mach_error_string(ret));
52         exit(1);
53     }
54
55     printf("reserve name: %s\n", name);
56     printf("accumulated total: %u:%09u\n", total.seconds, total.nanoseconds);
57     printf("computation:          %u:%09u\n", comptime.seconds, comptime.nanoseconds);
58     printf("period:                %u:%09u\n", period.seconds, period.nanoseconds);
59 }
60
61 main()
62 {
63     processor_set_name_t psetname;
64     mach_port_t *reserves;
65     unsigned int count;
66     kern_return_t ret;
67     int i;
68
69     /* get the default processor set */
70     ret = processor_set_default(mach_host_self(), &psetname);
71     if (ret != KERN_SUCCESS) {
72         printf("processor_set_default failed with %s\n",
73             mach_error_string(ret));
74         exit(1);
75     }
76
77     /* get a list of existing reserves on this processor set */
78     ret = processor_set_reserves(psetname, &reserves, &count);
79     if (ret != KERN_SUCCESS) {
80         printf("failed getting procset reserves, %s\n", mach_error_string(ret));
81         exit(-1);
82     }
83
84     /* display the relevant info */
85     for(i=0;i<count;i++) {
86         show_reserve((mach_reserve_t)reserves[i]);
87     }
88 }

```

Chapter 7

Hierarchical Reservation Scheduling

This section presents programming samples to use the two-level hierarchical reservation scheme in Real-Time Mach. Please see the report “Hierarchical Reservation in Real-Time Mach” for detailed information regarding this abstraction. Sample code is given below.

7.0.1 Hierarchical Reservation Example 1

```
1 /*
2  * RTMach example program to test hierarchical reserves.
3  */
4 #include <stdio.h>
5 #include <mach.h>
6 #include <rt/mach_reserves.h>
7 #include <rt/sched_policy.h>
8 #include <rt/rt_thread.h>
9 #include <rt/thread_attribute.h>
10 #include <rt/mach_clock.h>
11 #include <rt/timespec.h>
12
13 #define UPPER_LIMIT 80000000
14
15 thread_t thread1;
16 thread_t thread2;
17 thread_t thread3;
18 thread_t thread4;
19 thread_t thread5;
20 thread_t thread6;
21 thread_t thread7;
22 thread_t thread8;
23
24 kern_return_t rt_simple_thread_fork(void (*func)(), int *arg, int pri);
25
26 rt_thread_attr_data_t thread_attr1;
27 rt_thread_attr_data_t thread_attr2;
28 rt_thread_attr_data_t thread_attr3;
29 rt_thread_attr_data_t thread_attr4;
30 rt_thread_attr_data_t thread_attr5;
31 rt_thread_attr_data_t thread_attr6;
32 rt_thread_attr_data_t thread_attr7;
33 rt_thread_attr_data_t thread_attr8;
34
35 mach_port_t deadline_port1 = MACH_PORT_NULL;
```

```

36 mach_port_t deadline_port2 = MACH_PORT_NULL;
37 mach_port_t deadline_port3 = MACH_PORT_NULL;
38 mach_port_t deadline_port4 = MACH_PORT_NULL;
39 mach_port_t deadline_port5 = MACH_PORT_NULL;
40 mach_port_t deadline_port6 = MACH_PORT_NULL;
41 mach_port_t deadline_port7 = MACH_PORT_NULL;
42 mach_port_t deadline_port8 = MACH_PORT_NULL;
43
44 void func1(int *arg)
45 {
46     int index = 0;
47     for(index = 0; index < UPPER_LIMIT; index++)
48         ;
49     printf("T1 done\n");
50     thread_terminate(mach_thread_self());
51 }
52
53
54 void func2(int *arg)
55 {
56     int index = 0;
57     for(index = 0; index < UPPER_LIMIT; index++)
58         ;
59     printf("T2 done\n");
60     thread_terminate(mach_thread_self());
61 }
62
63
64 void func3(int *arg)
65 {
66     int index = 0;
67     for(index = 0; index < UPPER_LIMIT; index++)
68         ;
69     printf("T3 done\n");
70     thread_terminate(mach_thread_self());
71 }
72
73
74 void func4(int *arg)
75 {
76     int index = 0;
77     for(index = 0; index < UPPER_LIMIT; index++)
78         ;
79     printf("T4 done\n");
80     thread_terminate(mach_thread_self());
81 }
82
83 void func5(int *arg)
84 {
85     int index = 0;
86     for(index = 0; index < UPPER_LIMIT; index++)
87         ;
88     printf("T5 done\n");
89     thread_terminate(mach_thread_self());
90 }
91
92 void func6(int *arg)
93 {

```

```

94     int index = 0;
95     for(index = 0; index < UPPER_LIMIT; index++)
96         ;
97     printf("T6 done\n");
98     thread_terminate(mach_thread_self());
99 }
100
101 void func7(int *arg)
102 {
103     int index = 0;
104     for(index = 0; index < UPPER_LIMIT; index++)
105         ;
106     printf("T7 done\n");
107     thread_terminate(mach_thread_self());
108 }
109
110 void func8(int *arg)
111 {
112     int index = 0;
113     for(index = 0; index < UPPER_LIMIT; index++)
114         ;
115     printf("T8 done\n");
116     thread_terminate(mach_thread_self());
117 }
118
119 thread_t rt_periodic_thread_fork(rt_thread_attr_data_t *thread_attr,
120                                void (*func)(), int *arg, int priority,
121                                int period_secs, int period_nsecs,
122                                int deadline_secs, int deadline_nsecs,
123                                timespec_t starttime,
124                                mach_port_t *portp)
125 {
126     thread_t new_thread;
127     kern_return_t ret;
128     ret = rt_thread_attribute_init_edf(
129         priority, /* priority */
130         period_secs,
131         period_nsecs,
132         deadline_secs,
133         deadline_nsecs,
134         func,
135         arg,
136         portp,
137         thread_attr);
138     if (ret != KERN_SUCCESS) {
139         return ret;
140     }
141
142     (thread_attr->start_time).seconds = starttime.seconds;
143     (thread_attr->start_time).nanoseconds = starttime.nanoseconds;
144     thread_attr->abs_flag = TRUE;
145
146     ret = rt_thread_create(mach_task_self(), &new_thread, thread_attr);
147     return new_thread;
148 }
149
150 main()
151 {

```

```

152     mach_reserve_name_t reserve_name1;
153     mach_reserve_name_t reserve_name2;
154     mach_clock_t clock_port;
155     mach_reserve_name_t child_reserve_name1;
156     mach_reserve_name_t child_reserve_name2;
157     mach_reserve_name_t child_reserve_name3;
158     mach_reserve_name_t child_reserve_name4;
159
160     timespec_t compute1, period1, start1,
161             compute2, period2, start2,
162             child_compute1, child_period1,
163             child_compute2, child_period2,
164             child_compute3, child_period3,
165             child_compute4, child_period4,
166             threadstart, t1;
167
168     processor_set_name_t psetname;
169     mach_reserve_t reserve1;
170     mach_reserve_t reserve2;
171     mach_reserve_t child_reserve1;
172     mach_reserve_t child_reserve2;
173     mach_reserve_t child_reserve3;
174     mach_reserve_t child_reserve4;
175
176     kern_return_t ret;
177
178     ret = clock_get_port(mach_host_self(), &clock_port);
179     if (ret != KERN_SUCCESS)
180     {
181         printf("ERROR: got %s from clock_get_port\n",
182             mach_error_string(ret));
183     }
184
185     /* CREATE RESERVE */
186
187     /* use reservation scheduling */
188     ret = rt_set_scheduling_policy(SCHED_POLICY_HIER_RESERVES_FP);
189     if (ret != KERN_SUCCESS) {
190         printf("ERROR: got %s setting the scheduling policy\n",
191             mach_error_string(ret));
192         exit(-3);
193     }
194
195     /* get the default processor set */
196     ret = processor_set_default(mach_host_self(), &psetname);
197     if (ret != KERN_SUCCESS) {
198         printf("processor_set_default failed with %s\n",
199             mach_error_string(ret));
200         exit(1);
201     }
202
203     /*
204     * set start time to zero (immediate)
205     */
206     start1.seconds = 0;
207     start1.nanoseconds = 0;
208     start2.seconds = 0;
209     start2.nanoseconds = 0;

```

```

210
211     /*
212     * computation time of reserves
213     */
214     computel.seconds = 0;
215     computel.nanoseconds = 200000000;
216     compute2.seconds = 0;
217     compute2.nanoseconds = 100000000;
218
219     child_computel.seconds = 0;
220     child_computel.nanoseconds = 80000000;
221     child_compute2.seconds = 0;
222     child_compute2.nanoseconds = 160000000;
223     child_compute3.seconds = 0;
224     child_compute3.nanoseconds = 40000000;
225     child_compute4.seconds = 0;
226     child_compute4.nanoseconds = 80000000;
227
228
229     /*
230     * period of reserves
231     */
232     period1.seconds = 0;
233     period2.seconds = 0;
234     period1.nanoseconds = 500000000;
235     period2.nanoseconds = 250000000;
236     child_period1.seconds = 0;
237     child_period2.seconds = 0;
238     child_period3.seconds = 0;
239     child_period4.seconds = 0;
240     child_period1.nanoseconds = 600000000;
241     child_period2.nanoseconds = 1200000000;
242     child_period3.nanoseconds = 300000000;
243     child_period4.nanoseconds = 600000000;
244
245     ret = reserve_create(psetname, &reserve1);
246     if (ret != KERN_SUCCESS) {
247         printf("reserve_create 1 failed with %s\n",
248             mach_error_string(ret));
249         exit(1);
250     }
251
252     ret = reserve_create(psetname, &reserve2);
253     if (ret != KERN_SUCCESS) {
254         printf("reserve_create 2 failed with %s\n",
255             mach_error_string(ret));
256         exit(1);
257     }
258
259     ret = syscall_child_reserve_create(&child_reserve1);
260     if (ret != KERN_SUCCESS) {
261         printf("syscall_child_reserve_create 1 failed with %s\n",
262             mach_error_string(ret));
263         exit(1);
264     }
265
266     ret = syscall_child_reserve_create(&child_reserve2);
267     if (ret != KERN_SUCCESS) {

```

```

268         printf("syscall_child_reserve_create 2 failed with %s\n",
269                mach_error_string(ret));
270         exit(1);
271     }
272
273     ret = syscall_child_reserve_create(&child_reserve3);
274     if (ret != KERN_SUCCESS) {
275         printf("syscall_child_reserve_create 3 failed with %s\n",
276                mach_error_string(ret));
277         exit(1);
278     }
279
280     ret = syscall_child_reserve_create(&child_reserve4);
281     if (ret != KERN_SUCCESS) {
282         printf("syscall_child_reserve_create 4 failed with %s\n",
283                mach_error_string(ret));
284         exit(1);
285     }
286
287     /*
288      * Name new reserves
289      */
290
291     strcpy(reserve_name1, "Parent1");
292     ret = reserve_set_name(reserve1, reserve_name1);
293
294     if (ret != KERN_SUCCESS) {
295         printf("reserve_set_name 1 failed with %s\n",
296                mach_error_string(ret));
297         exit(1);
298     }
299
300     strcpy(reserve_name2, "Parent2");
301     ret = reserve_set_name(reserve2, reserve_name2);
302     if (ret != KERN_SUCCESS) {
303         printf("reserve_set_name 2 failed with %s\n",
304                mach_error_string(ret));
305         exit(1);
306     }
307
308     strcpy(child_reserve_name1, "Child1");
309     ret = reserve_set_name(child_reserve1, child_reserve_name1);
310     if (ret != KERN_SUCCESS) {
311         printf("reserve_set_name chl failed with %s\n",
312                mach_error_string(ret));
313         exit(1);
314     }
315
316     strcpy(child_reserve_name2, "Child2");
317     ret = reserve_set_name(child_reserve2, child_reserve_name2);
318     if (ret != KERN_SUCCESS) {
319         printf("reserve_set_name ch2 failed with %s\n",
320                mach_error_string(ret));
321         exit(1);
322     }
323
324     strcpy(child_reserve_name3, "Child3");
325     ret = reserve_set_name(child_reserve3, child_reserve_name3);

```

```

326     if (ret != KERN_SUCCESS) {
327         printf("reserve_set_name ch3 failed with %s\n",
328             mach_error_string(ret));
329         exit(1);
330     }
331
332     strcpy(child_reserve_name4, "Child4");
333     ret = reserve_set_name(child_reserve4, child_reserve_name4);
334     if (ret != KERN_SUCCESS) {
335         printf("reserve_set_name ch4 failed with %s\n",
336             mach_error_string(ret));
337         exit(1);
338     }
339
340     /*
341     * Actually make our reservation
342     */
343
344     ret = reserve_request(reserve1, compute1, period1, start1);
345     if (ret != KERN_SUCCESS) {
346         printf("reserve_request 1 failed with %s\n",
347             mach_error_string(ret));
348         exit(1);
349     }
350
351     ret = reserve_request(reserve2, compute2, period2, start2);
352     if (ret != KERN_SUCCESS) {
353         printf("reserve_request 2 failed with %s\n",
354             mach_error_string(ret));
355         exit(1);
356     }
357
358     ret = syscall_ch_res_req(child_reserve1, reserve1, child_compute1,
359         child_period1);
360     if (ret != KERN_SUCCESS) {
361         printf("syscall_ch_res_req() ch1 failed with %s\n",
362             mach_error_string(ret));
363         exit(1);
364     }
365
366     ret = syscall_ch_res_req(child_reserve2, reserve1, child_compute2,
367         child_period2);
368     if (ret != KERN_SUCCESS) {
369         printf("syscall_ch_res_req() ch2 failed with %s\n",
370             mach_error_string(ret));
371         exit(1);
372     }
373
374     ret = syscall_ch_res_req(child_reserve3, reserve2, child_compute3,
375         child_period3);
376     if (ret != KERN_SUCCESS) {
377         printf("syscall_ch_res_req() ch3 failed with %s\n",
378             mach_error_string(ret));
379         exit(1);
380     }
381
382     ret = syscall_ch_res_req(child_reserve4, reserve2, child_compute4,
383         child_period4);

```

```

384     if (ret != KERN_SUCCESS) {
385         printf("syscall_ch_res_req() ch4 failed with %s\n",
386             mach_error_string(ret));
387         exit(1);
388     }
389
390 /* NOW CREATE THREADS */
391
392     ret = clock_get_time(clock_port,&t1);
393     if(ret != KERN_SUCCESS)
394     {
395         printf("ERROR: got %s from clock_get_time\n",
396             mach_error_string(ret));
397     }
398     threadstart.seconds = 5;
399     threadstart.nanoseconds = 0;
400     timespec_add(threadstart,t1);
401
402     thread1 = rt_periodic_thread_fork(&thread_attr1,
403         func1, 0, 9,
404         20, 0, 20, 0, threadstart,
405         &deadline_port1);
406     printf("returned from thread fork 1\n");
407
408     thread2 = rt_periodic_thread_fork(&thread_attr2,
409         func2, 0, 9,
410         20, 0, 20, 0, threadstart,
411         &deadline_port2);
412     printf("returned from thread fork 2\n");
413
414     thread3 = rt_periodic_thread_fork(&thread_attr3,
415         func3, 0, 9,
416         20, 0, 20, 0, threadstart,
417         &deadline_port3);
418     printf("returned from thread fork 3\n");
419
420     thread4 = rt_periodic_thread_fork(&thread_attr4,
421         func4, 0, 9,
422         20, 0, 20, 0, threadstart,
423         &deadline_port4);
424     printf("returned from thread fork 4\n");
425
426     thread5 = rt_periodic_thread_fork(&thread_attr5,
427         func5, 0, 9,
428         20, 0, 20, 0, threadstart,
429         &deadline_port5);
430     printf("returned from thread fork 5\n");
431
432     thread6 = rt_periodic_thread_fork(&thread_attr6,
433         func6, 0, 9,
434         20, 0, 20, 0, threadstart,
435         &deadline_port6);
436     printf("returned from thread fork 6\n");
437
438     thread7 = rt_periodic_thread_fork(&thread_attr7,
439         func7, 0, 9,
440         20, 0, 20, 0, threadstart,
441         &deadline_port7);

```

```

442     printf("returned from thread fork 7\n");
443
444     thread8 = rt_periodic_thread_fork(&thread_attr8,
445                                     func8, 0, 9,
446                                     20, 0, 20, 0, threadstart,
447                                     &deadline_port8);
448     printf("returned from thread fork 8\n");
449
450     /*
451      * Bind threads
452      */
453
454     ret = thread_set_reserve(thread1, child_reserve1);
455     if (ret != KERN_SUCCESS) {
456         printf("thread_set_reserve 1 failed with %s\n",
457               mach_error_string(ret));
458         exit(1);
459     }
460
461     ret = thread_set_reserve(thread2, child_reserve1);
462     if (ret != KERN_SUCCESS) {
463         printf("thread_set_reserve 2 failed with %s\n",
464               mach_error_string(ret));
465         exit(1);
466     }
467
468     ret = thread_set_reserve(thread3, child_reserve2);
469     if (ret != KERN_SUCCESS) {
470         printf("thread_set_reserve 3 failed with %s\n",
471               mach_error_string(ret));
472         exit(1);
473     }
474
475     ret = thread_set_reserve(thread4, child_reserve2);
476     if (ret != KERN_SUCCESS) {
477         printf("thread_set_reserve 4 failed with %s\n",
478               mach_error_string(ret));
479         exit(1);
480     }
481
482     ret = thread_set_reserve(thread5, child_reserve3);
483     if (ret != KERN_SUCCESS) {
484         printf("thread_set_reserve 5 failed with %s\n",
485               mach_error_string(ret));
486         exit(1);
487     }
488
489     ret = thread_set_reserve(thread6, child_reserve3);
490     if (ret != KERN_SUCCESS) {
491         printf("thread_set_reserve 6 failed with %s\n",
492               mach_error_string(ret));
493         exit(1);
494     }
495
496     ret = thread_set_reserve(thread7, child_reserve4);
497     if (ret != KERN_SUCCESS) {
498         printf("thread_set_reserve 7 failed with %s\n",
499               mach_error_string(ret));

```

```

500             exit(1);
501         }
502
503         ret = thread_set_reserve(thread8, child_reserve4);
504         if (ret != KERN_SUCCESS) {
505             printf("thread_set_reserve 8 failed with %s\n",
506                 mach_error_string(ret));
507             exit(1);
508         }
509
510
511         /*
512          * Destroy the main thread
513          */
514         thread_terminate(mach_thread_self());
515     }

```

7.0.2 Hierarchical Reservation Example 2

```

1 /*
2  * RTMach example program to test hierarchical reserves and clocks.
3  */
4 #include <stdio.h>
5 #include <mach.h>
6 #include <rt/mach_reserves.h>
7 #include <rt/sched_policy.h>
8 #include <rt/rt_thread.h>
9 #include <rt/thread_attribute.h>
10 #include <rt/mach_clock.h>
11 #include <rt/timespec.h>
12 thread_t thread1;
13 thread_t thread2;
14 thread_t thread3;
15 thread_t thread4;
16
17 kern_return_t rt_simple_thread_fork(void (*func)(), int *arg, int pri);
18
19 rt_thread_attr_data_t thread_attr1;
20 mach_port_t deadline_port1 = MACH_PORT_NULL;
21
22 void func1(int *arg)
23 {
24     int index;
25     for(index = 0; index < 10000000; index++)
26         ;
27     printf("T1 done\n");
28     thread_terminate(mach_thread_self());
29 }
30
31 thread_t rt_periodic_thread_fork(rt_thread_attr_data_t *thread_attr,
32                                void (*func)(), int *arg, int priority,
33                                int period_secs, int period_nsecs,
34                                int deadline_secs, int deadline_nsecs,
35                                timespec_t starttime,
36                                mach_port_t *portp)
37 {
38     thread_t new_thread;
39     kern_return_t ret;

```

```

40 ret = rt_thread_attribute_init_edf(
41     priority, /* priority */
42     period_secs,
43     period_nsecs,
44     deadline_secs,
45     deadline_nsecs,
46     func,
47     arg,
48     portp,
49     thread_attr);
50 if (ret != KERN_SUCCESS) {
51     return ret;
52 }
53
54 (thread_attr->start_time).seconds = starttime.seconds;
55 (thread_attr->start_time).nanoseconds = starttime.nanoseconds;
56 thread_attr->abs_flag = TRUE;
57 ret = rt_thread_create(mach_task_self(), &new_thread, thread_attr);
58 return new_thread;
59 }
60
61 main()
62 {
63     mach_reserve_name_t reserve_name1;
64     mach_reserve_name_t child_reserve_name1;
65     mach_clock_t clock_port;
66     timespec_t computel, period1, start1,
67     threadstart, child_computel, child_period1, t1;
68     processor_set_name_t psetname;
69     mach_reserve_t reservel;
70     mach_reserve_t child_reservel;
71     kern_return_t ret;
72
73     ret = clock_get_port(mach_host_self(), &clock_port);
74     if (ret != KERN_SUCCESS)
75     {
76         printf("ERROR: got %s from clock_get_port\n",
77             mach_error_string(ret));
78     }
79 /* CREATE RESERVE */
80
81     /* use reservation scheduling */
82     ret = rt_set_scheduling_policy(SCHED_POLICY_HIER_RESERVES_FP);
83     if (ret != KERN_SUCCESS) {
84         printf("ERROR: got %s setting the scheduling policy\n",
85             mach_error_string(ret));
86         exit(-3);
87     }
88
89     /* get the default processor set */
90     ret = processor_set_default(mach_host_self(), &psetname);
91     if (ret != KERN_SUCCESS) {
92         printf("processor_set_default failed with %s\n",
93             mach_error_string(ret));
94         exit(1);
95     }
96
97     /*

```

```

98     * set start time to zero (immediate)
99     */
100    start1.seconds = 0;
101    start1.nanoseconds = 0;
102
103    /*
104     * computation time of reserves
105     */
106    computel.seconds = 0;
107    computel.nanoseconds = 10000000; /* 10 ms */
108    child_computel.seconds = 0;
109    child_computel.nanoseconds = 20000000; /* 5 ms */
110
111    /*
112     * period of reserves
113     */
114    periodl.seconds = 0;
115    periodl.nanoseconds = 45000000; /* 40 ms */
116    child_periodl.seconds = 0;
117    child_periodl.nanoseconds = 100000000; /* 50 ms */
118
119    ret = reserve_create(psetname, &reservel);
120    if (ret != KERN_SUCCESS) {
121        printf("reserve_create 1 failed with %s\n",
122              mach_error_string(ret));
123        exit(1);
124    }
125
126    ret = syscall_child_reserve_create(&child_reservel);
127    if (ret != KERN_SUCCESS) {
128        printf("syscall_child_reserve_create 1 failed with %s\n",
129              mach_error_string(ret));
130        exit(1);
131    }
132
133    /*
134     * Set a name for our new reserve
135     */
136
137    strcpy(reserve_name1, "Parent1");
138    ret = reserve_set_name(reservel, reserve_name1);
139
140    if (ret != KERN_SUCCESS) {
141        printf("reserve_set_name 1 failed with %s\n",
142              mach_error_string(ret));
143        exit(1);
144    }
145
146    strcpy(child_reserve_name1, "Child1");
147    ret = reserve_set_name(child_reservel, child_reserve_name1);
148    if (ret != KERN_SUCCESS) {
149        printf("reserve_set_name chl failed with %s\n",
150              mach_error_string(ret));
151        exit(1);
152    }
153
154    /*
155     * Actually make our reservation

```

```

156     */
157
158     ret = reserve_request(reserve1, compute1, period1, start1);
159     if (ret != KERN_SUCCESS) {
160         printf("reserve_request 1 failed with %s\n",
161             mach_error_string(ret));
162         exit(1);
163     }
164
165     ret = syscall_ch_res_req(child_reserve1, reserve1, child_compute1,
166         child_period1);
167     if (ret != KERN_SUCCESS) {
168         printf("syscall_ch_res_req() ch1 failed with %s\n",
169             mach_error_string(ret));
170         exit(1);
171     }
172
173     /* NOW CREATE THREAD */
174     ret = clock_get_time(clock_port, &t1);
175     if (ret != KERN_SUCCESS)
176     {
177         printf("ERROR: got %s from clock_get_time\n",
178             mach_error_string(ret));
179     }
180     threadstart.seconds = 30;
181     threadstart.nanoseconds = 0;
182     timespec_add(threadstart, t1);
183     thread1 = rt_periodic_thread_fork(&thread_attr1,
184         func1, 0, 15,
185         20, 0, 20, 0, threadstart,
186         &deadline_port1);
187     printf("returned from thread fork 1\n");
188
189
190     /*
191     * Bind our thread to this new reserve
192     */
193
194     ret = thread_set_reserve(thread1, child_reserve1);
195     if (ret != KERN_SUCCESS) {
196         printf("thread_set_reserve 1 failed with %s\n",
197             mach_error_string(ret));
198         exit(1);
199     }
200
201
202     /*
203     * Destroy the main thread
204     */
205     thread_terminate(mach_thread_self());
206 }

```

7.0.3 Hierarchical Reservation Example 3

```

1 /*
2 * RTMach example program
3 * create and bind two threads to a reserve
4 * under different scheduling policies

```

```

5  */
6
7 #include <stdio.h>
8 #include <mach.h>
9 #include <rt/mach_reserves.h>
10 #include <rt/sched_policy.h>
11 #include <rt/rt_thread.h>
12 #include <rt/thread_attribute.h>
13 #include <rt/mach_clock.h>
14 #include <rt/timespec.h>
15 thread_t thread1;
16 thread_t thread2;
17 kern_return_t rt_set_scheduling_policy(int policy);
18 kern_return_t rt_simple_thread_fork(void (*func>(), int *arg, int pri);
19
20 rt_thread_attr_data_t thread_attr1;
21 rt_thread_attr_data_t thread_attr2;
22 mach_port_t deadline_port1 = MACH_PORT_NULL;
23 mach_port_t deadline_port2 = MACH_PORT_NULL;
24
25 void func1(int *arg)
26 {
27     int index = 0;
28     for(index = 0; index <100000000; index++)
29         ;
30     printf("T1 done\n");
31     thread_terminate(mach_thread_self());
32 }
33
34
35 void func2(int *arg)
36 {
37     int index = 0;
38     for(index = 0; index < 100000000; index++)
39         ;
40     printf("T2 done\n");
41     thread_terminate(mach_thread_self());
42 }
43
44 thread_t rt_periodic_thread_fork(rt_thread_attr_data_t *thread_attr,
45                                 void (*func>(), int *arg, int priority,
46                                 int period_secs, int period_nsecs,
47                                 int deadline_secs, int deadline_nsecs,
48                                 mach_port_t *portp)
49 {
50     thread_t new_thread;
51     kern_return_t ret;
52     ret = rt_thread_attribute_init_edf(
53         priority, /* priority */
54         period_secs,
55         period_nsecs,
56         deadline_secs,
57         deadline_nsecs,
58         func,
59         arg,
60         portp,
61         thread_attr);
62     if (ret != KERN_SUCCESS) {

```

```

63     return ret;
64 }
65 ret = rt_thread_create(mach_task_self(), &new_thread, thread_attr);
66 return new_thread;
67 }
68
69
70
71 main()
72 {
73     mach_reserve_name_t reserve_name;
74     timespec_t compute, period, start;
75     processor_set_name_t psetname;
76     mach_reserve_t reserve;
77     kern_return_t ret;
78
79     /* CREATE RESERVE */
80
81     /* use reservation scheduling */
82     ret = rt_set_scheduling_policy(SCHED_POLICY_RESERVES_FP);
83     if (ret != KERN_SUCCESS) {
84         printf("ERROR: got %s setting the scheduling policy\n",
85             mach_error_string(ret));
86         exit(-3);
87     }
88
89     /* get the default processor set */
90     ret = processor_set_default(mach_host_self(), &psetname);
91     if (ret != KERN_SUCCESS) {
92         printf("processor_set_default failed with %s\n",
93             mach_error_string(ret));
94         exit(1);
95     }
96
97     /*
98      * set start time to zero (immediate)
99      */
100    start.seconds = 0;
101    start.nanoseconds = 0;
102
103    /*
104     * computation time of reserve
105     */
106    compute.seconds = 0;
107    compute.nanoseconds = 27000000; /* 27 ms */
108
109    /*
110     * period of reserve
111     */
112    period.seconds = 0;
113    period.nanoseconds = 40000000; /* 40 ms */
114
115    ret = reserve_create(psetname, &reserve);
116    if (ret != KERN_SUCCESS) {
117        printf("reserve_create failed with %s\n",
118            mach_error_string(ret));
119        exit(1);
120    }

```

```

121
122     /*
123     * Set a name for our new reserve
124     */
125     strcpy(reserve_name, "Example");
126     ret = reserve_set_name(reserve, reserve_name);
127     if (ret != KERN_SUCCESS) {
128         printf("reserve_set_name failed with %s\n",
129               mach_error_string(ret));
130         exit(1);
131     }
132
133     /*
134     * Actually make our reservation
135     */
136     ret = reserve_request(reserve, compute, period, start);
137     if (ret != KERN_SUCCESS) {
138         printf("reserve_request failed with %s\n",
139               mach_error_string(ret));
140         exit(1);
141     }
142
143
144
145     /* NOW CREATE THREADS */
146
147
148     thread1 = rt_periodic_thread_fork(&thread_attr1,
149                                     func1, 0, 13,
150                                     30, 0, 20, 0,
151                                     &deadline_port1);
152     thread2 = rt_periodic_thread_fork(&thread_attr2,
153                                     func2, 0, 14,
154                                     20, 0, 20, 0,
155                                     &deadline_port2);
156
157
158     /*
159     * Bind our thread to this new reserve
160     */
161
162
163     ret = thread_set_reserve(thread1, reserve);
164     if (ret != KERN_SUCCESS) {
165         printf("thread_set_reserve failed with %s\n",
166               mach_error_string(ret));
167         exit(1);
168     }
169
170     ret = thread_set_reserve(thread2, reserve);
171     if (ret != KERN_SUCCESS) {
172         printf("thread_set_reserve failed with %s\n",
173               mach_error_string(ret));
174         exit(1);
175     }
176
177     /*
178     * Destroy the main thread

```

```

179         */
180     thread_terminate(mach_thread_self());
181 }
182

```

7.0.4 Hierarchical Reservation Example 4

```

1 /*
2  * RTMach example program.
3  * Create and bind two threads to a reserve;
4  * create two other threads and bind them to another reserve.
5  * Investigate their behavior under different scheduling policies
6  */
7
8 #include <stdio.h>
9 #include <mach.h>
10 #include <rt/mach_reserves.h>
11 #include <rt/sched_policy.h>
12 #include <rt/rt_thread.h>
13 #include <rt/thread_attribute.h>
14 #include <rt/mach_clock.h>
15 #include <rt/timespec.h>
16 thread_t thread1;
17 thread_t thread2;
18 thread_t thread3;
19 thread_t thread4;
20
21 kern_return_t rt_set_scheduling_policy(int policy);
22 kern_return_t rt_simple_thread_fork(void (*func)(), int *arg, int pri);
23
24 rt_thread_attr_data_t thread_attr1;
25 rt_thread_attr_data_t thread_attr2;
26 rt_thread_attr_data_t thread_attr3;
27 rt_thread_attr_data_t thread_attr4;
28
29 mach_port_t deadline_port1 = MACH_PORT_NULL;
30 mach_port_t deadline_port2 = MACH_PORT_NULL;
31 mach_port_t deadline_port3 = MACH_PORT_NULL;
32 mach_port_t deadline_port4 = MACH_PORT_NULL;
33
34 void func1(int *arg)
35 {
36     int index = 0;
37     for(index = 0; index < 20000000; index++)
38         ;
39     printf("T1 done\n");
40     thread_terminate(mach_thread_self());
41 }
42
43
44 void func2(int *arg)
45 {
46     int index = 0;
47     for(index = 0; index < 20000000; index++)
48         ;
49     printf("T2 done\n");
50     thread_terminate(mach_thread_self());
51 }

```

```

52
53 void func3(int *arg)
54 {
55     int index = 0;
56     for(index = 0; index < 20000000; index++)
57         ;
58     printf("T3 done\n");
59     thread_terminate(mach_thread_self());
60 }
61
62 void func4(int *arg)
63 {
64     int index = 0;
65     for(index = 0; index < 20000000; index++)
66         ;
67     printf("T4 done\n");
68     thread_terminate(mach_thread_self());
69 }
70
71 thread_t rt_periodic_thread_fork(rt_thread_attr_data_t *thread_attr,
72                                 void (*func)(), int *arg, int priority,
73                                 int period_secs, int period_nsecs,
74                                 int deadline_secs, int deadline_nsecs,
75                                 mach_port_t *portp)
76 {
77     thread_t new_thread;
78     kern_return_t ret;
79     ret = rt_thread_attribute_init_edf(
80         priority, /* priority */
81         period_secs,
82         period_nsecs,
83         deadline_secs,
84         deadline_nsecs,
85         func,
86         arg,
87         portp,
88         thread_attr);
89     if (ret != KERN_SUCCESS) {
90         return ret;
91     }
92     ret = rt_thread_create(mach_task_self(), &new_thread, thread_attr);
93     return new_thread;
94 }
95
96
97
98 main()
99 {
100     mach_reserve_name_t reserve_name1, reserve_name2;
101     timespec_t computel, compute2, period1, period2, start1, start2;
102
103     processor_set_name_t psetname;
104     mach_reserve_t reserve1, reserve2;
105     kern_return_t ret;
106
107     /* CREATE RESERVES */
108
109     /* use reservation scheduling */

```

```

110     ret = rt_set_scheduling_policy(SCHED_POLICY_RESERVES);
111     if (ret != KERN_SUCCESS) {
112         printf("ERROR: got %s setting the scheduling policy\n",
113             mach_error_string(ret));
114         exit(-3);
115     }
116
117     /* get the default processor set */
118     ret = processor_set_default(mach_host_self(), &psetname);
119     if (ret != KERN_SUCCESS) {
120         printf("processor_set_default failed with %s\n",
121             mach_error_string(ret));
122         exit(1);
123     }
124
125     /*
126      * set start time to zero (immediate)
127      */
128     start1.seconds = 5;
129     start1.nanoseconds = 0;
130
131     /*
132      * computation time of reserve
133      */
134     computel.seconds = 0;
135     computel.nanoseconds = 5000000; /* 5 ms */
136
137     /*
138      * period of reserve
139      */
140     period1.seconds = 0;
141     period1.nanoseconds = 40000000; /* 40 ms */
142
143     ret = reserve_create(psetname, &reservel);
144     if (ret != KERN_SUCCESS) {
145         printf("reserve_create failed with %s\n",
146             mach_error_string(ret));
147         exit(1);
148     }
149
150     /*
151      * Set a name for our new reserve
152      */
153     strcpy(reserve_name1, "res_create4_reservel");
154     ret = reserve_set_name(reservel, reserve_name1);
155     if (ret != KERN_SUCCESS) {
156         printf("reserve_set_name failed with %s\n",
157             mach_error_string(ret));
158         exit(1);
159     }
160
161     /*
162      * Actually make our reservation
163      */
164     ret = reserve_request(reservel, computel, period1, start1);
165     if (ret != KERN_SUCCESS) {
166         printf("reserve_request failed with %s\n",
167             mach_error_string(ret));

```

```

168             exit(1);
169     }
170
171
172
173     /*
174     * set start time to zero (immediate)
175     */
176     start2.seconds = 5;
177     start2.nanoseconds = 0;
178
179     /*
180     * computation time of reserve
181     */
182     compute2.seconds = 0;
183     compute2.nanoseconds = 15000000; /* 30 ms */
184
185     /*
186     * period of reserve
187     */
188     period2.seconds = 0;
189     period2.nanoseconds = 20000000; /* 40 ms */
190
191     ret = reserve_create(psetname, &reserve2);
192     if (ret != KERN_SUCCESS) {
193         printf("reserve_create failed with %s\n",
194             mach_error_string(ret));
195         exit(1);
196     }
197
198     /*
199     * Set a name for our new reserve
200     */
201     strcpy(reserve_name2, "res_create4_reserve2");
202     ret = reserve_set_name(reserve2, reserve_name2);
203     if (ret != KERN_SUCCESS) {
204         printf("reserve_set_name failed with %s\n",
205             mach_error_string(ret));
206         exit(1);
207     }
208
209     /*
210     * Actually make our reservation
211     */
212     ret = reserve_request(reserve2, compute2, period2, start2);
213     if (ret != KERN_SUCCESS) {
214         printf("reserve_request failed with %s\n",
215             mach_error_string(ret));
216         exit(1);
217     }
218
219
220
221 /* NOW CREATE THREADS */
222
223
224
225

```

```

226     thread1 = rt_periodic_thread_fork(&thread_attr1,
227                                     func1, 0, 13,
228                                     30, 0, 20, 0,
229                                     &deadline_port1);
230     thread2 = rt_periodic_thread_fork(&thread_attr2,
231                                     func2, 0, 14,
232                                     20, 0, 20, 0,
233                                     &deadline_port2);
234     func3, 0, 13,
235
236     thread3 = rt_periodic_thread_fork(&thread_attr3,
237                                     func3, 0, 13,
238                                     30, 0, 20, 0,
239                                     &deadline_port3);
240     thread4 = rt_periodic_thread_fork(&thread_attr4,
241                                     func4, 0, 14,
242                                     20, 0, 20, 0,
243                                     &deadline_port4);
244
245     /*
246     * Bind threads to reservel, reserve2
247     */
248
249
250
251     ret = thread_set_reserve(thread1, reservel);
252     if (ret != KERN_SUCCESS) {
253         printf("thread_set_reserve failed with %s\n",
254               mach_error_string(ret));
255         exit(1);
256     }
257
258     ret = thread_set_reserve(thread2, reservel);
259     if (ret != KERN_SUCCESS) {
260         printf("thread_set_reserve failed with %s\n",
261               mach_error_string(ret));
262         exit(1);
263     }
264
265     ret = thread_set_reserve(thread3, reserve2);
266     if (ret != KERN_SUCCESS) {
267         printf("thread_set_reserve failed with %s\n",
268               mach_error_string(ret));
269         exit(1);
270     }
271
272     ret = thread_set_reserve(thread4, reserve2);
273     if (ret != KERN_SUCCESS) {
274         printf("thread_set_reserve failed with %s\n",
275               mach_error_string(ret));
276         exit(1);
277     }
278
279     /*
280     * Destroy the main thread
281     */
282     thread_terminate(mach_thread_self());
283 }

```

7.0.5 Hierarchical Reservation Example 5

```
1 /*
2  * RTMach example program to test hierarchical reserves.
3  */
4 #include <stdio.h>
5 #include <mach.h>
6 #include <rt/mach_reserves.h>
7 #include <rt/sched_policy.h>
8 #include <rt/rt_thread.h>
9 #include <rt/thread_attribute.h>
10 #include <rt/mach_clock.h>
11 #include <rt/timespec.h>
12
13 #define UPPER_LIMIT 80000000
14
15 thread_t thread1;
16 thread_t thread2;
17 thread_t thread3;
18 thread_t thread4;
19 thread_t thread5;
20 thread_t thread6;
21 thread_t thread7;
22 thread_t thread8;
23
24 kern_return_t rt_simple_thread_fork(void (*func)(), int *arg, int pri);
25
26 rt_thread_attr_data_t thread_attr1;
27 rt_thread_attr_data_t thread_attr2;
28 rt_thread_attr_data_t thread_attr3;
29 rt_thread_attr_data_t thread_attr4;
30 rt_thread_attr_data_t thread_attr5;
31 rt_thread_attr_data_t thread_attr6;
32 rt_thread_attr_data_t thread_attr7;
33 rt_thread_attr_data_t thread_attr8;
34
35 mach_port_t deadline_port1 = MACH_PORT_NULL;
36 mach_port_t deadline_port2 = MACH_PORT_NULL;
37 mach_port_t deadline_port3 = MACH_PORT_NULL;
38 mach_port_t deadline_port4 = MACH_PORT_NULL;
39 mach_port_t deadline_port5 = MACH_PORT_NULL;
40 mach_port_t deadline_port6 = MACH_PORT_NULL;
41 mach_port_t deadline_port7 = MACH_PORT_NULL;
42 mach_port_t deadline_port8 = MACH_PORT_NULL;
43
44 void func1(int *arg)
45 {
46     int index = 0;
47     for(index = 0; index < UPPER_LIMIT; index++)
48         ;
49     printf("T1 done\n");
50     thread_terminate(mach_thread_self());
51 }
52
53
54 void func2(int *arg)
55 {
```

```

56     int index = 0;
57     for(index = 0; index < UPPER_LIMIT; index++)
58         ;
59     printf("T2 done\n");
60     thread_terminate(mach_thread_self());
61 }
62
63
64 void func3(int *arg)
65 {
66     int index = 0;
67     for(index = 0; index < UPPER_LIMIT; index++)
68         ;
69     printf("T3 done\n");
70     thread_terminate(mach_thread_self());
71 }
72
73
74 void func4(int *arg)
75 {
76     int index = 0;
77     for(index = 0; index < UPPER_LIMIT; index++)
78         ;
79     printf("T4 done\n");
80     thread_terminate(mach_thread_self());
81 }
82
83 void func5(int *arg)
84 {
85     int index = 0;
86     for(index = 0; index < UPPER_LIMIT; index++)
87         ;
88     printf("T5 done\n");
89     thread_terminate(mach_thread_self());
90 }
91
92 void func6(int *arg)
93 {
94     int index = 0;
95     for(index = 0; index < UPPER_LIMIT; index++)
96         ;
97     printf("T6 done\n");
98     thread_terminate(mach_thread_self());
99 }
100
101 void func7(int *arg)
102 {
103     int index = 0;
104     for(index = 0; index < UPPER_LIMIT; index++)
105         ;
106     printf("T7 done\n");
107     thread_terminate(mach_thread_self());
108 }
109
110 void func8(int *arg)
111 {
112     int index = 0;
113     for(index = 0; index < UPPER_LIMIT; index++)

```

```

114         ;
115         printf("T8 done\n");
116         thread_terminate(mach_thread_self());
117     }
118
119     thread_t rt_periodic_thread_fork(rt_thread_attr_data_t *thread_attr,
120                                     void (*func)(), int *arg, int priority,
121                                     int period_secs, int period_nsecs,
122                                     int deadline_secs, int deadline_nsecs,
123                                     timespec_t starttime,
124                                     mach_port_t *portp)
125     {
126         thread_t new_thread;
127         kern_return_t ret;
128         ret = rt_thread_attribute_init_edf(
129             priority, /* priority */
130             period_secs,
131             period_nsecs,
132             deadline_secs,
133             deadline_nsecs,
134             func,
135             arg,
136             portp,
137             thread_attr);
138         if (ret != KERN_SUCCESS) {
139             return ret;
140         }
141
142         (thread_attr->start_time).seconds = starttime.seconds;
143         (thread_attr->start_time).nanoseconds = starttime.nanoseconds;
144         thread_attr->abs_flag = TRUE;
145
146         ret = rt_thread_create(mach_task_self(), &new_thread, thread_attr);
147         return new_thread;
148     }
149
150     main()
151     {
152         mach_reserve_name_t reserve_name1;
153         mach_reserve_name_t reserve_name2;
154         mach_clock_t clock_port;
155         mach_reserve_name_t child_reserve_name1;
156         mach_reserve_name_t child_reserve_name2;
157         mach_reserve_name_t child_reserve_name3;
158         mach_reserve_name_t child_reserve_name4;
159
160         timespec_t compute1, period1, start1,
161                 compute2, period2, start2,
162                 child_compute1, child_period1,
163                 child_compute2, child_period2,
164                 child_compute3, child_period3,
165                 child_compute4, child_period4,
166                 threadstart, t1;
167
168         processor_set_name_t psetname;
169         mach_reserve_t reserve1;
170         mach_reserve_t reserve2;
171         mach_reserve_t child_reserve1;

```

```

172     mach_reserve_t child_reserve2;
173     mach_reserve_t child_reserve3;
174     mach_reserve_t child_reserve4;
175
176     kern_return_t ret;
177
178     ret = clock_get_port(mach_host_self(), &clock_port);
179     if (ret != KERN_SUCCESS)
180     {
181         printf("ERROR: got %s from clock_get_port\n",
182             mach_error_string(ret));
183     }
184
185     /* CREATE RESERVE */
186
187     /* use reservation scheduling */
188     ret = rt_set_scheduling_policy(SCHED_POLICY_HIER_RESERVES_FP);
189     if (ret != KERN_SUCCESS) {
190         printf("ERROR: got %s setting the scheduling policy\n",
191             mach_error_string(ret));
192         exit(-3);
193     }
194
195     /* get the default processor set */
196     ret = processor_set_default(mach_host_self(), &psetname);
197     if (ret != KERN_SUCCESS) {
198         printf("processor_set_default failed with %s\n",
199             mach_error_string(ret));
200         exit(1);
201     }
202
203     /*
204     * set start time to zero (immediate)
205     */
206     start1.seconds = 0;
207     start1.nanoseconds = 0;
208     start2.seconds = 0;
209     start2.nanoseconds = 0;
210
211     /*
212     * computation time of reserves
213     */
214     compute1.seconds = 0;
215     compute1.nanoseconds = 200000000;
216     compute2.seconds = 0;
217     compute2.nanoseconds = 100000000;
218
219     child_compute1.seconds = 0;
220     child_compute1.nanoseconds = 80000000;
221     child_compute2.seconds = 0;
222     child_compute2.nanoseconds = 160000000;
223     child_compute3.seconds = 0;
224     child_compute3.nanoseconds = 40000000;
225     child_compute4.seconds = 0;
226     child_compute4.nanoseconds = 80000000;
227
228
229     /*

```

```

230     * period of reserves
231     */
232     period1.seconds = 0;
233     period2.seconds = 0;
234     period1.nanoseconds = 500000000;
235     period2.nanoseconds = 250000000;
236     child_period1.seconds = 0;
237     child_period2.seconds = 0;
238     child_period3.seconds = 0;
239     child_period4.seconds = 0;
240     child_period1.nanoseconds = 600000000;
241     child_period2.nanoseconds = 1200000000;
242     child_period3.nanoseconds = 300000000;
243     child_period4.nanoseconds = 600000000;
244
245     ret = reserve_create(psetname, &reserve1);
246     if (ret != KERN_SUCCESS) {
247         printf("reserve_create 1 failed with %s\n",
248             mach_error_string(ret));
249         exit(1);
250     }
251
252     ret = reserve_create(psetname, &reserve2);
253     if (ret != KERN_SUCCESS) {
254         printf("reserve_create 2 failed with %s\n",
255             mach_error_string(ret));
256         exit(1);
257     }
258
259     ret = syscall_child_reserve_create(&child_reserve1);
260     if (ret != KERN_SUCCESS) {
261         printf("syscall_child_reserve_create 1 failed with %s\n",
262             mach_error_string(ret));
263         exit(1);
264     }
265
266     ret = syscall_child_reserve_create(&child_reserve2);
267     if (ret != KERN_SUCCESS) {
268         printf("syscall_child_reserve_create 2 failed with %s\n",
269             mach_error_string(ret));
270         exit(1);
271     }
272
273     ret = syscall_child_reserve_create(&child_reserve3);
274     if (ret != KERN_SUCCESS) {
275         printf("syscall_child_reserve_create 3 failed with %s\n",
276             mach_error_string(ret));
277         exit(1);
278     }
279
280     ret = syscall_child_reserve_create(&child_reserve4);
281     if (ret != KERN_SUCCESS) {
282         printf("syscall_child_reserve_create 4 failed with %s\n",
283             mach_error_string(ret));
284         exit(1);
285     }
286
287     /*

```

```

288     * Name new reserves
289     */
290
291     strcpy(reserve_name1, "Parent1");
292     ret = reserve_set_name(reserve1, reserve_name1);
293
294     if (ret != KERN_SUCCESS) {
295         printf("reserve_set_name 1 failed with %s\n",
296             mach_error_string(ret));
297         exit(1);
298     }
299
300     strcpy(reserve_name2, "Parent2");
301     ret = reserve_set_name(reserve2, reserve_name2);
302     if (ret != KERN_SUCCESS) {
303         printf("reserve_set_name 2 failed with %s\n",
304             mach_error_string(ret));
305         exit(1);
306     }
307
308     strcpy(child_reserve_name1, "Child1");
309     ret = reserve_set_name(child_reserve1, child_reserve_name1);
310     if (ret != KERN_SUCCESS) {
311         printf("reserve_set_name ch1 failed with %s\n",
312             mach_error_string(ret));
313         exit(1);
314     }
315
316     strcpy(child_reserve_name2, "Child2");
317     ret = reserve_set_name(child_reserve2, child_reserve_name2);
318     if (ret != KERN_SUCCESS) {
319         printf("reserve_set_name ch2 failed with %s\n",
320             mach_error_string(ret));
321         exit(1);
322     }
323
324     strcpy(child_reserve_name3, "Child3");
325     ret = reserve_set_name(child_reserve3, child_reserve_name3);
326     if (ret != KERN_SUCCESS) {
327         printf("reserve_set_name ch3 failed with %s\n",
328             mach_error_string(ret));
329         exit(1);
330     }
331
332     strcpy(child_reserve_name4, "Child4");
333     ret = reserve_set_name(child_reserve4, child_reserve_name4);
334     if (ret != KERN_SUCCESS) {
335         printf("reserve_set_name ch4 failed with %s\n",
336             mach_error_string(ret));
337         exit(1);
338     }
339
340     /*
341     * Actually make our reservation
342     */
343
344     ret = reserve_request(reserve1, compute1, period1, start1);
345     if (ret != KERN_SUCCESS) {

```

```

346         printf("reserve_request 1 failed with %s\n",
347                mach_error_string(ret));
348         exit(1);
349     }
350
351     ret = reserve_request(reserve2, compute2, period2, start2);
352     if (ret != KERN_SUCCESS) {
353         printf("reserve_request 2 failed with %s\n",
354                mach_error_string(ret));
355         exit(1);
356     }
357
358     ret = syscall_ch_res_req(child_reserve1, reserve1, child_compute1,
359                             child_period1);
360     if (ret != KERN_SUCCESS) {
361         printf("syscall_ch_res_req() ch1 failed with %s\n",
362                mach_error_string(ret));
363         exit(1);
364     }
365
366     ret = syscall_ch_res_req(child_reserve2, reserve1, child_compute2,
367                             child_period2);
368     if (ret != KERN_SUCCESS) {
369         printf("syscall_ch_res_req() ch2 failed with %s\n",
370                mach_error_string(ret));
371         exit(1);
372     }
373
374     ret = syscall_ch_res_req(child_reserve3, reserve2, child_compute3,
375                             child_period3);
376     if (ret != KERN_SUCCESS) {
377         printf("syscall_ch_res_req() ch3 failed with %s\n",
378                mach_error_string(ret));
379         exit(1);
380     }
381
382     ret = syscall_ch_res_req(child_reserve4, reserve2, child_compute4,
383                             child_period4);
384     if (ret != KERN_SUCCESS) {
385         printf("syscall_ch_res_req() ch4 failed with %s\n",
386                mach_error_string(ret));
387         exit(1);
388     }
389
390 /* NOW CREATE THREADS */
391
392     ret = clock_get_time(clock_port, &t1);
393     if (ret != KERN_SUCCESS)
394     {
395         printf("ERROR: got %s from clock_get_time\n",
396                mach_error_string(ret));
397     }
398     threadstart.seconds = 5;
399     threadstart.nanoseconds = 0;
400     timespec_add(threadstart, t1);
401
402     thread1 = rt_periodic_thread_fork(&thread_attr1,
403                                     func1, 0, 9,

```

```

404         20, 0, 20, 0, threadstart,
405         &deadline_port1);
406     printf("returned from thread fork 1\n");
407
408     thread2 = rt_periodic_thread_fork(&thread_attr2,
409         func2, 0, 9,
410         20, 0, 20, 0, threadstart,
411         &deadline_port2);
412     printf("returned from thread fork 2\n");
413
414     thread3 = rt_periodic_thread_fork(&thread_attr3,
415         func3, 0, 9,
416         20, 0, 20, 0, threadstart,
417         &deadline_port3);
418     printf("returned from thread fork 3\n");
419
420     thread4 = rt_periodic_thread_fork(&thread_attr4,
421         func4, 0, 9,
422         20, 0, 20, 0, threadstart,
423         &deadline_port4);
424     printf("returned from thread fork 4\n");
425
426     thread5 = rt_periodic_thread_fork(&thread_attr5,
427         func5, 0, 9,
428         20, 0, 20, 0, threadstart,
429         &deadline_port5);
430     printf("returned from thread fork 5\n");
431
432     thread6 = rt_periodic_thread_fork(&thread_attr6,
433         func6, 0, 9,
434         20, 0, 20, 0, threadstart,
435         &deadline_port6);
436     printf("returned from thread fork 6\n");
437
438     thread7 = rt_periodic_thread_fork(&thread_attr7,
439         func7, 0, 9,
440         20, 0, 20, 0, threadstart,
441         &deadline_port7);
442     printf("returned from thread fork 7\n");
443
444     thread8 = rt_periodic_thread_fork(&thread_attr8,
445         func8, 0, 9,
446         20, 0, 20, 0, threadstart,
447         &deadline_port8);
448     printf("returned from thread fork 8\n");
449
450     /*
451     * Bind threads
452     */
453
454     ret = thread_set_reserve(thread1, child_reserve1);
455     if (ret != KERN_SUCCESS) {
456         printf("thread_set_reserve 1 failed with %s\n",
457             mach_error_string(ret));
458         exit(1);
459     }
460
461     ret = thread_set_reserve(thread2, child_reserve1);

```

```

462     if (ret != KERN_SUCCESS) {
463         printf("thread_set_reserve 2 failed with %s\n",
464             mach_error_string(ret));
465         exit(1);
466     }
467
468     ret = thread_set_reserve(thread3, child_reserve2);
469     if (ret != KERN_SUCCESS) {
470         printf("thread_set_reserve 3 failed with %s\n",
471             mach_error_string(ret));
472         exit(1);
473     }
474
475     ret = thread_set_reserve(thread4, child_reserve2);
476     if (ret != KERN_SUCCESS) {
477         printf("thread_set_reserve 4 failed with %s\n",
478             mach_error_string(ret));
479         exit(1);
480     }
481
482     ret = thread_set_reserve(thread5, child_reserve3);
483     if (ret != KERN_SUCCESS) {
484         printf("thread_set_reserve 5 failed with %s\n",
485             mach_error_string(ret));
486         exit(1);
487     }
488
489     ret = thread_set_reserve(thread6, child_reserve3);
490     if (ret != KERN_SUCCESS) {
491         printf("thread_set_reserve 6 failed with %s\n",
492             mach_error_string(ret));
493         exit(1);
494     }
495
496     ret = thread_set_reserve(thread7, child_reserve4);
497     if (ret != KERN_SUCCESS) {
498         printf("thread_set_reserve 7 failed with %s\n",
499             mach_error_string(ret));
500         exit(1);
501     }
502
503     ret = thread_set_reserve(thread8, child_reserve4);
504     if (ret != KERN_SUCCESS) {
505         printf("thread_set_reserve 8 failed with %s\n",
506             mach_error_string(ret));
507         exit(1);
508     }
509
510     /*
511     * Destroy the main thread
512     */
513     thread_terminate(mach_thread_self());
514 }
515 }

```

Chapter 8

Virtual Memory Management

This section illustrates the use of some of the Mach virtual memory system interfaces.

8.1 Allocating and Wiring

This program allocates 8192 bytes of virtual memory and “wires” it down. Wiring refers to creating a mapping to real physical memory and insuring that the virtual pages will not fault due to the normal paging behavior of the system. Note that allocating virtual memory does not automatically create a virtual to physical mapping. Such a mapping only occurs when a page is manipulated or “faulted.” Wiring the pages down insures that the mapping exists and is always resident. The program also “wires” down its code pages.

8.2 Allocating and Wiring Program Components

lines 1-2 These includes files are need to define the Mach data structures and calls.

line 12 Here we obtain the task port for the currently executing task. This is needed to indicate which address space is receiving the virtual memory allocation.

line 15 Here we obtain the host privileged port which is necessary to execute the `vm_wire` call. This level of privilege comes along with root privileges under Unix. This is necessary to prevent resource attacks against the system.

lines 19-24 Fault in the code pages of our task, and mark them as nonpageable.

lines 29-34 This allocates 8192 bytes of virtual memory at the next convenient location.

lines 38-43 Here we wire down the pages we have allocated. This “faults” them in, or creates a virtual to physical mapping, and requests that the system maintain this mapping until told otherwise.

8.3 Allocating and Wiring Program Text and Data

```
1 #include <mach.h>
2 #include <mach/mach_host.h>
3
4 main()
5 {
6     task_t          mytask;
7     mach_port_t     host_priv_port;
8     vm_offset_t     memory;
9     long            size;
```

```

10     kern_return_t    ret;
11
12     mytask = mach_task_self();           /* Which task is this?*/
13
14     /* Get the Host Privileged port to allow the user to wire pages */
15     host_priv_port = mach_host_priv_self();
16
17     /* Wire down the task's code pages- faults them all in, and marks
18        them nonpageable */
19     ret = task_wire_code(host_priv_port, mytask);
20     if (ret != KERN_SUCCESS) {
21         printf("task_wire_code failed. ret = %x (%s)\n", ret,
22              mach_error_string(ret));
23         exit(1);
24     }
25
26     size = 8192;
27
28     /* Allocate 8192 bytes of virtual memory */
29     ret = vm_allocate(mytask, &memory, size, TRUE);
30     if (ret != KERN_SUCCESS) {
31         printf("vm_allocate failed. ret = %x (%s)\n", ret,
32              mach_error_string(ret));
33         exit(1);
34     }
35
36     /* Fault in the memory that has just been allocated and wire it
37        to physical pages so it can't be paged out */
38     ret = vm_wire(host_priv_port, mytask, memory, size, VM_PROT_DEFAULT);
39     if (ret != KERN_SUCCESS) {
40         printf("vm_wire failed. ret = %x (%s)\n", ret,
41              mach_error_string(ret));
42         exit(1);
43     }
44
45 }

```

8.4 Sharing

This section describes how to use memory objects. The example program creates a memory object and shares it within one task. The choice of using one task is for simplicity only. It generally makes sense to share memory objects between tasks.

8.5 Sharing Program Components

lines 1-3 These includes files are need to define the Mach data structures and calls.

line 21 Here we obtain the task port for the currently executing task. This is needed to indicate which address space is receiving the virtual memory allocation.

line 24 Here we obtain the host privileged port which is necessary to manipulate the default pager port. This level of privilege comes along with root privileges under Unix.

lines 31-36 These lines are used to obtain the current default pager port. This is necessary in order to request the pager to take action on our behalf.

lines 39-44 Here we request that the default pager allocate a memory object of 8192 bytes for us. Note that this virtual memory while allocated, does not yet exist in our address space and it must be subsequently mapped in. Having the port to the memory object is sufficient to manipulate this memory, however, and this port may be passed between tasks.

lines 50-56 `vm_map` is used to map the allocated memory object into our address space at the next convenient location.

lines 64-70 Here we make another mapping for the same memory object, but we request that it be located 16384 bytes later.

lines 83-75 By modifying the first word of the first mapping but not the second, we can see that we have dual mapped this memory since the change appears in both mappings.

8.6 Sharing Program Text

```
1 #include <mach.h>
2 #include <mach/mach_host.h>
3 #include <mach/default_pager.h>
4
5 main()
6 {
7     task_t          mytask;
8     mach_port_t     host_priv_port;
9     mach_port_t     pager;
10    memory_object_t memory_object;
11    vm_size_t        size;
12
13    vm_offset_t      address1;
14    vm_offset_t      address2;
15    vm_offset_t      *addrptr1;
16    vm_offset_t      *addrptr2;
17
18    kern_return_t    ret;
19
20
21    mytask = mach_task_self();          /* Which task is this?*/
22
23    /* Get the Host Privileged port to allow the user to wire pages */
24    host_priv_port = mach_host_priv_self();
25
26    /*
27     *   Setting the default memory manger using MACH_PORT_NULL for the
28     *   pager argument, returns the current default pager port in the
29     *   pager argument
30     */
31    pager = MACH_PORT_NULL;
32    ret = vm_set_default_memory_manager(host_priv_port, &pager);
33    if (ret != KERN_SUCCESS) {
34        printf("vm_set_default_memory_manager failed. ret = %x\n",ret);
35        exit();
36    }
37
38    /* Allocate a 8192 byte memory object from the default pager */
39    size = 8192;
40    ret = default_pager_object_create(pager, &memory_object, size);
41    if (ret != KERN_SUCCESS) {
```

```

42         printf("default_pager_object_create failed. ret = %x\n",ret);
43         exit();
44     }
45
46     /*
47     *     Map the memory object into this task's address space at the
48     *     next convenient location.
49     */
50     address1 = 0;
51     ret = vm_map(mytask, &address1, size, 0, TRUE, memory_object, 0, FALSE,
52                 VM_PROT_DEFAULT, VM_PROT_DEFAULT, VM_INHERIT_DEFAULT);
53     if (ret != KERN_SUCCESS) {
54         printf("vm_map. ret = %x\n",ret);
55         exit();
56     }
57
58     /*
59     *     Map the memory object again, but at a location (size * 2)
60     *     bytes after the first mapping.  Since we have the memory
61     *     object port we can map this memory into any task, we use
62     *     the same task for simplicity
63     */
64     address2 = address1 + (size * 2);
65     ret = vm_map(mytask, &address2, size, 0, FALSE, memory_object, 0, FALSE,
66                 VM_PROT_DEFAULT, VM_PROT_DEFAULT, VM_INHERIT_DEFAULT);
67     if (ret != KERN_SUCCESS) {
68         printf("vm_map. ret = %x\n",ret);
69         exit();
70     }
71
72     /* Modify the first word in the first mapping */
73     addrptr1 = (vm_offset_t *)address1;
74     addrptr2 = (vm_offset_t *)address2;
75     *addrptr1 = 0xdead;
76
77     /* We see that the modification can be seen in both mappings */
78     printf("address 1 = %x:%x, address 2 = %x:%x\n",address1,*addrptr1,
79           address2,*addrptr2);
80 }

```

Chapter 9

Real-Time Synchronization

The current version of Real-Time Mach provides two kind of synchronization primitives: real-time mutex and condition variable. This section describe Real-Time synchronization facilities in Real-Time Mach.

9.1 Real-Time Mutex

The following program is an example of the real-time mutex primitives. The mutex policy should be specified in mutex attribute(line 36) when a mutex variable is created. Currently, *fifo(PRI_FIFO)*, *priority(PRI_PRI)*, and basic priority inheritance(*PRI_BPI*) are implemented. The main function allocates a mutex variable(line 37), and creates two periodic threads(line 55 and 73). Both real-time threads call *rt_mutex_lock* to get the mutex(line 96 and 120), then go into the critical section in each period. After executing the critical section, *rt_mutex_unlock* is called to release the mutex(line 108 and 132).

```
1 /*
2  * Demonstration of Real-Time mutex
3  */
4
5 #include <mach.h>
6 #include <rt/rt_thread.h>
7 #include <rt/rt_types.h>
8 #include <rt/rt_priority.h>
9 #include <rt/rt_sync_attr.h>
10 #include <stdio.h>
11
12 #define PRIORITY          1
13 #define PERIOD_SEC       1
14 #define PERIOD_NSEC      0
15
16 void          rtthread_1();
17 void          rtthread_2();
18 mach_port_t  mutex;
19
20 #define DELAY(x)                                     \
21 { int i;                                           \
22   for (i = 0; i < (x); i++) ;                      \
23 }
24
25 main()
26 {
27     rt_thread_attr_data_t  th_attr;
28     thread_t               thread;
```

```

29     rt_mutex_attr_data_t    mutex_attr;
30     kern_return_t           ret;
31
32     /*
33     * Allocate Mutex
34     */
35
36     mutex_attr.mutex_policy = PRI_BPI;          /* Specify mutex policy */
37     ret = rt_mutex_allocate(&mutex, &mutex_attr);
38                                           /* Allocate mutex */
39     if(ret != KERN_SUCCESS) {
40         printf("Cannot allocate mutex\n");
41         exit(1);
42     }
43
44     /*
45     * Create Thread 1
46     */
47     ret = rt_thread_attribute_init(PRIORITY, PERIOD_SEC, PERIOD_NSEC,
48                                   rtthread_1, NULL, NULL, &th_attr);
49                                           /* Initialize thread structure */
50     if (ret != KERN_SUCCESS) {
51         printf("Initializing thread attribute failed\n");
52         exit(1);
53     }
54
55     ret = rt_thread_create(mach_task_self(), &thread, &th_attr);
56                                           /* Create Real-Time thread */
57     if (ret != KERN_SUCCESS) {
58         printf("Creating real-time thread");
59         exit(1);
60     }
61
62     /*
63     * Create Thread 2
64     */
65     ret = rt_thread_attribute_init(PRIORITY, PERIOD_SEC, PERIOD_NSEC,
66                                   rtthread_2, NULL, NULL, &th_attr);
67                                           /* Initialize thread structure */
68     if (ret != KERN_SUCCESS) {
69         printf("Initializing thread attribute failed\n");
70         exit(1);
71     }
72
73     ret = rt_thread_create(mach_task_self(), &thread, &th_attr);
74                                           /* Create Real-Time thread */
75     if (ret != KERN_SUCCESS) {
76         printf("Creating real-time thread");
77         exit(1);
78     }
79
80     /*
81     * Root thread Sleeps here
82     */
83     sleep(10);
84
85     /*
86     * Bye

```

```

87     */
88     exit(0);
89 }
90
91 void
92 rtthread_1(void)
93 {
94     kern_return_t    ret;
95
96     ret = rt_mutex_lock(mutex);    /* Acquire mutex */
97     if(ret != KERN_SUCCESS) {
98         printf("lock failed\n");
99         thread_terminate(mach_thread_self());
100    }
101
102    printf("Thread #1 enters critical section\n");
103
104    DELAY(1000000);                /* Critical section is here */
105
106    printf("Thread #1 leaves critical section\n");
107
108    ret = rt_mutex_unlock(mutex);  /* Release mutex */
109    if(ret != KERN_SUCCESS) {
110        printf("unlock failed\n");
111        thread_terminate(mach_thread_self());
112    }
113 }
114
115 void
116 rtthread_2(void)
117 {
118     kern_return_t    ret;
119
120     ret = rt_mutex_lock(mutex);    /* Acquire mutex */
121     if(ret != KERN_SUCCESS) {
122         printf("lock failed\n");
123         thread_terminate(mach_thread_self());
124     }
125
126    printf("Thread #2 enters critical section\n");
127
128    DELAY(1000000);                /* Critical section is here */
129
130    printf("Thread #2 leaves critical section\n");
131
132    ret = rt_mutex_unlock(mutex);  /* Release mutex */
133    if(ret != KERN_SUCCESS) {
134        printf("unlock failed\n");
135        thread_terminate(mach_thread_self());
136    }
137 }

```

9.2 Condition Variable

The example program demonstrate the use of condition variable primitives. The main function allocates a shared condition variable using *rt_condition_allocate*(line 44) and creates two periodic threads, sender and receiver. When the *count* value is less than or equal to zero, the receiver thread suspends itself and waits for a signal from the sender

by calling `rt_condition_wait`(line 114). The sender thread increments the count and calls `rt_condition_signal` to send a signal to server(line 147). `rt_condition_wait` and `rt_condition_signal` must be called in a critical section.

```

1  /*
2  *   Demonstration of Real-Time condition variable
3  */
4
5  #include <mach.h>
6  #include <rt/rt_thread.h>
7  #include <rt/rt_types.h>
8  #include <rt/rt_priority.h>
9  #include <rt/rt_sync_attr.h>
10 #include <stdio.h>
11
12 #define PRIORITY          1
13 #define PERIOD_SEC       1
14 #define PERIOD_NSEC     0
15
16 void                    receiver(), sender();
17 mach_port_t            mutex;
18 mach_port_t            cond;
19 int                    count = 0;
20
21 main()
22 {
23     rt_thread_attr_data_t  th_attr;
24     thread_t              thread;
25     rt_mutex_attr_data_t  mutex_attr;
26     rt_cond_attr_data_t   cond_attr;
27     kern_return_t         ret;
28
29     /*
30      * Allocate Mutex
31      */
32     mutex_attr.mutex_policy = PRI_BPI;          /* Specify mutex policy */
33     ret = rt_mutex_allocate(&mutex, &mutex_attr);
34                                           /* Allocate mutex */
35     if(ret != KERN_SUCCESS) {
36         printf("Cannot allocate mutex\n");
37         exit(1);
38     }
39
40     /*
41      * Allocate Condition Variable
42      */
43     cond_attr.cond_policy = BLOCK_FIFO;        /* Specify condition policy */
44     ret = rt_condition_allocate(&cond, &cond_attr);
45                                           /* Allocate condition */
46     if (ret != KERN_SUCCESS) {
47         printf("Cannot allocate condition");
48         exit(1);
49     }
50
51     /*
52      * Create Receiver Thread
53      */
54     ret = rt_thread_attribute_init(PRIORITY, PERIOD_SEC, PERIOD_NSEC,
55                                   receiver, NULL, NULL, &th_attr);

```

```

56                                     /* Initialize thread structure */
57     if (ret != KERN_SUCCESS) {
58         printf("Initializing thread attribute failed\n");
59         exit(1);
60     }
61
62     ret = rt_thread_create(mach_task_self(), &thread, &th_attr);
63                                     /* Create Real-Time thread */
64     if (ret != KERN_SUCCESS) {
65         printf("Creating real-time thread");
66         exit(1);
67     }
68
69     /*
70      * Create Sender Thread
71      */
72     ret = rt_thread_attribute_init(PRIORITY, PERIOD_SEC, PERIOD_NSEC,
73                                   sender, NULL, NULL, &th_attr);
74                                     /* Initialize thread structure */
75     if (ret != KERN_SUCCESS) {
76         printf("Initializing thread attribute failed\n");
77         exit(1);
78     }
79
80     ret = rt_thread_create(mach_task_self(), &thread, &th_attr);
81                                     /* Create Real-Time thread */
82     if (ret != KERN_SUCCESS) {
83         printf("Creating real-time thread");
84         exit(1);
85     }
86
87     /*
88      * Root thread Sleeps here
89      */
90     sleep(10);
91
92     /*
93      * Bye
94      */
95     exit(0);
96 }
97
98 void
99 receiver(void)
100 {
101     kern_return_t    ret;
102
103     ret = rt_mutex_lock(mutex);      /* Acquire mutex */
104     if (ret != KERN_SUCCESS) {
105         printf("lock failed\n");
106         thread_terminate(mach_thread_self());
107     }
108
109     while (count <= 0) {
110         /*
111          * Wait a condition
112          */
113

```

```

114         ret = rt_condition_wait(cond, mutex);
115         if (ret != KERN_SUCCESS) {
116             printf("Condition wait failed\n");
117             exit(1);
118         }
119     }
120     printf("receiver subtracts from count. count = %d\n",count);
121     --count;
122
123     ret = rt_mutex_unlock(mutex);    /* Release mutex */
124     if(ret != KERN_SUCCESS) {
125         printf("unlock failed\n");
126         thread_terminate(mach_thread_self());
127     }
128 }
129
130 void
131 sender(void)
132 {
133     int    ret;
134
135     ret = rt_mutex_lock(mutex);      /* Acquire mutex */
136     if(ret != KERN_SUCCESS) {
137         printf("lock failed\n");
138         thread_terminate(mach_thread_self());
139     }
140
141     /*
142      * Send a Signal
143      */
144     count++;
145     printf("sender adds to count. count = %d\n",count);
146
147     ret = rt_condition_signal(cond);
148     if (ret != KERN_SUCCESS) {
149         printf("Condition signal failed\n");
150         exit(1);
151     }
152
153     ret = rt_mutex_unlock(mutex);    /* Release mutex */
154     if(ret != KERN_SUCCESS) {
155         printf("unlock failed\n");
156         thread_terminate(mach_thread_self());
157     }
158 }

```

Chapter 10

Real-Time IPC

Real-Time Mach provides a real-time version of IPC. This section describe Real-Time IPC facilities using mig. The example consists three parts: a server, a client and the mig routines.

10.1 Server Program

The following program is a simple server program. The main function allocates a real-time port (line 39). The port attributes, number of messages, and its size should be specified in port attribute (lines 34-38). The real-time port should be registered to name server so that clients know the port (line 49). Then, create a aperiodic server thread (line 67). The body of server thread calls `rt_mach_msg_server()` which is a library routine to receive messages from clients. In `rt_mach_msg_server()`, the server thread automatically associated with the real-time port. Each service routines (`timeget()` at line 76, `timeshow()` at line 84) is called when the server receives an appropriate message. These functions should be declared in mig definition file, and stub routines are generated by mig.

```
1 /*
2  * Demonstration of Real-Time IPC (server side)
3  */
4
5 #include <mach.h>
6 #include <mach/message.h>
7 #include <rt/rt_policy.h>
8 #include <rt/rt_thread.h>
9 #include <rt_ipc/rt_message.h>
10 #include <rt_ipc/rt_port.h>
11 #include <stdio.h>
12
13 #define IPC_BUF_SIZE    512
14 #define IPC_NUM_BUFS   5
15
16 #define PRIORITY        1
17 #define PERIOD_SEC      0    /* Aperiodic thread */
18 #define PERIOD_NSEC    0    /* Aperiodic thread */
19
20 extern          rt_ipc_server();
21 void            server();
22 mach_port_t     rt_port;
23
24 main()
25 {
26     rt_thread_attr_data_t  th_attr;
27     thread_t               thread;
```

```

28     rt_mach_port_attr_t    port_attr;
29     kern_return_t          ret;
30
31     /*
32     * Allocate Real-Time port
33     */
34     port_attr.size = IPC_BUF_SIZE;
35     port_attr.nbufs = IPC_NUM_BUFS;
36     port_attr.policy.dispatch = DISP_FIFO;
37     port_attr.policy.prio_inherit = PRI_BPI;
38     port_attr.policy.prio_handoff = HANDOFF_ON;
39     ret = rt_mach_port_allocate(mach_task_self(), MACH_PORT_RIGHT_RECEIVE,
40                               &port_attr, &rt_port);
41     if (ret != KERN_SUCCESS) {
42         printf("%s allocating volserver port\n", mach_error_string(ret));
43         exit(1);
44     }
45
46     /*
47     * Register Port
48     */
49     ret = netname_check_in(name_server_port, "rt-port",
50                           MACH_PORT_NULL, rt_port);
51     if (ret != KERN_SUCCESS) {
52         printf("%s registering server port\n", mach_error_string(ret));
53         exit(1);
54     }
55
56     /*
57     * Create Server Thread
58     */
59     ret = rt_thread_attribute_init(PRIORITY, PERIOD_SEC, PERIOD_NSEC,
60                                   server, NULL, NULL, &th_attr);
61     /* Initialize thread structure */
62     if (ret != KERN_SUCCESS) {
63         printf("Initializing thread attribute failed\n");
64         exit(1);
65     }
66
67     ret = rt_thread_create(mach_task_self(), &thread, &th_attr);
68     /* Create Real-Time thread */
69     if (ret != KERN_SUCCESS) {
70         printf("Creating real-time thread");
71         exit(1);
72     }
73
74     /*
75     * Root thread Sleeps here
76     */
77     sleep(20);
78
79     /*
80     * Bye
81     */
82     exit(0);
83 }
84
85 /* Server thread */

```

```

86 void
87 server()
88 {
89     kern_return_t ret;
90
91     ret = rt_mach_msg_server(rt_ipc_server, IPC_BUF_SIZE, rt_port);
92     printf("%s in server\n", mach_error_string(ret));
93
94     exit(-1);
95 }
96
97 kern_return_t
98 timeget(mach_port_t port, int *get_time)
99 {
100     time(get_time);                /* get time */
101
102     return KERN_SUCCESS;
103 }
104
105 kern_return_t
106 timeshow(mach_port_t port)
107 {
108     int    show_time;
109
110     time(&show_time);              /* get time */
111
112     printf("Server: %s\n", ctime(&show_time));    /* display time */
113
114     return KERN_SUCCESS;
115 }

```

10.2 Client Program

In the client program, the `main()` function obtains a real-time port from the name server (line 33), then creates a periodic thread. To send messages to the server, it calls `mig stub` routines (`timeget()` line 76, `timeshow()` line 84).

```

1 /*
2  * Demonstration of Real-Time IPC (client side)
3  */
4
5 #include <mach.h>
6 #include <rt/rt_thread.h>
7 #include <mach/message.h>
8 #include <rt_ipc/rt_message.h>
9 #include <rt_ipc/rt_port.h>
10 #include "rt_ipc.h"
11 #include <stdio.h>
12
13 #define IPC_BUF_SIZE    512
14 #define IPC_NUM_BUFS   5
15
16 #define PRIORITY        1
17 #define PERIOD_SEC      1
18 #define PERIOD_NSEC     0
19
20 void                    client();

```

```

21 mach_port_t      rt_port;
22
23 main()
24 {
25     rt_thread_attr_data_t  th_attr;
26     thread_t               thread;
27     rt_mach_port_attr_t    port_attr;
28     kern_return_t          ret;
29
30     /*
31      * Look up Real-Time port.
32      */
33     ret = netname_look_up(name_server_port, "*", "rt-port", &rt_port);
34     if (ret != KERN_SUCCESS) {
35         printf("Failed to get Real-Time port.\n");
36         exit(1);
37     }
38
39     /*
40      * Create Client Thread
41      */
42     ret = rt_thread_attribute_init(PRIORITY, PERIOD_SEC, PERIOD_NSEC,
43                                   client, NULL, NULL, &th_attr);
44     /* Initialize thread structure */
45     if (ret != KERN_SUCCESS) {
46         printf("Initializing thread attribute failed\n");
47         exit(1);
48     }
49
50     ret = rt_thread_create(mach_task_self(), &thread, &th_attr);
51     /* Create Real-Time thread */
52     if (ret != KERN_SUCCESS) {
53         printf("Creating real-time thread");
54         exit(1);
55     }
56
57     /*
58      * Root thread Sleeps here
59      */
60     sleep(10);
61
62     /*
63      * Bye
64      */
65     exit(0);
66 }
67
68 /* Client thread */
69 void
70 client()
71 {
72     int             get_time;
73     kern_return_t   ret;
74
75
76     ret = timeget(rt_port, &get_time);      /* Send request to get time */
77     if (ret != KERN_SUCCESS) {
78         printf("timeget request failed %s\n", mach_error_string(ret));

```

```

79         exit(1);
80     }
81
82     printf("Client: %s\n", ctime(&get_time));        /* display time */
83
84     ret = timeshow(rt_port);        /* Send request to display time */
85     if (ret != KERN_SUCCESS) {
86         printf("timeshow request failed %s\n", mach_error_string(ret));
87         exit(1);
88     }
89 }

```

10.3 MIG Definition File

The real-time version of the mig definition file is almost same as the original Mach MIG definition file. `rtroutine` and `simplertroutine` indicate the use of real-time IPC where `routine` and `simpleroutine` are used for normal IPC. `rtroutine` is used for synchronous communication, and `simplertroutine` is asynchronous.

```

1 /*
2  * Demonstration of Real-Time IPC (Mig file)
3  */
4
5 subsystem rt_ipc 51000;
6
7 import <mach.h>;
8 import <mach/message.h>;
9
10 #include <mach/std_types.defs>
11
12 rtoutine timeget(
13     port          :      mach_port_t;
14     out get_time  :      int
15 );
16
17 simplertroutine timeshow(
18     port          :      mach_port_t
19 );

```

Chapter 11

Device Management

This section shows how to create programs using the Mach device interface.

11.1 Device

11.1.1 Console

This program demonstrates the use of a console device. The program reads a character from console and writes the same character to console. *device_open* opens a device, and *device_read_inband* and *device_write_inband* read and write data to and from the device.

```
1 /*
2  * Demonstration of device interface
3  */
4
5 #include <mach.h>
6 #include <device/device_types.h>
7 #include <stdio.h>
8
9 char    char_getchar(mach_port_t console);
10
11 main()
12 {
13     mach_port_t    console;
14     char           c;
15     kern_return_t  rc;
16
17     while(c = char_getchar(console)) {
18         char_putchar(console, c);
19     }
20 }
21
22 init_console(mach_port_t *console)
23 {
24     kern_return_t  rc;
25
26     /*
27     * Open console device
28     */
29
30     rc = device_open(mach_master_device_port(), D_READ|D_WRITE,
31                     "console", console);
```

```

32
33     if (rc != KERN_SUCCESS) {
34         printf("console_init : console open error");
35         exit(1);
36     }
37 }
38
39 char_putchar(mach_port_t console, char c)
40 {
41     int    cw;
42
43     /*
44      * Write to console
45      */
46     device_write_inband(console, NULL, NULL, &c, 2, &cw);
47
48     if(cw == 0) {
49         return(-1);
50     } else {
51         return(c);
52     }
53 }
54
55 char
56 char_getchar(mach_port_t console)
57 {
58     char    c;
59     int     count;
60
61     /*
62      * Read from console
63      */
64     device_read_inband(console, NULL, NULL, IO_INBAND_MAX, &c, &count);
65
66     return(c);
67 }

```

11.1.2 Floppy Disk

The following example shows the use of accessing floppy disk. This example uses *device_read* and *device_write*. These functions use virtual copy, and *device_read_inband* and *device_write_inband* copy actually.

```

1 /*
2  * Demonstration of device interface
3  */
4
5 #include <mach.h>
6 #include <device/device_types.h>
7 #include <stdio.h>
8
9 #define SIZE    512
10
11 char write_buffer[SIZE];
12
13 device_t    disk_open(char *name);
14 void        disk_close(device_t dev);
15 void        disk_read(device_t dev, int blkno, int size,
16                    vm_address_t *addr);

```

```

17 void          disk_write(device_t dev, int blkno, int size,
18                vm_address_t addr);
19
20
21 main()
22 {
23     device_t    dev;
24     char        *read_buf;
25     int         i;
26
27     dev = disk_open("fd1b");
28
29     for (i = 0; i < 10; i++) {
30
31         /* Write to Floppy disk */
32         /* Contents of 'write_buffer' is written */
33         disk_write(dev, i, SIZE, (vm_address_t)write_buffer);
34
35         /* Read from Floppy disk */
36         /* Read buffer is automatically allocated
37            and return the address in 'read_buf' */
38         disk_read(dev, i, SIZE, (vm_address_t *)&read_buf);
39     }
40
41     disk_close(dev);
42 }
43
44
45 device_t
46 disk_open(char *name)
47 {
48     device_t    dev;
49     kern_return_t  ret;
50
51     /*
52      *      Open Floppy disk
53      */
54     printf("Opening Floppy Disk\n");
55     ret = device_open(mach_master_device_port(),
56                     D_READ|D_WRITE, name, &dev);
57     if (ret != KERN_SUCCESS) {
58         printf("device_open : %s\n", mach_error_string(ret));
59         exit(1);
60     }
61
62     return dev;
63 }
64
65 void
66 disk_close(device_t dev)
67 {
68     kern_return_t  ret;
69
70     /*
71      *      Close Floppy disk
72      */
73     printf("Closing Floppy Disk\n");
74     ret = device_close(dev);

```

```

75         if (ret != KERN_SUCCESS) {
76             printf("device_close : %s\n", mach_error_string(ret));
77             exit(1);
78         }
79     }
80
81 void
82 disk_read(device_t dev, int blkno, int size, vm_address_t *addr)
83 {
84     kern_return_t    ret;
85     unsigned int     count;
86
87     /*
88      *      Read from Floppy disk
89      */
90     printf("Reading from Floppy Disk\n");
91     ret = device_read(dev, D_READ, (recnum_t)blkno, size,
92                     (io_buf_ptr_t *) addr, &count);
93     if ((ret != KERN_SUCCESS) || (size != count)) {
94         printf("device_read : %s\n", mach_error_string(ret));
95         printf("block = %d, size = %d, count = %d\n",
96             blkno, size, count);
97         disk_close(dev);
98         exit(1);
99     }
100 }
101
102 void
103 disk_write(device_t dev, int blkno, int size, vm_address_t addr)
104 {
105     kern_return_t    ret;
106     unsigned int     count;
107
108     /*
109      *      Write from Floppy disk
110      */
111     printf("Writing to Floppy Disk\n");
112     ret = device_write(dev, NULL, (recnum_t)blkno,
113                     (io_buf_ptr_t)addr, size, &count);
114     if ((ret != KERN_SUCCESS) || (size != count)) {
115         printf("device_write : %s\n", mach_error_string(ret));
116         printf("block = %d, size = %d, count = %d\n",
117             blkno, size, count);
118         disk_close(dev);
119         exit(1);
120     }
121 }

```

11.2 Name Space for Device

The names for identifying devices are different from the names in Unix. The following is the list of the names which is frequently used in Gateway 2000.

IDE Disk hd0a, hd0d, hd0e

SCSI Disk sd0a, sd0d, sd0e

Floppy Disk fd0b, fd1b

Ethernet e10

Chapter 12

Pro Audio Spectrum 16 and Sound Blaster

12.0.1 The PAS-16

The Pro Audio Spectrum 16 is manufactured by MediaVision. Two PAS-16 cards can be used to obtain full-duplex support. SoundBlaster Vibra-16 chips configured from the BIOS are also supported with full-duplex support.

The correct jumper configuration for a PAS-16 for version MK83g of RealTime Mach is: J2, no jumpers; J10, 0x220; J12, 5-6; J11, 2-3. This may change in future versions; consult the version release notes and HWConfig notes distributed with your version for more information.

In the following, all information applicable to PAS is generally applicable to SoundBlaster Vibra-16 support as well. If you are using a SoundBlaster, replace all occurrences of `pas_` with `sb_`.

12.0.2 Running the PAS-16 utilities

The Device Server

A portion of the PAS-16 driver is implemented at the user-level. This is known as the “PAS server.” To run this server:

1. Be sure the nameserver (`snameS`) is running.
2. Ensure that the PAS-16 server is not already running. Running multiple copies of the server can have unpredictable results.
3. Start the server (`pas_serv`). If you are running UX, you will probably want to background it.
4. When you wish to stop an existing server from running, be sure to give it a chance to clean up shared memory state with the kernel. If you are running UX, this can be accomplished by killing the server with a standard signal such as HUP, INT, or TERM, and avoiding the use of SIGKILL (9).

PAS-16 Clients

sbu `sbu` is a simple client for reading data from files and sending this data to the PAS-16 server to be played back. It takes up to three arguments. The first (and only required) argument is the name of a file to read. `sbu` does not recognize soundfile headers- it simply plays back all of the data it finds in the file. The second argument, if present, is the sampling rate (in samples per second) at which the file should be played. The third argument, if present, is the volume (0..31) at which to play the file. If the environment variable `PASBITS` is set to 16, then the file is treated as containing 16-bit samples. Otherwise, it is assumed to contain 8-bit samples. `sbu` blocks until the file has finished playing.

sbi `sbi` is the input equivalent of `sbu`. Like `sbu`, the first three arguments (only the first is mandatory) are a filename, a sampling rate, and a volume. An optional fourth argument is a block size to record with (defaults to 8192 bytes). The `PASBITS` environment variable again specifies 8 or 16 bits; in its absence, a default of 8 bits is used. `sbi` records continuously into the file until killed (as with `^C` under UX, for example).

sbr `sbr` sends a sampling-rate change command to the server. It takes one argument- the new sampling rate, which it sets immediately. This can be used to change the sampling rate dynamically while `sbu` or `sbi` are running, for example.

pas_reset `pas_reset` resets the state of the user-level server, in-kernel device driver, and the PAS-16 hardware. This operation will disrupt any PAS-16 operations in progress.

passpeed `passpeed` provides a simple motif slider which can be used to adjust the sampling rate that the PAS-16 PCM engine is using. This can be used even while other PAS-16 clients are performing input or output operations to affect the results of these operations.

pasmix `pasmix` exports the “mixer” functionality of the PAS-16 MV501 chip to the user in a simple motif window. This “mixer” functionality is not a true mixer, but instead a set of volume controls for the various internal digital and analog paths within the MV101 pcm circuitry. This functionality is exported in the form of six vertical sliders. The first (“Output”) is the primary output volume, which can be set on a scale of 0-31. the second (“Input”) performs the same function on the input channel, again on a 0-31 scale. The next two sliders, “Treble” and “Bass,” act like similarly labelled controls on inexpensive stereo systems. These controls can be set on a 0-12 scale. Finally, there are two additional “master” volume controls, one each for the input and output channels. The first, “M. Out,” adjusts the output volume on a 0-63 scale. The second, “M. In,” adjusts the input volume on a 0-15 scale. Like `passpeed`, `passpec`’s sliders may be adjusted while other clients are performing input or output operations to affect the results.

passpec `passpec` is a simple demonstration program that reads data from the PAS-16 microphone and uses FFTs to display the relative volume levels in 32 frequency bands. The display is in an X-window.

moff `moff` turns off the PAS-16 microphone. This is useful when debugging new PAS-16 clients that exit unexpectedly, and may leave the PAS-16 in a state where the microphone and output speaker are both enabled, producing unpleasant feedback effects.

Chapter 13

Display Screen (DS) Library

This section illustrates the use of the Real-Time Mach Display Screen (DS) Library. The library provides routines for drawing graphics including lines, circles, bitmaps, and text fonts.

13.1 Program Overview

The program creates four periodic threads. Each thread moves a circle around on the screen. All of the threads have the same period which is 100 ms; the units used in the program are nanoseconds, so we see 100000000 ns.

All of the threads use the same function, called `circle_thread` to do their work. When a periodic thread invokes its associated function, an argument specified when the thread was created is passed during the call. In this example, the argument is used to allow the `circle_thread` function and then the `move_circle` function to determine which circle data to use for the move operation.

The program uses a clock device and a timer to limit the duration of execution. In this case, the main thread sleeps for 20 seconds and then exits, automatically killing the real-time threads it created.

13.2 Program Components

lines 5-16 These includes files are need to define the Mach data structures and calls.

lines 32-46 The tables `thread_period`, `thread_function`, and `thread_arg` give the information necessary to create the real-time threads. This information includes the period, the routine to run during each period, and the argument to pass to the routine for each thread.

lines 55-126 The main program initializes the DS library, initializes a timer that will be used to control the duration of the program, creates the periodic real-time threads, and waits for the timer to expire.

lines 68-74 Initialize the DS library and set the background pattern on the screen. Initialize the circle data structures that control the circle drawing.

lines 81-90 Set up the timer that will be used to limit the duration of this program.

lines 97-114 Create periodic real-time threads based on timing constraints specified in lines 32-46.

lines 119-123 Sleep on the timer for 20 seconds.

lines 131-135 Definition of the threads' function. Each thread moves its associated circle (and that's all it does).

lines 145-225 These are the data structures and routines for drawing circles.

13.3 Program Text

```
1 /*
2  * Display Screen (DS) library example.
3  */
4
5 #include <mach.h>
6 #include <mach_error.h>
7 #include <device/device_types.h>
8 #include <rt/mach_clock.h>
9 #include <rt/mach_timer.h>
10 #include <rt/rt_thread.h>
11 #include <rt/timespec.h>
12 #include <rt/mach_timer.h>
13 #include <rt/thread_attribute.h>
14 #include <rt/pset_attribute.h>
15 #include <rt/sched_policy.h>
16 #include <ds/ds_Draw.h>
17
18 #include <stdio.h>
19 #include "pattern.h"      /* need this to draw the background pattern */
20
21 #define NTHREAD          4
22
23 void          move_circle();
24 void          circle_thread();
25
26 /*
27  * The following tables (thread_period, thread_function, and thread_arg)
28  * give the information necessary to create the real-time threads.  This
29  * information includes the period, the routine to run during each period,
30  * and the argument to pass to the routine for each thread.
31  */
32 timespec_t    thread_period[NTHREAD]          = {{0,100000000}, /* ns */
33                                                  {0,100000000},
34                                                  {0,100000000},
35                                                  {0,100000000},
36                                                  };
37 void          (*thread_function[NTHREAD])()   = {circle_thread,
38                                                  circle_thread,
39                                                  circle_thread,
40                                                  circle_thread,
41                                                  };
42 long          thread_arg[NTHREAD]             = {0,
43                                                  1,
44                                                  2,
45                                                  3,
46                                                  };
47 thread_t      threads[NTHREAD];              /* thread handles */
48
49
50 /*
51  * The main program initializes the DS library, initializes a timer that
52  * will be used to control the duration of the program, creates the
53  * periodic real-time threads, and waits for the timer to expire.
54  */
55 main()
```

```

56 {
57     rt_thread_attr_data_t thattr;
58     timespec_t duration;
59     mach_clock_t clock;
60     mach_timer_t timer;
61     kern_return_t ret;
62     int i;
63
64     /*
65      * initialize the DS library and set the background pattern on the
66      * screen.
67      */
68     ds_DrawInit(0);
69     ds_DrawBackgroundPattern(Black, BlackWidth, BlackHeight);
70
71     /*
72      * initialize the circle data structures that control the circle drawing.
73      */
74     init_circle();
75
76     /*
77      * set up the timer that will be used to limit the duration of this
78      * program.
79      */
80     /* get a port on the default clock */
81     ret = clock_get_port(mach_host_self(), &clock);
82     if (ret != KERN_SUCCESS) {
83         fatalerror("ERROR: could not get clock port (%s).\n",
84                   mach_error_string(ret));
85     }
86
87     /* create a timer associated with the clock device */
88     if ((ret = timer_create(current_task(), &timer,
89                             clock)) != KERN_SUCCESS) {
90         fatalerror("timer create failed: %d\n", ret);
91     }
92
93     /*
94      * create periodic real-time threads based on timing constraints
95      * specified above.
96      */
97     for (i = 0; i < NTHREAD; i++) {
98         /* set up thread attributes to pass to rt_thread_create */
99         if ((ret = rt_thread_attribute_init(1,
100
101
102
103
104
105             fatalerror("rt_thread_attribute_init failed: %d,%d\n",i,ret);
106         }
107
108         /* create the thread and save the thread handle in threads[i] */
109         if ((ret = rt_thread_create(mach_task_self(),
110                                     &threads[i],
111                                     &thattr)) != KERN_SUC
112             fatalerror("rt_thread_create failed:%d,%d\n",i,ret);
113     }

```

```

114     }
115
116     /*
117     * sleep on the timer for 20 seconds
118     */
119     duration.seconds = 20;
120     duration.nanoseconds = 0;
121     if ((ret = timer_sleep(timer, duration, 0)) != KERN_SUCCESS) {
122         fatalerror("timer_sleep failed : %d\n", ret);
123     }
124
125     /* terminate */
126 }
127
128 /*
129 * Each thread moves its associated circle (and that's all it does).
130 */
131 void circle_thread(id)
132 long id;
133 {
134     move_circle(id);
135 }
136
137
138 /*****
139 *
140 * data structures and routines for drawing circles
141 *
142 *****/
143 */
144
145 #define CIRCLE_RADIUS 50
146
147 /*
148 * position and direction of a circle.
149 */
150 struct ppoint {
151     int    x;
152     int    y;
153     int    hdirection;
154     int    wdirection;
155 } pcircle[NTHREAD];
156
157
158 /*
159 * Set the initial positions and directions of the circles associated with
160 * the real-time threads to arbitrary values. Draw the initial circles.
161 */
162 init_circle()
163 {
164     int i;
165
166     pcircle[0].x = 10;
167     pcircle[0].y = 20;
168     pcircle[0].hdirection = 1;
169     pcircle[0].wdirection = 1;
170
171     pcircle[1].x = SCREEN_WIDTH - 103;

```

```

172     pcircle[1].y = 40;
173     pcircle[1].hdirection = -1;
174     pcircle[1].wdirection = 1;
175
176     pcircle[2].x = 70;
177     pcircle[2].y = SCREEN_HEIGHT - 23;
178     pcircle[2].hdirection = 1;
179     pcircle[2].wdirection = -1;
180
181     pcircle[3].x = SCREEN_WIDTH - 2;
182     pcircle[3].y = 88;
183     pcircle[3].hdirection = -1;
184     pcircle[3].wdirection = -1;
185
186     for(i = 0; i < NTHREAD; i++){
187         ds_DrawCircle(pcircle[i].x, pcircle[i].y, CIRCLE_RADIUS, WHITE);
188     }
189 }
190
191
192 /*
193  * Move the circle associated with one of the threads (indicated by the
194  * argument i). First, black out the previous circle, then calculate the
195  * new x and y positions based on the current direction and a velocity
196  * factor related to the thread number, i. Then draw the circle in the new
197  * position.
198  */
199 void move_circle(i)
200     int     i;
201 {
202     /* erase the old circle */
203     ds_DrawCircle(pcircle[i].x, pcircle[i].y, CIRCLE_RADIUS, BLACK);
204
205     /* calculate the new x; keep it within the bounds of the screen */
206     pcircle[i].x = pcircle[i].x + ((i+1) * 7 * pcircle[i].wdirection);
207     if (pcircle[i].x <= CIRCLE_RADIUS) {
208         pcircle[i].wdirection = 1;
209     }
210     else if (pcircle[i].x > SCREEN_WIDTH - CIRCLE_RADIUS) {
211         pcircle[i].wdirection = -1;
212     }
213
214     /* calculate the new y; keep it within the bounds of the screen */
215     pcircle[i].y = pcircle[i].y + ((i+1) * 7 * pcircle[i].hdirection);
216     if (pcircle[i].y <= CIRCLE_RADIUS) {
217         pcircle[i].hdirection = 1;
218     }
219     else if (pcircle[i].y >= SCREEN_HEIGHT - CIRCLE_RADIUS) {
220         pcircle[i].hdirection = -1;
221     }
222
223     /* draw the new circle */
224     ds_DrawCircle(pcircle[i].x, pcircle[i].y, CIRCLE_RADIUS, WHITE);
225 }
226
227
228 /*
229  * Print an error message and exit.

```

```
230 */
231 fatalerror(f, i1, i2, i3, i4)
232     char *f;
233     int i1, i2, i3, i4;
234 {
235     fprintf(stderr, f, i1, i2 , i3, i4);
236     exit(1);
237 }
```

Chapter 14

Timeline and Fault-Tolerant Scheduling

This work was carried out by the FORTS project at the University of Pittsburgh. The system calls used for Fault-tolerant real-time scheduling are `fttl_thread_attribute_init` and `fttl_thread_create`. Please see the user reference manual for corresponding information. Programming samples are provided below.

14.0.1 Timeline Scheduling Example 1

```
1 #include <rt/mach_clock.h>
2 #include <rt/timespec.h>
3 #include <stdio.h>
4 #include <mach.h>
5 #include <rt/sched_policy.h>
6 #include <rt/rt_thread.h>
7 #include <rt/thread_attribute.h>
8 #include <limits.h>
9
10 rt_thread_attr_data_t thread_attr;
11 mach_port_t deadline_port_one=MACH_PORT_NULL;
12
13
14 void tl_thread_simple(int *arg)
15 {
16     printf("Simple thread #%d\n", (int)arg);
17 }
18
19
20 /*
21  * this is called when the thread misses a deadline
22  * see the call to rt_thread_deadline_handler
23  * too see where call is generated
24  */
25 void deadlinefunc(timespec_t time, thread_t thread, int *arg)
26 {
27     printf("timeline thread %d missed it's deadline\n", (int)arg);
28     printf("At time = %d %d\n", time.seconds, time.nanoseconds);
29 }
30
31 void gettime(timespec_t *time)
32 {
33     mach_clock_t clock_port;
34     kern_return_t ret;
35
```

```

36     /* get a port on the default clock */
37     ret = clock_get_port(mach_host_self(), &clock_port);
38     if (ret != KERN_SUCCESS) {
39         printf("ERROR: could not get clock port.\n");
40         exit(-3);
41     }
42
43     ret = clock_get_time(clock_port, time);
44     if (ret != KERN_SUCCESS) {
45         printf("ERROR: got %s from clock_get_time\n",
46             mach_error_string(ret));
47         exit(-5);
48     }
49 }
50
51 main(int argc, char **argv)
52 {
53     thread_t new_thread;
54     timespec_t cur_time;
55     timespec_t st_time;
56     timespec_t ex_time;
57     timespec_t de_time;
58     kern_return_t ret;
59     int x;
60
61     /* set the scheduling policy to timeline */
62
63     ret = rt_set_scheduling_policy(SCHED_POLICY_TIMELINE);
64     if (ret != KERN_SUCCESS) {
65         printf("ERROR: got %s setting the scheduling policy\n",
66             mach_error_string(ret));
67         exit(-3);
68     }
69
70     /* get the current time */
71     gettime(&cur_time);
72     printf("the current time: seconds = %d, nanoseconds = %d\n",
73         cur_time.seconds, cur_time.nanoseconds);
74
75     /* the first thread will run 5 seconds from now and we give it an
76     exec time of 5 seconds (which it will not use) */
77     st_time.seconds = cur_time.seconds + 5;
78     st_time.nanoseconds = 0;
79     ex_time.seconds = 5;
80     ex_time.nanoseconds = 0;
81     de_time.seconds = cur_time.seconds + 10;
82     de_time.nanoseconds = 0;
83     ret = tl_thread_attribute_init(
84         st_time,          /* ready */
85         ex_time,         /* exec_time */
86         de_time,        /* deadline */
87         tl_thread_simple, /* entry point */
88         1,              /* argument */
89         &deadline_port_one, /* port */
90         &thread_attr);
91
92     if (ret != KERN_SUCCESS) {
93         printf("Got %s from tl_thread_attribute_init\n",

```

```

94                                     mach_error_string(ret));
95         exit(-23);
96     }
97
98     /* create and run the thread */
99     ret = tl_thread_create(mach_task_self(), &new_thread, &thread_attr);
100
101     if (ret != KERN_SUCCESS) {
102         printf("Got %s from tl_thread_create\n",
103             mach_error_string(ret));
104         exit(-23);
105     } else {
106         printf("Thread one created.\n");
107     }
108
109     ret = rt_thread_deadline_handler(
110         &new_thread,
111         &thread_attr,
112         deadlinefunc,
113         1 );
114
115     if (ret != KERN_SUCCESS) {
116         printf("Got %s from rt_thread_deadline_handler\n",
117             mach_error_string(ret));
118         exit(-23);
119     }
120
121     sleep(60); /* if we are alive longer than a minute - just die... */
122     exit(0);
123 }

```

14.0.2 Timeline Scheduling Example 2

```

1 #include <rt/mach_clock.h>
2 #include <rt/timespec.h>
3 #include <stdio.h>
4 #include <mach.h>
5 #include <rt/sched_policy.h>
6 #include <rt/rt_thread.h>
7 #include <rt/thread_attribute.h>
8 #include <limits.h>
9
10 rt_thread_attr_data_t thread_attr;
11
12 mach_port_t deadline_port_one=MACH_PORT_NULL;
13 mach_port_t deadline_port_two=MACH_PORT_NULL;
14 mach_port_t deadline_port_three=MACH_PORT_NULL;
15 mach_port_t deadline_port_four=MACH_PORT_NULL;
16
17
18 void tl_thread_simple(int *arg)
19 {
20     printf("Simple thread #%d\n", (int)arg);
21 }
22
23
24 void tl_thread(int *arg)
25 {

```

```

26     printf("Thread #%d will busy wait and blow it's deadline... \n",
27            (int)arg);
28     while(1);
29 }
30
31
32 void deadlinefunc(timespec_t time, thread_t thread, int *arg)
33 {
34     printf("timeline thread %d missed it's deadline\n", (int)arg);
35     printf("At time = %d %d\n", time.seconds, time.nanoseconds);
36 }
37
38
39 void gettime(timespec_t *time)
40 {
41     mach_clock_t clock_port;
42     kern_return_t ret;
43
44     /* get a port on the default clock */
45     ret = clock_get_port(mach_host_self(), &clock_port);
46     if (ret != KERN_SUCCESS) {
47         printf("ERROR: could not get clock port.\n");
48         exit(-3);
49     }
50
51     ret = clock_get_time(clock_port, time);
52     if (ret != KERN_SUCCESS) {
53         printf("ERROR: got %s from clock_get_time\n",
54                mach_error_string(ret));
55         exit(-5);
56     }
57 }
58
59
60 main(int argc, char **argv)
61 {
62     thread_t new_thread;
63     timespec_t cur_time;
64     timespec_t st_time;
65     timespec_t ex_time;
66     timespec_t de_time;
67     kern_return_t ret;
68     int x;
69
70     /* set the scheduling policy to timeline */
71     ret = rt_set_scheduling_policy(SCHED_POLICY_TIMELINE);
72     if (ret != KERN_SUCCESS) {
73         printf("ERROR: got %s setting the scheduling policy\n",
74                mach_error_string(ret));
75         exit(-3);
76     }
77
78     /* get the current time */
79     gettime(&cur_time);
80     printf("the current time: seconds = %d, nanoseconds = %d\n",
81            cur_time.seconds, cur_time.nanoseconds);
82
83     /* the first thread will run 5 seconds from now and we give it an

```

```

84     exec time of 5 seconds (which it will not use) */
85     st_time.seconds      = cur_time.seconds + 5;
86     st_time.nanoseconds  = 0;
87     ex_time.seconds      = 5;
88     ex_time.nanoseconds  = 0;
89     de_time.seconds      = cur_time.seconds + 10;
90     de_time.nanoseconds  = 0;
91     ret = tl_thread_attribute_init(
92         st_time,          /* ready */
93         ex_time,          /* exec_time */
94         de_time,          /* deadline */
95         tl_thread_simple, /* entry point */
96         1,                /* argument */
97         &deadline_port_one, /* port */
98         &thread_attr);
99
100    if (ret != KERN_SUCCESS) {
101        printf("Got %s from tl_thread_attribute_init\n",
102            mach_error_string(ret));
103        exit(-23);
104    }
105
106    /* create and run the thread */
107    ret = tl_thread_create(mach_task_self(), &new_thread, &thread_attr);
108
109    if (ret != KERN_SUCCESS) {
110        printf("Got %s from tl_thread_create\n",
111            mach_error_string(ret));
112        exit(-23);
113    } else {
114        printf("Thread one created.\n");
115    }
116
117    /* the second thread will run 20 seconds from now and we give it an
118    exec time of 5 seconds (which it will not use) */
119    st_time.seconds      = cur_time.seconds + 20;
120    st_time.nanoseconds  = 0;
121    ex_time.seconds      = 5;
122    ex_time.nanoseconds  = 0;
123    de_time.seconds      = cur_time.seconds + 25;
124    de_time.nanoseconds  = 0;
125    ret = tl_thread_attribute_init(
126        st_time,          /* ready */
127        ex_time,          /* exec_time */
128        de_time,          /* deadline */
129        tl_thread_simple, /* entry point */
130        2,                /* argument */
131        &deadline_port_two, /* port */
132        &thread_attr);
133
134    if (ret != KERN_SUCCESS) {
135        printf("Got %s from tl_thread_attribute_init\n",
136            mach_error_string(ret));
137        exit(-23);
138    }
139
140    /* create and run the thread */
141    ret = tl_thread_create(mach_task_self(), &new_thread, &thread_attr);

```

```

142
143     if (ret != KERN_SUCCESS) {
144         printf("Got %s from tl_thread_create\n",
145             mach_error_string(ret));
146         exit(-23);
147     } else {
148         printf("Thread two created.\n");
149     }
150
151
152     /* the third thread will run 14 seconds from now and we give it an
153     exec time of 4 seconds (which it will not use) */
154     st_time.seconds      = cur_time.seconds + 14;
155     st_time.nanoseconds = 0;
156     ex_time.seconds     = 4;
157     ex_time.nanoseconds = 0;
158     de_time.seconds     = cur_time.seconds + 24;
159     de_time.nanoseconds = 0;
160     ret = tl_thread_attribute_init(
161         st_time,
162         ex_time,
163         de_time,
164         tl_thread_simple,
165         3,
166         &deadline_port_three,
167         &thread_attr);
168
169     if (ret != KERN_SUCCESS) {
170         printf("Got %s from tl_thread_attribute_init\n",
171             mach_error_string(ret));
172         exit(-23);
173     }
174
175     ret = tl_thread_create(mach_task_self(), &new_thread, &thread_attr);
176
177     if (ret != KERN_SUCCESS) {
178         printf("Got %s from tl_thread_create\n",
179             mach_error_string(ret));
180         exit(-23);
181     } else {
182         printf("Thread three created.\n");
183     }
184
185
186
187     /* the last thread will run 10 seconds from now
188     even though it has a ready time in 3 seconds
189     because we give it an exec time of 4 seconds (which it WILL use) */
190     st_time.seconds      = cur_time.seconds + 3;
191     st_time.nanoseconds = 0;
192     ex_time.seconds     = 4;
193     ex_time.nanoseconds = 0;
194     de_time.seconds     = cur_time.seconds + 30;
195     de_time.nanoseconds = 0;
196     ret = tl_thread_attribute_init(
197         st_time,
198         ex_time,
199         de_time,

```

```

200             tl_thread,
201             4,
202             &deadline_port_four,
203             &thread_attr);
204
205     if (ret != KERN_SUCCESS) {
206         printf("Got %s from tl_thread_attribute_init\n",
207             mach_error_string(ret));
208         exit(-23);
209     }
210
211     /* create and run the thread */
212     ret = tl_thread_create(mach_task_self(), &new_thread, &thread_attr);
213
214     if (ret != KERN_SUCCESS) {
215         printf("Got %s from tl_thread_create\n",
216             mach_error_string(ret));
217         exit(-23);
218     } else {
219         printf("Thread four created.\n");
220     }
221
222
223
224
225
226
227     ret = rt_thread_deadline_handler(
228         &new_thread,
229         &thread_attr,
230         deadlinefunc,
231         4 );
232
233     if (ret != KERN_SUCCESS) {
234         printf("Got %s from rt_thread_deadline_handler\n",
235             mach_error_string(ret));
236         exit(-23);
237     }
238
239
240
241     sleep(60); /* if we are alive longer than a minute - just die... */
242     exit(0);
243 }

```

14.0.3 Timeline Scheduling Example 3

```

1 #include <rt/mach_clock.h>
2 #include <rt/timespec.h>
3 #include <stdio.h>
4 #include <mach.h>
5 #include <rt/sched_policy.h>
6 #include <rt/rt_thread.h>
7 #include <rt/thread_attribute.h>
8 #include <limits.h>
9
10 rt_thread_attr_data_t thread_attr;
11

```

```

12 mach_port_t deadline_port_one=MACH_PORT_NULL;
13 mach_port_t deadline_port_two=MACH_PORT_NULL;
14 mach_port_t deadline_port_three=MACH_PORT_NULL;
15
16
17
18 void tl_thread_simple(int *arg)
19 {
20     printf("Simple thread #%d\n", (int)arg);
21 }
22
23
24
25 void deadlinefunc(timespec_t time, thread_t thread, int *arg)
26 {
27     printf("timeline thread %d missed it's deadline\n", (int)arg);
28     printf("At time = %d %d\n", time.seconds, time.nanoseconds);
29 }
30
31
32
33 void gettime(timespec_t *time)
34 {
35     mach_clock_t clock_port;
36     kern_return_t ret;
37
38     /* get a port on the default clock */
39     ret = clock_get_port(mach_host_self(), &clock_port);
40     if (ret != KERN_SUCCESS) {
41         printf("ERROR: could not get clock port.\n");
42         exit(-3);
43     }
44
45     ret = clock_get_time(clock_port, time);
46     if (ret != KERN_SUCCESS) {
47         printf("ERROR: got %s from clock_get_time\n",
48             mach_error_string(ret));
49         exit(-5);
50     }
51 }
52
53
54
55 main(int argc, char **argv)
56 {
57     thread_t new_thread;
58     timespec_t cur_time;
59     timespec_t st_time;
60     timespec_t ex_time;
61     timespec_t de_time;
62     kern_return_t ret;
63     int x;
64
65     /* set the scheduling policy to timeline */
66     ret = rt_set_scheduling_policy(SCHED_POLICY_TIMELINE);
67     if (ret != KERN_SUCCESS) {
68         printf("ERROR: got %s setting the scheduling policy\n",
69             mach_error_string(ret));

```

```

70     exit(-3);
71 }
72
73 /* get the current time */
74     gettime(&cur_time);
75     printf("the current time: seconds = %d, nanoseconds = %d\n",
76           cur_time.seconds, cur_time.nanoseconds);
77
78 /* the first thread will run 5 seconds from now and we give it an
79    exec time of 5 seconds (which it will not use) */
80     st_time.seconds      = cur_time.seconds + 5;
81     st_time.nanoseconds = 0;
82     ex_time.seconds      = 5;
83     ex_time.nanoseconds = 0;
84     de_time.seconds      = cur_time.seconds + 10;
85     de_time.nanoseconds = 0;
86     ret = tl_thread_attribute_init(
87         st_time,          /* ready */
88         ex_time,          /* exec_time */
89         de_time,          /* deadline */
90         tl_thread_simple, /* entry point */
91         1,                /* argument */
92         &deadline_port_one, /* port */
93         &thread_attr);
94
95     if (ret != KERN_SUCCESS) {
96         printf("Got %s from tl_thread_attribute_init\n",
97               mach_error_string(ret));
98         exit(-23);
99     }
100
101     /* create and run the thread */
102     ret = tl_thread_create(mach_task_self(), &new_thread, &thread_attr);
103
104     if (ret != KERN_SUCCESS) {
105         printf("Got %s from tl_thread_create\n",
106               mach_error_string(ret));
107         exit(-23);
108     } else {
109         printf("Thread one created.\n");
110     }
111
112 /* the second thread will run 20 seconds from now and we give it an
113    exec time of 5 seconds (which it will not use) */
114     st_time.seconds      = cur_time.seconds + 20;
115     st_time.nanoseconds = 0;
116     ex_time.seconds      = 5;
117     ex_time.nanoseconds = 0;
118     de_time.seconds      = cur_time.seconds + 25;
119     de_time.nanoseconds = 0;
120
121     printf("Before rt_thread_attribute_init #2\n");
122     ret = rt_thread_attribute_init(
123         st_time,          /* ready */
124         ex_time,          /* exec_time */
125         de_time,          /* deadline */
126         tl_thread_simple, /* entry point */
127         2,                /* argument */

```

```

128             &deadline_port_two, /* port */
129             &thread_attr);
130 printf("After rt_thread_attribute_init #2\n");
131
132     if (ret != KERN_SUCCESS) {
133         printf("Got %s from tl_thread_attribute_init\n",
134             mach_error_string(ret));
135         exit(-23);
136     }
137
138     /* create and run the thread */
139     ret = tl_thread_create(mach_task_self(), &new_thread, &thread_attr);
140
141     if (ret != KERN_SUCCESS) {
142         printf("Got %s from tl_thread_create\n",
143             mach_error_string(ret));
144         exit(-23);
145     } else {
146         printf("Thread two created.\n");
147     }
148
149     /* the third thread will run 14 seconds from now and we give it an
150     exec time of 4 seconds (which it will not use)
151     st_time.seconds      = cur_time.seconds + 14;
152     st_time.nanoseconds = 0;
153     ex_time.seconds     = 4;
154     ex_time.nanoseconds = 0;
155     de_time.seconds     = cur_time.seconds + 24;
156     de_time.nanoseconds = 0;
157     ret = tl_thread_attribute_init(
158         st_time,
159         ex_time,
160         de_time,
161         tl_thread_simple,
162         3,
163         &deadline_port_three,
164         &thread_attr);
165
166     if (ret != KERN_SUCCESS) {
167         printf("Got %s from tl_thread_attribute_init\n",
168             mach_error_string(ret));
169         exit(-23);
170     }
171
172     ret = tl_thread_create(mach_task_self(), &new_thread, &thread_attr);
173
174     if (ret != KERN_SUCCESS) {
175         printf("Got %s from tl_thread_create\n",
176             mach_error_string(ret));
177         exit(-23);
178     } else {
179         printf("Thread three created.\n");
180     }
181
182
183     ret = rt_thread_deadline_handler(&new_thread, &thread_attr,
184         deadlinefunc, 3 );
185

```

```

186         if (ret != KERN_SUCCESS) {
187             printf("Got %s from rt_thread_deadline_handler\n",
188                   mach_error_string(ret));
189             exit(-23);
190         }
191     */
192
193     sleep(60); /* if we are alive longer than a minute - just die... */
194     exit(0);
195 }
196

```

14.0.4 Timeline Scheduling Example 4

```

1  /*
2  * tl_thread.c
3  */
4  /*
5  * RTMach example program
6  * create a timelined thread
7  */
8  /*
9  *     Created - under TL0.0.c kernel, timeline thread creation will simply
10 *     return KERN_FAILURE an no thread is ever created.
11 *     [94/08/09     epc]
12 */
13
14 #include <rt/mach_clock.h>
15 #include <rt/timespec.h>
16 #include <stdio.h>
17 #include <mach.h>
18 #include <rt/sched_policy.h>
19 #include <rt/rt_thread.h>
20 #include <rt/thread_attribute.h>
21 #include <limits.h>
22
23 /*
24     four tl_threads will be created... though they are scheduled 1,2,3,4
25     They should run 1432 due to the time constaints we used
26 */
27
28 rt_thread_attr_data_t thread_attr;
29
30 mach_port_t deadline_port_one=MACH_PORT_NULL;
31 mach_port_t deadline_port_two=MACH_PORT_NULL;
32 mach_port_t deadline_port_three=MACH_PORT_NULL;
33 mach_port_t deadline_port_four=MACH_PORT_NULL;
34
35 void tl_thread_simple(int *arg)
36 {
37     printf("Simple thread #%d\n", (int)arg);
38 }
39
40 void tl_thread(int *arg)
41 {
42     printf("Thread #%d will busy wait and blow it's deadline... \n",
43           (int)arg);
44     while(1);

```

```

45 }
46
47 /*
48  * this is called when the thread misses a deadline
49  * see the call to rt_thread_deadline_handler
50  * too see where call is generated
51  */
52 void deadlinefunc(timespec_t time, thread_t thread, int *arg)
53 {
54     printf("timeline thread %d missed it's deadline\n", (int)arg);
55     printf("At time = %d %d\n", time.seconds, time.nanoseconds);
56 }
57
58 void gettime(timespec_t *time)
59 {
60     mach_clock_t clock_port;
61     kern_return_t ret;
62
63     /* get a port on the default clock */
64     ret = clock_get_port(mach_host_self(), &clock_port);
65     if (ret != KERN_SUCCESS) {
66         printf("ERROR: could not get clock port.\n");
67         exit(-3);
68     }
69
70     ret = clock_get_time(clock_port, time);
71     if (ret != KERN_SUCCESS) {
72         printf("ERROR: got %s from clock_get_time\n",
73             mach_error_string(ret));
74         exit(-5);
75     }
76 }
77
78 main(int argc, char **argv)
79 {
80     thread_t new_thread;
81     timespec_t cur_time;
82     timespec_t st_time;
83     timespec_t ex_time;
84     timespec_t de_time;
85     kern_return_t ret;
86     int x;
87
88     /* set the scheduling policy to timeline */
89
90     ret = rt_set_scheduling_policy(SCHED_POLICY_TIMELINE);
91     if (ret != KERN_SUCCESS) {
92         printf("ERROR: got %s setting the scheduling policy\n",
93             mach_error_string(ret));
94         exit(-3);
95     }
96
97     /* get the current time */
98
99     gettime(&cur_time);
100     printf("the current time: seconds = %d, nanoseconds = %d\n",
101         cur_time.seconds, cur_time.nanoseconds);
102

```

```

103  /* the first thread will run 5 seconds from now and we give it an
104     exec time of 5 seconds (which it will not use) */
105
106     st_time.seconds      = cur_time.seconds + 5;
107     st_time.nanoseconds = 0;
108     ex_time.seconds      = 5;
109     ex_time.nanoseconds = 0;
110     de_time.seconds      = cur_time.seconds + 10;
111     de_time.nanoseconds = 0;
112     ret = tl_thread_attribute_init(
113         st_time,          /* ready */
114         ex_time,          /* exec_time */
115         de_time,          /* deadline */
116         tl_thread_simple, /* entry point */
117         1,                 /* argument */
118         &deadline_port_one, /* port */
119         &thread_attr);
120
121     if (ret != KERN_SUCCESS) {
122         printf("Got %s from tl_thread_attribute_init\n",
123             mach_error_string(ret));
124         exit(-23);
125     }
126
127     /* create and run the thread */
128     ret = rt_thread_create(mach_task_self(), &new_thread, &thread_attr);
129
130     if (ret != KERN_SUCCESS) {
131         printf("Got %s from tl_thread_create\n",
132             mach_error_string(ret));
133         exit(-23);
134     } else {
135         printf("Thread one created.\n");
136     }
137
138  /* the second thread will run 20 seconds from now and we give it an
139     exec time of 5 seconds (which it will not use) */
140
141     st_time.seconds      = cur_time.seconds + 20;
142     st_time.nanoseconds = 0;
143     ex_time.seconds      = 5;
144     ex_time.nanoseconds = 0;
145     de_time.seconds      = cur_time.seconds + 25;
146     de_time.nanoseconds = 0;
147     ret = rt_thread_attribute_init(
148         st_time,          /* ready */
149         ex_time,          /* exec_time */
150         de_time,          /* deadline */
151         tl_thread_simple, /* entry point */
152         2,                 /* argument */
153         &deadline_port_two, /* port */
154         &thread_attr);
155
156     if (ret != KERN_SUCCESS) {
157         printf("Got %s from tl_thread_attribute_init\n",
158             mach_error_string(ret));
159         exit(-23);
160     }

```

```

161
162     /* create and run the thread */
163     ret = rt_thread_create(mach_task_self(), &new_thread, &thread_attr);
164
165     if (ret != KERN_SUCCESS) {
166         printf("Got %s from tl_thread_create\n",
167             mach_error_string(ret));
168         exit(-23);
169     } else {
170         printf("Thread two created.\n");
171     }
172
173     /* the third thread will run 14 seconds from now and we give it an
174     exec time of 4 seconds (which it will not use) */
175
176     st_time.seconds      = cur_time.seconds + 14;
177     st_time.nanoseconds = 0;
178     ex_time.seconds     = 4;
179     ex_time.nanoseconds = 0;
180     de_time.seconds     = cur_time.seconds + 24;
181     de_time.nanoseconds = 0;
182     ret = tl_thread_attribute_init(
183         st_time,          /* ready */
184         ex_time,         /* exec_time */
185         de_time,         /* deadline */
186         tl_thread_simple, /* entry point */
187         3,               /* argument */
188         &deadline_port_three, /* port */
189         &thread_attr);
190
191     if (ret != KERN_SUCCESS) {
192         printf("Got %s from tl_thread_attribute_init\n",
193             mach_error_string(ret));
194         exit(-23);
195     }
196
197     /* create and run the thread */
198     ret = rt_thread_create(mach_task_self(), &new_thread, &thread_attr);
199
200     if (ret != KERN_SUCCESS) {
201         printf("Got %s from tl_thread_create\n",
202             mach_error_string(ret));
203         exit(-23);
204     } else {
205         printf("Thread three created.\n");
206     }
207
208     /* the last thread will run 10 seconds from now
209     even though it has a ready time in 3 seconds
210     because we give it an exec time of 4 seconds (which it WILL use) */
211
212     st_time.seconds      = cur_time.seconds + 3;
213     st_time.nanoseconds = 0;
214     ex_time.seconds     = 4;
215     ex_time.nanoseconds = 0;
216     de_time.seconds     = cur_time.seconds + 30;
217     de_time.nanoseconds = 0;
218     ret = tl_thread_attribute_init(

```

```

219             st_time,             /* ready */
220             ex_time,             /* exec_time */
221             de_time,             /* deadline */
222             tl_thread,           /* entry point */
223             4,                   /* argument */
224             &deadline_port_four, /* port */
225             &thread_attr);
226
227     if (ret != KERN_SUCCESS) {
228         printf("Got %s from tl_thread_attribute_init\n",
229               mach_error_string(ret));
230         exit(-23);
231     }
232
233     /* create and run the thread */
234     ret = rt_thread_create(mach_task_self(), &new_thread, &thread_attr);
235
236     if (ret != KERN_SUCCESS) {
237         printf("Got %s from tl_thread_create\n",
238               mach_error_string(ret));
239         exit(-23);
240     } else {
241         printf("Thread four created.\n");
242     }
243
244     ret = rt_thread_deadline_handler(
245         &new_thread,
246         &thread_attr,
247         deadlinefunc,
248         4
249         );
250
251     if (ret != KERN_SUCCESS) {
252         printf("Got %s from rt_thread_deadline_handler\n",
253               mach_error_string(ret));
254         exit(-23);
255     }
256
257     sleep(60); /* if we are alive longer than a minute - just die... */
258     exit(0);
259 }

```

14.0.5 Timeline Scheduling Example 5

```

1 #include <rt/mach_clock.h>
2 #include <rt/timespec.h>
3 #include <stdio.h>
4 #include <mach.h>
5 #include <rt/sched_policy.h>
6 #include <rt/rt_thread.h>
7 #include <rt/thread_attribute.h>
8 #include <limits.h>
9
10 rt_thread_attr_data_t thread_attr;
11 mach_port_t deadline_port_four=MACH_PORT_NULL;
12
13
14

```

```

15 void tl_thread_simple(int *arg)
16 {
17     printf("Simple thread #%d\n", (int)arg);
18 }
19
20
21
22 void tl_thread(int *arg)
23 {
24     printf("Thread #%d will busy wait and blow it's deadline... \n",
25           (int)arg);
26     while(1);
27 }
28
29
30
31 void deadlinefunc(timespec_t time, thread_t thread, int *arg)
32 {
33     printf("timeline thread %d missed it's deadline\n", (int)arg);
34     printf("At time = %d %d\n", time.seconds, time.nanoseconds);
35 }
36
37
38
39 void gettime(timespec_t *time)
40 {
41     mach_clock_t clock_port;
42     kern_return_t ret;
43
44     /* get a port on the default clock */
45     ret = clock_get_port(mach_host_self(), &clock_port);
46     if (ret != KERN_SUCCESS) {
47         printf("ERROR: could not get clock port.\n");
48         exit(-3);
49     }
50
51     ret = clock_get_time(clock_port, time);
52     if (ret != KERN_SUCCESS) {
53         printf("ERROR: got %s from clock_get_time\n",
54               mach_error_string(ret));
55         exit(-5);
56     }
57 }
58
59
60
61 main(int argc, char **argv)
62 {
63     thread_t new_thread;
64     timespec_t cur_time;
65     timespec_t st_time;
66     timespec_t ex_time;
67     timespec_t de_time;
68     kern_return_t ret;
69     int x;
70
71     /* set the scheduling policy to timeline */
72

```

```

73     ret = rt_set_scheduling_policy(SCHED_POLICY_TIMELINE);
74     if (ret != KERN_SUCCESS) {
75         printf("ERROR: got %s setting the scheduling policy\n",
76             mach_error_string(ret));
77         exit(-3);
78     }
79
80     /* get the current time */
81
82     gettimeofday(&cur_time);
83     printf("the current time: seconds = %d, nanoseconds = %d\n",
84         cur_time.seconds, cur_time.nanoseconds);
85
86
87     st_time.seconds      = cur_time.seconds + 3;
88     st_time.nanoseconds = 0;
89     ex_time.seconds      = 6;
90     ex_time.nanoseconds = 0;
91     de_time.seconds      = cur_time.seconds + 30;
92     de_time.nanoseconds = 0;
93     ret = tl_thread_attribute_init(
94         st_time,          /* ready */
95         ex_time,          /* exec_time */
96         de_time,          /* deadline */
97         tl_thread,        /* entry point */
98         4,                /* argument */
99         &deadline_port_four, /* port */
100        &thread_attr);
101
102     if (ret != KERN_SUCCESS) {
103         printf("Got %s from tl_thread_attribute_init\n",
104             mach_error_string(ret));
105         exit(-23);
106     }
107
108     /* create and run the thread */
109     ret = rt_thread_create(mach_task_self(), &new_thread, &thread_attr);
110
111     if (ret != KERN_SUCCESS) {
112         printf("Got %s from tl_thread_create\n",
113             mach_error_string(ret));
114         exit(-23);
115     } else {
116         printf("Thread four created.\n");
117     }
118
119     ret = rt_thread_deadline_handler(
120         &new_thread,
121         &thread_attr,
122         deadlinefunc,
123         4
124     );
125
126     if (ret != KERN_SUCCESS) {
127         printf("Got %s from rt_thread_deadline_handler\n",
128             mach_error_string(ret));
129         exit(-23);
130     }

```

```

131
132     sleep(60); /* if we are alive longer than a minute - just die... */
133     exit(0);
134 }

```

14.0.6 Fault-Tolerant Timeline Scheduling Example 1

```

1 #include <rt/mach_clock.h>
2 #include <rt/timespec.h>
3 #include <stdio.h>
4 #include <mach.h>
5 #include <rt/sched_policy.h>
6 #include <rt/rt_thread.h>
7 #include <rt/thread_attribute.h>
8 #include <limits.h>
9
10 rt_thread_attr_data_t thread_attr;
11 mach_port_t deadline_port_one=MACH_PORT_NULL;
12 thread_t new_thread;
13
14
15 /*
16  * this is called when the thread misses a deadline
17  * see the call to rt_thread_deadline_handler
18  * too see where call is generated
19  */
20 void deadlinefunc(timespec_t time, thread_t thread, int *arg)
21 {
22     int ret;
23
24     printf("timeline thread %d missed it's deadline\n", (int)arg);
25     printf("At time = %d %d\n", time.seconds, time.nanoseconds);
26
27     printf("Global thread_t = %ld, arg thread_t = %ld\n",new_thread,thread);
28
29     ret = thread_terminate(thread);
30     if (ret != KERN_SUCCESS) printf("Cannot terminate thread %d\n",thread);
31
32 }
33
34 void gettime(timespec_t *time)
35 {
36     mach_clock_t clock_port;
37     kern_return_t ret;
38
39     /* get a port on the default clock */
40     ret = clock_get_port(mach_host_self(), &clock_port);
41     if (ret != KERN_SUCCESS) {
42         printf("ERROR: could not get clock port.\n");
43         exit(-3);
44     }
45
46     ret = clock_get_time(clock_port, time);
47     if (ret != KERN_SUCCESS) {
48         printf("ERROR: got %s from clock_get_time\n",
49             mach_error_string(ret));
50         exit(-5);
51     }

```

```

52 }
53
54
55
56
57 void tl_thread_simple(int *arg)
58 {
59     printf("Simple thread #%d\n", (int) arg);
60 }
61
62
63 void tl_thread(int *arg)
64 {
65     timespec_t cur_time;
66
67     gettimeofday(&cur_time);
68     printf("Thread #%d will blow it's deadline, time = %d %d\n",
69           (int) arg, cur_time.seconds, cur_time.nanoseconds);
70     while(1);
71 }
72
73
74
75 main(int argc, char **argv)
76 {
77     timespec_t cur_time;
78     timespec_t st_time;
79     timespec_t ex_time;
80     timespec_t de_time;
81     kern_return_t ret;
82     int x;
83     timespec_t fault_int;
84
85     syscall_set_fault_flag(1);
86
87
88
89     fault_int.seconds = 100;
90     fault_int.nanoseconds = 0;
91     ret = set_fttl_scheduling_policy(fault_int);
92     if (ret != KERN_SUCCESS) {
93         printf("ERROR: got %s setting the scheduling policy\n",
94               mach_error_string(ret));
95         exit(-3);
96     }
97
98
99
100
101     gettimeofday(&cur_time);
102     printf("the current time: seconds = %d, nanoseconds = %d\n",
103           cur_time.seconds, cur_time.nanoseconds);
104
105     st_time.seconds      = cur_time.seconds + 5;
106     st_time.nanoseconds = 0;
107     ex_time.seconds     = 5;
108     ex_time.nanoseconds = 0;
109     de_time.seconds     = cur_time.seconds + 15;

```

```

110     de_time.nanoseconds = 0;
111     ret = fttl_thread_attribute_init(
112         st_time,
113         ex_time,
114         de_time,
115         tl_thread,
116         1,
117         &deadline_port_one,
118         &thread_attr);
119
120     if (ret != KERN_SUCCESS) {
121         printf("Got %s from tl_thread_attribute_init\n",
122             mach_error_string(ret));
123         exit(-23);
124     }
125
126     ret = fttl_thread_create(mach_task_self(), &new_thread, &thread_attr);
127
128     if (ret != KERN_SUCCESS) {
129         printf("Got %s from tl_thread_create\n",
130             mach_error_string(ret));
131         exit(-23);
132     } else {
133         printf("Thread one created, ID=%ld\n", new_thread);
134     }
135
136     ret = rt_thread_deadline_handler(
137         new_thread,
138         &thread_attr,
139         deadlinefunc,
140         1 );
141
142     if (ret != KERN_SUCCESS) {
143         printf("Got %s from rt_thread_deadline_handler\n",
144             mach_error_string(ret));
145         exit(-23);
146     }
147
148     sleep(20);
149     exit(0);
150
151
152 }
153

```

14.0.7 Fault-Tolerant Timeline Scheduling Example 2

```

1 #include <rt/mach_clock.h>
2 #include <rt/timespec.h>
3 #include <stdio.h>
4 #include <mach.h>
5 #include <rt/sched_policy.h>
6 #include <rt/rt_thread.h>
7 #include <rt/thread_attribute.h>
8 #include <limits.h>
9
10 rt_thread_attr_data_t thread_attr;
11

```

```

12 mach_port_t deadline_port_one=MACH_PORT_NULL;
13 mach_port_t deadline_port_two=MACH_PORT_NULL;
14 mach_port_t deadline_port_three=MACH_PORT_NULL;
15 mach_port_t deadline_port_four=MACH_PORT_NULL;
16
17 thread_t tmp_thread;
18
19 void gettime(timespec_t *time)
20 {
21     mach_clock_t clock_port;
22     kern_return_t ret;
23
24     /* get a port on the default clock */
25     ret = clock_get_port(mach_host_self(), &clock_port);
26     if (ret != KERN_SUCCESS) {
27         printf("ERROR: could not get clock port.\n");
28         exit(-3);
29     }
30
31     ret = clock_get_time(clock_port, time);
32     if (ret != KERN_SUCCESS) {
33         printf("ERROR: got %s from clock_get_time\n",
34             mach_error_string(ret));
35         exit(-5);
36     }
37 }
38
39
40
41 void tl_thread_simple(int *arg)
42 {
43     timespec_t cur_time;
44     int i;
45
46     gettime(&cur_time);
47     printf("Simple thread #%d started at seconds = %d %d\n",
48         (int)arg, cur_time.seconds, cur_time.nanoseconds);
49     for(i=0;i<100000;i++);
50     printf("Simple thread #%d finished\n", (int)arg);
51 }
52
53
54 void tl_thread(int *arg)
55 {
56     timespec_t cur_time;
57
58     gettime(&cur_time);
59     printf("Thread #%d will blow it's deadline, time = %d %d\n",
60         (int)arg, cur_time.seconds, cur_time.nanoseconds);
61     while(1);
62 }
63
64
65 void deadlinefunc(timespec_t time, thread_t thread, int *arg)
66 {
67     kern_return_t ret;
68
69     printf("timeline thread %d missed it's deadline\n", (int)arg);

```

```

70 printf("At time = %d %d\n", time.seconds, time.nanoseconds);
71
72 ret = thread_terminate(thread);
73 if (ret != KERN_SUCCESS) printf("Cannot terminate thread %d\n",thread);
74 }
75
76
77 main(int argc, char **argv)
78 {
79     thread_t new_thread;
80     timespec_t cur_time;
81     timespec_t st_time;
82     timespec_t ex_time;
83     timespec_t de_time;
84     timespec_t fint;
85     kern_return_t ret;
86     int x;
87
88     /* set the scheduling policy to timeline */
89     fint.seconds = 100;
90     fint.nanoseconds = 0;
91     ret = set_fttl_scheduling_policy(fint);
92     if (ret != KERN_SUCCESS) {
93         printf("ERROR: got %s setting the scheduling policy\n",
94             mach_error_string(ret));
95         exit(-3);
96     }
97
98     /* get the current time */
99     gettime(&cur_time);
100    printf("the current time: seconds = %d, nanoseconds = %d\n",
101        cur_time.seconds,cur_time.nanoseconds);
102
103
104
105    st_time.seconds      = cur_time.seconds + 5;
106    st_time.nanoseconds = 0;
107    ex_time.seconds     = 5;
108    ex_time.nanoseconds = 0;
109    de_time.seconds     = cur_time.seconds + 15;
110    de_time.nanoseconds = 0;
111    ret = fttl_thread_attribute_init(
112        st_time,
113        ex_time,
114        de_time,
115        tl_thread,
116        1,
117        &deadline_port_one,
118        &thread_attr);
119
120    if (ret != KERN_SUCCESS) {
121        printf("Got %s from tl_thread_attribute_init\n",
122            mach_error_string(ret));
123        exit(-23);
124    }
125
126    ret = fttl_thread_create(mach_task_self(), &new_thread, &thread_attr);
127

```

```

128     if (ret != KERN_SUCCESS) {
129         printf("Got %s from tl_thread_create\n",
130             mach_error_string(ret));
131         exit(-23);
132     } else {
133         printf("Thread one created.\n");
134     }
135
136     ret = rt_thread_deadline_handler(
137         new_thread,
138         &thread_attr,
139         deadlinefunc,
140         1 );
141
142     if (ret != KERN_SUCCESS) {
143         printf("Got %s from rt_thread_deadline_handler\n",
144             mach_error_string(ret));
145         exit(-23);
146     }
147
148     st_time.seconds      = cur_time.seconds + 10;
149     st_time.nanoseconds = 0;
150     ex_time.seconds      = 2;
151     ex_time.nanoseconds = 0;
152     de_time.seconds      = cur_time.seconds + 30;
153     de_time.nanoseconds = 0;
154     ret = fttl_thread_attribute_init(
155         st_time,
156         ex_time,
157         de_time,
158         tl_thread_simple,
159         2,
160         &deadline_port_two,
161         &thread_attr);
162
163     if (ret != KERN_SUCCESS) {
164         printf("Got %s from tl_thread_attribute_init\n",
165             mach_error_string(ret));
166         exit(-23);
167     }
168
169     ret = fttl_thread_create(mach_task_self(), &new_thread, &thread_attr);
170
171     if (ret != KERN_SUCCESS) {
172         printf("Got %s from tl_thread_create\n",
173             mach_error_string(ret));
174         exit(-23);
175     } else {
176         printf("Thread two created.\n");
177     }
178
179
180     st_time.seconds      = cur_time.seconds + 14;
181     st_time.nanoseconds = 0;
182     ex_time.seconds      = 2;
183     ex_time.nanoseconds = 0;
184     de_time.seconds      = cur_time.seconds + 24;
185     de_time.nanoseconds = 0;

```

```

186     ret = fttl_thread_attribute_init(
187         st_time,
188         ex_time,
189         de_time,
190         tl_thread_simple,
191         3,
192         &deadline_port_three,
193         &thread_attr);
194
195     if (ret != KERN_SUCCESS) {
196         printf("Got %s from tl_thread_attribute_init\n",
197             mach_error_string(ret));
198         exit(-23);
199     }
200
201     ret = fttl_thread_create(mach_task_self(), &new_thread, &thread_attr);
202
203     if (ret != KERN_SUCCESS) {
204         printf("Got %s from tl_thread_create\n",
205             mach_error_string(ret));
206         exit(-23);
207     } else {
208         printf("Thread three created.\n");
209     }
210
211     st_time.seconds      = cur_time.seconds + 15;
212     st_time.nanoseconds = 0;
213     ex_time.seconds      = 1;
214     ex_time.nanoseconds = 0;
215     de_time.seconds      = cur_time.seconds + 50;
216     de_time.nanoseconds = 0;
217     ret = fttl_thread_attribute_init(
218         st_time,
219         ex_time,
220         de_time,
221         tl_thread,
222         4,
223         &deadline_port_four,
224         &thread_attr);
225
226     if (ret != KERN_SUCCESS) {
227         printf("Got %s from tl_thread_attribute_init\n",
228             mach_error_string(ret));
229         exit(-23);
230     }
231
232     ret = fttl_thread_create(mach_task_self(), &new_thread, &thread_attr);
233
234     if (ret != KERN_SUCCESS) {
235         printf("Got %s from tl_thread_create\n",
236             mach_error_string(ret));
237         exit(-23);
238     } else {
239         printf("Thread four created.\n");
240     }
241
242     ret = rt_thread_deadline_handler(
243         new_thread,

```

```

244             &thread_attr,
245             deadlinefunc,
246             4 );
247
248     if (ret != KERN_SUCCESS) {
249         printf("Got %s from rt_thread_deadline_handler\n",
250             mach_error_string(ret));
251         exit(-23);
252     }
253
254     sleep(60); /* if we are alive longer than a minute - just die... */
255     exit(0);
256 }
257

```

14.0.8 Fault-Tolerant Timeline Scheduling Example 3

```

1 #include <rt/mach_clock.h>
2 #include <rt/timespec.h>
3 #include <stdio.h>
4 #include <mach.h>
5 #include <rt/sched_policy.h>
6 #include <rt/rt_thread.h>
7 #include <rt/thread_attribute.h>
8 #include <limits.h>
9
10 rt_thread_attr_data_t thread_attr;
11
12 mach_port_t deadline_port_one=MACH_PORT_NULL;
13 mach_port_t deadline_port_two=MACH_PORT_NULL;
14 mach_port_t deadline_port_three=MACH_PORT_NULL;
15 mach_port_t deadline_port_four=MACH_PORT_NULL;
16
17
18 void tl_thread_simple(int *arg)
19 {
20     printf("Simple thread #%d\n", (int)arg);
21 }
22
23
24 void tl_thread(int *arg)
25 {
26     printf("Thread #%d will busy wait and blow it's deadline... \n",
27         (int)arg);
28     while(1);
29 }
30
31
32 void deadlinefunc(timespec_t time, thread_t thread, int *arg)
33 {
34     printf("timeline thread %d missed it's deadline\n", (int)arg);
35     printf("At time = %d %d\n", time.seconds, time.nanoseconds);
36 }
37
38
39 void gettime(timespec_t *time)
40 {
41     mach_clock_t clock_port;

```

```

42     kern_return_t ret;
43
44     /* get a port on the default clock */
45     ret = clock_get_port(mach_host_self(), &clock_port);
46     if (ret != KERN_SUCCESS) {
47         printf("ERROR: could not get clock port.\n");
48         exit(-3);
49     }
50
51     ret = clock_get_time(clock_port, time);
52     if (ret != KERN_SUCCESS) {
53         printf("ERROR: got %s from clock_get_time\n",
54             mach_error_string(ret));
55         exit(-5);
56     }
57 }
58
59
60 main(int argc, char **argv)
61 {
62     thread_t new_thread;
63     timespec_t cur_time;
64     timespec_t st_time;
65     timespec_t ex_time;
66     timespec_t de_time;
67     kern_return_t ret;
68     int x;
69     timespec_t fint;
70
71     /* set the scheduling policy to FT timeline */
72     fint.seconds = 100;
73     fint.nanoseconds = 0;
74     ret = set_fttl_scheduling_policy(fint);
75
76     if (ret != KERN_SUCCESS) {
77         printf("ERROR: got %s setting the scheduling policy\n",
78             mach_error_string(ret));
79         exit(-3);
80     }
81
82     /* get the current time */
83     gettimeofday(&cur_time);
84     printf("the current time: seconds = %d, nanoseconds = %d\n",
85         cur_time.seconds, cur_time.nanoseconds);
86
87     /* the first thread will run 5 seconds from now and we give it an
88     exec time of 5 seconds (which it will not use) */
89     st_time.seconds = cur_time.seconds + 5;
90     st_time.nanoseconds = 0;
91     ex_time.seconds = 5;
92     ex_time.nanoseconds = 0;
93     de_time.seconds = cur_time.seconds + 15;
94     de_time.nanoseconds = 0;
95     ret = fttl_thread_attribute_init(
96         st_time, /* ready */
97         ex_time, /* exec_time */
98         de_time, /* deadline */
99         tl_thread_simple, /* entry point */

```

```

100             1,                /* argument */
101             &deadline_port_one, /* port */
102             &thread_attr);
103
104     if (ret != KERN_SUCCESS) {
105         printf("Got %s from tl_thread_attribute_init\n",
106                mach_error_string(ret));
107         exit(-23);
108     }
109
110     /* create and run the thread */
111     ret = fttl_thread_create(mach_task_self(), &new_thread, &thread_attr);
112
113     if (ret != KERN_SUCCESS) {
114         printf("Got %s from tl_thread_create\n",
115                mach_error_string(ret));
116         exit(-23);
117     } else {
118         printf("Thread one created.\n");
119     }
120
121     /* the second thread will run 20 seconds from now and we give it an
122        exec time of 5 seconds (which it will not use) */
123     st_time.seconds = cur_time.seconds + 10;
124     st_time.nanoseconds = 0;
125     ex_time.seconds = 5;
126     ex_time.nanoseconds = 0;
127     de_time.seconds = cur_time.seconds + 30;
128     de_time.nanoseconds = 0;
129     ret = fttl_thread_attribute_init(
130         st_time,          /* ready */
131         ex_time,          /* exec_time */
132         de_time,          /* deadline */
133         tl_thread_simple, /* entry point */
134         2,                /* argument */
135         &deadline_port_two, /* port */
136         &thread_attr);
137
138     if (ret != KERN_SUCCESS) {
139         printf("Got %s from tl_thread_attribute_init\n",
140                mach_error_string(ret));
141         exit(-23);
142     }
143
144     /* create and run the thread */
145     ret = fttl_thread_create(mach_task_self(), &new_thread, &thread_attr);
146
147     if (ret != KERN_SUCCESS) {
148         printf("Got %s from tl_thread_create\n",
149                mach_error_string(ret));
150         exit(-23);
151     } else {
152         printf("Thread two created.\n");
153     }
154
155
156
157     st_time.seconds = cur_time.seconds + 10;

```

```

158     st_time.nanoseconds = 0;
159     ex_time.seconds     = 5;
160     ex_time.nanoseconds = 0;
161     de_time.seconds     = cur_time.seconds + 40;
162     de_time.nanoseconds = 0;
163     ret = fttl_thread_attribute_init(
164         st_time,          /* ready */
165         ex_time,         /* exec_time */
166         de_time,         /* deadline */
167         tl_thread_simple, /* entry point */
168         3,               /* argument */
169         &deadline_port_three, /* port */
170         &thread_attr);
171
172     if (ret != KERN_SUCCESS) {
173         printf("Got %s from tl_thread_attribute_init\n",
174             mach_error_string(ret));
175         exit(-23);
176     }
177
178     /* create and run the thread */
179     ret = fttl_thread_create(mach_task_self(), &new_thread, &thread_attr);
180
181     if (ret != KERN_SUCCESS) {
182         printf("Got %s from tl_thread_create\n",
183             mach_error_string(ret));
184         exit(-23);
185     } else {
186         printf("Thread three created.\n");
187     }
188
189
190     sleep(30); /* if we are alive longer than a minute - just die... */
191     exit(0);
192 }
193

```