Mach Kernel Monitor  
(with applications using the PIE environment)

22 February 1990

Ted Lehr  
David Black  
Department of Electrical and Computer Engineering  
and School of Computer Science  
Carnegie Mellon University  
Pittsburgh PA 15213

Abstract

Factors such as the decomposition of parallel programs affect their performance. Measurements of parallel program performance are improved if supported by information such as how programs are scheduled. This manual describes how to use MKM, the Mach (context-switch) kernel monitor. Special examples of data obtained by using MKM are shown via the PIE performance monitoring environment.
1 The Mach Kernel Monitor: The Need to Monitor Context-Switching

The performance of computations on parallel machines is affected by user designed attributes such as the parallel decomposition of the computations and by operating system actions such as scheduling. One way of determining how computations are scheduled is to detect and time-stamp the context-switches of their threads.

Certain Mach configurations (ie. those with the EXP extension) include the Mach Kernel Monitor (MKM) for monitoring kernel-level behavior. Currently, MKM only monitors context-switches of user selectable threads. It permits simultaneous monitoring of independent computations so that multiple users may selectively observe as many computations as they desire, collecting only the data about those they are interested in regardless of what else is running on the system.

After discussing the implementation and system calls of MKM, this manual gives some examples using the PIE\(^1\) performance monitoring and visualization system. Using the PIE examples, the scheduling information collected by MKM fulfills the double role of visualizing scheduling performance in general as well as visualizing the influence of the scheduler on user algorithms.

2 Implementation

An MKM monitor consists of:

- A data structure consisting of buffers for storing information about detected events (in the current implementation, a monitor can detect only context-switch events) and state information.
- A list of threads that can be observed by the monitor.
- Event detection sensors within Mach context-switching code.
- Entries within thread data structures for assigning monitors to threads.

These data structures are operated upon by several monitor system calls. Future versions of the monitor may contain data structures for monitoring message sends and receives or paging behavior. Currently, the only calls recognized by a Mach kernel monitor are ones concerned with detection of context-switches.

The `monitor_create` call creates a monitor within the calling task. The call returns the monitor id and the size of the event buffer in the kernel. The user uses this size to allocate an appropriately sized user buffer into which `monitor_read` writes context-switch data read from the kernel. `monitor_create` returns the monitor in a suspended state. `monitor_resume` starts the monitor. `monitor_suspend` permits the user to suspend (pause) monitoring. `monitor_terminate` destroys the monitor. `monitor_read` reads the context-switch event data from the kernel into a user supplied buffer. `monitor_read` calls are valid as long as its argument is a valid, non-terminated monitor. `thread_monitor` enables individual threads for monitoring. When a thread calls `thread_monitor` the

\(^1\)For introduction to PIE, see "Visualizing Performance Debugging," in *IEEE Computer*, October 1989, by Ted Lehr, et al.
associated monitor will detect each time the thread context-switches. thread_unmonitor disables a thread from being monitored.

```c
#include <mach.h>
#include <mach/kernel_event.h>
#include <mach/monitor.h>

main()
{
    int buf_size;
    monitor_t my_monitor;
    int j, num_events = MONITOR_MIG_BUFFER_SIZE;
    kern_mon_buffer_t kernel_events;

    buf_size = REQUESTED_SIZE;
    monitor_create(task_self(), &this_monitor, &buf_size);
    monitor_resume(this_monitor);
    thread_monitor(this_monitor, UNIQUE_ID, thread_self());

    for(i = 0; i < 300; i++) sleep(1);
    thread_unmonitor(this_monitor, thread_self());
    kernel_events = (kern_mon_buffer_t)malloc(sizeof(kern_mon_data_t)*MONITOR_MIG_BUFFER_SIZE);
    while (num_events == MONITOR_MIG_BUFFER_SIZE) {
        monitor_read(this_monitor, kernel_events, &num_events)
        for (j = 0; j < num_events; j++) {
            printf("%8.8x %8.8x %8.8x %8.8x %8.8x\n",
                kernel_events[j].third_element,
                kernel_events[j].second_element,
                kernel_events[j].hi_time,
                kernel_events[j].lo_time,
                kernel_events[j].first_element);
        }
    }
    monitor_terminate(this_monitor);
}
```

Figure 1: Simple Example Program Using Monitor

Figure 1 shows an example program using monitoring. The program runs as a single thread. The constant REQUESTED_SIZE is assigned to buf_size. This is the requested size of the monitor buffers allocated inside the kernel. After creating and resuming (starting) the monitor, the program enables itself for monitoring. UNIQUE_ID is some constant that the user knows is unique across all the threads in his program. In this case, since there is only one thread, the value of UNIQUE_ID is arbitrary.

After executing a sleep statement 300 times, the thread is disabled for monitoring. Then a user routine is called, print_events, that repeatedly calls monitor_read and prints the events until the kernel_buffer is empty. Note that the user buffer is only MONITOR_MIG_BUFFER_SIZE big because of a limitation on MIG buffer sizes. Thus, the program must call monitor_read until it returns less than MONITOR_MIG_BUFFER_SIZE events. When this occurs, the program knows that the kernel buffer is empty. When print_events returns, the monitor is terminated. Note that in this program, monitor_suspend is never called.
The context-switch data structure saved for each relevant context-switch is:

```c
typedef
struct
{
    unsigned event_type;  /* type */
    unsigned first_element;  /* stopped thread */
    unsigned second_element;  /* started thread */
    unsigned third_element;  /* flag and cpu */
    unsigned hi_time;  /* hi time stamp */
    unsigned lo_time;  /* lo time stamp */
} kern_mon_data_t, *kern_mon_buffer_t;
```

The members of the structure consist of the type of event (currently, only one type of kernel event is detected: context-switches), the stopped-thread and started-thread, the processor on which the threads switched, and a timestamp separated into seconds and microseconds fields. If one of the threads is unknown, its id is set to -1. The most significant bit of the processor field is 1 if that event overwrote a previous, unread event (i.e. overflow).

**NOTE:**
Currently, there is no internal protection guarding against requesting too much memory for buffers. It is suggested that no more than a half megabyte should be requested. A rule of thumb is that the greater the thread-to-processor ratio, $t$, of one’s computation, the more context-switches there will be. Assume that when $t > 1.0$, there will be ten context-switches per second. The size of the internal buffers then depends on how often they are read. `monitor_read` retrieves at most $n = \text{sizeof(kern_mon_data_t)} \times \text{MONITOR_MIG_BUFFER_IZE}$ bytes each time the buffers are read. Assume that once one `monitor_read` is made to read the buffers, the user repeats the call until the buffers are empty. In such case, if the user makes bursts of `monitor_read` calls once every $d$ seconds, a good buffer size would be $10d \times n$.

Figure 2 is schematic depicting a case in which two independent non-communicating Mach tasks have created separate MKM monitors. Each monitor is represented by a port in its parent task. Thus, the task that creates a monitor obtains rights to the port that represents the monitor; only tasks that possesses such rights can access the monitor. In our example, unless task B gives task A rights to the monitor created by B, task A cannot access it.

Figure 2 also shows non-intersecting sets of circular buffers allocated to each monitor for holding context-switch events. A buffer is assigned to each processor in order to eliminate contention between processors for buffers. When a thread context-switches, a software context-switch sensor detects which, if any, monitor is tracking the thread and writes an event to the appropriate buffer. Eventually, a task holding rights to a monitor will release those rights and terminate the monitor. This can be done either explicitly while the task is alive, or implicitly when the task terminates. In either case, the termination of a particular monitor is accomplished by the kernel.
Figure 2: General Kernel Monitor Architecture
monitor_create
#include <mach.h>
#include <mach/kernel_event.h>
#include <mach/monitor.h>

kern_return_t monitor_create(owner_task, new_monitor, buffer_size)
    task_t    owner_task;
    monitor_t *new_monitor;  /* out */
    int *buffer_size;  /* out */

Description
monitor_create creates a new monitor within the task specified by owner_task argument.
buffer_size is the requested size (in number of events) for the monitor kernel buffer used to hold
context-switch events. When monitor_create returns, buffer_size is maximum number of events
that kernel buffer may hold before it overflows. When the monitor is created send rights to its monitor
kernel port are given to it and returned in new_monitor to the caller. The new monitor is returned in a
suspended state. To get a new monitor to run, first monitor_create is called to get the new monitor's
identifier,(monitor). Then monitor_resume is called to get the monitor to execute.

Arguments
owner_task The task which is to contain the new monitor.
new_monitor The new monitor.
buffer_size The size (in number of events) of the monitor buffer in kernel.

Returns
KERN_SUCCESS A new monitor has been created.
KERN_INVALID_ARGUMENT
parent_task is not a valid task.

Notes
Currently, there is no internal protection guarding against requesting too much memory for buffers. It is
suggested that no more than a half megabyte should be requested. A rule of thumb is that the greater
the thread-to-processor ratio, t, of one's computation, the more context-switches there will be. Assume
that when t > 1.0, there will be ten context-switches per second. The size of the internal buffers then
depends on how often they are read. Each time the buffers are read, (see monitor_read) at most \( n =
\text{sizeof(kern_mon_data_t)} \times \text{MONITOR_MIG_BUF_SIZE} \) bytes are retrieved. Assume that once one
monitor_read is made to read the buffers, the user repeats the call until the buffers are empty. In such
case, if the user makes bursts of monitor_read calls once every \( d \) seconds, a good buffer size would
be \( 10d \times n \).

See Also
monitor_resume, monitor_terminate, monitor_suspend, monitor_read,
thread_monitor, thread_unmonitor, monitor


**monitor_resume**

```c
#include <mach.h>
#include <mach/kernel_event.h>
#include <mach/monitor.h>

kern_return_t monitor_resume(target_monitor)
    monitor_t target_monitor;
```

**Arguments**

- `target_monitor` The monitor to be resumed.

**Description**

Sets the state of `target_monitor` to MONITOR_RUN. When the monitor is in this state, it can detect events.

**Returns**

- `KERN_SUCCESS` The monitor has been resumed.
- `KERN_FAILURE` The monitor state is MONITOR_SHUTDOWN.
- `KERN_INVALID_ARGUMENT` `target_monitor` is not a monitor or its port is no longer valid.

**See Also**

`monitor_create`, `monitor_terminate`, `monitor_suspend`, `monitor_read`, `thread_monitor`, `thread_unmonitor`, `monitor`
monitor_suspend

#include <mach.h>
#include <mach/kernel_event.h>
#include <mach/monitor.h>

kern_return_t monitor_suspend(target_monitor);
    monitor_t target_monitor;

Arguments

 target_monitor  The monitor to be suspended.

Description

Sets the state of target_monitor to MONITOR_PAUSE. No events are detected when the monitor is in this state although any previously detected events may be read by monitor_read.

Returns

KERN_SUCCESS  The monitor has been suspended.
KERN_FAILURE  The monitor state is MONITOR_SHUTDOWN.
KERN_INVALID_ARGUMENT  target_monitor is not a monitor or its port is no longer valid.

See Also

monitor_create, monitor_terminate, monitor_resume, monitor_read, thread_monitor, thread_unmonitor, monitor
monitor_terminate

#include <mach.h>
#include <mach/kernel_event.h>
#include <mach/monitor.h>

kern_return_t monitor_terminate(target_monitor)
    monitor_t target_monitor;

Description
monitor_terminate destroys the monitor specified by target_monitor.

Arguments
target_monitor The monitor to be destroyed.

Returns
KERN_SUCCESS The monitor has been destroyed.
KERN_INVALID_ARGUMENT target_monitor is not a valid monitor or its monitor port no longer exists.

See Also
monitor_create, monitor_resume, monitor_suspend, monitor_read, thread_monitor, thread_unmonitor, monitor
monit or_read

#include <mach.h>
#include <mach/kernel_event.h>
#include <mach/monitor.h>

/* only current interpretation of kernel_event */
type def struct kernel_event { /* unit kernel event */
  unsigned event_type; /* the type of kernel event */
  unsigned first_element; /* the stopped thread */
  unsigned second_element; /* the started thread */
  unsigned third_element; /* flag and cpu number */
  unsigned hi_time; /* hi time stamp */
  unsigned lo_time; /* lo time stamp */
} kern_mon_event, *kern_mon_event_t, kern_mon_data_t,
*k kern_mon_buffer_t;

kern_return_t monitor_read(target_monitor, buffer, events_read)
    monitor_t target_monitor;
    kern_mon_buffer_t buffer;
    int events_read;

Description

monitor_read returns events detected by target_monitor into the buffer argument. 
events_read is the number of events returned. Each call to monitor_read is limited to returning a
maximum of MONITOR_MIG_BUF_SIZE events, a limitation of MIG buffer size. Buffer data is aligned
around event boundaries but it is the responsibility of user code to properly declare and allocate buffer

Arguments

target_monitor The monitor for which events are being read.
buffer The user’s buffer into which the events will be written.
events_read The number of events read by the call.

Returns

KERN_SUCCESS The monitor buffer was successfully read.
KERN_INVALID_ARGUMENT target_monitor is not a monitor or the monitor port is not valid.

Notes

A rule of thumb is that the greater the thread-to-processor ratio, t, of one’s computation, the more context-
switches there will be. Assume that when t > 1.0, there will be ten context-switches per second. In order
to prevent the internal buffers from overflowing, monitor_read should be called at least once every
MONITOR_MIG_BUF_SIZE/10 seconds which in the current implementation is about every 30 seconds.
Or, if each time monitor_read is called it is immediately followed by repeated calls until the internal
buffers are empty, these bursts of repeated calls should occur every every B/(10 x
sizeof(kern_mon_data_t)) seconds where B is the size of the internal buffer in bytes.
See Also

monitor_resume, monitor_terminate, monitor_suspend, monitor_read,
thread_monitor, thread_unmonitor, monitor
thread_monitor
#include <mach.h>
#include <mach/kernel_event.h>
#include <mach/monitor.h>

kern_return_t thread_monitor(target_monitor, unique_id, target_thread)
    monitor_t target_monitor;
    int unique_id;
    thread_t target_thread;

Arguments

 target_monitor The monitor which will observe target_thread
 unique_id An id for thread identification outside kernel.
 target_thread The thread which will be monitored.

Description

thread_monitor enables target_thread for monitoring by target_monitor argument. The caller
is responsible for keeping unique_id unique among all threads that target_monitor observes.
target_thread can be observed by only one monitor at a time.

Returns

 KERN_SUCCESS The thread has been enabled to be observed by monitor.
 KERN_FAILURE The monitor state is MONITOR_SHUTDOWN or thread is not active.
 KERN_INVALID_ARGUMENT target_monitor is not a monitor, target_thread is not a thread, or the monitor
 port is not valid.

See Also

 monitor_create, monitor_terminate, monitor_resume, monitor_suspend, monitor_read, thread_unmonitor, monitor
thread_unmonitor
#include <mach.h>
#include <mach/kernel_event.h>
#include <mach/monitor.h>

kern_return_t thread_unmonitor(target_monitor, target_thread)
    monitor_t target_monitor;
    thread_t target_thread;

Arguments

target_monitor The monitor which observes target_thread
target_thread The thread which will be disabled.

Description

thread_unmonitor disables target_thread from being monitored by target_monitor.

Returns

KERN_SUCCESS The thread has been disabled from monitoring.
KERN_INVALID_ARGUMENT target_monitor is a valid monitor, target_thread is not a thread, or the monitor
    port is not valid.

See Also

monitor_create, monitor_terminate, monitor_resume, monitor_suspend, monitor_read, thread_monitor, monitor
3 Examples of Context-Switch Monitoring

Before giving examples of some results achieved using MKM, a brief description of the PIE environment is necessary. The Parallel Programming and Instrumentation Environment (PIE) is a software development environment for debugging performance using special development and data analysis tools. PIE is a “computational laboratory” in which programmers design experiments to evaluate the real behavior of computations.

PIE is a portable system whose basic platform is a workstation running the X Window System. It supports several sequential and parallel languages. Currently, PIE’s monitoring instrumentation runs on Vax and Sun workstations, Encore Multimax, and Warp. Although PIE can be ported to other Unix-like operating systems, its current form is implemented on top of the Mach operating system.

Figure 3: Organization of PIE

Figure 3 depicts the general organization of PIE. This manual only presents the visualization component of PIE, called PIEscope, because of its relevance in showing context-switches detected by MKM. The context-switches shown in the examples were detected by MKM and retrieved by PIE using the monitor_read call. The begin and end of the threads were recorded using simple methods in PIE.

3.1 Interpreting the PIE Figures

The two views in Figure 4 are examples of PIE’s principle formats for representing performance information. These particular views show the execution of a matrix multiplication computation. The top view is an example of the execution barscope view; the bottom view is an example of the cpu barscope...
Figure 4: Execution and Cpu Views of Incomplete Gang Scheduling

view. In the execution barscope view, time is measured in microseconds on the horizontal while the threads of the computation are ordered on the vertical. This particular view shows the part of the execution from about 0.0 to 36.7 seconds. The time of execution for each thread is depicted by the dark rectangles.

The cpu barscope shows thread-to-processor assignments during the execution of a computation. As in the execution barscope, the cpu barscope displays time in microseconds on the horizontal. On the vertical, however, the processors used by the computation are ordered and arbitrarily numbered on the vertical. Opposite each cpu are alternating sets of patterned rectangles. A patterned rectangle represents an identifiable executing thread while a white rectangle is a period when none of the respective computation’s threads are running on the associated cpu. PIE allows a user to arbitrarily assign unique colors or patterns to as many threads per cpu-view as he wishes.

3.2 Gang Scheduling

Sometimes, the number of threads is greater than the number of processors. The two views of Figure 4 are execution and cpu barscopes of an entire matrix multiply computation for which five processors were allocated. In each view, a pair of time cursors delimit approximately the same period in time, one in which the collector, again represented by dark diagonally slashed bars, is running on cpu 12.
3.3 Comparing Schedulers for a Uniprocessor

PIE and MKM can also monitor sequential computations. The three views in Figure 5 are execution barscopes that were used to evaluate the Mach scheduler. Each view shows the same two threads time-sharing one processor. As one moves down the Figure, the views represent newer versions of Mach. Because the views depict uniprocessor executions, cutting a vertical swath through a view at any point slices through only one running thread ... only a single black rectangle.

The top view of Figure 5 depicts an execution on the old `XF29` kernel. The next view depicts the same computation on the less primitive `CS5a` kernel. The `X96` kernel in the bottom view is the most advanced.

The views show that the two threads do not behave identically on three kernels. `XF29` uses a simple scheduling algorithm that switches the threads roughly every 100 milliseconds. The schedulers of the latter two kernels use a progressive algorithm which attempt to increase the length of the time slices allocated to each thread. Despite good intentions, the progressive algorithm of `CS5a` fails to dramatically reduce the number of context-switches because it permits threads to switch to themselves repeatedly as indicated by the preponderance of "squiggles." `X96` corrects this drawback by preempting a thread from switching to itself.
# Table of Contents

1 The Mach Kernel Monitor: The Need to Monitor Context-Switching  
2 Implementation  
3 Examples of Context-Switch Monitoring  
   3.1 Interpreting the PIE Figures  
   3.2 Gang Scheduling  
   3.3 Comparing Schedulers for a Uniprocessor
List of Figures

Figure 1: Simple Example Program Using Monitor 2
Figure 2: General Kernel Monitor Architecture 4
Figure 3: Organization of PIE 13
Figure 4: Execution and Cpu Views of Incomplete Gang Scheduling 14
Figure 5: The Execution of Two Threads on Three Kernels: XF29, CS5a, X96 15