Efficient Logging without Compromising Testability

Technical Report MRTC87

Joel Huselius (joel.huselius@mdh.se)
Department of Computer Science and Engineering
Mälardalens University, Västerås, Sweden

December 12, 2002

Abstract

We present a structure for logging entries originating from a monitoring-process on a real-time system. The main contribution to that *logging structure* is an algorithm for choosing and evicting entries from a multi-queue structure. The algorithm is a garbage collection algorithm for usage when storing data in a finite space during monitoring of real-time systems. In order not to compromise the testability of the system, the algorithm has therefore been designed to have a constant execution time for a given setup.

An example of application is in a monitoring/replay approach to facilitate debugging.

1 Introduction

Debugging of real-time systems requires the use of monitoring, logging of monitoring data, and replay [2]. On-line, during the execution of the system, information is extracted (the system is monitored) and stored (logged) to a persistent medium. Thereby, deterministic off-line re-execution (replay) of the system is facilitated.

Monitoring is performed by inserting probes to relevant parts of the system. Probing is either performed by inserting software probes into the code, by adding some extra hardware to snoop the system, or by some hybrid approach. In this report, we assume the use of probes for monitoring that are implemented in software. A comprehensive investigation of issues related to replay, monitoring, and debugging, is presented in [2].

As real-time systems are normally very long running systems, intended to run without interruption for days or years, monitoring these systems require very large memory resources. However, as many real-time systems are small, and embedded into devices, memory resources are also often limited. As a solution to this problem, older entries are evicted according to some scheme as resources are exhausted [4].

1.1 Probe Effect

When performing monitoring with probes implemented in software, the *probe effect* [1] is a potential problem. The effect can become visible when the composition of the set of resources required to perform a given monitoring task is altered. Examples of such resources may be execution time, accesses to physical or virtual devices (e.g. semaphores or disk drives), etc. The probe effect, when significant enough, is manifested by alternations in the scheduling and the temporal and functional operation of the system, thereby invalidating all previous verification of the system.

Thus, for example, a probe effect can be visible if a software is modified, or if the platform of the system is changed.

As a consequence of this, the perturbation from the probes must not change over time. Thus, probes should be left in the system even after the end of the verification process [6] [8, s3.1;p51].

1.2 The Effect of Jitter on Testability

There is a direct relation between the amount of jitter in the system and the number of possible execution scenarios. Many execution scenarios require larger testing efforts. Thus, the larger the jitter is, the harder it is to test the system - the testability is reduced as the jitter increases [7].

There are several factors that control the amount of jitter in the system [7]: *Execution time jitter* depends on differences between the best-case execution time and the worst-case execution time of a system. *Start jitter* describes the jitter induced in low priority tasks as they are affected by the execution of - and the execution time jitter in - high priority tasks. Clock synchronization algorithms alter the frequencies of local clocks in a distributed system; this gives rise to a *clock synchronization jitter*. Differing communication delays in distributed systems result in a *communication latency jitter*.

1.3 Data- and Control-Flow

An execution consists of two parts [3]: The *data-flow* describes alterations in data structures that are under control of the taskset (e.g. local and global variables, input and output), while the *control-flow* describes run-time events during the execution (e.g. calls to functions and system calls, occurred interrupts and preemptions).

The complete data- and control-flow of an execution, together with the application code, defines an execution of the system. Thus, for a successful replay, both data- and control-flow should be monitored.

1.4 Outline

Section 2 present a problem formulation: Can the architecture used to store monitored events contribute to improve replay performence? Our proposed solution, together with some evaluation of an implementation, is presented in Section 3. Section 4 present some ideas for future work.

The paper is concluded in Section 5.

Enclosed in Appendix A is a C-language implementation of our proposed method.

2 Storing Monitoring Entries

Traditionally, the *logging structure* for storing monitoring entries has been a single circular queue [4], which implies FIFO rules. Garbage collection in a circular queue is trivial; entries can be stored in seemingly chronological order on the medium using a simple modulus-function, and as space is exhausted the oldest entry is replaced with the newest entry. Thus, the on-line performance of the garbage collection algorithm ensures that no large penalty in the form of increased execution time is imposed on the system.

However, there are performance related drawbacks to the simplistic FIFO scheme. These issues do not concern the on-line performance of the algorithm, but the off-line usefulness-ratio of the stored information. That is to say how many of the stored entries that can be used in a replay. With the circular queue logging structure, no respect is paid to the relative context of the information that is expunged and the information that is allowed to remain. The usefulness of the information handled is ignored. Thus, we cannot assume that the final product is optimal with respect to the off-line usefulness of stored data.

One example is found in the monitoring of transactions to persistent storage mediums, such as hard-disks. As an instance of a block written on a hard-disk may be read several times, it is more efficient to perform the monitoring at the time of the write-operation. However, as a disk is a persistent storage medium, the log-entries should be evicted in an order that reflects the ordering of the sequence of read-actions to them.

Furthermore, out of a complexity perspective for the programmer, it is desirable to allow replay of only a subset of the system. As only a subset of the system is replayed, it is not required to monitor the entire system - leading to that less of the limited memory resources must be allocated. But, as mentioned in Section 1.1, probes should not be removed from, or added to, the system because it invalidates previous verification efforts. However, if the functionality of the garbage collection algorithm could be altered without introducing a probe effect, memory resources could be saved.

Therefore, a more complex logging structure should be used.

In order to allow an optimal off-line usage of the information, the logging structure should respect at least some of the information available about the different entries that presently remain in the monitoring log.

2.1 Orthogonal Jitter

However, the logging structure may increase the jitter in the system, which will lead to that the testability of the system is compromised (see Section 1.2). The effect on testability will be greater if the jitter of the logging structure is depending on factors orthogonal to the factors that control the original jitter in the system.

We define a determinism in the temporal domain: a definition of a function is deterministic in

the temporal domain if any two executions of the function, provided with the same input, will deliver the same output in the same time and require the same amount of computation.

Thus, inputs to functions must describe, not only the ordinary explicit inputs, but also temporal side effects, interference from other executions etc. There is a definition of every function, such that it has the smallest set of inputs possible while still being deterministic in the temporal domain.

We assume that $f(a_1, a_2 \ldots, a_n)$ is the smallest definition of a function f, such that it is deterministic in the temporal domain. If adding the logging structure leads to that the defintion $f(a_1, a_2 \ldots, a_n)$ is no longer deterministic in the temporal domain, the logging structure has a jitter that is orthogonal to the jitter of the original system. Consequently, the testability of the function is reduced as the logging structure is added.

Thus, the jitter of the logging structure should not be allowed to have an execution time that varies depending on the entries that reside in the log. As the execution time of our proposed algorithm is constant, and no additional synchronization is required, the increase of jitter is constant. Thus, the testability of the system is not compromised by the use of our algorithm.

3 Streams of Monitoring Entries

In the context of data-flow monitoring, we advocate a logging structure based on a constant execution time eviction scheduler (CETES). In CETES, many *streams* can be defined. The definition of a stream is application specific. Every stream has its own queue of monitoring entries, and all streams share the same pool of memory. An *eviction scheduler* decides which stream that should release its oldest entry in favour for a new entry to some (the same or other) stream. Depending on the type of monitoring performed, examples of how to partition the monitoring effort into streams could be by block on a hard-disk, by ipc-queues,[1] by transactions, or by individual tasks.

In this report, we present an eviction scheduler that can prioritize one or more streams, and which has a constant execution time. The constant execution time enables supervisors of the system to modify the priorities of streams either during run-time, or during setup of the system, while keeping the testability of the system uncompromised.

3.1 FIFO Queues

As we have partitioned the monitoring data into streams, we can see that there are two ways to implement a logging structure with FIFO rules. We will differentiate between local FIFO (LFIFO) and global FIFO (GFIFO) logging structures.

Similar to the CETES logging structure, LFIFO will also base its logging on the concept of streams. One circular queue is established for each stream of monitoring data. The lengths of these queues are allowed to vary between queues and between executions, without compromising testability, but the total memory area used for the logging structure is not.

The GFIFO logging structure has no concept of streams, and the scope of the monitoring efforts

[1] IPC: Inter-Process Communication

in not adjustable once the procedure of testing the system has been commenced. All monitoring entries share the same memory pool, and there is no room for adjustments.

3.2 Implementation

The CETES algorithm has been implemented in the C-language, the source of the scheduler is provided in Appendix A.

Our main philosophy when designing the logging structure, with its multi-queue-structure and its eviction scheduler, has been to eliminate all special cases as the probability of differing execution times between cases is believed to be large.

When removing items from a queue, it is normally a special case if the queue is empty before the attempted removal. Thus, we settled for an architecture that trades space for the absence of special cases; the queue of a stream must never be empty![2]

Initially, during setup of the logging structure, all streams in the system are assigned at least one entry. Extra entries are initially placed in an idle-stream.

As our logging structure would be concurrently utilized by several execution entities (tasks, processes, etc.), and operate on a shared pool of memory, a mutual exclusion problem must be addressed. We have chosen to handle the problem by implementing the probes as *kernel probes* [5, p40;s4.3.3], which is to say to incorporate the probes into system calls and interrupt routines of the operating system. Thus, we avoid the introduction of any additional points of synchronization to achieve mutual exclusion; we only extend the use of those that already must be present in the system.

3.2.1 Limitations

A limitation in the current implementation is that all streams must have equally sized entries. Thus, it is difficult or at least not space- and time-efficient to use the same instance of the CETES logging structure for monitoring of the entire system.

If entries with non-uniform sizes are allowed to share the same pool of memory, care must be taken to ensure that the system will not suffer form fragmentation. This is also, as it is an example of memory allocation, an example of the classic bin-packing problem.

Furthermore, our solution has a larger execution time due to increased complexity in the algorithm. As the amount of consumed resources should be kept to a minimum this overhead presents a problem, but as the overhead is constant and no additional synchronization is required the problem is reduced; the testability of the system is uncompromised. Provided that probes are allowed to remain resident in the system, the solution does not incur a probe effect on the system either.

[2]In our implementation, we also allow a small extension to this rule; the user can specify a shortest length on a particular stream, but that length must never be shorter then one (1).

3.2.2 Eviction Mechanism

The eviction mechanism presented here is intended for use in the context of monitoring of ipc-queue-traffic. It is possible that other scenarios would require redesign of the algorithm.

Because the stream that is scheduled for eviction will never be an empty stream, the actual eviction of an entry is a very trivial process once it has been established which stream to evict from.

We partition every ipc-queue into one stream, so that every ipc-queue has its own stream for monitoring entries. Constraints can be associated with each stream. The implementation in Appendix A, allows the following constraints:

Spatial stream length. The minimum number of physical entries in the stream can be set to a positive integer larger then zero. As a default, the minimum spatial length is set to 1 (one).

Temporal stream length. The minimum span, in the temporal domain, of entries from the stream can be specified. As a default, the minimum temporal length is set to 1 (one).

Stream priority. Streams can be prioritized. As long as spatial- and temporal- constraints on streams with lower priorities are still maintained, only the streams with the highest priority will be served,

The search for an entry to evict will then browse the next-to-last entries in every available stream. Out of the streams from which entries can be removed without violating temporal- or spatial- constraints, the browse will find the node which has the oldest time-stamp, and that stream to which the found entry belongs will be chosen for eviction. As it is not certain that every stream has a next-to-last entry, a termination node is kept as the default.

By considering the age of the next-to-last entries in all streams, respect is paid to the feasible time span of the replay with the entries that will remain after the eviction. We thereby ensure that the usefulness-ratio will be greater then when only considering the age of the last entries (as is done in a solution using FIFO rules).

As a feature in the implementation, we have added the concept of priorities. The browse will primarily choose low-priority streams to evict messages from. As low prioritized streams will soon shrink, one may use temporal- or spatial- constraints that allow these to contain more entries then one. Thus, the presence of some entries in these streams can be ensured. Because of the constant execution time of the algorithm, altering constraints of streams between system executions can be performed without increasing the probe effect or compromising the testability of the system. Thus, these features can be used to shift the focus of the monitoring activities.

3.3 Performance

In this section, we will investigate a setup of a monitoring session performed on three ipc-queues. Both the local and the global FIFO logging structures, as well as our proposed CETES structure, will be investigated.

3.3.1 Setup

A simulation of the execution of a monitored system is performed. There are three ipc-queues in the system, ipc_0, ipc_1, ipc_2. Each of these is monitored as read actions are committed to them. Monitored events are logged, and each ipc-queue is matched as one stream. Reads are issued to queues at periodicities $T = \langle T_{max}, T_{min} \rangle$ ordered as follows: $T_2 = \langle 20, 10 \rangle$, $T_1 = \langle 30, 15 \rangle$, $T_0 = \langle 50, 30 \rangle$. At each read operation committed to ipc_n, $s_n = \langle s_{n_{max}}, s_{n_{min}} \rangle$ entries are consumed.

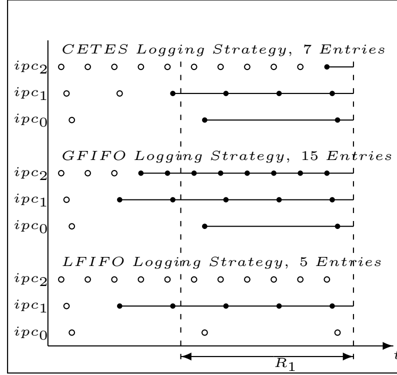

Figure 1: Coverage for Logging Structures, Section 3.3.2.

3.3.2 Example 1

According to the fault hypothesis, the stream from ipc_1 must be at least R_1 time units long. There are no restrictions posed on the other streams, but monitored and logged events should be kept at best effort. Thus, the logging structure used must ensue this as it controls which entries that remain in the log.

In this example, we let $R_1 = 65$, $s_0 = s_1 = s_2 = \langle 1, 0 \rangle$. There are no restrictions concerning the lengths of the logs from the monitoring of ipc-queues ipc_2 and ipc_0.

See Figure 1, the top graph displays the contents of our proposed CETES logging structure at the end of the simulation. The middle graph displays the behaviour of the global FIFO (GFIFO) logging structure, and the lower graph shows the local FIFO (LFIFO) logging structure, at the same point in time. Circles represent occurred events, $\bullet$ those events that remain in the log at the end of the simulation, $\circ$ those that do not.

$$L_{GFIFO} = \sum_{n=0}^{2} \left\lceil \frac{R_1}{T_{n_{min}}} \right\rceil * s_{n_{max}} \tag{1}$$

$$L_{LFIFO} = \left\lceil \frac{R_1}{T_{1_{min}}} \right\rceil * s_{1_{max}} \tag{2}$$

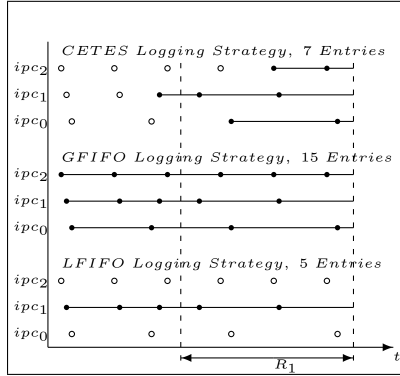

Figure 2: Coverage for Logging Structures, Section 3.3.3.

Note, that in order to ensure that the log contains a sufficient number of entries from the monitoring of ipc_0, the GFIFO logging structure requires that the log can accommodate L_{GFIFO} entries, as described by Equation 1. Thus, $L_{GFIFO} = 15$. Equation 2 describes the number of entries required for the LFIFO logging structure. Thus, $L_{LFIFO} = 5$. The CETES structure requires $L_{CETES} = 7$ entries: L_2 and then one for each of the other streams, as described by equation 3.

$$L_{CETES} = \left\lceil \frac{R_1}{T_{1_{min}}} \right\rceil * s_{1_{max}} + 2 \tag{3}$$

3.3.3 Example 2

In a following execution of the same system, we may experience another scenario. Firgure 1 displays another feasible execution scenario from the same settings. The logging structures use the same setup as in the previous example.

3.3.4 Discussion

Comparing the three different logging structures, we can see that all of them seem able to comply to the requirements posed on the contents of the log concerning ipc_1. However, the proposed CETES logging structure is, in both examples, performing much better with respect to monitoring the remaining ipc-queues.

3.4 Code Auditing

The implementation of CETES (attached in Appendix A) has been successfully compiled into assembler language for several different architectures, and the produced assembler source has been audited. The investigated architectures, and the used compilers are accounted in Table 1.

We have established that the resulting code conforms to the structure shown in Figure 3 in all

surveyed cases, where the conditional branch is always executed a fixed number of times, namely as many times as there are streams defined in the system. Thus, the complexity of the proposed CETES logging structure is $\mathcal{O}(N)$, where N is an application specific constant.

Table 1: Controlled Architectures.

Architecture	Compiler	Compiler Flags
Intel x86	gcc 3.1	`-S`
Sun SPARC	gcc 2.95.2	`-O1 -S`
Motorola 68k	m68k-coff-gcc 2.7.0	`-O1 -Wa,-al`

Note that the examined SPARC-architecture required the use of extra special code for making the greater-then-comparison between two integers. A normal C-syntax `A > B` was found to result in a conditional branch - as our aim was to create a software which conform to the structure in Figure 3, that branch must be eliminated. In order to resolve this problem, a construction using binary shift, addition, and subtraction, was produced. Assuming unsigned integers, where `S` is the size of the data-type to which both `A` and `B` belongs, it is true that `(!(((A)-(B+1))>>(S)))` $\Leftrightarrow$ `A > B`.

By proving that it is possible to create this software, we can conclude that, given the right platform implementation, it is possible to have an eviction scheduler with a constant execution time. The constraint which we pose on that architecture is that every instance of a given assembler instruction takes a fixed number of processor clock cycles to complete.

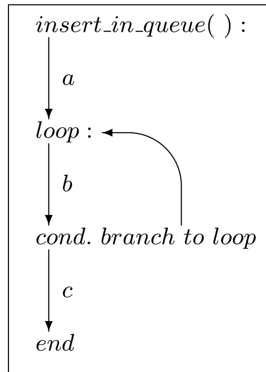

Figure 3: Program Structure of the Eviction Scheduler

3.4.1 Overhead

As previously mentioned, our implementation does require some extra overhead in memory resources. This overhead can be divided into four sub categories, the overhead per stream, per entry, the number of extra streams, and the number of extra entries.

The overhead for each stream equals the size of the header for a stream, which currently amounts to: eighth (8) `int`'s, four (4) `char`'s, and one (1) `long`. The characters are used to name streams,

and their number may vary for each instance of the implementation, the integers may potentially be replaced for smaller units, would the application allow.

For each entry, there is a time-stamp with the size of one (1) `long`, two `int`'s for indexing, and a buffer for the data. The integers could possibly be replaced with some smaller data-type, and the size of the buffer for the data is application specific.

We have previously mentioned that our implementation initially uses an idle stream for un-used entries. Thus, one additional stream-header is required, and as no stream can be empty, one additional entry accompanies the idle stream. It is possible to remove the idle stream and use one of the other streams instead. For pedagogic reasons, we have chosen not to do so in our implementation. As the eviction scheduler will base its calculation on the next-to-last entry in every stream, and streams are only guaranteed to have at least one entry, an extra termination node is required. That single termination node is used by every stream in the logging structure.

In Table 2, we present the execution overhead as figures on the number of instructions required for implementations of both our algorithm and the traditional GFIFO-algorithm. Note that the counts presented are closely tied to the particular implementation, and that reducing the number of instructions in the implementation has not been a primary goal. It is however some form of indication on the amount of computation overhead required.

Table 2: Instruction Count, see Figure 3 for a, b, c.

Architecture	a	b	c	GFIFO
SPARC	16	77	40	27
Intel x86	8	184	60	38
Motorola 68k	7	85	33	35

4 Future Work

The mechanism described in Section 3 is intended for use for storing data-flow monitoring entries. As a replay cannot be performed correctly without valid information about both data- and control-flow through the entire interval of the replay, correlating the storing of these entries is potentially interesting - there is for example no gain in keeping control-flow information about an interval for which no data-flow information is available.

We will in our future work describe how to integrate the CETES structure with a structure for control-flow monitoring.

We will also investigate how, during replay re-produced, intermediate task-input can be used to reduce the amount of monitored messages that need be logged during the reference execution. We will investigate mechanisms for logging and evicting monitored intermediate data- and control-flow events so that no more events then required for successful replay are kept in the log.

It is our intention to facilitate this by using the last event in each stream to decide how many of the events in the log that need be kept from a circular queue of such events.

A very important problem that must be solved, in order to allow efficient monitoring for incremental replay, is how to decide how entries in logs of transactions to ipc-queues and hard disk blocks can be invalidated.

5 Conclusions

We have here presented an algorithm that allows monitoring of computer systems to shift focus without compromising the testability of the system. Two features of the algorithm makes this shift feasible without introducing any probe effect; the constant execution time, and the possibility to grade monitoring effort using priorities and temporal as well as spatial constraints.

Furthermore, the presented work can contribute to that the relative usefulness of logged data increase. When the algorithm searches for an entry to evict from the monitoring log, respect is paid to the feasible time span of the replay with the remaining entries.

References

[1] Jason Gait. A Probe Effect in Concurrent Programs. *Software-Practise and Experience*, 16(3):225 – 233, March 1986.

[2] Joel Huselius. Debugging Parallel Systems: A State of the Art Report. Technical Report 63, Mälardalen University, Department of Computer Science and Engineering, September 2002.

[3] Bernhard Plattner. Real-Time Execution Monitoring. *IEEE Transactions on Software Engineering*, SE-10(6):756 – 764, November 1984.

[4] Darlene Stewart and Morven Gentleman. Non-Stop Monitoring and Debugging on Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 263 – 269. IEEE Computer Society, May 1997.

[5] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Kungliga Tekniska Högskolan, Sweden, May 2000.

[6] Henrik Thane and Hans Hansson. Using Deterministic Replay for Debugging of Distributed Real-Time Systems. In *the 12th Euromicro Conference on Real-Time Systems*, pages 265 – 272. IEEE Computer Society, June 2000.

[7] Henrik Thane and Hans Hansson. Testing Distributed Real-Time Systems. *Elsevier Microprocessors and Microsystems*, 24(9):463 – 478, February 2001.

[8] Jeffrey Tsai et al. *Distributed Real-Time Systems: Monitoring Visualization and Debugging and Analysis*. Wiley-Interscience, 1996.

A C Source for the Constant Execution Time Eviction Scheduler

```
/* FILE ev_sched.h
   Written 2002-11-12 by Joel Huselius jhi@mdh.se
   M\"{a}lardalen University at the
   Department of Computer Science and Engineering

   A Constant Execution Time Eviction Scheduler

*/

//#define EV_SCH_DEBUG /* UN-COMMENT LINE FOR AUX-OUTPUTS */
#ifdef EV_SCH_DEBUG
#include <stdio.h>
#define EV_SCH_DEBUG_PRINT printf
#else
#define EV_SCH_DEBUG_PRINT //
#endif

#ifndef EV_SCHED_H
#define EV_SCHED_H

#include <stdlib.h>

/* MACRO DEFINITIONS FOR GENERAL USE */

#define TRUE  1
#define FALSE 0

/*
MACRO DEFINITIONS FOR BIT-OPERATIONS:
*/
#define AGTBL(A,B,S)   (!(((long)(A)-(B+1))>>(S)))
#define AGTEQBL(A,B,S) (!(((long)(A)-(B))>>(S)))
#define AEQBL(A,B,S)     (A==B)

#define AGTBI(A,B,S)         (A>=(B+1))
#define AGTEQBI(A,B,S)       (A>=B)
#define AEQBI(A,B,S)         (A==B)

/* MACRO DEFINITIONS FOR QUEUES */

#define MON_ENTRIES 8
#define PRIO_LEVELS 5
#define Q_NAME_LEN  4
```

```
#define NO_IPC_Q    80
#define ENTRY_SIZE  sizeof(int)

#define NO_Q      4
#define Q_FREE    0

/* DATA STRUCTURES */

typedef struct mon_entry_s{
  int next;
  char contence[ENTRY_SIZE];
  int prev;
  unsigned long time;
}mon_entry_t;

typedef struct queue_s{
  int id;
  int ipc_id;
  int strt;
  int stop;
  int length;
  int minimum_length;
  int priority;
  int occupied;
  char name[Q_NAME_LEN];
  unsigned long minimum_age;
}queue_t;

typedef struct map_s{
  int ipc_queue;
  int mon_queue;
}map_t;

/* FUNCTON DECLARATIONS */

void init_q(void);
int insert_in_q(queue_t* q_ptr, void* val_p, unsigned long time);
int delete_in_q(queue_t* q_ptr);
void exit_q(void);

void set_priority(queue_t* q_ptr, int prio);
void set_minimum_length(queue_t* q_ptr, int min_len);
void set_minimum_age(queue_t* q_ptr, long min_age);
void print_queues(void);

/* GLOBAL VARIABLES */

extern mon_entry_t monit_area[MON_ENTRIES+1];
```

```
extern mon_entry_t* monit;
extern queue_t queues[NO_Q];
extern char mon_bit_mask[NO_Q];
extern map_t map[NO_IPC_Q];

#endif
```

```
/* FILE ev_sched.c
   Written 2002-11-12 by Joel Huselius jhi@mdh.se
   M\"{a}lardalen University at the
   Department of Computer Science and Engineering

   A Constant Execution Time Eviction Scheduler

*/

#include "ev_sched.h"

mon_entry_t monit_area[MON_ENTRIES+1];
mon_entry_t* monit=&(monit_area[1]); /* Make index (-1) the
                                        termination entry */
queue_t queues[NO_Q];
char mon_bit_mask[NO_Q];
map_t map[NO_IPC_Q];

/* CODE */

void init_q(void){
  int i;

  monit[-1].prev            =-1;
  monit[-1].next            =-1;
  monit[-1].time            =0;
  memset((void*)&(monit[-1].contence),0,sizeof(ENTRY_SIZE));

  for(i=0;(i-NO_Q)>>((sizeof(int)*8)-1);i++){
    queues[i].occupied       =((-(i==0)) &(TRUE))
                             |((-(i!=0)) &(FALSE));
    queues[i].id             =i;
    queues[i].strt           =i;
    queues[i].stop           =((-(i==0)) &(MON_ENTRIES-1))
                             |((-(i!=0)) &(i));
    queues[i].length         =((-(i==0)) &(MON_ENTRIES-NO_Q+1))
                             |((-(i!=0)) &(1));
    queues[i].minimum_length =1;
```

```
    queues[i].minimum_age    =1;
    queues[i].priority       =((-(i==0)) &(0))
                             |((-(i!=0)) &(1));
    queues[i].name[0]        = '\0';

    monit[i].prev            =-1;
    monit[i].next            =((-(i==0)) &NO_Q)
                             |((-(i!=0)) &(-1));
    monit[i].time            =i;

    memset((void*)&(monit[i].contence),0,sizeof(ENTRY_SIZE));
  }
  for(i=NO_Q;(i-MON_ENTRIES)>>((sizeof(int)*8)-1);i++){
    monit[i].prev    =((-(i==NO_Q))          &(0))
                     |((-(i!=NO_Q))          &(i-1));
    monit[i].next    =((-(i==MON_ENTRIES-1)) &(-1))
                     |((-(i!=MON_ENTRIES-1)) &(i+1));
    monit[i].time    =i;

    memset((void*)&(monit[i].contence),0,sizeof(ENTRY_SIZE));
  }
  return;
}

void exit_q(void){
}

int insert_in_q(queue_t* q_ptr, void* val_p,
                unsigned long time){
  queue_t* q_evict;
  int free_idx;
  /***** Start of the Eviction Scheduler *****/
  int j;
  int part_a, part_b;
  int q_id, q_prio=0;
  int j_older, j_too_long, j_lower_prio;
  int j_equal_prio, j_too_old, j_safe;
  int q_not_too_long,q_not_too_old;
  long j_prev,q_id_prev;
  static int latest_stream=Q_FREE;

  q_id=latest_stream;

  j=0;
  do{
    q_prio=queues[q_id].priority;
    q_id_prev=monit[queues[q_id].stop].prev;
    j_prev=monit[queues[j].stop].prev;
```

```
  j_older       =AGTEQBL(monit[q_id_prev].time,
                         monit[j_prev].time,
                         (sizeof(long)*8)-1);

  q_not_too_old =AGTBL(queues[q_id].minimum_age,
                       time-(monit[q_id_prev].time),
                       (sizeof(long)*8)-1);

  q_not_too_long=AGTEQBL(queues[q_id].minimum_length,
                         queues[q_id].length,
                         (sizeof(long)*8)-1);

  j_lower_prio  =AGTBL(q_prio,
                       queues[j].priority,
                       (sizeof(int)*8)-1);

  j_equal_prio  =AEQBL(queues[j].priority,
                       q_prio,
                       (sizeof(int)*8)-1);

  part_a=((q_not_too_old)
         |(q_not_too_long)
         |(j_lower_prio)
         |(j_equal_prio&j_older)
         );

  j_too_old     =AGTEQBL(time-(monit[j_prev].time),
                         queues[j].minimum_age,
                         (sizeof(long)*8)-1);

  j_too_long    =AGTBL(queues[j].length,
                       queues[j].minimum_length,
                       (sizeof(long)*8)-1);

  j_safe=AGTBL(queues[j].length,1,(sizeof(int)*8)-1);

  part_a=-(part_a
          &j_safe&j_too_long&j_too_old
          );

  part_b=(~part_a);
  q_id=(part_a&(j))^(part_b&(q_id));

  j++;
}while((j-NO_Q)>>((sizeof(int)*8)-1));
```

```
  EV_SCH_DEBUG_PRINT("%3lld Evicted from %x, Inserted to %x\n",
                     time,q_id,q_ptr->id);

  latest_stream=q_ptr->id;
  q_evict=&(queues[q_id]);

  /***** End of the Eviction Scheduler *****/

  free_idx=delete_in_q(q_evict);

  memcpy(monit[free_idx].contence,val_p,ENTRY_SIZE);
  monit[free_idx].time =time;
  monit[free_idx].prev    =-1;
  monit[free_idx].next    =q_ptr->strt;

  monit[q_ptr->strt].prev =free_idx;
  q_ptr->strt=free_idx;
  q_ptr->length++;

  return free_idx;
}

int delete_in_q(queue_t* q_ptr){
  int free_idx;

  free_idx=q_ptr->stop;
  q_ptr->stop=monit[q_ptr->stop].prev;
  monit[q_ptr->stop].next=-1;
  q_ptr->length--;
  return free_idx;
}

void set_priority(queue_t* q_ptr, int prio){
  q_ptr->priority=prio;
}

void set_minimum_length(queue_t* q_ptr, int min_len){
  q_ptr->minimum_length=min_len;
}

void set_minimum_age(queue_t* q_ptr, long min_age){
  q_ptr->minimum_age=min_age;
}

void print_queues(void){
  int i,j,n;
```

```
  for(i=0;i<NO_Q;i++){
    printf("q_%d\t",i);
    n=queues[i].strt;
    for(j=0;j<queues[i].length;j++){
      printf("<%3d>",monit[n].time);
      n=monit[n].next;
    }
    printf("\n");
  }
}
```