One-sided Mutual Exclusion A new Approach to Mutual Exclusion Primitives

Christian Siebert

christian.siebert@cs.tu-chemnitz.de

Abstract

Todays operating systems are designed to manage multiple processes within uniprocessor systems, within several processors and even within distributed computer systems. Fundamental to this design is concurrency, especially mutual exclusion of multiple processes. All mutual exclusion techniques so far are designed to bother all involved processes.
The first part of this paper shows an overview of the most common existing techniques that solves this problem. A new approach with a completely different design technique will be discussed afterwards. It is not always necessary to involve all processes and still maintain mutual exclusion. A couple of implementations for different operating systems and user/kernel mode applications will be compared to existing techniques.

1 Introduction

Interleaved execution (in single-processor multiprogramming systems) and overlapping execution (in multiple-processor systems) can be viewed as examples of concurrent processing. Scheduling policies, interrupt handling behaviour and other facts make it nearly impossible to determine the chronological execution order of instructions from several processes. Additional benefits in processing efficiency and in program structuring are the positive results of such execution techniques but sharing of resources (like devices or memory) becomes critical. There have to exist controlling possibilities to guarantee a maximum number of processes per critical resource. The part of contiguous instructions accessing a critical resource is called a critical section.
The most classical concurrency problems are the reader/writer problem and the producer/consumer problem (see [1]).
Depending on the type of process interaction, there are several potential control problems like data coherence, deadlock, starvation and mutual exclusion. This paper will focus mainly on the last problem.

2 Mutual exclusion

The most usual solution is to encapsulate critical sections by a pair of controlling functions.

```
{ ... several processes could stay at this point ... }
enter_critical_section(resource_id);
    { ... this is the critical section ... }
    { at most one process could stay at this point }
exit_critical_section(resource_id);
{ ... several processes could stay at this point ... }
```

Any process attempting to enter its critical section while another process stays already there (for the same resource) has to wait until the other process has left its critical section. When no other process is in a critical section, any process requesting entry to its critical section can enter without delay.
There are different mechanisms for implementations of *enter_critical_section()* and *exit_critical_section()*.

2.1 Decentralized software approach

The following software approach for mutual exclusion depends on shared memory and assumes elementary mutual exclusion at the memory access level. Simultaneous read and/or write accesses to the same memory location are serialized by the memory hardware.

2.1.1 Peterson's algorithm (1981)

A global array *inside[]* indicates the position of each process in relation to the critical section (initialized with no's), and the global variable *prior* resolves conflicts. A critical section is protected this way:

```
inside[this_process] := yes;
prior := the_other_process;
while (inside[the_other_process] = yes) and
    (prior = the_other_process) do { nothing };

 { critical section }

inside[this_process] := no;
```

Even though Peterson's algorithm could be easily generalized to the case of n processes, this is a constant number that must be defined before using.

2.1.2 Problems

Experiments with decentralized mutual exclusion algorithms on multiprocessor machines shows that they can fail! Although the algorithms are theoretical correct, they don't work on most modern processors (like PowerPC 603-G4, Intel Pentium II/III/4 and AMD K5/K6/Athlon). To get speed improvements, these processors use multiple functional units and implement speculative execution based on Tomasulo's algorithm (see [4]). Instructions are *executed out of order* and they are *committed in order* if they depend on each other. Most decentralized mutual exclusion algorithms write to one set of memory locations (*inside[this_process]*) and test another set of memory locations (*inside[the_other_process]*). One processor can't find the necessary global dependencies between those memory accesses.
Therefore we need a special kind of support ...

2.2 Hardware support

2.2.1 Special machine instructions

Processor designers have proposed several machine instructions that carry out more than only one action atomically. Examples are reading and writing (like *exchange* or *test_and_set*) or testing and conditional writing (like *compare_and_swap*) of single (or more) memory locations within one instruction fetch cycle.
These instructions simplify algorithms for mutual exclusion and work for any number of processes on a single processor and on multiple processors with shared memory. One disadvantage is that while a process is (busy) waiting for acccess to a critical section, it continues to consume processor time (so called spinlocks).

2.3 Operating system mechanisms

2.3.1 Semaphores

To hide the complexity of methods for process cooperation and to add additional features like blocking behaviour without processor time consumption, operating system functionality has been extended by semaphores. A semaphore can be viewed as an integer variable with three defined operations:

- initialization to a nonnegative value
- *wait* operation to decrement the value; process gets blocked if the value becomes negative
- *signal* operation to increment the value; one blocked process (if existing) gets unblocked

All three operations are assumed to be atomic, they cannot be interrupted.
A queue is used to hold all blocked processes waiting on the semaphore. *Strong semaphores* use the fairest queue policy first-in-first-out (FIFO) and are typically provided by operating systems. Semaphores with another policy are called *weak semaphores.*
A *binary semaphore* (similar to a semaphore initialized with the value 1, thus only having the two states {free, used}) for every critical resource could be used for the mutual exclusion problem by using *wait* as *enter_critical_section()* and *signal* as *exit_critical_section().*

3 Additional techniques

Several other mutual exclusion primitives (e.g. interrupt disabling on uniprocessor systems, reader/writer spinlocks and semaphores, monitors, pipes, signals, bolt variables, eventcounter and tickets, path expressions, serializers, read-copy update, conditional critical sections) have already been implemented or suggested.

4 One-sided mutual exclusion

Looking through all those existing mutual exclusion primitives, you might notice that every involved process needs to do something directly to maintain mutual exclusion. Every process has to call a function pair similar to *enter_critical_section()* and *exit_critical_section()*, regardless of the underlying primitive. Sure, these functions could be made fast, consuming maybe only a minimal constant time. However, the total amount of processing time, necessary for mutual exclusion, depends mainly on the:

- number of processes involved in mutual exclusion
- frequency of passing through a critical section
- number of collisions (more than one process wants to enter the critical section)
- type of resource usage (e.g. always read/write or read only for some processes)

Focusing only on the second point, it would be better to unbalance synchronization time for processes passing through a critical section with major differences in frequency (high frequency $\Rightarrow$ shorter time; low frequency $\Rightarrow$ longer time). But by keeping even constant times for every access, the total time for mutual exclusion still depend on this frequency variable and a process passing through a critical section very often would dominate the total time.

Thus the question arises, whether it is possible to exempt some processes (called *passive* from now on) from doing anything for mutual exclusion and to impose additional work on the other processes (called *active* from now on) to keep mutual exclusion. From a theoretical point of view, it would become the fastest possible mutual exclusion method for such cases, simply by adjusting the frequencies (this would become a threshold value for the decision to use this method or not).

4.1 Meta-Information

It seems to be impossible to maintain mutual exclusion when some processes (the passive ones) do really nothing, not even setting a single bit, for this purpose. The following sections will show that it is possible to create either one general passive process or several reader processes and still maintain mutual exclusion.

Now focus on some more details about critical sections:

1. there is a well defined entry (location of *enter_critical_section()*)
2. there is a well defined exit (location of *exit_critical_section()*)
3. critical sections should be kept short (to minimize waiting time in case of collisions)

Of course the usual function pair have to be removed from the passive processes, to achieve our new aim. So we first exchange them with labels to keep the information of the well defined entry and exit locations. Labels get removed by compilers; there is no instruction added to your code by inserting labels.

How could this information be used for mutual exclusion? Well, another criterion has to be added for critical sections:

4. all instructions processed within a critical section (of passive processes) have to be located between the entry and exit locations

This implies forbidding of function calls within critical sections. Since critical sections should be kept short, function calls should be avoided in any case. Alternatively, making the affected functions inline gives the compiler a hint to place the instructions correct (similar to macro expansion) - according the new rule. The correct output of the compiler should be checked in this case.

Every process knows its next instruction to be executed, either contained within a processor register called instruction pointer (or program counter) or stored somewhere (i.e. within a *process control block*) when it is currently not executed on any processor.

4.2 Controlling a passive process

How is it possible to prevent passive processes to enter their critical sections while another active process is currently inside a critical section?

Breakpoints, mostly used by debuggers, can be utilized to stop a process when it wants to execute an instruction located at the breakpoint. Although there are processors supporting hardware breakpoints (like 80386 and up, see [2]), software breakpoints are more general. They are not limited to a constant number (like four for 80386+) and are available on any processor. Short instructions, like interrupt calls or similar constructs for entering another code part, get placed at the breakpoint location for enabling the breakpoint. To reverse this procedure, the new breakpoint instructions have to be replaced by the old instructions.

4.3 Inspecting a passive process

Until now we are able to prevent passive processes to enter a critical section by setting breakpoints to the entry locations of the critical sections. But how is it possible to ensure that they are not already inside the critical section and still are?

The fourth criterion gives us the possibility to use the instruction pointer as a source of information for this question. Since all instructions processed within a critical section have to be located between the entry and exit locations, we get the values of the instruction pointers of the passive processes (after setting the breakpoints) and compare them with the corresponding locations. If all instruction pointers are smaller than the entry location or larger than the exit location of the corresponding process, then all passive processes are definitly outside their critical sections and we could safely enter our own critical section (remember: the passive processes can't enter because of the breakpoints). Otherwise, at least one of the inspected passive processes is potentially currently staying inside its critical section and we have to wait until all passive processes have left their critical sections.

4.4 General implementation

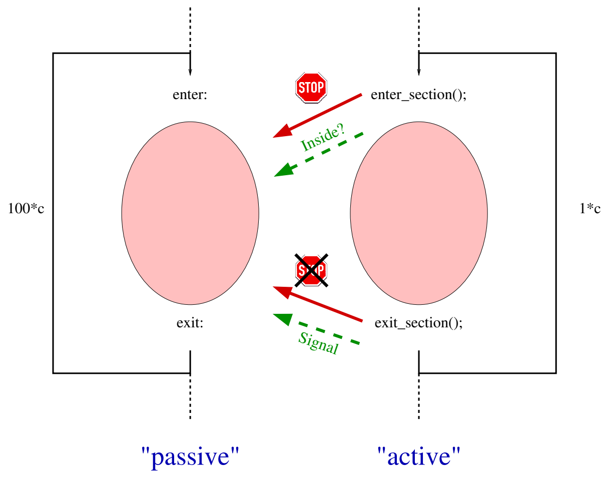
Several active processes could use a usual mutual exclusion primitive for their own process group and an additional one-sided mutual exclusion primitive to involve the passive processes. To keep it simple, we should concentrate on the case for only one active process and one passive process.

The following code (of course only for the active process) is a framework for the next implementations:

```
procedure enter_critical_section();
var
    ip : Address;
begin
    set_breakpoint(passive_process, entry);
    set_breakpoint(passive_process, exit);
    ip := instruction_pointer(passive_process);
    if (entry < ip) and (ip < exit) then
        wait_until_exit(passive_process);
end;
```

```
procedure exit_critical_section();
begin
    remove_breakpoint(passive_process, entry);
    remove_breakpoint(passive_process, exit);
    optional_signal(passive_process);
end;
```

Depending on the kind of blocking behaviour, there are two main differences. A spinning version could use a kind of "*here: jump here*" instruction for the breakpoints and a conditional loop around the range check (incl. getting the instruction pointer). A waiting version could install different functionality at the breakpoints. The first breakpoint should execute a real wait (i.e. for a signal) whereas the second breakpoint informs the active process.



4.5 A first real implementation

Because of the fact that this method disturbs the security model of modern operating systems, an older operating system without real separation of user/kernel mode and without virtual memory has been choosen for a first implementation. My old AMIGA 1200 from the year 1992 with one MC68020 processor, no MMU and the normal AMIGA operating system version 3.1 fits this needs.

4.5.1 One-sided semaphore

Functions for the passive process (only initialization and finalization):

- *create_one_sided_semaphore()*
  - initializes a semaphore for this process
  Supply the entry and exit location of the section, the type of section and which process is the active one. Normally the section type is "critical section". But it is also possible to invert its meaning and to create a "safe section" (e.g. for checkpointing).
- *destroy_one_sided_semaphore()*
  - complement to *create_one_sided_semaphore()*

Functions for the active process:

- *open_one_sided_semaphore()*
  - opens a created semaphore for this process
- *close_one_sided_semaphore()*
  - complement to *open_one_sided_semaphore()*
- *one_sided_semaphore_lock()*
  - similar to *wait()* or *P()*
- *one_sided_semaphore_unlock()*
  - similar to *signal()* or *V()*

The implementation is straight forward as explained above with one exception: There is no public description of where the instruction pointer gets saved and I don't have access to the source for the internal functions (i.e. scheduler). But as expected, the instruction pointer could be found within the (user) stack of its process.

Here are the results of the benchmark tests with two processes (both working on shared data structures) and three different primitives to achieve mutual exclusion (no semaphore ⇒ inconsistency, signal semaphore, one-sided semaphore):

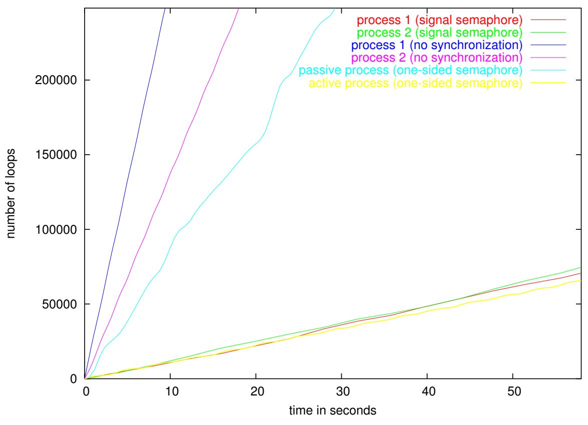


Fig. 1. semaphores (worst case: 1-1 access)

Figure 1 shows the worst case scenario for this method because both processes passing through the critical sections with almost equal frequency (minor deviations can be explained with the behaviour of the scheduler and the periodic instruction sequence of the two processes). The same benchmark for different access behaviour (one process enters a critical section approximately 25 times more frequent than the other process) leads to the following results:

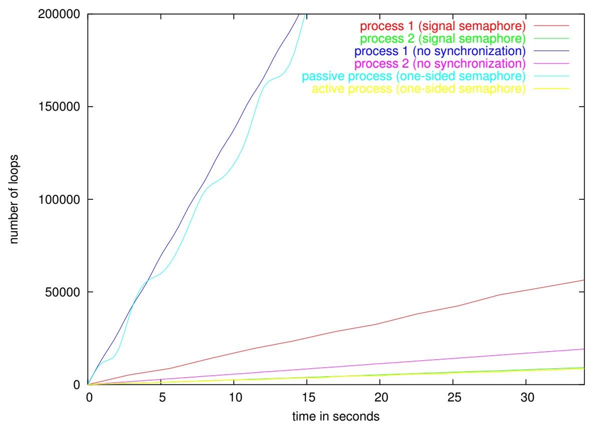


Fig. 2. semaphores (25-1 access)

This implementation of the one-sided semaphore is nearly always faster than the original semaphore construct, supplied (and hopefully tuned) for more than five years with this operating system (available since exec-library v33, release 1.2, 1987). One explanation is the simplified semaphore semantic: This one-sided semaphore works only exactly for two processes (active and passive). So there is no need for a real queue to hold all interested processes. Most of the usual startup overhead could be done during the create and open calls.
The most important fact we could get from these diagrams is the fact that our aim can be achieved: The passive process becomes nearly as fast as a process without any primitive, when the number of collisions (proportional to the access frequency difference) decreases.

4.5.2 One-sided spinlock

To avoid all possible implementation problems that could fake the benchmarks, I tried to implement a one-sided spinlock for this plattform.

The shortest possible spinlock (in assembler-notation [3]) looks like this:

{ enter_critical_section() functionality }

```
spin_loop:
    tas lock_variable
    jne spin_loop
```

{ exit_critical_section() functionality }

```
    clrb lock_variable
```

The *tas* instruction is an atomic test-and-set machine instruction that sets a variable to one and holds its previous value in the flags for the conditional jump afterwards. The *clr* instruction sets this variable back to zero (to allow another process to enter the c.s.).

Both functions for the passive process gets replaced by simple labels (as before for the one-sided semaphore). Since this becomes a spinlock, only the entry breakpoint is necessary. Within an initial phase the two labels have to be stored in a simple array and the (passive) task pointer has to be determined.

{ enter_critical_section() functionality }

```
    movew ♯breakpoint_instruction,section_start
ossl_loop:
    movel passive_task(54)(0),d0
    cmp2l section_bounds,d0
    jcc ossl_loop
```

{ exit_critical_section() functionality }

```
ossl_end:
    movew ♯old_instruction,section_start
```

The first *move* instruction installs the breakpoint at the entry of the critical section (for the passive process). The instruction pointer is hidden within the task structure and is read with a double indirect *move*. The nice *cmp2* instruction checks this pointer against two boundaries (in our case the section boundaries) and informs us of its relative location. If it is located within this section, the *jcc* instruction jumps back to the beginning of this busy waiting loop. In the end, the exit functionality simply removes the breakpoint.

A benchmark (similar to the semaphore benchmark) using three different kinds of spinlock primitives (no spinlock ⇒ inconsistency, usual spinlock, one-sided spinlock) produces these results:

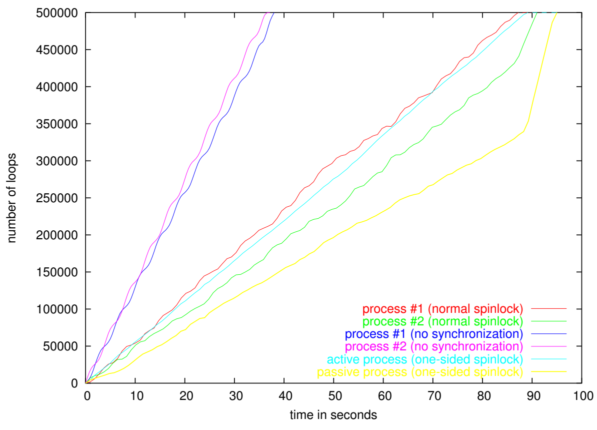


Fig. 3. spinlock (worst case: 1-1 access)

Figure 3 shows again the worst case scenario for this method (as before). Strangely enough the passive

process becomes the slowest process. An explanation could be the fact that this passive process has no influence in a collision case whereas the active process has this power with the breakpoint facility. The active process gets a higher "virtual" priority if it uses one-sided primitives. When no collisions occur anymore, the passive process becomes faster than a process with the usual spinlock (comparision of the steep inclines after the other process finishes).
The same benchmark for a different access behaviour (one process enters a critical section approximately 7 times more frequent than the other process) shows that the one-sided spinlock is better than the normal spinlock:

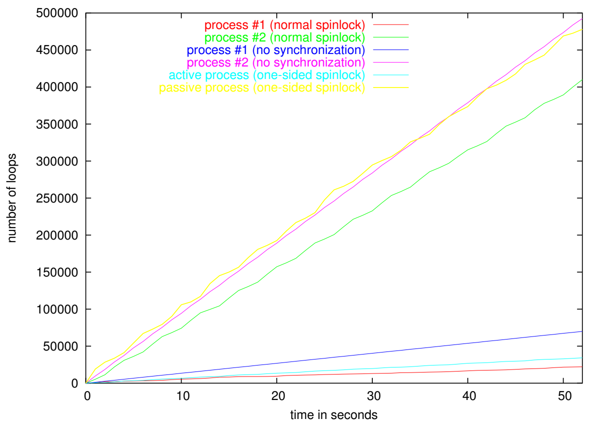


Fig. 4. spinlock (1-7 access)

Both processes (active and passive) gets faster access to a critical section than with the usual (two-sided) spinlock!

4.6 A second real implementation (kernel-mode)

Both controlling and inspecting of another process (described as before) is normally prohibited for user processes. So I transformed the one-sided semaphore to a real Linux (2.4) Kernel object.
Since references to the system call table are not longer supported by linux, a first implementation as a loadable module has been given up. A second attempt resulted in direct patching of the kernel sources.
All structures have been hidden (from the user) in kernel space and all functions became real system calls. The syntax of all functions is similar to the one from the first real implementation except an additional user-tag for create and open calls (like for message queues) to support multiple semaphores per process.

But as system calls are very expensive on todays operating systems/architectures (on average 150ns for an empty system call, called directly via "int $0x80" on a 1,8 GHz Athlon with a 2.4.26 Kernel) the final results are worse (similar to the third implementation, see below).

4.7 Feasibility for replacement of Linux Kernel spinlocks

A spinlock at user level becomes very inefficient because of the restrictions of todays operating systems. Maybe there is some kernel object that could profit from using one-sided spinlocks? So I patched the kernel sources again to count the number of spinlock calls. Three special atomic 64 bit counters, readable through the virtual */proc/stat* file had been installed. Here are some results:

number of calls per second			
working type	spinlock	readlock	writelock
idle state	1,500	85	62
kernel compile	69,316	4,116	3,012
yield bombing	6,197,947	17	203

Against my expectations all spinlocks are used rarely. The Linux Kernel avoids spinlocks if possible and that is very good for the overall performance. The Linux 2.6 Kernel is even better in his locking behaviour. Only "constructed" stress tests lead to high spinlock counts. So I have found no real kernel object which is worthwhile to be protected by one-sided spinlocks (the performance gain would be either hardly measurable or even negative).

4.8 A third real implementation (user-mode)

Because of the bad results for the kernel-mode semaphore, I decided to implement a user-mode semaphore. The instruction pointer information is simply read from the */proc* file system. Software breakpoints are prepared by a mprotect() call to make this memory segment writeable.
Obviously the results could never become faster than for the kernel implementation but there is no big difference in the results. Here is the comparison between *pthread mutexes* and the user implementation of the one-sided semaphore, measured on a dual Intel(R) XEON(TM) CPU with 2.40GHz:

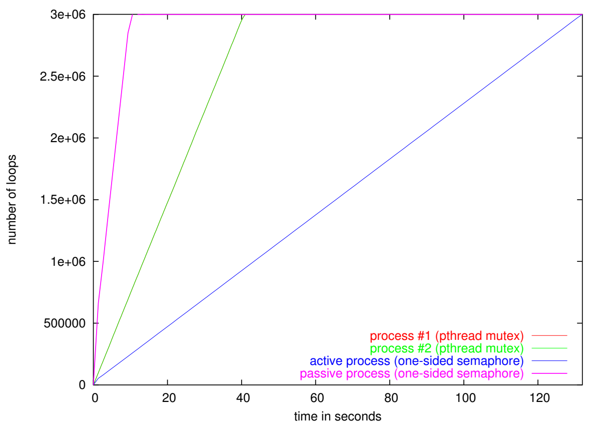


Fig. 5. semaphore (worst case: 1-1 access)

As before, figure 5 shows the worst case scenario. Notice that both threads with the pthread mutexes run nearly exact with the same speed. The passive process is faster than a process using mutexes, but the active one becomes very slow.

Now the results for different access behaviour (one process enters a critical section approximately 56 times more frequent than the other process):

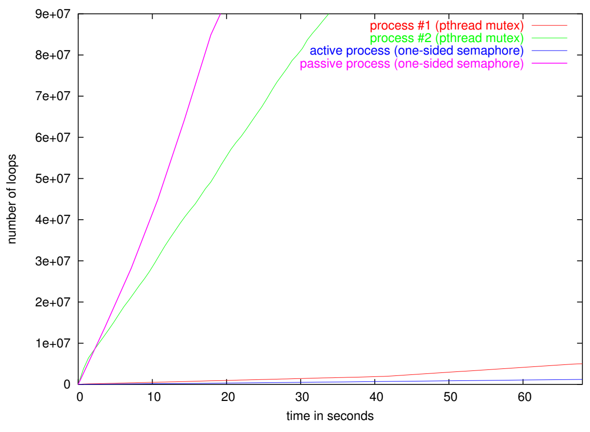


Fig. 6. semaphore (1-56 access)

The active process remains (as expected) slower than a process using normal synchronization techniques. But as the access behaviour differs more and more, both lines converges. Thus the total time for mutual exclusion gets reduced, because the active process is much faster than a process that synchonizes with normal techniques.

5 Outlook and improvements

Normal primitives for mutual exclusion have been developed for more than fourty years now. Special hardware support has been added to make them faster and faster. These special machine instructions, mostly used to build other higher level primitives, are very slow compared to normal processor instructions (see spinlock analysis). If there would be a processor instructions to (only!) read the instruction pointer register(s) in symmetric multiprocessor systems (which should be more than easy) and public readable data structures with the same information for currently non-running processes, this one-sided mutual exclusion method could become a real alternative to existing methods. Today it might only be relevant for very big access differences like for checkpointing purposes, where the normal working process could run without additional delays and a second checkpointing process would be able to force this working process into a stable state where a lightweight checkpointing becomes possible.

References

[1] Stallings, William: *Operating systems - internals and design principles, ISBN 0-13-031999-6, 2001*

[2] The IA-32 Intel(R) Architecture Software Developer's Manual, 2004 *http://www.intel.com/design/Pentium4/documentation.htm*

[3] MOTOROLA M68000 FAMILY Programmer's Reference Manual, 1992

[4] John L. Hennessy, David A. Patterson: *Computer Architecture - A Quantitative Approach, ISBN 1-55860-724-2, 2003*