RJ2516(32946)4/26/79
Computer Science

Research Report

THE CONVOY PHENOMENON

Mike Blasgen
Jim Gray
Mike Mitoma
Tom Price

IBM Research Laboratory
San Jose, California 95193

May 1977 (Revised January 1979)

IBM Research Division
Yorktown Heights, New York · San Jose, California · Zurich, Switzerland

RJ2516(32946)4/26/79
Computer Science

THE CONVOY PHENOMENON

Mike Blasgen
Jim Gray
Mike Mitoma
Tom Price

IBM Research Laboratory
San Jose, California 95193

May 1977 (Revised January 1979)

ABSTRACT: A congestion phenomenon on high-traffic locks is described and a non-FIFO strategy to eliminate such congestion is presented.

CONVOYS DESCRIBED

When driving on a two-lane road with no passing one often encounters clusters of cars. This is because a fast-moving car will soon bump into a slow one. The equilibrium state of such a system is for everyone to be in a convoy behind the slowest car.

In System R [Astrahan], transactions often bump into one another when contending for shared resources. This contention appears as conflicting requests for locks. (You may know locks by the names semaphore, monitor, latch or queue.) Typically, access to these resources follows the protocol:

```
LOCK <resource>;
<operate on resource>;
UNLOCK <resource>;
```

If other processes request the lock while it is granted then they are placed in a queue of waiters and suspended. When the lock becomes available, requests are granted in first-come first-served order. Setting and clearing a lock costs ten instuctions if no waiting is involved. If waiting is involved,it costs 50 instructions plus two process dispatches (i.e. several thousand instructions). (Note: the System R lock manager is much fancier than this [Gray],]Naumanm.)

In what follows three statistics about a particular lock will be of interest: The execution interval of a lock is the average number of instructions executed between successive requests for that lock by a process. The duration of a lock is the average number of instructions executed while the lock is held. The collision cross section of a lock is the fraction of time it is granted. In a uni-processor the collision cross section is (duration/(duration+interval)) ignoring the wait time and task switching time if a request must wait. There are three high-traffic locks in System R regulating access to the buffer pool, recovery log and to system entry/exit. Estimates of these statistics for the high traffic resources of System R are shown below.

RESOURCE	EXECUTION INTERVAL (instructions)	DURATION HELD (instructions)	CROSS SECTION
BUFFER POOL	1000	60	6%
ENTRY-EXIT	1500	70	5%
LOG	20000	300*	1.5%

* The log lock is sometimes held during a log write to disk so this is an average.

In the remainder consider a hypothetical lock L with an execution interval of 1000 instructions, a duration of 100 instructions and hence a cross section of 10%.

Consider what happens if a process P1 stops (goes into wait state) while it holds high traffic lock L:

* All other processes will be scheduled and will more or less immediately request L.
* Each such transaction will find lock L busy and so will wait for it.

The static situation is now:

* P1 is sleeping.
* All other processes are waiting for the lock.

lock L:

```
******
* P1 *<---|P2|<---|P3|<---...<---|Pn|
******
holder    convoy of waiters........
```

The dynamic situation is then:
* The system sleeps until P1 wakes up.
* P1 runs and releases L (after 100 instructions).
* P2 is granted L by P1.
* P1 executes 1000 instructions more and then
* P1 requests L (again) but L is busy (because there are many processes ahead of P1 in the queue and not all these processes will be dispatched before P1 re-requests L).
* P1 enqueues on the lock and goes to sleep.

In an N-process M-processor system, with N>>M, the lock queue will contain N-M processes and each processes will have an execution interval of 1000 instructions. Thus the system is in a situation of lock thrashing. Most of the CPU is dedicated to task switching.

This is a stable phenomenon: new processes are sucked into the convoy and if a process leaves the convoy (for I/O wait or lock wait), when it returns the convoy will probably still exist.

CONVOYS IN SYSTEM R

We have observed the convoy phenomenon in System R. Convoys have also been observed by others in MVS [MVS] and IMS [Obermark].

In System R on VM/370 a process can experience one of five flavors of wait: page fault wait, IO wait, lock wait, quantum runout wait, and time slice wait. When a process stops waiting, it is marked dispatchable. When the dispatcher dispatches a process it is given a quantum The process runs until it page faults, does an I/O, a lock wait or exhausts the quantum. At any of these times the dispatcher subtracts the consumed time from the process time slice. If the result is negative the process goes into time slice wait if the CPU is a scarce resource. If the process still has some time slice left, the process goes into wait state. The dispatcher could do this much faster, but VM/370 requires about 2000 instructions to switch processes.

Clearly, the duration of a high traffic lock should be kept to a minimum. System R code follows the protocols:
* Never lock wait while a high traffic lock is held.
* Never do I/O while a high traffic lock is held (the log lock is an occasional exception to this.)
* Never page fault while a high traffic lock is held (i.e. only access frequently used pages.)

If these rules are followed and if I/O is frequent enough so that almost every quantum ends with an I/O wait or a low-traffic lock wait then the high traffic

locks should almost always be free when a task sleeps.

With probability P a process ends its quantum with an I/O wait or a low-traffic lock wait. P is very close to 1. However, 1-P is not zero if the scheduler is pre-enptive (because the process may not do any I/O for a long time thereby getting quantum runout or it may page fault.) So if the lock is held 10% of the time, some one will sleep holding the lock with probability 0.10*(1-P). This probability is small, but its consequences are so disasterous as to make it a real problem. Namely, such an event will create a convoy on that lock.

Hence, the dispatcher should never interrupt a process holding a high-traffic lock (low traffic locks do not create convoys). Put another way pre-emptive scheduling is bad for a transaction system. The consequence of pre-empting a process which holds a high traffic lock is that a convoy will immediately form on the lock and that it will persist for a very long time. (Note that page faults are a source of pre-emption).

Most of us are stuck with a pre-emptive scheduler (i.e. general purpose operating system with virtual memory). Hence convoys will occur. The problem is to make them evaporate quickly when they do occur rather than have then persist forever.

Before adopting the solutions outined below, 92% of lock waits were for the three high traffic locks. After adopting the solutions described below, lock waits on high traffic locks were reduced by a factor of ten and only 40% of lock waits were for the high traffic locks.

POSSIBLE SOLUTIONS

The simplest solution of all is to run one transaction at a time. Then no locking is required. However, our experiments indicate that even with convoys the response and throughput of System R is improved by multi-programming. This is because I/O and computation can overlap and because short transactions are not delayed by very long ones.

The next simplest solution is to avoid locks. One can go a very long way with shrewd use of atomic machine instructions (compare and swap) and other programming tricks. For example, the system entry-exit lock was eliminated by such techniques. We have done a lot of this, but have been unable to completely eliminate high traffic locks from our programs.

Another strategy is to reduce the traffic on locks by refining:
* the lock granularity (how much is locked),
* the lock mode (non-exclusive requests).

To give an example of finer granularity, IMS was convoying on a lock which controlled the OSAM buffer pool. By partitioning the buffer pool into disjoint sub-pools and associating a lock with each sub-pool the execution interval was increased so that buffer pool convoys disappeared. It was replaced by convoys on the "PI" and log locks. This demonstrates that eliminating one bottleneck simply exposes the next one [Obermark].

To give an example of non-exclusive lock modes, observe that processes adding to the System R log need not acquire it in exclusive mode. Rather, adders can get

it in shared mode and only processes which want to write the log to disk (a relatively rare event) need acquire the log lock in exclusive mode.

Using non-exclusive modes reduces the probability P that one will wait for a request, and refining the granularity (more different locks) increases the execution interval between requests for the same lock (decreasing the traffic on a particular lock).

These techniques make convoys less likely and less stable. But we suspect that convoys will continue to occur. In particular, there was no easy way to fix system entry-exit lock convoys using mode or granularity techniques (we had to resort to special logic for this problem).

We also considered two strategies which seem to have few virtues: spin locks and integration of the dispatcher and lock manager.

Spin locks come in two flavors:

* Busy wait: holds the CPU until quantum runout.
* Lazy wait: branches to the dispatcher and tests the lock the next time it is dispatched.

Spin locks eliminate convoys (as explained below convoys are caused by first-come first-served scheduling, spin locks don't have FCFS the discipline). In one set of experiments we performed, busy wait locks increased system execution time (elapsed) by 75%. Lazy wait locks increased execution time by 20%. That is, the CPU time wasted by spinning is greater than the cost of convoys.

Another way to solve the convoy problem is to involve the dispatcher. Notice in the example that P1 stupidly gave up the lock to P2. If P1 had hung on to the lock until it waited, and P2 did the same then the convoy would flush itself immediately. The obvious solution is to have the dispatcher know about locks and have the dispatcher grant locks when tasks are switched. The arguments against this approach are:

* The book-keeping associated with giving up a lock at task switch is intimidating.
* For reasons of modularity, the dispatcher should not know about locks, they are a higher level notion.
* The solution does not generalize to multiple processors. * The solution does not address pre-emption due to page faults.

A SOLUTION

The key issue of convoys is associated with the granting of locks in first-come first-served order. So we elect to grant all lock requests in random order in the hope that eventually everyone will get service. In theory, some process might "starve" (never be granted its request) but in fact the underlying scheduler and the stochastic nature of real systems cause each process to eventually get service.

The proposed solution is:

```
*    When releasing a lock, broadcast to all waiters that the lock is free:
        DO;
          CONVOY=LATCH.QUEUE;          /* atomic pair */
          LATCH=FREE;                  /* atomic pair */
          DO WHILE(CONVOY ¬= NIL);
            WAKEUP FIRST OF CONVOY;    /* CAR of list */
            CONVOY = REST OF CONVOY;   /* CDR of list */
          END;
        END;
*    When acquiring a lock:
          DO WHILE (LATCH ¬ MINE);
            IF LATCH = FREE THEN LATCH = MINE;
            ELSE
               ENQUEUE ON LATCH;
               SLEEP;
          END;
```

First consider the properties of this algorithm on a uni-processor. If a convoy exists, the releasor (P1) of the lock will

* dequeue all members of the convoy from the lock.
* mark the lock as free.
* signal all members of the convoy.

Now the releasor continues to run (after all he just woke up from a wait and so has almost a full quantum.) If he needs the lock again it is free. If he goes into I/O or lock wait, he will not hold the lock. So with probability P (>.99), he will terminate with the lock free. Now one of the members of the convoy will run (P2). He will start with the lock free and so he will be able to acquire it. (There is no queue on the lock.) With probability P he will terminate with the lock free. If there are N processes in the convoy, it will disappear with probability P**N.

Now consider the multiprocessor case. The logic above applies to waiters in a convoy but there is another problem. If a lock is held 10% of the time, then the probability that two processors will bump into one another is .01. If they do bump into one another, a convoy will form. They will convoy on one another. The multi-processor case is much like the page fault case: the other processors look at the lock at a random (and frequent) points. Hence they look at it during critical sections. The solution seems to be for the lock requestor to spin for a few instructions in the hope that the lock will become free. If the lock does not free, the process should enqueue and sleep. This spin time is affected by the probability the lock is held, the number of processors, the cost of task switch, the time to service interrupts, and the expected length of the convoy. The worst case is immediately after a convoy is broadcast to. 500 instructions might be nice spin time for System R on two CPUs. One would spin for 500 instructions in the lock request operation.

Dieter Gawlick points out that the broadcast approach is likely to cause contention whenever a convoy is breaking up. He cites the situation in which all processes are waiting for the log lock being held by the checkpoint process (in IMS Fast Path). When the checkpoint completes, the waiting processes will all contend for the log lock.

Gawlick suggests that instead of broadcasting, simply mark the latch as free and

wakeup only the first waiter in the queue. In this way the current holder and first waiter can contend for the lock but access to the lock is approximately first come first served. This solution is workable and in fact has been used to solve a convoy problem of IMS. But our analysis indicates that all members of the convoy will be woken up almost immediately if the latch has a short execution interval because each unlock wakes up a new waiting process as it marks the latch free. Hence both solutions seem acceptable.

REFERENCES

[Astrahan] Astrahan, et. al., "System R: A Relational Approach to Data Management," ACM TODS, Vol. 1,No. 2, June 1976, pp. 97-137.

[Gray] Gray J. N., Lorie R. P., Putzolu G. F., and Traiger I. L., "Granularity of Locks and Degrees of Consistency in a Shared Data Base", Modeling in Data Base Management Systems, Nijssen editor, North Holland, 1976. pp. 365-394.

[Nauman] Nauman J. S., "Observations on Sharing in Data Base Systems" IBM Technical Report TR 03.047, IBM Santa Teresa Lab., IBM, Coyote California. May 1978.

[MVS] OS/VS2 System Logic Library, Vol. 4, page 159, Form No. SY28-0716, IBM, White Plains, 1977.

[Obermark] Obermark R.L., Personal communication, IBM Systems Center, Palo Alto California.