Semaphores in Plan 9

Sape Mullender

Bell Laboratories
2018 Antwerp, Belgium

Russ Cox[*]

MIT CSAIL
Cambridge, Massachusetts 02139

1. Introduction

Semaphores are now more than 40 years old. Edsger W. Dijkstra described them in EWD 74 [Dijkstra, 1965 (in Dutch)]. A semaphore is a non-negative integer with two operations on it, *P* and *V*. The origin of the names *P* and *V* is unclear. In EWD 74, Dijkstra calls semaphores *seinpalen* (Dutch for signalling posts) and associates *V* with *verhoog* (increment/increase) and *P* with *prolaag*, a non-word resembling *verlaag* (decrement/decrease). He continues, *''Opm. 2. Vele seinpalen nemen slechts de waarden 0 en 1 aan. In dat geval fungeert de V-operatie als 'baanvak vrijgeven'; de P-operatie, de tentatieve passering, kan slechts voltooid worden, als de betrokken seinpaal (of seinpalen) op veilig staat en passering impliceert dan een op onveilig zetten.''* (''Remark 2. Many signals assume only the values 0 and 1. In that case the V-operation functions as 'release block'; the P-operation, the tentative passing, can only be completed, if the signal (or signals) involved indicates clear, and passing then implies setting it to stop.'') Thus, it may be that *P* and *V* were inspired by the railway terms *passeer* (pass) and *verlaat* (leave).

We discard the railway terminology and use the language of locks: *P* is *semacquire* and *V* is *semrelease*. The C declarations are:

```
int  semacquire(long *addr, int block);
long semrelease(long *addr, long count);
```

Semacquire waits for the semaphore value **addr* to become positive and then decrements it, returning 1; if the *block* flag is zero, *semacquire* returns 0 rather than wait. If *semacquire* is interrupted, it returns -1. *Semrelease* increments the semaphore value by the specified count.

Plan 9 [Pike et al., 1995] has traditionally used a different synchronization mechanism, called rendezvous. Rendezvous is a symmetric mechanism; that is, it does not assign different roles to the two processes involved. The first process to call rendezvous will block until the second does. In contrast, semaphores are an asymmetric mechanism: the process executing *semacquire* can block but the process executing

[*] Now at Google, Mountain View, California 94043

semrelease is guaranteed not to. We added semaphores to Plan 9 to provide a way for a real-time process to wake up another process without running the risk of blocking. Since then, we have also used semaphores for efficient process wakeup and locking.

2. Hardware primitives

The implementations in this paper assume hardware support for atomic read-modify-write operations on a single memory location. The fundamental operation is ‘‘compare and swap,’’ which behaves like this C function *cas*, but executes atomically:

```
int
cas(long *addr, long old, long new)
{
    /* Executes atomically. */
    if(*addr != old)
        return 0;
    *addr = new;
    return 1;
}
```

In one atomic operation, *cas* checks whether the value **addr* is equal to *old* and, if so, changes it to *new*. It returns a flag telling whether it changed **addr*.

Of course, *cas* is not implemented in C. Instead, we must implement it using special hardware instructions. All modern processors provide a way to implement compare and swap. The x86 architecture (since the 486) provides a direct compare and swap instruction, CMPXCHG. Other processors—including the Alpha, ARM, MIPS, and PowerPC—provide a pair of instructions called load linked (LL) and store conditional (SC). The LL instruction reads from a memory location, and SC writes to a memory location only if (1) it was the memory location used in the last LL instruction, and (2) that location has not been changed since the LL. On those systems, compare and swap can be implemented in terms of LL and SC.

The implementations also use an atomic addition operation *xadd* that atomically adds to a value in memory, returning the new value. We don’t need additional hardware support for *xadd*, since it can be implemented using *cas*:

```
long
xadd(long *addr, long delta)
{
    long v;

    for(;;){
        v = *addr;
        if(cas(addr, v, v+delta))
            return v+delta;
    }
}
```

3. User-space semaphores

We implemented *semacquire* and *semrelease* as kernel-provided system calls. For efficiency, it is useful to have a semaphore implementation that, if there is no contention, can run entirely in user space, only falling back on the kernel to handle contention. Figure 1 gives the implementation. The user space semaphore, a *Usem*, consists of a user-level semaphore value *u* and a kernel value *k*:

```
typedef struct Usem Usem;
struct Usem {
    long    u;
    long    k;
};
```

When *u* is non-negative, it represents the actual semaphore value. When *u* is negative, the semaphore has value zero: acquirers must wait on the kernel semaphore *k* and releasers must wake them up.

```
void
usemacquire(Usem *s)
{
    if(xadd(&s->u, -1) < 0)
        while(semacquire(&s->k, 1) < 0){
            /* Interrupted, retry */
        }
}

void
usemrelease(Usem *s)
{
    if(xadd(&s->u, 1) <= 0)
        semrelease(&s->k, 1);
}
```

If the semaphore is uncontended, the *xadd* in *usemacquire* will return a non-negative value, avoiding the kernel call. Similarly, the *xadd* in *usemrelease* will return a positive value, also avoiding the kernel call.

4. Thread Scheduling

In the Plan 9 thread library, a program is made up of a collection of processes sharing memory. A thread is a coroutine assigned to a particular process. Within a process, threads schedule cooperatively. Each process manages the threads assigned to it, and the process schedulers run almost independently. The one exception is that a thread in one process might go to sleep (for example, waiting on a channel operation) and be woken up by a thread in a different process. The two processes need a way to coordinate, so that if the first has no runnable threads, it can go to sleep in the kernel, and then the second process can wake it up.

The standard Plan 9 thread library uses rendezvous to coordinate between processes. The processes share access to each other's scheduling queues: one process is manipulating another's run queue. The processes must also share a flag protected by a spin lock to coordinate, so that either both processes decide to call rendezvous or neither does.

For the real-time thread library, we wanted to remove as many sources of blocking as possible, including these locks. We replaced the locked run queue with a non-blocking array-based implementation of a producer/consumer queue. That implementation is beyond the scope of this paper. After making that change, the only lock remaining in the scheduler was the one protecting the ‘‘whether to rendezvous’’ flag. To eliminate that one, we replaced the rendezvous with a user-space semaphore counting the number of threads on the queue.

To wait for a thread to run, the process's scheduler decrements the semaphore. If the run queue is empty, the *usemacquire* will block until it is not. Having done so, it is guaranteed that there is a thread on the run queue:

```
// Get next thread to run
static Thread*
runthread(void)
{
    Proc *p;

    p = thisproc();
    usemacquire(&p->nready);
    return qget(&p->ready);
}
```

Similarly, to wake up a thread (even one in another process), it suffices to add the thread to its process's run queue and then increment the semaphore:

```
// Wake up thread t to run in its process.
static void
wakeup(Thread *t)
{
    Proc *p;

    p = t->p;
    qput(&p->ready, t);
    usemrelease(&p->nready);
}
```

This implementation removes the need for the flag and the lock; more importantly, the process executing *threadwakeup* is guaranteed never to block, because it executes *usemrelease*, not *usemacquire*.

5. Replacing spin locks

The Plan 9 user-level *Lock* implementation is an adapted version of the one used in the kernel. A lock is represented by an integer value: 0 is unlocked, non-zero is locked. A process tries to grab the lock by using a test-and-set instruction to check whether the value is 0 and, if so, set it to a non-zero value. If the lock is unavailable, the process loops, trying repeatedly. In a multiprocessor kernel, this is a fine lock implementation: the lock is held by another processor, which will unlock it soon. In user space, this implementation has bad interactions with the scheduler: if the lock is held by another process that has been preempted, spinning for the lock will not accomplish anything. The user-level lock implementation addresses this by rescheduling itself (with *sleep(0)*) between attempts after the first thousand unsuccessful attempts. Eventually it backs off more, sleeping for milliseconds at a time between lock attempts.

We replaced these spin locks with a semaphore-based implementation. Using semaphores allows the process to tell the kernel exactly what it is waiting for, avoiding bad interactions with the scheduler like the one above. The semaphore-based implementation represents a lock as two values, a user-level key and a kernel semaphore:

```
struct Lock
{
    long key;
    long sem;
};
```

The *key* counts the number of processes interested in holding the lock, including the one that does hold it. Thus if *key* is 0, the lock is unlocked. If *key* is 1, the lock is held. If *key* is larger than 1, the lock is held by one process and there are *key*−1 processes waiting to acquire it. Those processes wait on the semaphore *sem*.

```
void
lock(Lock *l)
{
    if(xadd(&l->key, 1) == 1)
        return; // changed from 0 -> 1: we hold lock
    // otherwise wait in kernel
    while(semacquire(&l->sem, 1) < 0){
        /* interrupted; try again */
    }
}

void
unlock(Lock *l)
{
    if(xadd(&l->key, -1) == 0)
        return; // changed from 1 -> 0: no contention
    semrelease(&l->sem, 1);
}
```

Like the user-level semaphore implementation described above, the lock implementation handles the uncontended case without needing to enter the kernel.

The one significant difference between the user-level semaphores above and the semaphore-based locks described here is the interpretation of the user-space value. Plan 9 convention requires that a zeroed *Lock* structure be an unlocked lock. In contrast, a zeroed *Usem* structure is analogous to a locked lock: a *usemacquire* on a zeroed *Usem* will block.

6. Kernel Implementation of Semaphores

Inside the Plan 9 kernel, there are two kinds of locks: the spin lock *Lock* spins until the lock is available, and the queuing lock *QLock* reschedules the current process until the lock is available. Because accessing user memory might cause a lengthy page fault, the kernel does not allow a process to hold a *Lock* while accessing user memory. Since the semaphore is stored in user memory, then, the obvious implementation is to acquire a *QLock*, perform the semaphore operations, and then release it. Unfortunately, this implementation could cause *semrelease* to reschedule while acquiring the *QLock*, negating the main benefit of semaphores for real-time processes. A more complex implementation is needed. This section documents the implementation. It is not necessary to understand the rest of the paper and can be skipped on first reading.

Each *semacquire* call records its parameters in a *Sema* data structure and adds it to a list of active calls associated with a particular *Segment* (a shared memory region). The *Sema* structure contains a kernel *Rendez* for use by sleep and wakeup (see [Pike et al., 1991]), the address, and a *waiting* flag:

```
struct Sema
{
    Rendez;
    long *addr;
    int   waiting;
    Sema *next;
    Sema *prev;
};
```

The list is protected by a *Lock*, which cannot cause the process to reschedule. The semaphore value **addr* is stored in user memory. Thus, we can access the list only when holding the lock and we can access the semaphore value only when not holding the lock. The helper functions

```
void    semqueue(Segment *s, long *addr, Sema *p);
void    semdequeue(Segment *s, long *addr, Sema *p);
void    semwakeup(Segment *s, long *addr, int n);
```

all manipulate the segment's list of *Sema* structures. They acquire the associated *Lock*, perform their operations, and release the lock before returning. *Semqueue* and *semdequeue* add *p* to or remove *p* from the list. *Semwakeup* walks the list looking for *n* *Sema* structures with *p.waiting* set. It clears *p.waiting* and then wakes up the corresponding process.

Using those helper functions, the basic implementation of *semacquire* and *semrelease* is:

```
int
semacquire(Segment *s, long *addr)
{
    Sema phore;

    semqueue(s, addr, &phore);
    for(;;){
        phore.waiting = 1;
        if(canacquire(addr))
            break;
        sleep(&phore, semawoke);
    }
    semdequeue(s, &phore);
    semwakeup(s, addr, 1);
    return 1;
}

long
semrelease(Segment *s, long *addr, long n)
{
    long v;

    v = xadd(addr, n);
    semwakeup(s, addr, n);
    return v;
}
```

(This version omits the details associated with returning –1 when interrupted and also with non-blocking calls.)

Semacquire adds a *Sema* to the segment's list and sets *phore.waiting*. Then it attempts to acquire the semaphore. If it is unsuccessful, it goes to sleep. To avoid missed wakeups, *sleep* calls *semawoke* before committing to sleeping; *semawoke* simply checks *phore.waiting*. Eventually, *canacquire* returns true, breaking out of the loop. Then *semacquire* removes its *Sema* from the list and returns.

The call to *semwakeup* at the end of *semacquire* corrects a subtle race that we found using Spin. Suppose process A calls *semacquire* and the semaphore has value 1. *Semacquire* queues its *Sema* and sets *phore.waiting*, *canacquire* succeeds (the semaphore value is now 0), and *semacquire* breaks out of the loop. Then process B calls *semacquire*: it adds itself to the list, fails to acquire the semaphore (the value is 0), and goes to sleep. Now process C calls *semrelease*: it increments the semaphore (the value is now 1) and looks for a single *Sema* in the list to wake up. It finds A's, checks that *phore.waiting* is set, and then calls the kernel *wakeup* to wake A. Unfortunately, A never went to sleep. The wakeup is lost on A, which had already acquired the semaphore. If A simply removed its *Sema* from the list and returned, the semaphore value would be 1 with B still asleep. To account for the possibly lost wakeup, A must trigger one extra *semwakeup* as it returns. This avoids the race, at the cost of an unnecessary (but harmless) wakeup when the race has not happened.

7. Performance

To measure the cost of semaphore synchronization, we wrote a program in which two processes ping-pong between two semaphores:

> Process 1 blocks on the acquisition of Semaphore 1,
> Process 2 releases Semaphore 1 and blocks on Semaphore 2,
> Process 1 releases Semaphore 2 and blocks on Semaphore 1,

This loop executes a million times. We also timed a program that does two million acquires and two million releases on a semaphore initialized to two million, so that none of the calls would block. In both cases, there were a total of four million system calls; the ping-pong case adds two million context switches. Table 1 gives the results.

		time per system call (microseconds)		
processor	*cpus*	*ping-pong*	*semacquire*	*semrelease*
PentiumIII/Xeon, 598 MHz	1	2.18	1.35	1.91
PentiumIII/Xeon, 797 MHz	2	0.887	0.949	1.38
PentiumIV/Xeon, 2196 MHz	4	0.970	1.38	1.84
AMD64, 2201 MHz	2	1.08	0.266	0.326

Table 1 Semaphore system call performance.

		time per lock operation (microseconds)	
processor	*cpus*	*spin locks*	*semaphore locks*
PentiumIII/Xeon, 598 MHz	1	5.4	5.4
PentiumIII/Xeon, 797 MHz	2	18.2	5.6
AMD64, 2201 MHz	2	22.6	2.5
PentiumIV/Xeon, 2196 MHz	4	43.8	4.9

Table 2 Performance of spin locks versus semaphore locks.

Next, we looked at lock performance, comparing the conventional Plan 9 locks from *libc* to the new ones using semaphores for sleep and wakeup. We ran Doug McIlroy's power series program [McIlroy, 1990], which spends almost all its time in channel communication. The Plan 9 thread library's channel implementation uses a single global lock to coordinate all channel activity, inducing a large amount of lock contention. The application creates a thousand processes and makes 207,631 lock calls. The number of locks (in the semaphore version) that require waiting (i.e., a semacquire is done) varies wildly. In 20 runs, the smallest number we saw was 127, the largest was 490, and the average was 288.

Table 2 shows the performance results. Surprisingly, the performance difference was most pronounced on multiprocessors. Naively, one would expect that spinning would have some benefit on multiprocessors whereas it could have no benefit on uniprocessors, but it turns out that spinning without rescheduling (the first 1000 tries) has no effect on performance. Contention only occurs some 500 or so times, and the time it takes to spin 500,000 times is in the noise. The difference between uniprocessors and multiprocessors here is that on uniprocessors, the first *sleep(0)* will put the process waiting for the lock at the back of the ready queue so that, by the time it is scheduled again, the lock will likely be available. On multiprocesssors, contention from other processes running simultaneously makes yielding less effective. It is also likely that the repeated atomic read-modify-write instructions, as in the tight loop of the spin lock, can slow the entire multiprocessor.

The performance of the semaphore-based lock implementation is sometimes much better, and never noticeably worse, than the spin locks. We will replace the spin lock implementation in the Plan 9 distribution soon.

8. Comparison with other approaches

Any operating system with cooperating processes must provide an interprocess synchronization mechanism. It is instructive to contrast the semaphores described here with mechanisms in other systems.

Many systems—for example, BSD, Mach, OS X, and even System V UNIX—provide semaphores [Bach, 1986]. In all those systems, semaphores must be explicitly allocated and deallocated, making them more cumbersome to use than *semacquire* and *semrelease*. Worse, semaphores in those systems occupy a global id space, so that it is possible to run the system out of semaphores just by running programs that allocate semaphores but neglect to deallocate them (or crash). The Plan 9 semaphores identify semaphores by a shared memory location: two processes are talking about the same semaphore if **addr* is the same word of physical memory in both. Further, there is no kernel-resident semaphore state except when *semacquire* is blocking. This makes the semaphore leaks of System V impossible.

Linux provides a lower-level system call named futex [Franke and Russell, 2002]. Futex is essentially ‘‘compare and sleep,’’ making it a good match for compare and swap-based algorithms. Futex also matches processes based on shared physical memory, avoiding the System V leak problem. Because futex only provides ‘‘compare and sleep’’ and ‘‘wakeup,’’ futex-based algorithms are required to handle the uncontended cases in user space, like our user-level semaphore and new lock implementations do. This makes futex-based implementations efficient; unfortunately, they are also quite subtle. The original example code distributed with futexes was wrong; a correct version was only published a year later [Drepper, 2003]. In contrast, semaphores are less general but easier to understand and to use correctly.

References

[Bach, 1986]
M.J. Bach, *The Design of the UNIX Operating System*, Prentice Hall, 1986

[Dijkstra, 1965]
E.W. Dijkstra, ''Over Seinpalen'', *EWD74*, 1965.
(http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF,
http://www.cs.utexas.edu/users/EWD/transcriptions/EWD00xx/EWD74.html)

[Drepper, 2003]
U. Drepper, ''Futexes are Tricky,'' published online at
http://people.redhat.com/drepper/futex.pdf.

[Franke and Russell, 2002]
''Fuss, Futexes, and Furwocks: Fast Userlevel Locking in Linux,'' *Proceedings of the 2002 Ottawa Linux Symposium*, Ottawa, Canada, 2002, pp. 479–495.

[Holzmann, 1991]
G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991

[Pike et al., 1991]
R. Pike, D. Presotto, K. Thompson, and G. Holzmann, ''Process sleep and wakeup on a shared memory multiprocessor,'' *Proceedings of the Spring 1991 EurOpen Conference*, Tromsø, Norway, 1991, pp. 161–166.

[Pike et al., 1995]
R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom, ''Plan 9 from Bell Labs'', *Computing Systems*, **8**(3), Summer 1995, pp. 221–254

[Plan 9, 2000]
Plan 9 Manual, 3rd edition published online at
http://plan9.bell-labs.com/sys/man