Spinlock versus Semaphore

What is the basic differences between Semaphores & Spinlock? & In what best situations or conditions, we can use these.

c++ c linux unix operating-system

edited May 21 '12 at 7:26
Mahmoud Al-Qudsi
15.4k 7 33 74

asked Oct 12 '08 at 18:58
iankits
806 3 14 29

10 Answers

Spinlock and semaphore differ mainly in four things:

1. What they are
A *spinlock* is one possible implementation of a lock, namely one that is implemented by busy waiting ("spinning"). A semaphore is a generalization of a lock (or, the other way around, a lock is a special case of a semaphore). Usually, *but not necessarily*, spinlocks are only valid within one process whereas semaphores can be used to synchronize between different processes, too.

A lock works for mutual exclusion, that is **one** thread at a time can acquire the lock and proceed with a "critical section" of code. Usually, this means code that modifies some data shared by several threads.
A *semaphore* has a counter and will allow itself being acquired by **one or several** threads, depending on what value you post to it, and (in some implementations) depending on what its maximum allowable value is.

Insofar, one can consider a lock a special case of a semaphore with a maximum value of 1.

2. What they do
As stated above, a spinlock is a lock, and therefore a mutual exclusion (strictly 1 to 1) mechanism. It works by repeatedly querying and/or modifying a memory location, usually in an atomic manner. This means that acquiring a spinlock is a "busy" operation that possibly burns CPU cycles for a long time (maybe forever!) while it effectively achieves "nothing".
The main incentive for such an approach is the fact that a context switch has an overhead equivalent to spinning a few hundred (or maybe thousand) times, so if a lock can be acquired by burning a few cycles spinning, this may overall very well be more efficient. Also, for realtime applications it may not be acceptable to block and wait for the scheduler to come back to them at some far away time in the future.

A semaphore, by contrast, either does not spin at all, or only spins for a very short time (as an optimization to avoid the syscall overhead). If a semaphore cannot be acquired, it blocks, giving up CPU time to a different thread that is ready to run. This may of course mean that a few milliseconds pass before your thread is scheduled again, but if this is no problem (usually it isn't) then it can be a very efficient, CPU-conservative approach.

3. How they behave in presence of congestion
It is a common misconception that spinlocks or lock-free algorithms are "generally faster", or that they are only useful for "very short tasks" (ideally, no synchronization object should be held for longer than absolutely necessary, ever).
The one important difference is how the different approaches behave *in presence of congestion*.

A well-designed system normally has low or no congestion (this means not all threads try to acquire the lock at the exact same time). For example, one would normally *not* write code that acquires a lock, then loads half a megabyte of zip-compressed data from the network,

decodes and parses the data, and finally modifies a shared reference (append data to a container, etc.) before releasing the lock. Instead, one would acquire the lock only for the purpose of accessing the *shared resource*.
Since this means that there is considerably more work outside the critical section than inside it, naturally the likelihood for a thread being inside the critical section is relatively low, and thus few threads are contending for the lock at the same time. Of course every now and then two threads will try to acquire the lock at the same time (if this *couldn't* happen you wouldn't need a lock!), but this is rather the exception than the rule in a "healthy" system.

In such a case, a spinlock *greatly* outperforms a semaphore because if there is no lock congestion, the overhead of acquiring the spinlock is a mere dozen cycles as compared to hundreds/thousands of cycles for a context switch or 10-20 million cycles for losing the remainder of a time slice.

On the other hand, given high congestion, or if the lock is being held for lengthy periods (sometimes you just can't help it!), a spinlock will burn insane amounts of CPU cycles for achieving nothing.
A semaphore (or mutex) is a much better choice in this case, as it allows a different thread to run *useful* tasks during that time. Or, if no other thread has something useful to do, it allows the operating system to throttle down the CPU and reduce heat / conserve energy.

Also, on a single-core system, a spinlock will be quite inefficient in presence of lock congestion, as a spinning thread will waste its complete time waiting for a state change that cannot possibly happen (not until the releasing thread is scheduled, which *isn't happening* while the waiting thread is running!). Therefore, given *any* amount of contention, acquiring the lock takes around 1 1/2 time slices in the best case (assuming the releasing thread is the next one being scheduled), which is not very good behaviour.

4. How they're implemented
A semaphore will nowadays typically wrap `sys_futex` under Linux (optionally with a spinlock that exits after a few attempts).
A spinlock is typically implemented using atomic operations, and without using anything provided by the operating system. In the past, this meant using either compiler intrinsics or non-portable assembler instructions. Meanwhile both C++11 and C11 have atomic operations as part of the language, so apart from the general difficulty of writing provably correct lock-free code, it is now possible to implement lock-free code in an entirely portable and (almost) painless way.

edited Jun 21 '13 at 10:14

answered Jun 20 '13 at 18:54

Damon
29.9k 5 56 86

> “Also, on a single-core system, a spinlock will be quite inefficient in presence of lock congestion, as a spinning thread will waste its complete time waiting for a state change that cannot possibly happen” : there is also (at least on Linux) the `spin_trylock` , which returns immediately with an error code, if the lock could not be acquired. A spin-lock is not always that harsh. But using `spin_trylock` requires, for an application, to be properly designed that way (probably a queue of pending operations, and here, selecting the next one, leaving the actual on the queue). – Hibou57 Jul 30 '13 at 1:11

very simply, a semaphore is a "yielding" synchronisation object, a spinlock is a 'busywait' one. (there's a little more to semaphores in that they synchronise several threads, unlike a mutex or guard or monitor or critical section that protects a code region from a single thread)

You'd use a semaphore in more circumstances, but use a spinlock where you are going to lock for a very short time - there is a cost to locking especially if you lock a lot. In such cases it can be more efficient to spinlock for a little while waiting for the protected resource to become unlocked. Obviously there is a performance hit if you spin for too long.

typically if you spin for longer than a thread quantum, then you should use a semaphore.

answered Oct 12 '08 at 19:06

gbjbaanb
33.9k 4 56 104

Over and above what Yoav Aviram and gbjbaanb said, the other key point used to be that you would never use a spin-lock on a single-CPU machine, whereas a semaphore would make sense on such a machine. Nowadays, you are frequently hard-pressed to find a machine without multiple cores, or hyperthreading, or equivalent, but in the circumstances that you have just a single CPU, you must use semaphores. (I trust the reason is obvious; if the single CPU is busy waiting for something else to release the spin-lock, but it is running on the only CPU, the lock is unlikely to be released.)

answered Oct 12 '08 at 19:53

Jonathan Leffler
335k 45 349 619

7 I'd like to second how important it is not to use spinlocks on single threaded systems. They are a the ticked to priority inversion problems. And trust me: You don't want to debug these kind of bugs. – Nils Pipenbrinck Oct 12 '08 at 20:06

2 spinlocks are all over in the Linux kernel, regardless if you have one ore more CPUs. What do you mean exactly? – Prof. Falken May 7 '10 at 10:24

@Amigable: by definition, a spinlock means that the current thread on the CPU is waiting for something else to release the locked object. If the only active thing that can change the lock is the current CPU, the lock will not be freed by spinning. If something else - a DMA transfer or other I/O controller can release the lock, all well and good. But spinning when nothing else can release the lock is not very sensible - you might as well yield the CPU to another process now as wait to be preempted. – Jonathan Leffler May 7 '10 at 14:46

1 I may very well be wrong, but I was under the impression that a re-entrant (single CPU) Linux kernel may interrupt a running spin lock. – Prof. Falken May 10 '10 at 12:29

2 @Amigable: there's a chance I'm wrong too, but I think I'm close to the classic definition of a spinlock. With pre-emptive scheduling, a process might spin on a lock until the end of its time slice, or until an interrupt causes it to yield, but if another process must provide the condition that allows the spinlock to lock, a spinlock is not a good idea on a single CPU machine. The system I work on has spinlocks and has a configurable upper bound on the number of spins before it goes into a non-busy wait mode. This is a user-level spin-lock; there might be a difference down in the kernel. – Jonathan Leffler May 10 '10 at 13:18

From Linux Device Drivers by Rubinni

> Unlike semaphores, spinlocks may be used in code that cannot sleep, such as interrupt handlers

edited Nov 17 '11 at 4:23

Jack
5,964 1 19 41

answered Nov 17 '11 at 3:15

Zbyszek
171 1 2

I am not a kernel expert but here are few points:

Even uniprocessor machine can use spin-locks if kernel preemption is enabled while compiling the kernel. If kernel preemption is disabled then spin-lock (perhaps) expands to a *void* statement.

Also, when we are trying to compare Semaphore vs Spin-lock, I believe semaphore refers to the one used in kernel - NOT the one used for IPC (userland).

Basically, spin-lock shall be used if critical section is small (smaller than the overhead of sleep/wake-up) and critical section does not call anything that can sleep! A semaphore shall be used if critical section is bigger and it can sleep.

Raman Chalotra.

edited Aug 24 '13 at 9:07

Viet
5,069 17 66 116

answered Jun 21 '10 at 18:47

Raman Chalotra
81 1 1

Spinlock refers to an implementation of inter-thread locking using machine dependent assembly instructions (such as test-and-set). It is called a spinlock because the thread simply waits in a loop ("spins") repeatedly checking until the lock becomes available (busy wait).

Spinlocks are used as a substitute for mutexes, which are a facility supplied by operating systems (not the CPU), because spinlocks perform better, if locked for a short period of time.

A Semaphor is a facility supplied by operating systems for IPC, therefor it's main purpose is inter-process-communication. Being a facility supplied by the operating system it's performance will not be as good as that of a spinlock for inter-thead locking (although possible). Semaphores are better for locking for longer periods of time.

That said - implementing splinlocks in assembly is tricky, and not portable.

edited Oct 12 '08 at 20:49

answered Oct 12 '08 at 19:18

yoav.aviram
1,060 9 17

4 All multi-threading CPUs need a spinlock instruction ("test and set") and it's always implemented as a single instruction in hardware because there would otherwise always be a race condition in which more than one thread thought it "owned" the protected resource. – Richard T Oct 12 '08 at 19:33

I'm not sure you understand semaphores... see what Dijkstra said: cs.cf.ac.uk/Dave/C/node26.html – gbjbaanb Oct 12 '08 at 19:52

POSIX makes a distinction between a semaphore shared by threads, and a semaphore shared by processes. – Greg Rogers Oct 12 '08 at 20:19

2 Semaphores are for inter-process synchronization, not communication. – Johan Bezem Nov 15 '11 at 10:03

I would like to add my observations, more general and not very Linux-specific.

Depending on the memory architecture and the processor capabilities, you might need a spin-lock in order to implement a semaphore on a multi-core or a multiprocessor system, because in such systems a race condition might occur when two or more threads/processes want to acquire a semaphore.

Yes, if your memory architecture offers the locking of a memory section by one core/processor delaying all other accesses, and if your processors offers a test-and-set, you may implement a semaphore without a spin-lock (but very carefully!).

However, as simple/cheap multi-core systems are designed (I'm working in embedded systems), not all memory architectures support such multi-core/multiprocessor features, only test-and-set or equivalent. Then an implementation could be as follows:

- acquire the spin-lock (busy waiting)
- try to acquire the semaphore
- release the spin-lock
- if the semaphore was not successfully acquired, suspend the current thread until the semaphore is released; otherwise continue with the critical section

Releasing the semaphore would need to be implemented as follows:

- acquire the spin-lock
- release the semaphore
- release the spin-lock

Yes, and for simple binary semaphores on an OS-level it would be possible to use only a spin-lock as replacement. But only if the code-sections to be protected are really very small.

As said before, if and when you implement your own OS, make sure to be careful. Debugging such errors is fun (my opinion, not shared by many), but mostly very tedious and difficult.

answered Nov 15 '11 at 10:02

Johan Bezem
1,592 1 9 32

A "mutex" (or "mutual exclusion lock") is a signal that two or more asynchronous processes can use to reserve a shared resource for exclusive use. The first process that obtains ownership of the "mutex" also obtains ownership of the shared resource. Other processes must wait for for the first process to release it's ownership of the "mutex" before they may attempt to obtain it.

The most common locking primitive in the kernel is the spinlock. The spinlock is a very simple single-holder lock. If a process attempts to acquire a spinlock and it is unavailable, the process will keep trying (spinning) until it can acquire the lock. This simplicity creates a small and fast lock.

answered May 21 '12 at 7:18
user1011455

Spinlock is used if and only if you are pretty certain that your expected result will happen very shortly, before your thread's execution slice time expires.

Example: In device driver module, The driver writes "0" in hardware Register R0 and now it needs to wait for that R0 register to become 1. The H/W reads the R0 and does some work and writes "1" in R0. This is generally quick(in micro seconds). Now spinning is much better than going to sleep and interrupted by the H/W. Of course, while spinning, H/W failure condition needs to be taken care!

There is absolutely no reason for a user application to spin. It doesn't make sense. You are going to spin for some event to happen and that event needs to be completed by another user level application which is never guaranteed to happen within quick time frame. So, I will not spin at all in user mode. I better to sleep() or mutexlock() or semaphore lock() in user mode.

answered Jun 21 '13 at 5:59

CancerSoftware
152 4

From what is the difference between spin locks and semaphores? by Maciej Piechotka:

> Both manage a limited resource. I'll first describe difference between binary semaphore (mutex) and spin lock.
>
> Spin locks perform a busy wait - i.e. it keeps running loop:
>
> ```
> while (try_acquire_resource ());
>  ...
> release();
> ```
>
> It performs very lightweight locking/unlocking but if the locking thread will be preempted by other which will try to access the same resouce the second one will simply try to acquitre resource untill it run out of it CPU quanta.
> On the other hand mutex behave more like:
>
> ```
> if (!try_lock()) {
>     add_to_waiting_queue ();
>     wait();
> }
> ...
> process *p = get_next_process_from_waiting_queue ();
> p->wakeUp ();
> ```
>
> Hence if the thread will try to acquire blocked resource it will be suspended till it will be avaible for it. Locking/unlocking is much more heavy but the waiting is 'free' and 'fair'.
>
> Semaphore is a lock that is allowed to be used multiple (known from initialization) number of times - for example 3 threads are allowed to simultainusly hold the resource but no more. It is used for example in producer/consumer problem or in general in queues:
>
> ```
> P(resources_sem)
> resource = resources.pop()
> ...
> resources.push(resources)
> V(resources_sem)
> ```

Difference between semaphore, mutex & spinlock?

Locking in Linux

edited Dec 4 '13 at 13:59 answered Jun 24 '13 at 11:20


Gilles
41.5k 9 87 149


vinayak jadi
343 1 11

1 Seems to be a copy/paste of this ;-) : what is the difference between spin locks and semaphores? – Hibou57 Jul 30 '13 at 0:30