A vertical red bar on the left side of the slide contains various white and dark red icons representing technology and computing. These include a cloud with a keyhole, a database cylinder, a server rack, a person silhouette, a window with a cursor, and various arrows and symbols like 'X' and 'O'.

Efficient Userspace Optimistic Spinning Locks

Waiman Long

Principal Software Engineer, Red Hat

Linux Plumbers Conference, Sep 2019



Locking in Userspace

Most Linux applications use the synchronization primitives provided by the pthread library, such as:

- Mutex - `pthread_mutex_lock()`
- Read/write lock - `pthread_rwlock_wrlock()/pthread_rwlock_rdlock()`
- Condition variable - `pthread_cond_signal()/pthread_cond_wait()`
- Spin lock - `pthread_spin_lock()`

The top three are sleeping locks whereas the last one is a spinning lock as the name implies.

A number of large enterprise applications (e.g. Oracle, SAP HANA), however, implement their own locking code to eke out a bit more performance than can be provided by the pthread library.

Those locking libraries generally use the kernel `futex(2)` API to implement the sleeping locks. The system V semaphore API has also been used to support userspace locking in the past, but using `futex(2)` is generally more performant.

Userspace Spinning vs. Sleeping Locks

It is generally advised that spinning lock should only be used in userspace when the lock critical sections are short and the lock is not likely to be heavily contended.

Both spinning lock and sleeping lock have their own problem as shown below:

Spinning Lock:

- Lock cacheline contention
- Lock starvation (unfairness)
- A long wait can waste a lot of CPU cycles

Sleeping Lock:

- Waiter wakeup latency (in 10s to 100s of us)
- Lock starvation

The Linux kernel solves the spin lock cacheline contention and unfairness problems by putting the lock waiters in a queue spinning on their own cachelines. Similar type of queued spin lock may not be applicable to userspace* because of the lock owner and waiter preemption problems which aren't an issue in the kernel as preemption can be disabled.

Userspace Spinning vs. Sleeping Locks (Cont)

- The sleeping lock wakeup latency limits the maximum locking rate on any given lock which can be a problem in large multithreaded applications that can have many threads contending on the same lock.
- For applications that needs to scale up to large number of threads on systems with hundreds of CPUs. There is a dilemma in deciding if a spinning lock or a sleeping lock should be used for a certain type of data that need to be protected.
- In fact, some large enterprise applications have performance benchmark tests that can stress a spin lock with hundreds of threads. This can result in over 50% of CPU time spent in the spin lock code itself if the system isn't properly tuned.
- A typical userspace spin lock implement can be a regular TAS spinlock with exponential backoff* to reduce lock cacheline contention.
- It will be hard to implement a fair and efficient userspace spinning lock without assistance from the kernel.

Optimistic Spinning Futex

- There are two main types of futexes for userspace locking implementation – wait-wake (WW) futex and priority inversion (PI) futex.
- Most userspace locking libraries use the WW futexes for their locking implementation. PI futexes are only used in special cases where real time processes require a bound latency response from the system.
- Both types of futexes require lock waiters to sleep in the kernel until the lock holder which calls into the kernel to wake them up.
- Optimistic spinning (OS) futex, formerly throughput-optimized (TP) futex, is a new type of futex that is a hybrid spinning/sleeping lock. It uses a kernel mutex to form a waiting queue with the mutex owner being the head of the queue. The head waiter will spin on the futex as long as the lock owner is running. Otherwise, the head waiter will go to sleep.
- Both userspace mutex and rwlock can be supported by the OS futex, but not the condition variable.

Why Optimistic Spinning Futex

In term of performance, there are two main problems with the use of WW futexes for userspace locks.

- 1) Lock waiters are put to sleep until the lock holder calls into the kernel to wake up the waiters after releasing the lock. So there is the wakeup latency. In addition, the act of calling into the kernel to do the wakeup also delay the task from doing other useful work in the userspace.
- 2) With a heavily contended futex, the userspace locking performance will be constrained by hash bucket spinlock in the kernel. With a large number of contending threads, it is possible that more than 50% of the CPU cycles can be spent waiting for the spinlock*.

With an OS futex, the lock waiters in the kernel are not sleeping unless the lock holder has slept. Instead, they are spinning in the kernel for the lock to be released. Consequently, the lock holder doesn't need to go into the kernel to wake up the waiters after releasing the lock. This allows the tasks to continue doing their work without any delay and avoid the kernel hash bucket lock contention problem. Once the lock is free, the lock waiter can immediate grab the lock in the kernel or return to userspace to acquire it.

OS Futex Design Principles

The design goals of OS futexes is to maximize the locking throughput while still maintaining a certain amount of fairness and deterministic latency (no lock starvation).

To improve locking throughput:

- 1) Optimistic spinning – spin on lock owner while it is running on CPU. If the lock owner is sleeping, the head lock waiter will sleep too. In this case, the lock owner will need to go to the kernel to wake up the lock waiter at unlock time.
- 2) Lock stealing – it is known performance enhancement technique provided that safeguard is in place to prevent lock starvation. As long as the head waiter in the kernel is not sleeping and hence set the FUTEX_WAITERS bit, lockers from userspace is free to steal the lock.
- 3) Userspace locking – OS futexes have the option to use either userspace locking or kernel locking. WW futexes support only userspace locking. Userspace locking provides better performance (shorter lock hold time), but is also more prone to lock starvation.

To combat lock starvation, OS futexes also have a builtin lock hand-off mechanism to force lock handoff to the head waiter in the kernel after a certain time threshold. This is similar to what the kernel mutexes and rw semaphores are doing.

Using OS Futexes for Userspace Mutexes

OS futexes are similar to PI futexes in how they should be used.

A thread in userspace will need to atomically put its thread ID into the futex word to get an exclusive lock. If that fails, the thread can choose to issue the following `futex(2)` call to try to lock the futex.

```
futex(uaddr, FUTEX_LOCK, flags, timeout, NULL, 0);
```

The flags can be `FUTEX_OS_USLOCK` to make the lock waiter returns to userspace to retry locking, or it can be 0 to acquire the lock directly in the kernel before going back to userspace.

At unlock time, the task can simply atomically clear the futex word as long as the `FUTEX_WAITERS` bit isn't set. If it is set, it has to issue the following `futex(2)` to do the unlock.

```
futex(uaddr, FUTEX_UNLOCK, 0, NULL, NULL, 0);
```

It is relatively simple to use OS futex to implement userspace mutexes.

Using OS Futexes for Userspace Rwlocks

OS futexes support shared locking for implementing userspace rwlock. A special upper bit in the futex word (FUTEX_SHARED_FLAG) is used to indicate shared locking. The lower 24 bits of the futex word is then used for reader count.

In order to acquire a shared lock, the task has to atomically put (FUTEX_SHARED_FLAG + 1) to the futex word. If the FUTEX_SHARED_FLAG bit has already been set, the task just need to atomically increment the reader count. If the futex is exclusively owned, the following futex(2) call can be used to acquire a shared lock in the kernel.

```
futex(uaddr, FUTEX_LOCK_SHARED, flags, timeout, NULL, 0);
```

The supported flag is FUTEX_OS_PREFER_READER to make the kernel code prefer reader a bit more than writer.

At unlock time, a task can just atomically decrement the reader count. If the count reaches 0, it has to atomically clear the FUTEX_SHARED_FLAG as well.

It gets a bit more complicated if the FUTEX_WAITERS bit is set as a FUTEX_UNLOCK_SHARED futex(2) call need to be used and it has to be sure that it is the only read-owner that does the unlock.

Userspace Mutex Performance Data

A mutex locking microbenchmark run on a 2-socket 48-core 96-thread x86-64 system with a 5.3 based kernel. 96 mutex locking threads with short critical section were run for 10s using WW futex, OS futex and Glibc pthread_mutex.

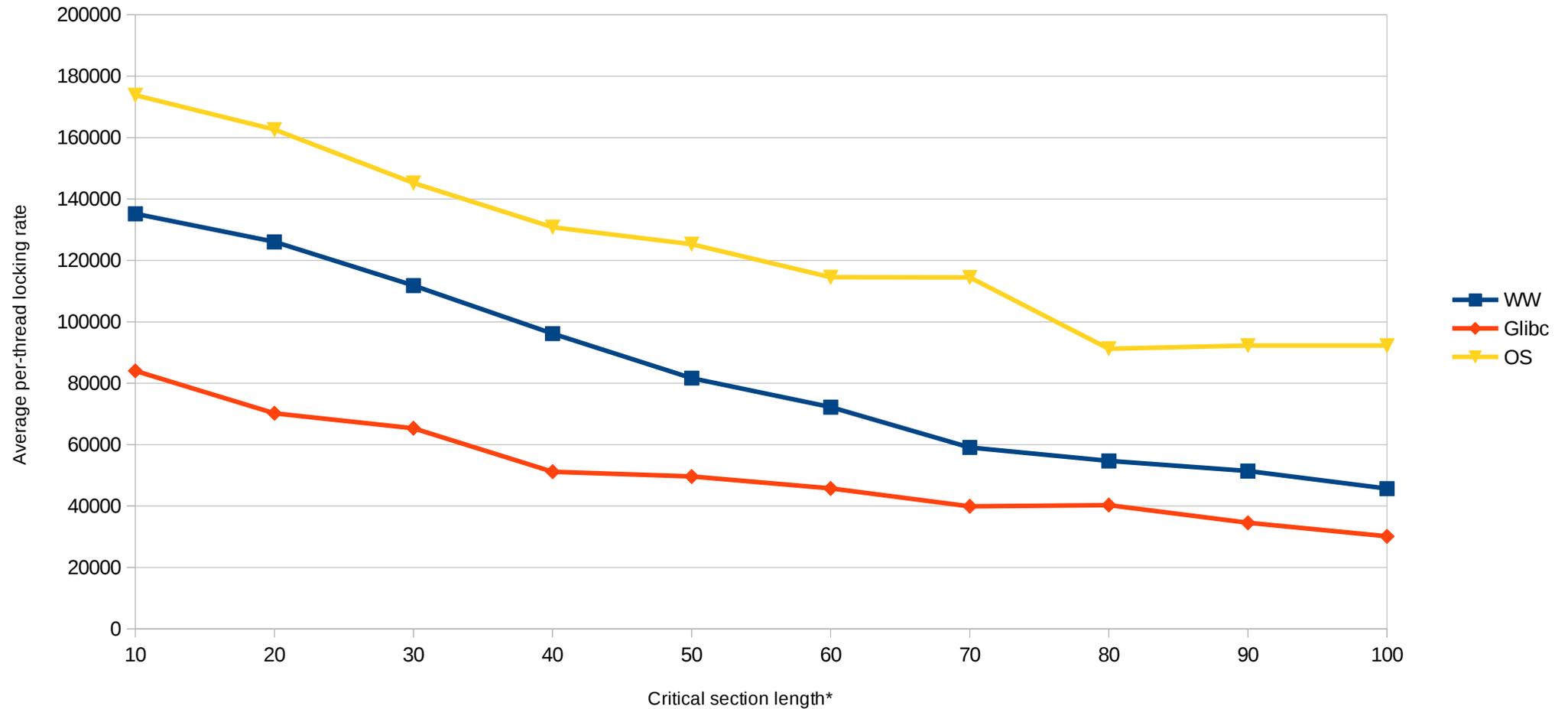
Futex Type	WW	OS	Glibc
Total locking ops	155,646,231	187,826,170	89,292,228
Per-thread average ops/s (std deviation)	162,114 (0.53%)	195,632 (1.83%)	93,005 (0.43%)
Futex lock syscalls	8,219,362	7,453,248	N/A
Futex unlock syscalls	13,792,295	19	N/A
# of kernel locks	N/A	276,442	N/A

The OS futex saw a 20.7% increase in locking rate when compared with the WW futex.

One characteristic of OS futex is the small number of futex unlock syscalls that need are issued.

Mutex Performance Chart

Mutex Locking Rates



* The number of pause instructions ran in the critical section.

Userspace Rwlock Performance Data 1

A rwlock microbenchmark run on a 2-socket 48-core 96-thread x86-64 system with a 5.3 based kernel. 96 rwlock locking threads doing equal number of read and write locks with short critical section were run for 10s.

Futex Type	WW	OS	Glibc
Total locking ops	41,849,400	130,361,414	45,124,138
Per-thread average ops/s (std deviation)	43,579 (0.17%)	135,779 (1.01%)	47,003 (0.60%)
Futex lock syscalls	8,557,148	6,065,240	N/A
Futex unlock syscalls	12,775,288	10	N/A
# of kernel locks	N/A	605,817	N/A

The OS futex saw a 212% increase in locking rate when compared with the WW futex.

Userspace Rwlock Performance Data 2

A rwlock microbenchmark run on a 2-socket 48-core 96-thread x86-64 system with a 5.3 based kernel. 48 writer threads and 48 reader threads with short critical section were run for 10s.

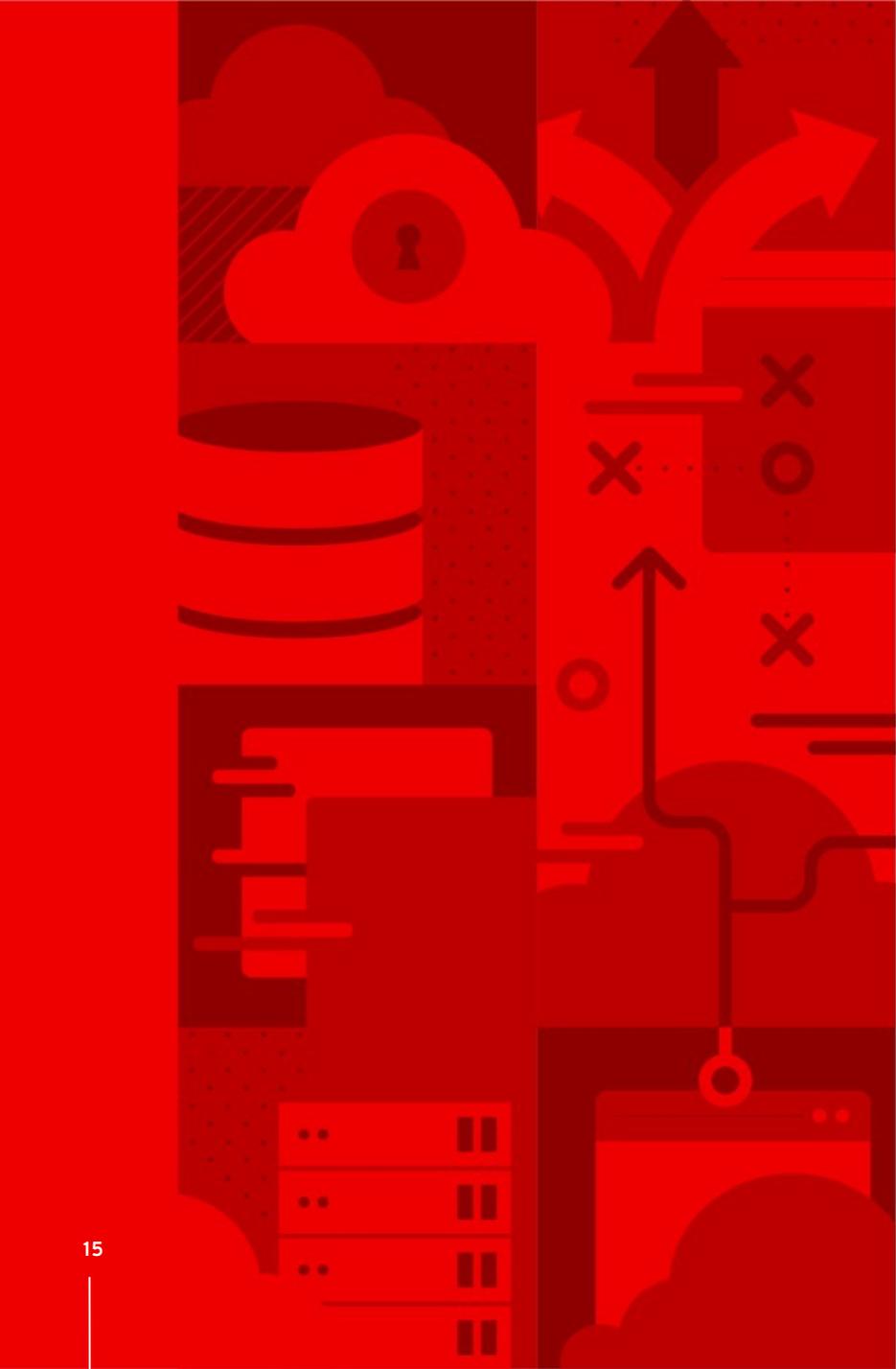
Futex Type (Prefer readers)	WW	OS	Glibc
Per-thread writer locking rate	0	39,076 (1.03%)	0
Per-thread reader locking rate	149,132 (0.42%)	138,714 (0.55%)	117,498 (0.57%)

Futex Type (Prefer writers)	WW	OS	Glibc
Per-thread writer locking rate	11,092 (0.23%)	44,916 (1.01%)	41,208 (0.64)
Per-thread reader locking rate	67,600 (0.51%)	141,420 (0.61%)	87 (5.09%)

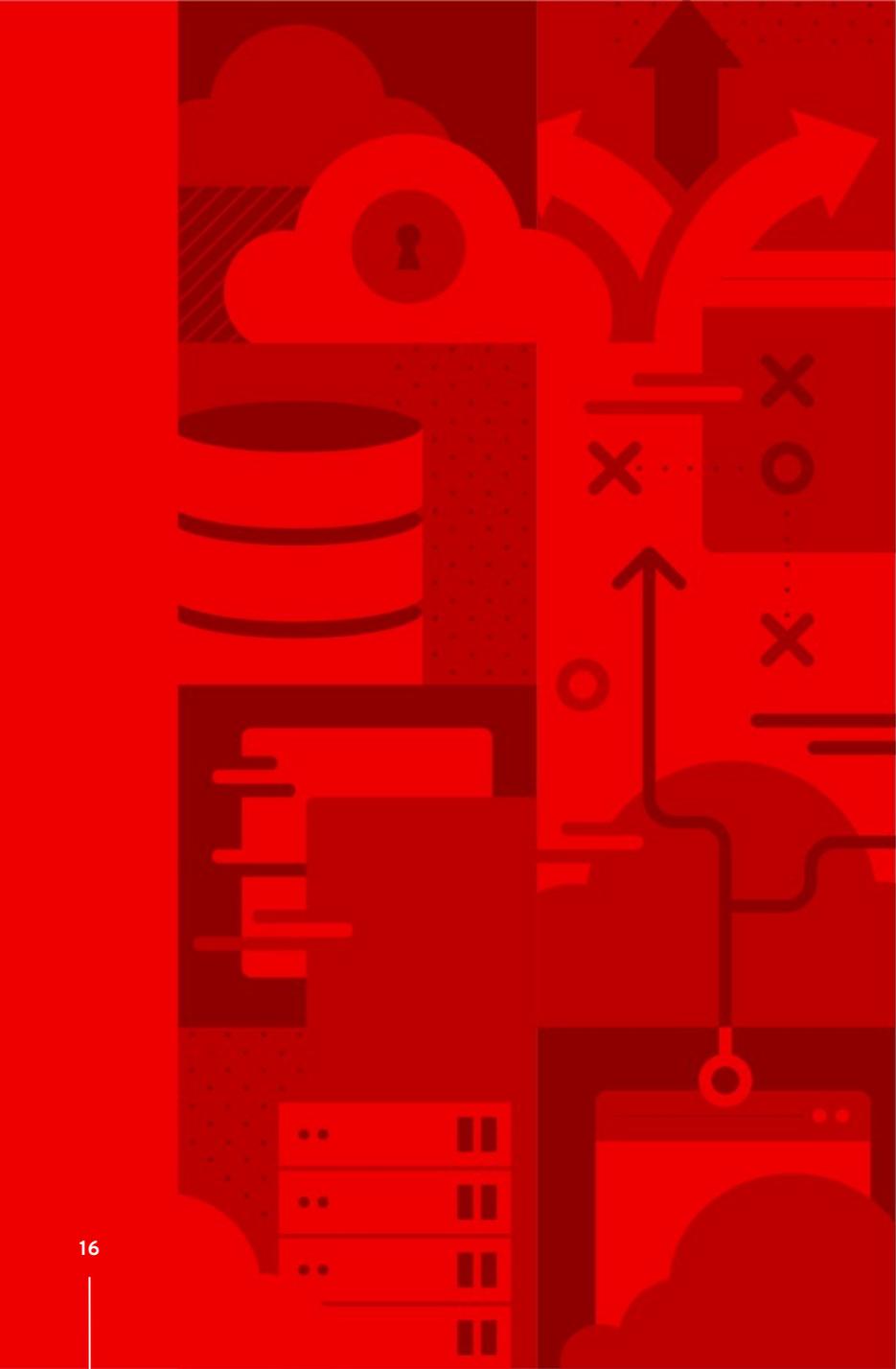
The OS futex lock handoff mechanism ensures that lock starvation will not happen.

Future Works Ahead

- Find a user for OS futexes, e.g. glibc or applications like PostgreSQL.
- Add a `mutex_lock()` variant that support timeout as `futex(2)` supports a timeout argument.
- Investigate various alternatives of doing more userspace optimistic spinning before going into the kernel and its impact on overall performance. For example, a bit in the futex word can be used to indicate if the lock owner is running. This bit can be set by the lock waiters spinning in the kernel or it can be set/cleared by the scheduler during context switch.



Q & A



Thank you

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

 [linkedin.com/company/red-hat](https://www.linkedin.com/company/red-hat)

 [facebook.com/redhatinc](https://www.facebook.com/redhatinc)

 [youtube.com/user/RedHatVideos](https://www.youtube.com/user/RedHatVideos)

 twitter.com/RedHat