Shared semaphore between user and kernel spaces

Asked 8 years, 9 months ago Modified 1 year, 5 months ago Viewed 9k times

23

10

Short version

Is it possible to share a semaphore (or any other synchronization lock) between user space and kernel space? Named POSIX semaphores have kernel persistence, that's why I was wondering if it is possible to also create, and/or access them from kernel context.

Searching the internet didn't help much due to the sea of information on normal usage of POSIX semaphores.

Long version

I am developing a unified interface to real-time systems in which I have some added book keeping to take care of, protected by a semaphore. These book keepings are done on resource allocation and deallocation, which is done in non-real-time context.

With RTAI, the thread waiting and posting a semaphore however needs to be in real-time context. This means that using RTAI's named semaphore means switching between real-time and non-real-time context on every wait/post in user space, and worse, creating a short real-time thread for every sem/wait in kernel space.

What I am looking for is a way to share a normal Linux or POSIX semaphore between kernel and user spaces so that I can safely wait/post it in non-real-time context.

Any information on this subject would be greatly appreciated. If this is not possible, do you have any other ideas how this task could be accomplished?[1]

[1] One way would be to add a system call, have the semaphore in kernel space, and have user space processes invoke that system call and the semaphore would be all managed in kernel space. I would be happier if I didn't have to patch the kernel just because of this though.

c linux-kernel semaphore

Share Improve this question Follow

edited Jan 22, 2018 at 14:35

Guy Avraham

3,096 ● 3 ● 35 ● 46

asked Jun 30, 2013 at 14:19

Shahbaz

44.5k ● 18 ● 111 ● 175

I'd imagine there'd be some sort of complications involving context switches on the user side...just a guess though. – Drew McGowen Jun 30, 2013 at 14:43

@DrewMcGowen, I'm guessing since the kernel is aware of the semaphore, there should be `do_sem_wait` kind of functions in the kernel (does the kernel still have the `do_X` for kernel side of `X`?) or something that would take care of those issues. From the user perspective, everything is a normal POSIX semaphore. In fact, I'm just trying to access that same POSIX semaphore from the kernel. – Shahbaz Jun 30, 2013 at 17:19

Where in the kernel? Do you expect to share a user space semaphore with the **scheduler**? Do you see a problem? You have to be more specific about *kernel space*. You can not use `up()` or `down()` in an interrupt handler yet alone some of the page fault and other hairy places that is kernel space. – artless noise Jul 5, 2013 at 1:50

@artlessnoise, just to be clear, I'm not interested in interrupt handlers. – Shahbaz Jul 5, 2013 at 8:20

I would think it is only kernel threads that make sense? A lot of the sub-systems are invoked through fault handlers as well. – artless noise Jul 5, 2013 at 12:14

8 Answers

Sorted by:
Reset to default

Trending (recent votes count more)

Help us improve our answers.
Are the answers below sorted in a way that puts the best answer at or near the top?

Take a short survey I'm not interested

15

+500

Well, you were in the right direction, but not quite -

Linux named POSIX semaphore are based on FUTex, which stands for Fast User-space Mutex. As the name implies, while their implementation is assisted by the kernel, a big chunk of it is done by user code. Sharing such a semaphore between kernel and user space would require re-implementing this infrastructure in the kernel. Possible, but certainly not easy.

SysV Semaphores on the other hand are implemented completely in kernel and are only accessible to user space via standard system calls (e.g. `sem_timedwait()` and friends).

This means that every SysV related operations (semaphore creation, taking or release) is actually implemented in the kernel and you can simply call the underlying kernel function from your code to take the same semaphore from the kernel is needed.

Thus, your user code will simply call `sem_timedwait()`. That's the easy part.

The kernel part is just a little bit more tricky: you have to find the code that implement `sem_timedwait()` and related calls in the kernel (they are are all in the file ipc/sem.c) and create a replica of each of the functions that does what the original function does without the calls to `copy_from_user(...)` and `copy_to_user(..)` and friends.

The reason for this is that those kernel function expect to be called from a system call with a pointer to a user buffer, while you want to call them with parameters in kernel buffers.

Take for example `sem_timedwait()` - the relevant kernel function is `sys_timedwait()` in ipc/sem.c (see here: http://lxr.free-electrons.com/source/ipc/sem.c#L1537). If you copy this function in your kernel code and just remove the parts that do `copy_from_user()` and `copy_to_user()` and simply use the passed pointers (since you'll call them from kernel space), you'll get kernel equivalent functions that can take SysV semaphore from kernel space, along side user space - ***so long as you call them from process context in the kernel*** (if you don't know what this last sentence mean, I highly recommend reading up on Linux Device Drivers, 3rd edition).

Best of luck.

Share Improve this answer Follow

edited Nov 19, 2020 at 9:24
Louis Go
1,926 ● 2 ● 13 ● 23

answered Jul 4, 2013 at 6:31

gby
14.5k ● 39 ● 57

this looks very promissing. By *user context in kernel*, do you mean code that runs as a result of system call (let's say for example in a `/sys` file handler or `ioctl`) or [any kernel code that is not in interrupt context](#)? In other words, if you have a normal kernel module, is the `__init` function in *user context in kernel*? How about code that is inside a kthread? – Shahbaz Jul 4, 2013 at 8:33

I searched the book for `user context` and didn't see much. I'd study it a bit further when I get more time. In the meantime, it would be nice if you cleared that up a bit. – Shahbaz Jul 4, 2013 at 8:38

@shahbaz I've meant any kernel code that is not in interrupt context. I should have probably used the term: "process context". Sorry for the confusion. – gby Jul 4, 2013 at 12:01

ok then that's great. I'll try it in the weekend. Hopefully the implementation of sysV semaphores are stable, so I wouldn't have to take care of kernel version, but that's something I'd have to check myself. – Shahbaz Jul 4, 2013 at 12:20

@Shahbaz sorry to disappoint you, but SysV IPC just changed in 3.10 ... Linux kernel **internals** should never considered stable. – gby Jul 4, 2013 at 12:25

6

One solution I can think of is to have a `/proc` (or `/sys` or whatever) file on a main kernel module where writing `0`/`1` to it (or read from/write to it) would cause it to issue an `up`/`down` on a `semaphore`. Exporting that semaphore allows other kernel modules to directly access it while user applications would go through the `/proc` file system.

I'd still wait to see if the original question has an answer.

Share Improve this answer Follow

answered Jul 3, 2013 at 8:59

Shahbaz
44.5k ● 18 ● 111 ● 175

Unfortunately, I don't think this will work since `procfs` and `sysfs` operations are non-atomic and leave the program susceptible to race conditions. – Vilhelm Gray Jul 3, 2013 at 18:39

@VilhelmGray, inside the read/write handlers of the virtual file, I would call `up` and `down`. Is there a problem if two processes simultaneously try to read from a `/proc` file by itself? (For example if the file simply doesn't do anything?) If not, then there shouldn't be any race conditions since the actual synchronization is done by the well-working `up`/`down` functions. – Shahbaz Jul 4, 2013 at 8:20

Sorry about that, I misunderstood your method: I thought you were reading a `0`/`1` and then calling the `up`/`down` in userspace. If you handling the semaphore completely from kernelspace - and just using the `/proc` file to request control - then you should have no issue. – Vilhelm Gray Jul 8, 2013 at 13:02

I went with sysfs in the end as I was more comfortable with it, but ioctl would also work. – Shahbaz Aug 1, 2013 at 15:00

FYI, last I tried (kernel 3.something), there was a gotcha here: [stackoverflow.com/q/28052643/912144](#) – Shahbaz Jan 22, 2018 at 14:39

1

Multiple solutions exist in **Linux/GLIBC but none permit to share explicitly a semaphore between user and kernel spaces**. The kernel provides solutions to suspend threads/processes and the most efficient is the futex. Here are some details about the state of the art of the current implementations to synchronize user space applications.

SYSV services

The Linux System V (SysV) semaphores are a legacy of the eponymous Unix OS. They are based on system calls to lock/unlock semaphores. The corresponding services are:

- semget() to get an identifier
- semop() to make operations on the semaphores (e.g. incrementation/decrementation)
- semctl() to make some control operations on the semaphores (e.g. destruction)

The GLIBC (e.g. **2.31 version**) does not provide any added value on top of those services. The library service directly calls the eponymous system call. For example, *semop()* (in *sysdeps/unix/sysv/linux/semtimedop.c*) directly invokes the corresponding system call:

```
int
__semtimedop (int semid, struct sembuf *sops, size_t nsops,
          const struct timespec *timeout)
{
  /* semtimedop wire-up syscall is not exported for 32-bit ABIs (they have
     semtimedop_time64 instead with uses a 64-bit time_t).  */
#if defined __ASSUME_DIRECT_SYSVIPC_SYSCALLS && defined __NR_semtimedop
  return INLINE_SYSCALL_CALL (semtimedop, semid, sops, nsops, timeout);
#else
  return INLINE_SYSCALL_CALL (ipc, IPCOP_semtimedop, semid,
                  SEMTIMEDOP_IPC_ARGS (nsops, sops, timeout));
#endif
}
weak_alias (__semtimedop, semtimedop)
```

Nowadays, SysV semaphores (as well as other SysV IPC like shared memory and message queues) are considered deprecated because as they need a system call for each operation, they slow down the calling processes with systematic context switches. New applications should use POSIX compliant services available through the GLIBC.

POSIX services

POSIX semaphores are based on Fast User Mutexes (FUTEX). The principle consists to **increment/decrement the semaphore counter in user space** with atomic operations as long as there is no contention. But when there is contention (multiple threads/processes want to "lock" the semaphore at the same time), a *futex()* system call is done to either wake up waiting threads/processes when the semaphore is "unlocked" or suspend threads/processes waiting for the semaphore to be released. From performance point of view, this makes a big difference compared to the above SysV services which systematically required a system call for any operation. The POSIX services are implemented in GLIBC for the user space part of the operations (atomic operations) with a switch into kernel space only when there is contention.

For example, in GLIBC **2.31**, the service to lock a semaphore is located in *nptl/sem_waitcommon.c*. It checks the value of the semaphore to decrement it with an atomic operation (in *__new_sem_wait_fast()*) and invokes the *futex()* system call (in *__new_sem_wait_slow()*) to suspend the calling thread **only if** the semaphore was equal to 0 before the attempt to decrement it.

```
static int
__new_sem_wait_fast (struct new_sem *sem, int definitive_result)
{
[...]
  uint64_t d = atomic_load_relaxed (&sem->data);
  do
```

```
    {
      if ((d & SEM_VALUE_MASK) == 0)
    break;
      if (atomic_compare_exchange_weak_acquire (&sem->data, &d, d - 1))
    return 0;
    }
  while (definitive_result);
  return -1;
[...]
}
[...]
static int
__attribute__ ((noinline))
__new_sem_wait_slow (struct new_sem *sem, clockid_t clockid,
             const struct timespec *abstime)
{
  int err = 0;

[...]
  uint64_t d = atomic_fetch_add_relaxed (&sem->data,
      (uint64_t) 1 << SEM_NWAITERS_SHIFT);

  pthread_cleanup_push (__sem_wait_cleanup, sem);

  /* Wait for a token to be available.  Retry until we can grab one.  */
  for (;;)
    {
      /* If there is no token available, sleep until there is.  */
      if ((d & SEM_VALUE_MASK) == 0)
    {
      err = do_futex_wait (sem, clockid, abstime);
[...]
```

The POSIX services based on the futex are for examples:

- sem_init() to create a semaphore
- sem_wait() to lock a semaphore
- sem_post() to unlock a semaphore
- sem_destroy() to destroy a semaphore

To manage mutex (i.e. binary semaphores), it is possible to use the pthread services. They are also based on the futex. For examples:

- pthread_mutex_init() to create/initialize a mutex
- pthread_mutex_lock/unlock() to lock/unlock a mutex
- pthread_mutex_destroy() to destroy a mutex

Share Improve this answer Follow

edited Nov 19, 2020 at 14:31

answered Nov 19, 2020 at 11:38

Rachid K.

3,109 ● 2 ● 7 ● 24

I read through your answer but did not find about "how to share" semaphoe between user and kernel space. Did I miss anything or you're trying to indicate SysV is deprecated since it requires system call? – Louis Go Nov 19, 2020 at 14:09

@LouisGo: You didn't miss anything. I wanted to point out the fact that multiple solutions already exist in Linux/GLIBC but none permit to share explicitly a semaphore between user and kernel. The kernel provides a solutions to suspend threads/processes and the most efficient is the futex. – Rachid K. Nov 19, 2020 at 14:22

Could you place the conclusion in the beginning of the answer? That would help for future user. Per your

descriptions, any technique uses futex would not allow communication between userspace and kernel. So whom try to find the solution would know that futex won't work in this use case. – Louis Go Nov 19, 2020 at 14:27

2

I would like to answer this differently: you don't want to do this. There are good reasons why there is no interface to do this kind of thing and there are good reasons why all other kernel subsystems are designed and implemented to never need a lock shared between user and kernel space. The complexity of lock ordering and implicit locking in unexpected places will quickly get out of hand if you start playing around with userland that can prevent the kernel from doing certain things.

Let me recall a very long debugging session I did around 15 years ago to at least shed some light what complex problems you can run into. I was involved in developing a file system where the large portion of the code was in userland. Something like FUSE.

The kernel would do a filesystem operation, package it into a message and send it to the userland daemon and wait for a reply. The userland daemon reads the message, does stuff and writes a reply to the kernel which wakes up and continues with the operation. Simple concept.

One thing you need to understand about filesystems is locking. When you're looking up a name of a file, for example "foo/bar", the kernel somehow gets the node for the directory "foo" then *locks it* and asks it if it has the file "bar". The filesystem code somehow finds "bar", *locks it* and then unlocks "foo". The locking protocol is quite straight forward (unless you're doing a rename), parent always gets locked before the child and the child is locked before the parent lock is released. The lookup message for the file is what would get sent to our userland daemon while the directory was still locked, when the daemon replied the kernel would proceed to *first lock* "*bar*" and then unlock "foo".

I don't even remember the symptoms we were debugging, but I remember the issue was not trivially reproducible, it required hours and hours of filesystem torture programs until it manifested itself. But after a few weeks we figured out what was going on. Let's say that the full path to our file was "/a/b/c/foo/bar". We're in the process of doing a lookup on "bar", which means that we're holding the lock on "foo". The daemon is a normal userland process so some operations it does can block and can be preempted too. It's actually talking over the network so it can block for a long time. While we're waiting for the userland daemon some other process want to look up "foo" for some reason. To do this, it has the node for "c", locked of course, and asks it to look up "foo". It manages to find it and attempts to lock it (it has to be locked before we can release the lock on "c") and waits for the lock on "foo" to be released. Another process comes in an wants to look up "c", it of course ends up waiting for that lock while holding the lock on "b". Another process waits for "b" and holds "a". Yet another process wants "a" and holds the lock on "/".

This is not a problem, not yet. This sometimes happens in normal filesystems too, locks can cascade all the way up to the root, you wait for a while for a slow disk, the disk responds, the congestions eases up and everyone gets their locks and everything keeps running fine. In our case though, the reason for holding the lock a long time was because the remote server for our distributed filesystem didn't respond. X seconds later the userland daemon times out and just before responding to the kernel that the lookup operation on "bar" has failed it logs a message to syslog with a timestamp. One of the things that the timestamp needs is the timezone information, so it needs to open "/etc/localtime", of course to do that, it needs to start looking up "/etc" and for that it needs to lock "/". "/" is already locked by someone else, so the userland daemon waits for that someone else to unlock "/" while that someone else waits through a chain of 5 processes and locks for the daemon to respond. The system ends up in a total deadlock.

Now, maybe your code will not have problems like this. You're talking about a real-time system so there might be a level of control you have that normal kernels don't. But I'm not sure if adding an unexpected layer of locking complexity would even let you keep real time properties of the system, or really make sure that nothing you do in userland will ever create a deadlock cascade. If you don't page, if you never touch any file descriptor, if you never do memory operations and a bunch of other things I can't really think of right now you could get away with a lock shared between userland and kernel, but it will be hard and you'll probably find unexpected problems.

Share Improve this answer Follow

answered Jul 10, 2013 at 8:01

Art

19.1k ● 1 ● 30 ● 54

Thanks for the answer, it was informative. I believe though, that this won't happen. The real-time part already uses RTAI locking mechanisms that can be shared between the two spaces just fine, I've used them extensively before and there's no problem there. The lock in this question is for the non-real-time, book-keeping part of the software, which is essentially always in the form: `lock-access shared memory-unlock`. – Shahbaz Jul 10, 2013 at 8:33

So only problem that can arise is if I call a function within the critical section that tries again to lock the mutex, which is a problem regardless of whether I share it between user and kernel spaces or if it's just one of them. Besides, the kernel space part is really no different from the user space part. No interrupt handling or anything a user-space process can't do. The only difference is that as far as RTAI is concerned, the same hard real-time application running in kernel space has smaller latency in real-time operations. – Shahbaz Jul 10, 2013 at 8:35

Well, I don't know about your specifics. You might get away with it. I just wanted to paint a picture why this normally is never done. Remember that the userland process while holding the lock can end up with memory management locks (faults), the kernel side can end up with other unexpected locks (interrupts). Userland can crash (exit involves lots of locks) and dump core (filesystem locks), signal handlers (more unexpected locking), etc. all while remembering to unlock the shared lock so that you don't block the kernel when userland exits unexpectedly. – Art Jul 10, 2013 at 9:09

I would redesign the communication protocol to use a circular lockless buffer. Or a normal pipe, or something like that. But that's just me being conservative. – Art Jul 10, 2013 at 9:09

I wouldn't mind going for a better design, but I don't see how you could manage a simple critical section access with circular buffers or pipes. The shared semaphore is really just a mutex, with the basic `lock-access-unlock` pattern. – Shahbaz Jul 10, 2013 at 9:54

3

I'm not really experienced on this by any means, but here's my take. If you look at glibc's implementation of sem_open, and sem_wait, it's really just creating a file in /dev/shm, mmap'ing a struct from it, and using atomic operations on it. If you want to access the named semaphore from user space, you will probably have to patch the tmpfs subsystem. However, I think this would be difficult, as it wouldn't be straightforward to determine if a file is meant to be a named semaphore.

An easier way would probably be to just reuse the kernel's semaphore implementation and have the kernel manage the semaphore for userspace processes. To do this, you would write a kernel module which you associate with a device file. Then define two ioctl's for the device file, one for wait, and one for post. Here is a good tutorial on writing kernel modules, including setting up a device file and adding I/O operations for it. http://www.freesoftwaremagazine.com/articles/drivers_linux. I don't know exactly how to implement an ioctl operation, but I think you can just assign a function to the ioctl member of the file_operations struct. Not sure what the function signature should be, but you could probably figure it out by digging around in the kernel source.

Share Improve this answer Follow

answered Jul 4, 2013 at 0:42

Zhehao Mao

1,791 ● 13 ● 12

Thanks for the pointer to the glibc implementations. Your suggestion is very similar to my own answer (except using `ioctl` instead of `sysfs`). I'd give it some thought. – Shahbaz Jul 4, 2013 at 8:24

2

As I'm sure you know, even the best working solution to this would likely be very ugly. If I were in your place, I would simply concede the battle and use rendezvous points to sync the processes

Share Improve this answer Follow

answered Jul 3, 2013 at 19:28

Mason Hemmel

185 ● 3

That's a poor reason to avoid implementing a proper solution. In fact, a working solution may not even be that difficult to implement. Just from the top of my head I can think of a possible elegant solution: perhaps using `mmap` to share the memory between user and driver, from which the same memory can be used on both sides to read and write the atomic values (i.e. interact with the semaphore). – Vilhelm Gray Jul 3, 2013 at 19:42

2

I have read your project's README and I have the following observations. Apologies in advance:

Firstly there already is a universal interface to real time systems. It is called POSIX; certainly VxWorks, Integrity and QNX are POSIX compliant and in my experience there are very few problems with portability if you develop within the POSIX API. Whether POSIX is sane or not is another matter, but it's the one we all use.

[The reason most RTOSes are POSIX compliant is because one of the big markets for them is defence equipment. And the US DoD won't let you use an OS for their non-IT equipment (eg Radars) unless it is POSIX compliant... This has pretty much made it commercially impossible to do an RTOS without giving it POSIX]

Secondly Linux itself can be made into a pretty good real time OS by applying the PREMPT_RT patch set. Of all the RTOSes out there this is probably the best one at the moment from the point of view of making efficient use of all these multi core CPUs. However it's not quite such a hard-realtime OS as the others, so its quid pro quo.

RTAI takes a different approach of in effect placing their own RTOS underneath Linux and making Linux nothing more than one task running in their OS. This approach is ok up to a point, but the big penalty of RTAI is that the real time bit is now (as far as I can tell) **not** POSIX compliant (though the API looks like they've just stuck rt_ on the front of some POSIX function names) and interaction with other things is now, as you're discovering, quite complicated.

PREEMPT_RT is a much more intrusive patch set than RTAI, but the payback is that everything else (like POSIX and valgrind) stays completely normal. Plus nice things like FTrace are available. Book keeping is then a case of merely using existing tools, not having to write new ones. Also it looks like PREEMPT_RT is gradually worming its way into the mainstream Linux kernel anyway. That would render other patch sets like RTAI pretty much pointless.

So Linux + PREEMPT_RT gives us realtime POSIX plus a bunch of tools, just like all the other RTOSes out there; commonality across the board. Which kinda sounds like the goal of your project.

I apologise for not helping with the with the "how" of your project, and it is highly ungentlemanly of me to query the "why?" of it too. But I feel it is important to know that there are established things out there that seem to heavily overlap with what you're trying to do. Unseating King POSIX is going to be difficult.

Share Improve this answer Follow

answered Jul 5, 2013 at 6:24

bazza

6,481 ● 13 ● 20

1 I'm aware of most of what you said, but thanks for the effort. Firstly, regarding PREEMPT_RT, as you said, "it's not quite real-time", which in real-time world translates to "not real-time". Maybe in general such a system would be fine, but without any guarantee, it can't be relied on for critical applications. – Shahbaz Jul 5, 2013 at 8:24

1 Regarding POSIX, I know that it's ubiquitous, but there are a few problems in my way. First and foremost, I need to, one way or another, fit RTAI there somewhere and they don't play that nicely together (RTAI being basically the only real real-time functionality for Linux). Second, POSIX is extremely irregular, there is a whole different way for making a shared semaphore and a shared readers-writer lock for example. A nice and clean API could be attractive (at least to myself). – Shahbaz Jul 5, 2013 at 8:29

1 Third, there are some functionality I'd rather have immediatly available. For example: `get_and_reserve_an_unused_name()`. Currently, most real-time people deal with very basic problems in practice. Their systems are highly static and small, they don't think of error recovery etc. POSIX may be fine for that. What I'm trying to acheive with URT is build a substructure that allows more dynamic and fault tolerant applications. Call it futuristic if you want. – Shahbaz Jul 5, 2013 at 8:31

@Shahbaz, sounds like you've done a good deal of thinking about it all! For what it's worth I'm using PREEMPT_RT at the moment, and it's pretty good. I've not used RTAI, though their approach gives them good control over scheduling so I'm not surprised if it is a more reliable and speedy scheduler. – bazza Jul 5, 2013 at 18:14

PREEMPT_RT is certainly pretty good, and I gather that it's not that far behind RTAI in terms of maximum latency, etc. Yes, POSIX *is* clunky, but it's what we've got. The risk of doing something new is that it becomes a support nightmare. For someone like me (an application developer), using something other than POSIX is a big risk. – bazza Jul 5, 2013 at 18:37

0

I was thinking about ways that kernel and user land share things directly i.e. without syscall/copyin-out cost. One thing I remembered was the RDMA model where the kernel writes/reads directly from user space, with synchronization of course. You may want to explore that model and see if it works for your purpose.

Share Improve this answer Follow

answered Jul 10, 2013 at 17:07

lsk

512 ● 4 ● 5

It might, but probably the more difficult part is how the user space application could lock the kernel space application. – Shahbaz Jul 11, 2013 at 8:43

IIRC, there was something about doorbells, registering callbacks and such for sync. You may have to dig into this for details. – lsk Jul 11, 2013 at 16:07

Interesting. I would definitely look into it. – Shahbaz Jul 11, 2013 at 16:31

Does RDMA stand for *Remote DMA*? Wikipedia says that's for direct memory access from memory of one computer to another. Is that what you're suggesting I look into? Or is it another RDMA? – Shahbaz Jul 12, 2013 at 16:21