Locking and Interprocess Communication

September 16, 2017

Group members:
Mallikarjuna(CS15B010),
Tejavardhan Reddy(CS15B014),
G. Pranav(CS15B015),
Giridhar S(CS15B016),
K. Mithilesh(CS15B020),
Lokesh K(CS15B049)
(CS13B032)

1 INTRODUCTION

The aim of the assignment is:

1. To understand the need of locking and an overview of machanisms.
2. To understand various locking mechanisms in use. .

2 Need of Locking

Locking arises due to the need of communicaion between the applications such as

when data generated by one process are transferred to another.
when data are shared.
when processes are forced to wait for each other.
when resource usage needs to be coordinated.

To prevent the CPUs from getting into each other's way, it is necessary to protect some kernel areas by means of locks; these ensure that access is restricted to one CPU at a time.

Assume there is counter being used to name folders by a process using two different interfaces.Lets assume it happens via the following steps

1. It reads the data from the interface.
2. It opens a file with the serial number count.
3. It increments the serial number by 1.
4. It writes the data to the file and closes it.

So here if the sheduler is activated just after first interface creates the folder and before incrementing the counter.This leads to a problem because second

interface also creates with same counter. Though it appears to happen rarely ,this one difficult time is enough to run into problems.

Though one may feel changing the code might help , but on closer observation it is always possible to create such a case.

Situations in which several processes interfere with each other when accessing resources are generally referred to as race conditions

Situations leading to race conditions are few and far between, thus begging the question as to whether it's worth making the — sometimes very considerable — effort to protect code against their occurrence.

3 Different Methods of Locking

3.1 Atomic Operations

In the view of the kernel, these are operations that are performed as if they consisted of a single assembly language instruction.In general, special lock instructions are used to prevent the other processors in the system from working until the current processor has completed the next action. And we want these lock instructions to not be disturbed hence we call them as Atomic Operations.

The Atomic variables must be accessed by their own functions and cannot be processed with normal operators such as ++.

3.2 SpinLocks

Spinlocks are used to protect short code sections that comprise just a few C statements and are therefore quickly executed and exited. Most kernel data structures have their own spinlock that must be acquired when critical elements in the structure are processed

Spinlocks are implemented by means of the spinlock_t data structure, which is manipulated using spin_lock and spin_unlock.spin_lock takes account of two situations:

1. If lock is not yet acquired from another place in the kernel, it is reserved for the current processor. Other processors may no longer enter the following area.

2. If lock is already acquired by another processor, spin_lock goes into an endless loop to repeatedly check whether lock has been released by spin_unlock. Once this is the case, lock is acquired, and the critical section of code is entered.

spin_lock is defined as an atomic operation to prevent race conditions arising when spinlocks are acquired.

3.3 Semahpores

```
struct semaphore {
    atomic_t count;
    int sleepers;
    wait_queue_head_t wait;
};
```

count specifies how many processes may be in the critical region protected by the semaphore at the same time. count == 1 is used in most cases (semaphores of this kind are also known as mutex semaphores because they are used to implement mutual exclusion).

sleepers specifies the number of processes waiting to be allowed to enter the critical region. Unlike spinlocks, waiting processes go to sleep and are not woken until the semaphore is free; this means that the relevant CPU can perform other tasks in the meantime.

wait is used to implement a queue to hold the task structures of all processes sleeping on the semaphore

```
down(&mutex);
     /* Critical section */
up(&mutex);
```

The count of the semaphore is decremented with down when the critical section is entered. When the counter has reached 0, no other process may enter the section.

3.4 Read Copy Update (RCU) mechanism

RCU performs very well in terms of performance impact, if at a slight cost in memory requirements, which is, however, mostly negligible.

The constraints that RCU places on potential users:

Accesses to the shared resource should be Read Only most of the time, and writes should be correspondingly rare. The kernel cannot go to sleep within a region protected by RCU.

The protected resource must be accessed via a pointer.

The mechanism keeps track of all users of the pointer to the shared data structure. When the structure is supposed to change, a copy (or a new instance that is filled in appropriately, this does not make any difference) is first created and the change is performed there. After all previous readers have finished their reading work on the old copy, the pointer can be replaced by a pointer to the new, modified copy. Notice that this allows read access to happen concurrently with write updates.

```
rcu_read_lock();
    p = rcu_dereference(ptr);
    if (p != NULL) {
        awesome_function(p);
    }
rcu_read_unlock();
```

If the object pointed at by ptr has to be modified, this must be done with

```
rcu_assign_pointer(ptr, new_ptr);
```

3.5 Memory and Optimization Barriers

Consider the technique of reordering the instructions to achieve a better performance. This optimization works perfectly fine, as long as the results are identical. However, it can be hard to decide for a compiler or processor if the result of a reordering will really match the intended purpose. This phenomenon is common when data is written to I/O registers. While locks are sufficient to ensure atomicity, they cannot always guarantee time ordering of code that is subjected to optimizations by compilers and processors.

To prevent these side-effects during the reordering, kernel provides the functions like $mb(), rmb(), wmb(), smb\_mb(), smp\_rmb(), smp\_wmb()$ that are availabe in the library $< system.h >$. rmb() is a read memory barrier. It guarantees that all read operations issued before the barrier are completed before any read operations after the barrier are carried out. wmb does the same thing, but this time for write accesses. mb() combines the effects of rmb() and wmb(). $smb\_mb(), smp\_rmb(), andsmp\_wmb()$ act as the hardware memory barriers as described above, but only when they are used on SMP systems. $barrier()$ ensures that the compiler does not process any read or write requests following the barrier before read or write requests issued before the barrier have been completed.

Note that the functions above tend to disable the optimizations and hence, the performance degrades.

3.6 Reader/Writer Locks

The mechanisms described above do not differentiate between situations in which data structures are simply read and those in which they are actively manipulated. Usually, any number of processes are granted concurrent read access to data structures, whereas write access must be restricted exclusively to a single task. To ensure this, the kernel provides additional semaphore and spinlock versions that are known accordingly as Reader/Writer semaphores and Reader/Writer spinlocks.

$read\_lock()$ and $read\_unlock()$ must be executed before and after a critical region to which a process requires read access. The kernel grants any number of read processes concurrent access to the critical region. $write\_lock$ and $write\_unlock$ are used for write access. The kernel ensures that only one writer (and no readers) is in the region.

3.7 The Big Kernel Lock

The Big Kernel Lock or BKL is used to lock the entire kernel to ensure that no processors run in parallel in kernel mode.The complete kernel is locked using $lock\_kernel$ and its unlocking counterpart is $unlock\_kernel$.

Although the BKLs are still present in the kernel, they can be replced with other locking mechanisms that are presented above to improve the performance and scalability.

3.8 Mutexes

Although semaphores can be used to implement the functionality of mutexes, the overhead imposed by the generality of semaphores is often not necessary. Because of this, the kernel contains a separate implementation of special-purpose mutexes that are not based on semaphores.
Mutexes are of two types: Classical Mutex and real-time mutex.

Classical Mutex:-

```
struct mutex {
atomic_t count;
spinlock_t wait_lock;
struct list_head wait_list;
};
```

Here, the count is 1 if the mutex is unlocked,0 if only a single process is using the mutex, negative if the mutex is locked and any processes are waiting on the mutex to be unlocked. This special casing helps to speed up the code because in the usual case, no one will be waiting on the mutex.

mutex_lock and *mutex_unlock* are used to lock and unlock a mutex, respectively. In addition, the kernel also provides the function *mutex_trylock* , which tries to obtain the mutex, but will return immediately if this fails because the mutex is already locked. Finally, *mutex_trylock* can be used to check if a given mutex is locked or not.

Real-Time Mutex:-

In contrast to the regular mutexes, Real-time mutexes implement priority inheritance, which, in turn, allows for solving the effects of priority inversion. They need to be explicitly enabled at compile time by selecting the configuration option $config\_rt\_mutex$.

```
struct rt_mutex {
spinlock_t wait_lock;
struct plist_head wait_list;
struct task_struct *owner;
};
```

When a low-priority process acquires the mutex and a higher priority process is waiting for the mutex, priority of the lower-priority process is temporarily boosted up to the priority level of the higher prority process to prevent the intervension of a process with intermediate priority. The change in contrast to regular mutexes is that the tasks on the waiter lists are sorted by priority. Whenever the waiter list changes, the kernel can consequently adjust the priority of the owner up or down.

3.9 Approximate Per-CPU Counters

Counters can become a bottleneck if a system is equipped with a large number of CPUs: Only one CPU at a time may modify the value; all other CPUs need to wait for the operation to finish until they can access the counter again. For some counters, it is not necessary to know the exact value at all times. Hence, we can introduce per-CPU counters to store an approximation of the value as it serves quite as well as the proper value would do.

If a processor wants to modify the value of the counter, it does not perform this modification by directly changing the counter value. Instead, the desired modification is stored in the CPU-specific entry of the array associated with the counter. If the changes in one of the CPU-specific array elements sum up to a value above or below a threshold that is considered to be large, the proper counter value is changed.

```
struct percpu_counter {
spinlock_t lock;
long count;
long *counters;
};
```

count is the proper value of the counter, and lock is a spinlock to protect the counter when the exact value is required. The CPU-specific array buffering counter manipulations is given by counters .

3.10 Lock Contention and Fine-Grained Locking

Locking needs to fulfill two purposes that are often hard to achieve simultaneously:
1.Code must be protected against concurrent access, which would lead to failures.
2.The impact on performance must be as little as possible.
Having both things at the same time is especially complicated when data are heavily used by the kernel.

Consider a really important data structure that is accessed very often. If the whole data structure is protected by only a single lock, then chances are high that the lock is acquired by some other part of the system when one part want to get hold of it. Lock contention is said to be high in this case, and the lock becomes a hotspot of the kernel. To remedy this situation, it is customary to identify independent parts of a data structure and use multiple locks to protect the elements. This solution is known as fine-grained locking. While the approach is beneficial for scalability on really big machines, it raises two other problems:

1. Taking many locks increases the overhead of an operation, especially on smaller SMP machines.

2. When a data structure is protected by multiple locks, then cases naturally arise when a process needs to access a lock that is held by other process. This, again may lead to deadlocks.

Achieving fine-grained locking for good scalability while making sure to avoid deadlocks is therefore currently among the kernel's foremost challenges.

4 Locking Issues

4.1 Contention:

Contention is said to occur when several threads/processes require frequent access to the same lock. When this is the case, some threads/processes have to wait until a lock (or a whole set of locks, depending on the process' requirement) is released. This creates a "single lane bridge" situation. Problems due to contention are not noticeable when traffic is low (i.e. non-concurrent or low-concurrency situations). However, as traffic (i.e. concurrency) increases, a bottleneck is created. This creates severe efficiency issues.

Overall, Lock Contention problems have a relatively low impact. They manifest themselves by impacting and limiting scalability. As concurrency increases, system throughput does not increase and may even degrade

4.2 Long Term Blocking

Long Term Blocking is similar to Lock Contention in that it involves an object or lock that is frequently accessed by a large number of database sessions. Where it differs is that in this case, one session does not release the lock immediately. Instead, the lock is held for a long period of time and while that lock is held, all dependent sessions will be blocked.

If one of the threads holding a lock faces some issue that takes time to fix; as the case is when it dies, stalls, blocks, page faults or enters an infinite loop, then other threads waiting for the lock may be forced to wait for a really long time, or forever.

Long Term Blocking tends to be a much bigger problem than Lock Contention. It can bring an entire area of functionality or even a whole system to a stand still. The locks involved in these scenarios may not be "hot" enough to lead to Lock Contention problems under normal circumstances. As such, these problems may be intermittent and very dependent on certain coincidental activity. These are the most likely to lead to "disasters" in production due to the combination of devastating impact and difficulty to reproduce.

The consequences of long term blocking problems may be abandonment. However, these problems can also often lead to further problems as frustrated users re-submit their requests. This can compound and exacerbate the problem by leading to a larger queue and consuming additional resource. In this way, the impact can expand to consume an entire system.

4.3 Overhead:

Requesting locks, waiting for locks, and releasing locks add processing overhead.

A program that supports multiprocessing always does the same lock and unlock processing, even though it is running in a uniprocessor or is the only user in a multiprocessor system of the locks in question.

When one thread requests a lock held by another thread, the requesting thread may spin for a while or be put to sleep and, if possible, another thread dispatched. This consumes processor time.

The existence of widely used locks places an upper bound on the throughput of the system. For example, if a given program spends 20 percent of its execution

time holding a mutual-exclusion lock, at most five instances of that program can run simultaneously, regardless of the number of processors in the system. In fact, even five instances would probably never be so nicely synchronized as to avoid waiting for one another.

Thus, the use of locks adds overhead for each access to a resource, even when the chances for collision are very rare.

4.4 Debugging:

Bugs associated with locks are time dependent and can be very subtle and extremely hard to replicate, such as deadlocks.

4.5 Composability:

One of lock-based programming's biggest problems is that "locks don't compose": it is hard to combine small, correct lock-based modules into equally correct larger programs without modifying the modules or at least knowing about their internals.

Consider the following Program:

```
class Account:
    member Integer balance
    member Lock mutex
    method deposit(Integer n)
           mutex.lock()
           balance := balance + n
           mutex.unlock()
    method withdraw(Integer n)
           deposit(-n)
```

Consider the following function.

```
function transfer(Account from, Account to, integer amount )
    from.withdraw(amount)
    to.deposit(amount)
```

In a concurrent program, this algorithm is incorrect because when one thread is halfway through transfer, another might observe a state where amount has been withdrawn from the first account, but not yet deposited into the other account: money has gone missing from the system. This problem can only be fixed completely by taking locks on both account prior to changing any of the two accounts, but then the locks have to be taken according to some arbitrary, global ordering to prevent deadlock:

```
function transfer( Account from, Account to , integer amount )
    if (from < to)    // arbitrary ordering on the locks
        from.lock()
        to.lock()
    else
        to.lock()
        from.lock()
```

```
from.withdraw(amount)
to.deposit(amount)
from.unlock()
to.unlock()
```

This solution gets more complicated when more locks are involved, and the transfer function needs to know about all of the locks, so they cannot be hidden.

4.6 Convoying:

A lock convoy occurs when multiple threads of equal priority contend repeatedly for the same lock.[1] Unlike deadlock and livelock situations, the threads in a lock convoy do progress; however, each time a thread attempts to acquire the lock and fails, it relinquishes the remainder of its scheduling quantum and forces a context switch. The overhead of repeated context switches and underutilization of scheduling quanta degrade overall performance.

Lock convoys often occur when concurrency control primitives such as critical sections serialize access to a commonly used resource, such as a memory heap or a thread pool. They can sometimes be addressed by using non-locking alternatives such as lock-free algorithms or by altering the relative priorities of the contending threads.

4.7 Deadlock :

A deadlock occurs when a process or thread enters waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process then the system is in deadlock situation.

4.7.1 Self Deadlock

If a process or thread which has accuired a specific lock and before releasing the accuired lock if it tries to accuire the smae lock again then it will lead to deadlock .
i.e a prcoess has a spin lock s and before leaving the spin lock it tries to accuire the same spinlock s again then the process will be spinning for lock s but it will be forver spinning as s can't released by the same programme since its spinning over a lock.
spinlocks, rwlocks and mutexes are not recursive in Linux.
In slightly diffrent context suppose if region shared by a softirq and user context. If we use a spin_lock() call to protect it, it is possible that the user context will be interrupted by the softirq while it holds the lock, and the softirq will then spin forever trying to get the same lock.

4.7.2 ABBA deadlock or deadly embrace

It involve two or more locks. If a process acuires a lock and then is waiting for another lock which is acquired by some another process and that process is waiting for the lock that is acuired by previous process then its a deadlock situtation as both process are waiting for each other to complete their execution

but none of them is able to do so.
i.e Say there're two threads thread A and thread B executing simultaneously and two locks X1 and X2 (protecting some critical region). A acquires X1 and B acquires X2 and then A tries to acquire X2 and B tries to acquire X1.

```
thread_A()
{
    . . .
    spinlock_lock(&X1)
    critical_code()
    spinlock_lock(&X2) //deadlock if thread_B has already acquired X2
    . . .
}

thread_B()
{
    . . .
    spinlock_lock(&X2)
     critical_code()
    spinlock_lock(&X1) //deadlock if thread_A has already acquired X1
    . . .
}
```

suppose A is running in CPU1 and B is running in CPU2 then

CPU1	CPU2
grab lock X1 – ok	grab lock X2 – ok
grab lock X2 – spin	grab lock X1 – spin

so the two CPUs will spin forever, waiting for the other to give up their lock.

4.8 Priority inverion :

Generally thread which has higher priority have to be executed first and is scheduled ahead by the scheduler.and if their is some lower priority thread running scheduler preempt and run the higher priotiy task.
But in case of locking the situation may arise where a low priority running task has accuired a lock s and a higher priority task came to be scheduled also requires the same lock s. so before scheduling higher priority taks scheduler has to wait till the lock is released by the lower priority task as too avoid deadlock. But it arises a new issue of priority inversion as lower priorty task has been executed before higher priority task.

4.9 Instability :

The optimal balance between lock overhead and lock contention can be unique to the problem domain (application) and sensitive to design, implementation, and even low-level system architectural changes. These balances may change over the life cycle of an application and may entail tremendous changes to update or re-balance.

4.10 Racing timers issue :

Timers can produce their own special problems with races. Consider a collection of objects (list, hash, etc) where each object has a timer which is due to destroy it.
If we want to destroy the entire collection (say on module removal), we might do the following:

```
spin_lock_bh(&list_lock);

while (list) {
        struct foo *next = list->next;
        del_timer(&list->timer);
        kfree(list);
        list = next;
}

spin_unlock_bh(&list_lock);
```

Sooner or later, this will crash on SMP, because a timer can have just gone off before the spin_lock_bh(), and it will only get the lock after we spin_unlock_bh(), and then try to free the element (which has already been freed).