

Take me for a spin



Implementing Locks in C

Charlotta Spik, Hannes Rabo, Fredrik Carlsson
2018-11-27
KTH Kista, Stockholm



Total Store Order

- Write operations are ordered
- Read operations can move around
 - A read can be done before a write is completed
- So even if a write is performed before a read, the read can be done first
 - We will then read an old value



Peterson's Algorithm

- Has an array of “flags” with size 2, one flag for each process
- Has a shared global variable “turn” to represent whose turn it is
- When a process wants to execute its critical section, it sets its flag to true and turn as the index of the other process
 - The flag indicates that the process wants to enter the critical section, but setting turn to the other process' index will allow the other process to run first
- While the the other process' flag is true and it is the other process' turn, spin
- After a process is done in the critical section, it sets its flag to false
- The other process can now enter the critical section



Peterson's Algorithm - Problem!

- The algorithm is dependent on that the process first sets its own value, then reads the value of the other process
- Because of total store order, this can be violated!
- Before the write, flag = 1, is complete, the other process can read the value as 0 and assume the other thread is not interested in the critical section
- Both threads will then enter the critical section and overwrite each other's value
- We solve this with atomic operations



The solution: test-and-set

- Atomic operation
- Returns the old value while simultaneously setting the new value
- If no one holds the lock:
 - Simultaneously set value to 1 while returning 0 => success
 - The value is now 1, which means someone has the lock
 - Set the lock to 0 again when done in critical section
- If another thread has the lock:
 - Simultaneously set value to 1 while returning 1 => failure
 - 1 means that another thread has the lock
 - try until test-and-set returns 0



Compare and Swap

- Similar to “test-and-set”
- Check if the lock is 0, and if so, sets it to 1 and acquire the lock
- If not 0, spin until it is

`__sync_val_compare_and_swap`



Spinning Wastes Time

- The thread spends all its time slot just waiting for a value to change
- Another solution is to make the thread yield while waiting, and wake up the thread when the value has changed
- For this we use `futex()`
 - This is a system call that provides a method for waiting until a certain condition becomes true
 - For example when a lock has been released
 - `FUTEX_WAIT` to put thread to sleep while waiting
 - `FUTEX_WAKE` to wake up a thread when the condition has changed



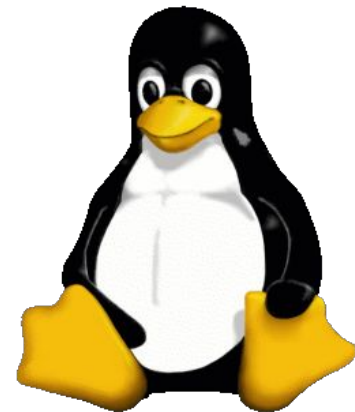
Spinning or Sleeping?

- Sometimes, doing a context switch between threads actually takes longer than waiting for the value to change
 - Maybe the lock will soon be released
- This will lead to an unnecessary switch to a new thread that does not get to execute for long before switching back
- In this case, spinlocks are a better choice



UNIX - SYS_futex

- Futex - Fast Userspace Mutex Lock
- The way mutexes are handled natively in UNIX
- Not the way to do it yourself - Use the POSIX standard (pthread_mutex)





POSIX Threading - pthreads

- Managing pthreads
 - pthread_create
 - pthread_join

```
pthread_t myThread;  
pthread_create(&myThread, NULL, myFunction, "Simple argument");  
pthread_join(myThread, NULL);
```

- Threads only takes one input argument
 - Can pass several values to a thread by using a **struct**

```
typedef struct args {int myInt; char myChar;} args;  
args myArgument;  
myArgument . myInt = 42;  
myArgument . myChar = 'F';
```

```
pthread_create(&myThread, NULL, myFunction, &myArgument);
```



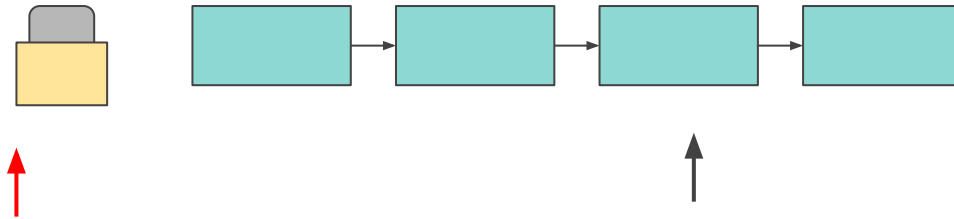
POSIX Threading - pthread_mutex

- Managing mutex locks
 - pthread_mutex_lock
 - pthread_mutex_unlock
 - PTHREAD_MUTEX_INITIALIZER

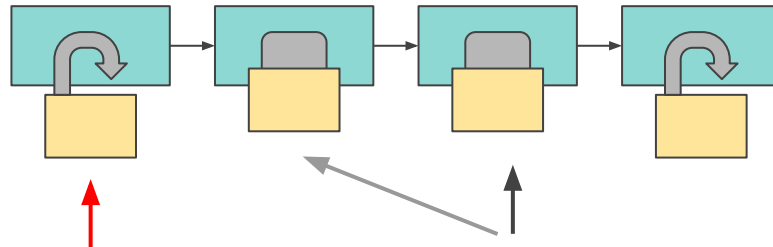


Locking you Lists

One “master lock”



One lock in each node - **Hand-over-Hand-locking** locking





Nifty Swifty Keywords

Compiling and including PThread

- gcc my_program.c -o my_program -lpthread

Waits until specified thread terminates

- pthread_join

Creates a thread for a specified function

- pthread_create

Disable compiler optimizations of a variable

- Volatile

Parse string to int

- Atoi



2017-06-07

You and your colleague Alan Peterson have realized that you need to implement a lock to protect shared data structures. Alan proposes that each thread should have a tag that signals that the thread wants to enter the critical section. If a thread sets its own tag and then checks that no other thread has set its tag the problem is solved. If another thread also has the tag set the tag is reset and tries again sometime later. There is a risk for starvation but this is problem that is acceptable.

What will you tell your colleague?



2017-06-07

You and your colleague Alan Peterson have realized that you need to implement a lock to protect shared data structures. Alan proposes that each thread should have a tag that signals that the thread wants to enter the critical section. If a thread sets its own tag and then checks that no other thread has set its tag the problem is solved. If another thread also has the tag set the tag is reset and tries again sometime later. There is a risk for starvation but this is a problem that is acceptable.

What will you tell your colleague?

Answer:

Alan, wtf? You know that on a modern processor we can't guarantee sequential consistency, but only guarantee to have **Total Store Order**. Hence there's a chance that we might end up with several threads inside the critical section.



2017-06-07

Assume we have a **single linked sorted list** that we wish to update by adding or removing elements. We want several threads to be able to perform operations in parallel and therefore implement a list where **each cell is protected by a lock**.

The operations require that we hold two locks when we remove an item but this is easily implemented. We can choose to use `pthread_mutex_lock()` to implement the lock operations but we could also use simple spin-locks. **What is the advantage and disadvantage of using spin-locks and how are the conditions changed if we have more threads and less cores?**



2017-06-07

Assume we have a **single linked sorted list** that we wish to update by adding or removing elements. We want several threads to be able to perform operations in parallel and therefore implement a list where **each cell is protected by a lock**.

The operations require that we hold two locks when we remove an item but this is easily implemented. We can choose to use `pthread_mutex_lock()` to implement the lock operations but we could also use simple spin-locks. **What is the advantage and disadvantage of using spin-locks and how are the conditions changed if we have more threads and less cores?**

Answer:

The advantage with spin-locks would be that the locks are held for a short time and a taken lock is probably released within a few clock cycles. If we spin we can **avoid suspending** a process.

The down side is of course **if the process that holds the lock is suspended**. If we increase the number of threads or have fewer cores the risk for this increase which would then be an argument against using spin-locks