#uni/semester3/Betriebssysteme/chapter28/chapter-notes

➡ Fundamental problem in concurrent programming, is that we want to execute a series of instructions atomically, but due to interrupts we couldn't. Putting locks around critical section, executes it as it were a single atomic instruction.

1. Locks: The Basic Idea
   ➡ To use a lock we add some code around the critical section: balance = balance + 1

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

   ➡ mutex is a **lock variable**. It holds the state of the lock.
   ➡ It is either **available** (or unlocked or free) and thus nu thread holds the lock. Or it is **acquired** (or locked or held), and thus only one thread holds the lock and is presumably in critical section.
   ➡ We also can store in the lock variable, which thread holds the lock or a queue for ordering lock acquisition.
   ➡ On lock() is checked if any other thread holds the lock and if not the thread will acquire the lock. This thread is called **owner** of the lock. If then another thread calls lock() it will be stuck in lock() and thus not enter the critical section. Once the owner calls unlock() the state is changed. To free if there is no other thread waiting for that lock. If there is, then one of those threads will notice or be told, and thus will acquire the lock.
   ➡ Threads are entities created by programmers but scheduled by the OS. Locks yield some of that control back to the programmer, ensuring that at a given time, no more than one thread can be ever active within a critical section. Locks help make the chaos of OS scheduling into a more controlled activity.

2. Pthread Locks
   ➡ POSIX uses the name **mutex** for a lock. It means **mutual exclusion** between threads, so if a thread is in a critical section it excludes others from entering. This does the same as above in POSIX:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper; exits on failure
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

➡ POSIX passes a variable to lock and unlock, so we can use different locks to protect different variables. This can increase concurrency. Using one big lock that is used for any critical section is a **coarse-grained** approach. Using many different ones to protect different data structures is a **fine-grained** approach and it allows more threads to be in locked code at once.

3. Building A Lock

   ➡ To build a lock we will need hardware and OS support. Next we will study how to use different hardware primates that have been added to the instruction sets of various computer architectures, in order to build a mutual exclusive primitive like a lock. As well how the OS gets involved to complete the picture.

4. Evaluating Locks

   ➡ First criteria of a lock is does it provide **mutual exclusion**? Does it prevent threads from entering a critical section?

   ➡ Does each thread contending for the lock get a fair shot at getting it once it is free? (**fairness**)

   ➡ How is the **performance**? What is the overhead when a thread grabs and releases the lock? What happens when multiple threads want to get the lock? What happens when there are multiple CPUs and threads on each CPU wanting to grab the lock?

5. Controlling Interrupts

   ➡ One early solution what to disable interrupts for critical sections:

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

➡ It works on a single processor system. By turning off interrupts (special hardware instruction) we make sure that the code inside critical section won't be interrupted. When we are finished we again use a hardware instruction to re-enable interrupts.

➡ It is simply and it works.

➡ Negatives are many:

➡ Security concerns, because it allows any calling thread to perform a privileged operation and it requires to trust that this facility won't be abused. A greedy program can use lock() and monopolise the processor. Malicious or buggy program could go in an endless loop, never unlocking (only solution is reboot).

➡ This doesn't work on multiple CPUs because mutuale threads are running on different CPUs, and disabling interrupts doesn't prevent them from going into the critical sections.

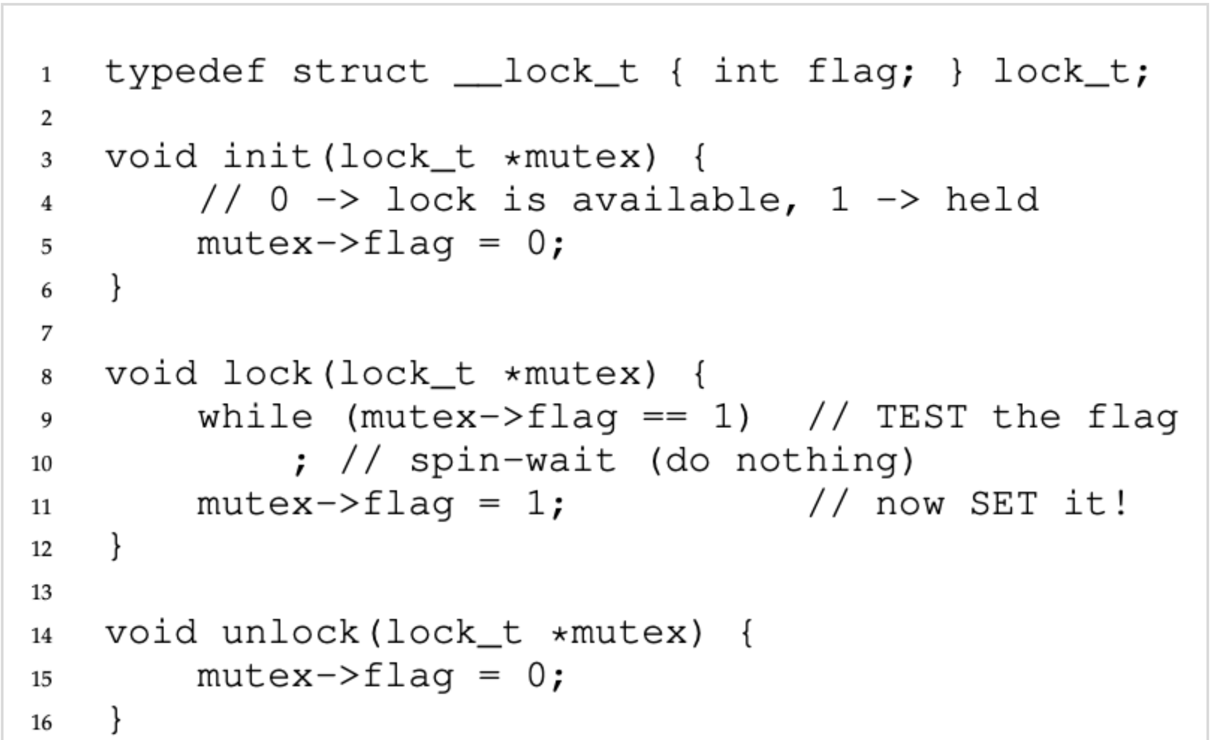
➡ Turning off interrupts for extended periods of time can lead to interrupts becoming lost, which lead to serious system problems. Imagine if OS missed that a disk device has finished a read request. How will the OS know to wake the process waiting for the read?

➡ Approach is also inefficient. Code that masks or unmasks interrupts tends to be executed slowly by modern CPUs.

➡ For this reasons turning off interrupts is used in some limited contexts as a mutual exclusion primitive. Namely by the OS in order to guarantee atomicity when accessing its own data structures or avoid bad interrupt handling situations.

6. A Failed Attempt: Just Using Loads/Stores

➡ Next example is building a lock using a single flag variable.



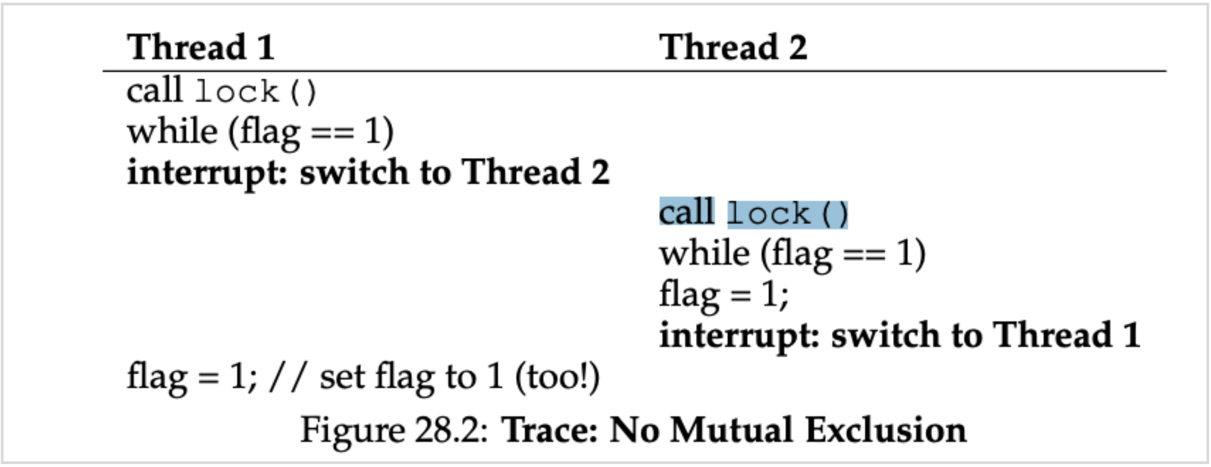
```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Figure 28.1: First Attempt: A Simple Flag

➡ Idea is, that when a thread enters the lock it it sets a flag to 1. And on unlock it sets it to 0. If on lock, the flag is already 1, then it waits until it is 0.

➡ The code has to problems, one of correctness and one of performance.

➡ The correctness problem is showed in the following image, with a flag=0



Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Figure 28.2: **Trace: No Mutual Exclusion**

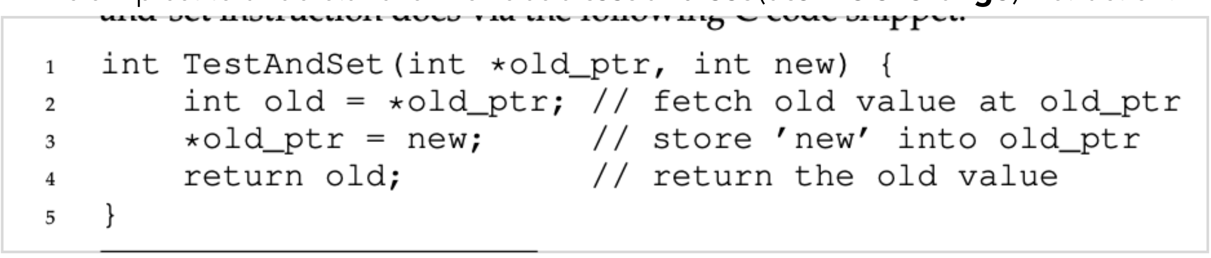
➡ The problem is that both threads enter the critical section aka set the flag to 1. This happens because in thread 1, the interrupts comes right before setting the flag to 1.

➡ Endlessly checking the flag is known as **spin-waiting**. This is bad for performance and is a big waste especially on a uniprocessor.

7. Building Working Spin Locks with Test-And-Set

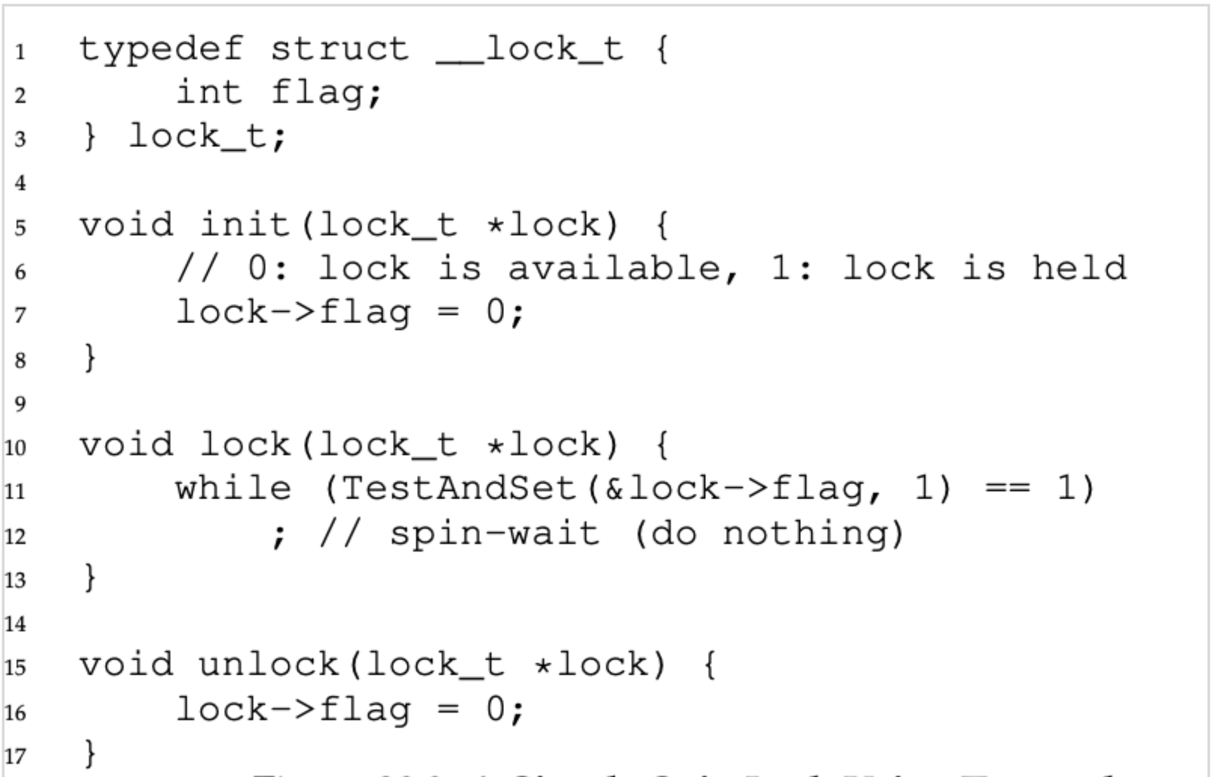
➡ We need hardware support for locking. Today all systems provide this type of support, even for single CPU systems.

➡ The simplest to understand is knows as a **test-and-set** (**atomic-exchange**) instruction:



```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

➡ This instruction returns the old value pointed by old_ptr, and simultaneously updated said value to new. The key is that this sequence is performed **atomically**! This slightly more powerful instruction is enough to build a simple **spin lock** as following:



```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.3: A Simple Spin Lock Using Test-and-set

➡ When a thread calls the lock method, it will test the value of flag and atomically set the value to 1. On unlock() its sets the flag back to 0.

➡ By making both the test (of the old lock value) and set (of the new lock value) a single atomic operation, we ensure that only one thread acquires the lock.

➡ This type of lock referred as a **spin lock**. It is the simplest type of lock tu build and simply spins using CPU cycles until the lock becomes available. To work correctly on a single processor it need a **preemptive scheduler** (one that will interrupt a thread via a timer, in order to run a different thread). Without preemption spin lock doesn't make sense, as a thread spinning on a CPU will never release a lock.

➡ Dekker's algorithm builds a lock with just loads and stores (assuming they are both atomic).

➡ Peterson's algorithm is an improvement and the idea is to ensure that two threads never enter a critical section at the same time.

```
int flag[2];
int turn;

void init() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}
void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
        ; // spin-wait while it's not your turn
}
void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

➡ I think turn is used in the while loop, to check if actually it is the right turn. Because of interrupt it can lead to a turn not finished.

➡ This idea of developing locks without hardware support didn't get far, as people realised it is much easier to assume a little hardware support. The above code doesn't work on modern hardware.

➡ remain in while, only when the flag is set to the other thread and no interrupt happened between turn = 1 - self and while.

➡ The check for turn == 1 - self in while, avoids a deadlock, where second thread runs lock, sets its flag but can't continue, because the flag of other thread is set to 1.

8. Evaluating Spin Locks

   ➡ Correctness: The spin lock provides mutual exclusion, it mean it only allows one thread to enter a critical section at a time.

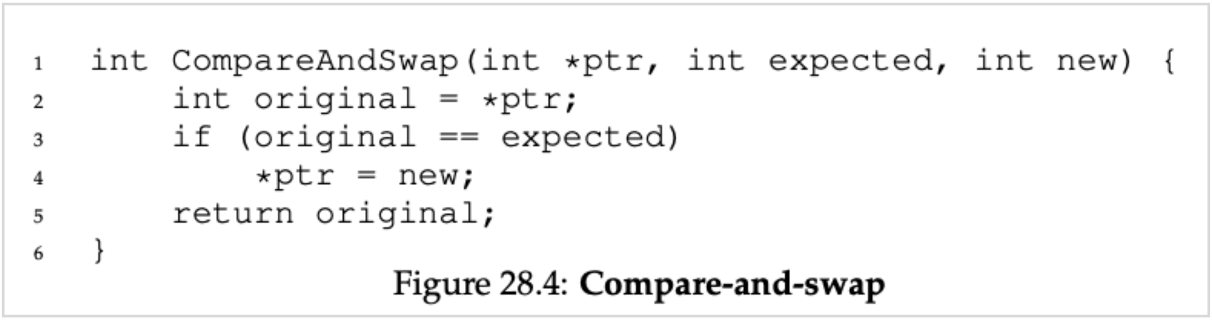
   ➡ Spin locks don't provide any fairness guarantees. A thread spinning may spin forever. Simple spin locks are not fair and may lead to starvation.

   ➡ How is performance?

   ➡ In the case of threads competing for the lock on a single processor the performance overheads can be quite painful. If the thread holding the lock is preempted within a critical section, the scheduler might run every other thread (N - 1), each trying to acquire the lock. Each of these threads will spin for the duration of a time slice before giving up the CPU, waste!

➡ On multiple CPUs spin locks work pretty well if the number of threads roughly equals the number of CPUs. Thread A on CPU 1 and Thread B on CPU 2 both contending for a lock. If Thread A grabs the lock, and then Thread B tries too, B will spin. Spinning to wait for a lock held on another processor doesn't waste as many cycles in this case, and thus can be effective.
➡ Always pretend you are a **malicious scheduler** that interrupts at the most inopportune of times, in order to find mistakes of concurrent execution!

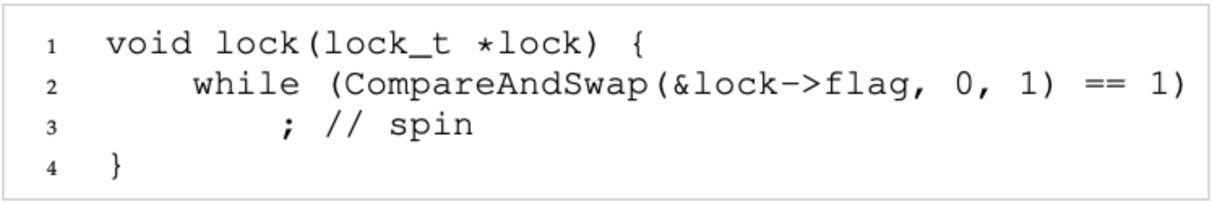
9. Compare-And-Swap
➡ Another hardware primitive is **compare-and-swap** instruction or **compare-and-exchange**:



```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

Figure 28.4: **Compare-and-swap**

➡ Returns original, and sets the pointer when expect equals original. By returning the original we allow the calling Cole to know whether it succeeded or not. This method should be also executed **atomically**!
➡ Building a lock is very similar to the test and set. Only this changes:



```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

➡ See book for C-callable x86 version of compare and swap.
➡ Compare-and-swap a more powerful instruction than test-and-set. We will make use of its power for **lock-free synchronization**.
➡ I think it is more powerful, because in some cases you can know what the value of the pointer is, and you can pass it as the expected parameter. Thus you can save some time. Doesn't really make sense though..

10. Load-Linked and Store-Conditional
➡ The **load linked** and **sotre conditional** instructions can be used in tandem to build locks and other concurrent structures.

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

Figure 28.5: **Load-linked And Store-conditional**

➡ The store-conditional only succeeds when no intervening store to the address has taken place. Success means 1 and updates the value at ptr. If it fails the value at ptr is not updated and 0 is returned.

➡ Lock with these instructions:

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

Figure 28.6: **Using LL/SC To Build A Lock**

➡ Failure for the StoreConditional means the following: One thread calls lock() and executes load linked, returning 0 as the lock is not held. Before it can attempt the store conditional it is interrupted and another thread enters the lock code, also executing the load linked instruction and also getting 0. Both will attempt the store conditional but only one will be able to do so and update the flag to 1. The other thread will fail the store conditional, because the value was updated since the last load linked and thus has to acquire the lock again.

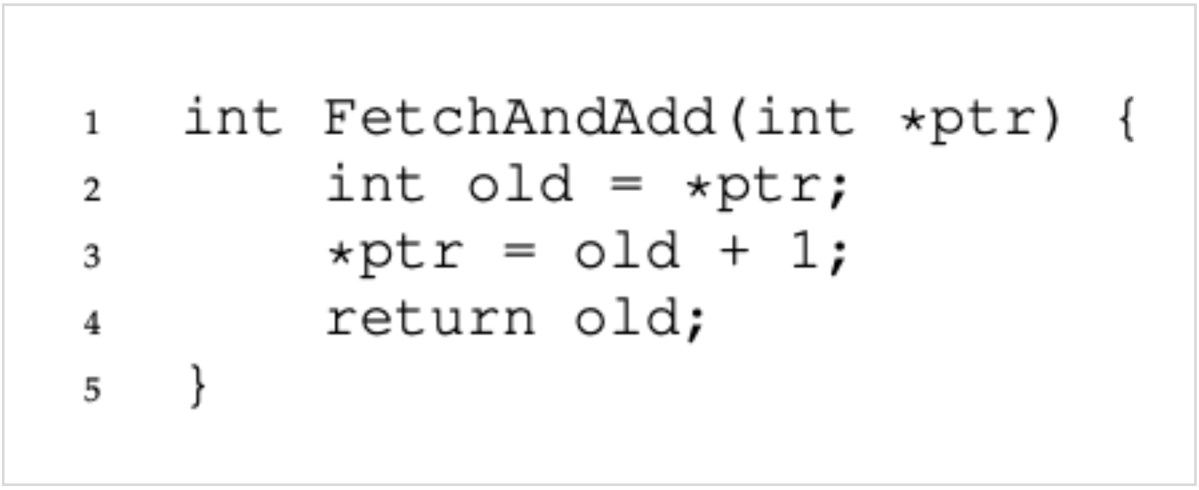
➡ A smaller version:

```
void lock(lock_t *lock) {
    while (LoadLinked(&lock->flag) ||
           !StoreConditional(&lock->flag, 1))
        ; // spin
}
```

➡ Works because is ||, which breaks if first condition wasn't fulfilled.

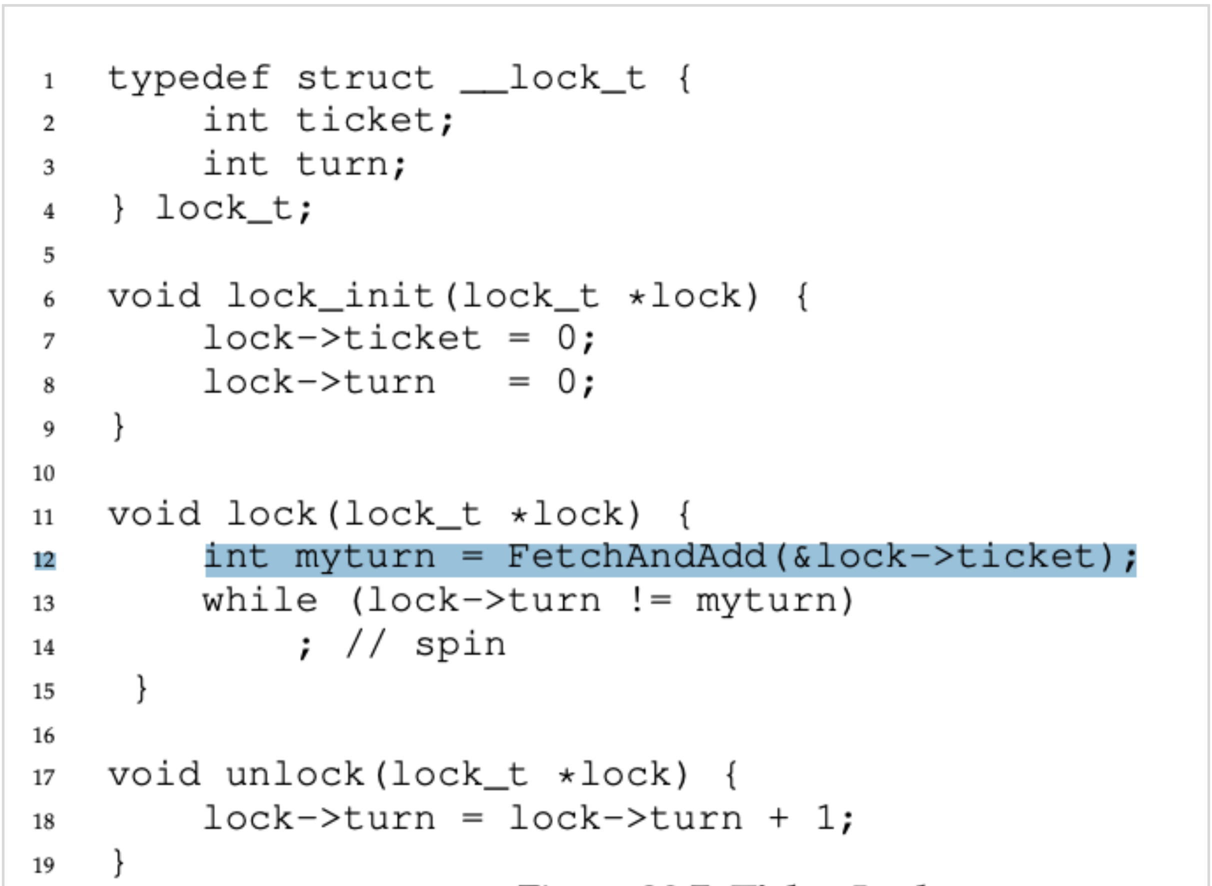
11. Fetch-And-Add

➡ fetch-and-add is a primitive that atomically increments a value while returning the old value at a particular address.



```c
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

➡ We will use it to build a ticket lock:



```c
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

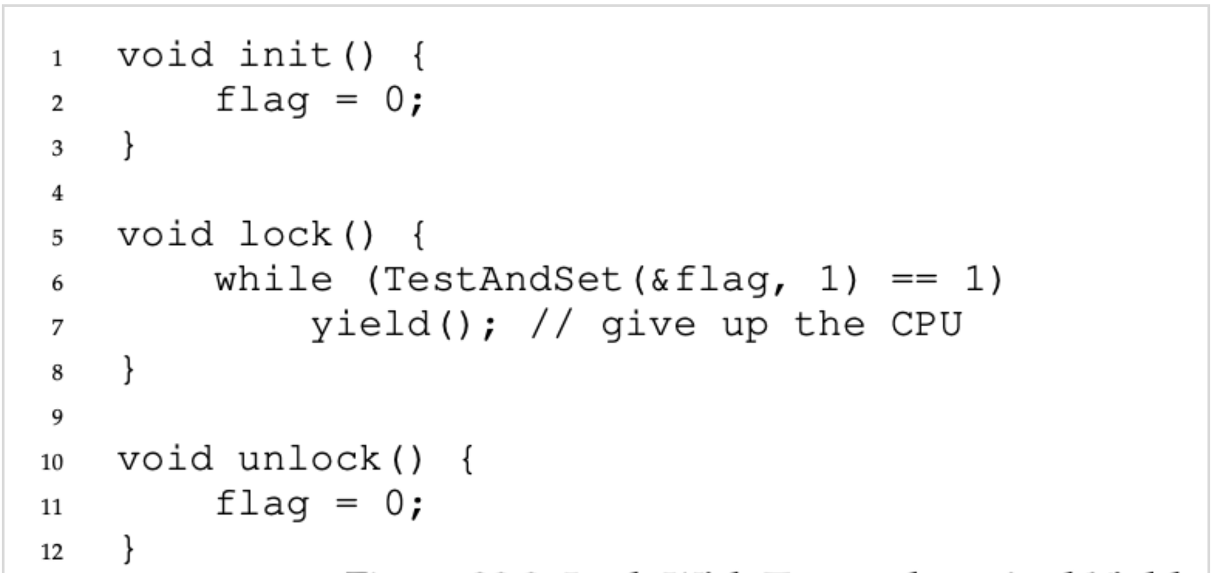
Figure 28.7: Ticket Locks

➡ When a thread wants to acquire a lock, it first does an atomic fetch-and-add on the ticket value. That value is now considerer the thread's "turn". The globally shared lock->turn is used to determine which thread's turn it is. When myTurn == turn then it is that thread's turn to enter the critical section. Unlock simply increments the turn, so that the next waiting threads get a change at entering.

➡ This solution ensures progress for all threads. Once a thread is assigned a ticket value (line 12) it will be scheduled at some point in the future. This wasn't the case in the last approaches!

12. Too Much Spinning: What Now?
➜ These all approaches are pretty simple, but in some cases can be very inefficient. Spinning wastes CPU cycles, and a thread does that when trying to acquire a held lock. It spins until an interrupt come. The problem gets worse with N threads contending for a lock; N - 1 time slices may be wasted in a similar manner, simply spinning and waiting for a single thread to release.
➜ In order to build a lock that doesn't needlessly waste time spinning will also need some OS support.

13. A Simple Approach: Just Yield, Baby
➜ When a thread is going to spin, instead give up the CPU to another thread.



```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Figure 28.8: **Lock With Test-and-set And Yield**

➜ In this approach we assume an operating system primitive **yield()** which a thread call call when it wants to give up the CPU and let another thread to run. Yield is a system call that moves the caller from the running state to the ready state. The yielding thread deschedules itself.
➜ When there are not too many threads, especially only two, this approach works well.
➜ But if we have a lot, like 100, the following happens. One thread holds the lock and before realising it, an interrupt comes. Now all the other 99 will each call lock() and yield the CPU. Assuming a round robin scheduler, all of these will run first before calling the thread that holds the lock. This is better than spinning but still costly. The cost of a context switch can be substantial.
➜ There is also a starvation problem. A thread might get into an endless yield loop while other threads enter and exit the critical section. We need an approach to address this

problem.

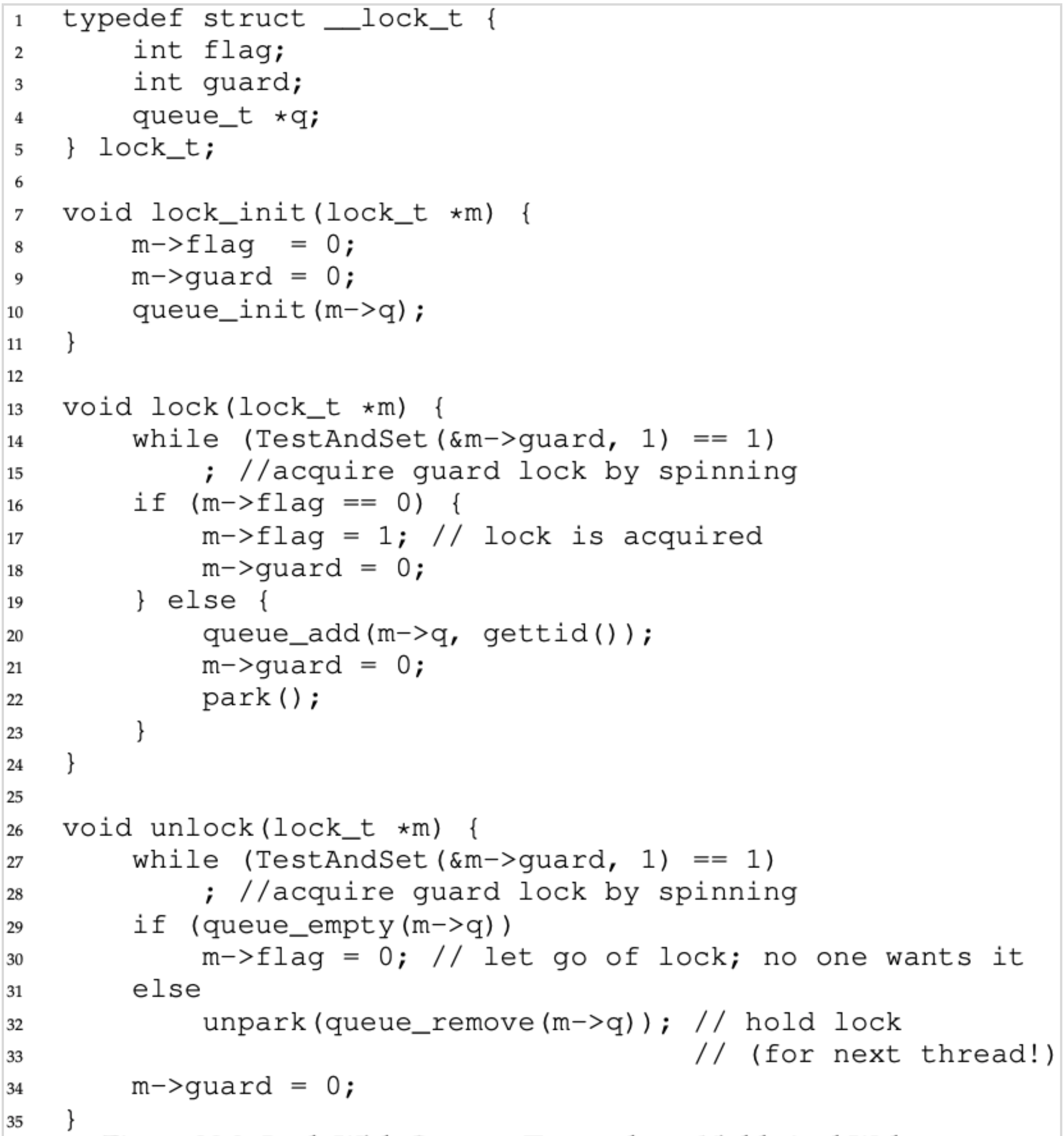
14. Using Queues: Sleeping Instead Of Spinning

➡ Problem with previous approaches is that they leave too much control to the scheduler. If it does a bad choice, then a thread runs that either spins waiting for the lock, or yield the CPU immediately. This is wasteful and no prevention of starvation.

➡ Thus we must exert some control over which thread runs next, after the owner of the lock releases it. Thus we need some OS support and a queue to keep track of which threads are wanting to acquire the lock.

➡ park() puts the calling thread to sleep

➡ unpark(threadID) wakes a particular thread.

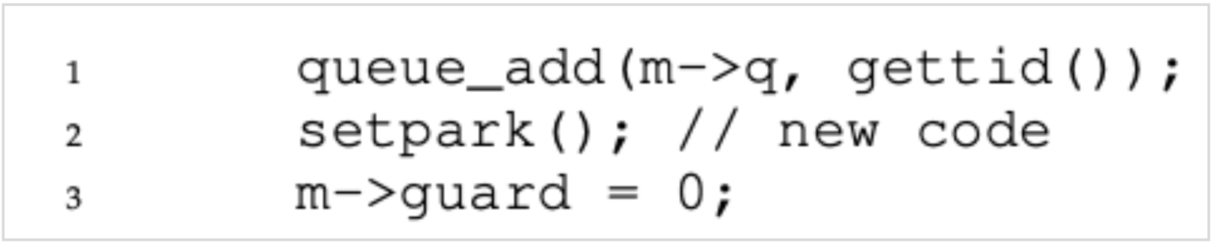


```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

➡ We combine here test-and-set idea with a queue of lock waiters. We use a queue to

help control who gets the lock next and thus avoid starvation.
➡ The guard is used as a spin-lock around the flag and queue manipulations. Guard thus lets only one thread do some acquiring or releasing. This approach doesn't avoid spin-waiting entirely. A thread might be interrupted while acquiring or releasing a lock, and cause other threads to spin-wait for this one to run again. However the time spent spinning is quite limited and thus this may be reasonable.
➡ In lock(), if thread can't acquire the lock, we add ourselves to a queue and set guard to 0, and yield CPU. What happens if the release of the guard lock would come after park()? It would block all acquisitions and releases. In order for it to be award there must be a release, but the release won't happen, because the guard wasn't set to 0. The program gets stuck.
➡ When a thread gets woken up, it doesn't set the flag back to 0. This is a necessity. When it wakes up it doesn't hold the guard at that point and thus cannot even try to set the flag to 1. Thus we just pass the lock directly from the thread realising it to the next thread in queue.
➡ There is a race condition just before the call to park(). With bad timing an interrupt occurs just before the thread calls park() and goes to sleep. It the thread switched to then released that lock, the other thread would then sleep forever. This problem is called **wakeup/waiting race**.
➡ setpark() is a third system call used to solve this problem. It allows a thread to indicate that it is about to park. If it is then interrupted and another threads calls unpack before park is actually called, the subsequent park returns immediately instead of sleeping. The modification of lock() looks like this:



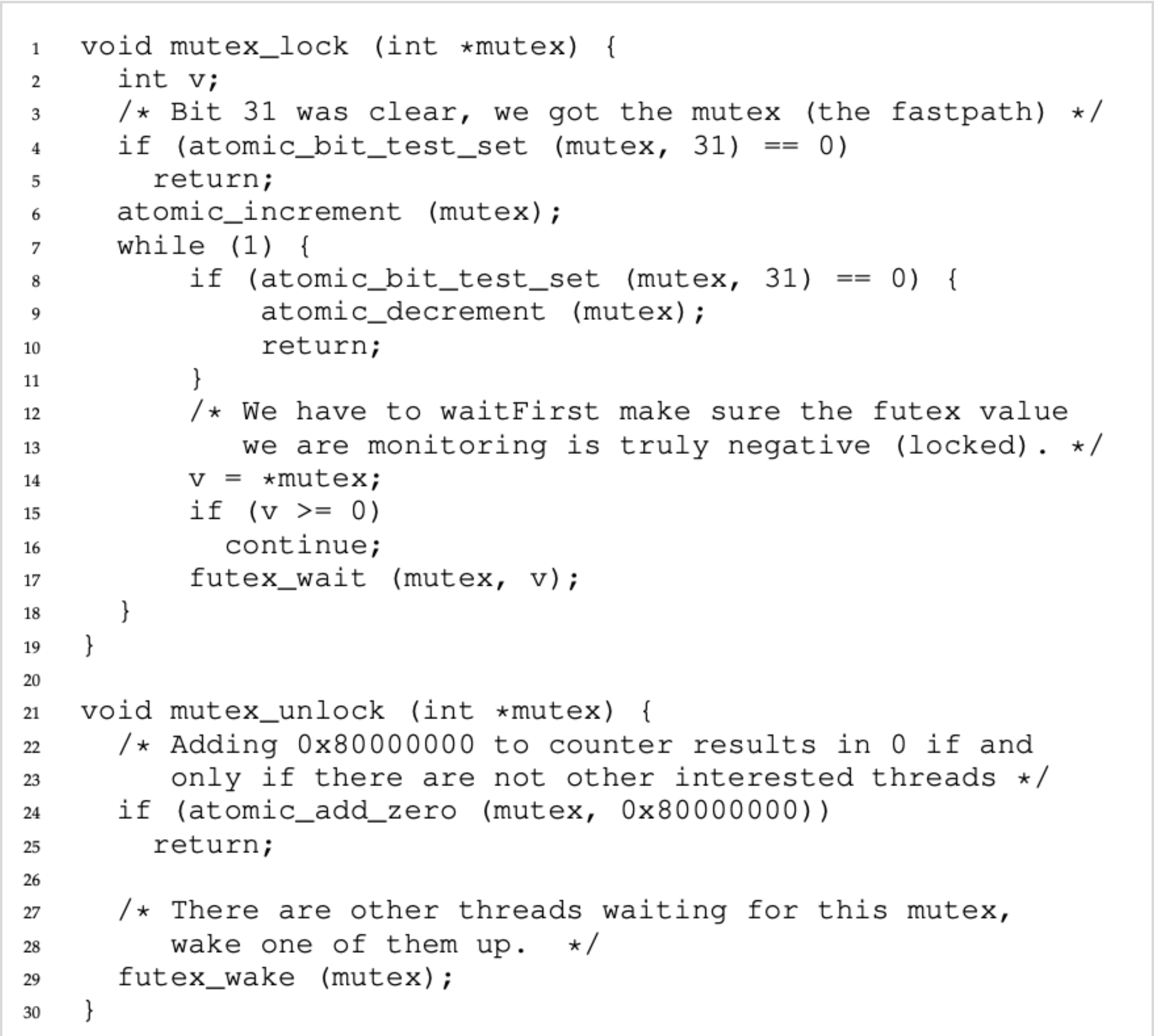
```
1        queue_add(m->q, gettid());
2        setpark(); // new code
3        m->guard = 0;
```

➡ I think in line 4 there still must be a park() call, from what I read from the text.
➡ A different solution would be to pass the guard into the kernel. In this case, the kernel could atomically release the lock and dequeue the running thread.

➡ One reason to avoid spinning is performance. Another one is the correctness, namely the problem of **priority inversion**. When we have two threads 1 and 2, and 2 has a higher priority than 1. If both are ready, the scheduler will always choose thread 2. T1 only runs, when T2 is in a blocked state. So if T2 is blocked, T1 runs and acquires a lock and enters critical section. T2 becomes ready again and scheduler runs it. It also wants to acquire the same lock! It will start to spin, but scheduler will always choose it over T1. So T2 will spin forever!

➡ Problem of inversion doesn't get away with spinning. We have T1, 2 and 3. T3 has highest priority and T1 lowest. T1 grabs a lock, and then T3 starts. Scheduler chooses T3 over T1. T3 then tries to acquire the lock, but it can't because T1 has it. If T2 starts to run, it will have a higher priority than T1 and thus it will run. T3 > T2 but it is stuck waiting for T1, which may never run because T2 > T1. So T2 controls the CPU. Having higher priority is like not having higher priority.
➡ Solution for spinning is to avoid using spin locks. Generally a higher priority thread waiting for a lower one can temporarily boost the lower's thread priority. This method is knows as **priority inheritance**.
➡ A last solution is to make sure all threads have the same priority.

15. Different OS, Different Support
➡ Linux provides a similar OS support. It offers **futex** which provides a more in kernel functionality. Each futex has associated with it a specific physical memory location as well as a per-futex in-kernel queue.
➡ futex_wait(address, expected) puts the calling thread to sleep when value at address is equal to expected. If not equal then the call returns immediately.
➡ futex_wake(address) wakes one thread that is waiting on the queue.



```
1  void mutex_lock (int *mutex) {
2    int v;
3    /* Bit 31 was clear, we got the mutex (the fastpath) */
4    if (atomic_bit_test_set (mutex, 31) == 0)
5      return;
6    atomic_increment (mutex);
7    while (1) {
8        if (atomic_bit_test_set (mutex, 31) == 0) {
9            atomic_decrement (mutex);
10           return;
11       }
12       /* We have to waitFirst make sure the futex value
13          we are monitoring is truly negative (locked). */
14       v = *mutex;
15       if (v >= 0)
16         continue;
17       futex_wait (mutex, v);
18   }
19 }
20
21 void mutex_unlock (int *mutex) {
22   /* Adding 0x80000000 to counter results in 0 if and
23      only if there are not other interested threads */
24   if (atomic_add_zero (mutex, 0x80000000))
25     return;
26
27   /* There are other threads waiting for this mutex,
28      wake one of them up.  */
29   futex_wake (mutex);
30 }
```

Figure 28.10: **Linux-based Futex Locks**

➡ Code is from lowlevellock.h in the nptl library. It uses a single integer to track both whether the lock is held or not (the high bit of the integer) and the number of Waites on the lock (all the other bits). If lock is negative, it is held, because the high bit is set and that bit determines the sign of an integer.

➡ It also optimises for the common case. When only one thread acquiring and releasing a lock, very little work is done (the atomic bit test-and-set and an atomic add to release the lock) (lines 4 and 24).

16. Two-Phase Locks

➡ The linux approach is referred to as a **two-phase lock**. This kind of lock realises that spinning can be useful, especially if the lock if about to be released. So in the first phase, the lock spins for a while, hoping that it can acquire the lock.

➡ If the lock is not acquired during the first spin phase, a second phase begins, where the caller is put to sleep. The linux code above is a form of such lock, but it only spins once. It could spin in a loop for a fixed amount of time before using futex to support the sleep.

➡ This is another example of a **hybrid** approach where combining two ideas results in a better one. However making a single general purpose lock, good for all possible use cases, is quite a challenge.

17. Summary
➡ This is how real locks are built these days. Some hardware support (powerful instructions) plus some OS support (park, unpack in Solaris or futex in Linux).