Concurrency and related terminology

CSE 536 Spring 2026

jedimaestro@asu.edu

https://elixir.bootlin.com/linux/v6.8/source/kernel

/ kernel / sched / core.c

```
6606  */
6607 static void __sched notrace __schedule(unsigned int sched_mode)
6608 {
6609         struct task_struct *prev, *next;
6610         unsigned long *switch_count;
6611         unsigned long prev_state;
6612         struct rq_flags rf;
6613         struct rq *rq;
6614         int cpu;
6615
6616         cpu = smp_processor_id();
6617         rq = cpu_rq(cpu);
6618         prev = rq->curr;
6619
6620         schedule_debug(prev, !!sched_mode);
6621
6622         if (sched_feat(HRTICK) || sched_feat(HRTICK_DL))
6623                 hrtick_clear(rq);
6624
6625         local_irq_disable();
6626         rcu_note_context_switch(!!sched_mode);
```

```
/*
 * Make sure that signal_pending_state()->signal_pending() below
 * can't be reordered with __set_current_state(TASK_INTERRUPTIBLE)
 * done by the caller to avoid the race with signal_wake_up():
 *
 * __set_current_state(@state)          signal_wake_up()
 * schedule()                             set_tsk_thread_flag(p, TIF_SIGPENDING)
 *                                        wake_up_state(p, state)
 *   LOCK rq->lock                          LOCK p->pi_state
 *   smp_mb__after_spinlock()               smp_mb__after_spinlock()
 *     if (signal_pending_state())          if (p->state & @state)
 *
 * Also, the membarrier system call requires a full memory barrier
 * after coming from user-space, before storing to rq->curr.
 */
rq_lock(rq, &rf);
smp_mb__after_spinlock();

/* Promote REQ to ACT */
rq->clock_update_flags <<= 1;
update_rq_clock(rq);
rq->clock_update_flags = RQCF_UPDATED;

switch_count = &prev->nivcsw;
```

/ mm / memory.c

```c
/*
 * These routines also need to handle stuff like marking pages dirty
 * and/or accessed for architectures that don't do it in hardware (most
 * RISC architectures).  The early dirtying is also good on the i386.
 *
 * There is also a hook called "update_mmu_cache()" that architectures
 * with external mmu caches can use to update those (ie the Sparc or
 * PowerPC hashed page tables that act as extended TLBs).
 *
 * We enter with non-exclusive mmap_lock (to exclude vma changes, but allow
 * concurrent faults).
 *
 * The mmap_lock may have been released depending on flags and our return value.
 * See filemap_fault() and __folio_lock_or_retry().
 */
static vm_fault_t handle_pte_fault(struct vm_fault *vmf)
{
        pte_t entry;
```

```
spin_lock(vmf->ptl);
entry = vmf->orig_pte;
if (unlikely(!pte_same(ptep_get(vmf->pte), entry))) {
        update_mmu_tlb(vmf->vma, vmf->address, vmf->pte);
        goto unlock;
}
if (vmf->flags & (FAULT_FLAG_WRITE|FAULT_FLAG_UNSHARE)) {
        if (!pte_write(entry))
                return do_wp_page(vmf);
        else if (likely(vmf->flags & FAULT_FLAG_WRITE))
                entry = pte_mkdirty(entry);
}
entry = pte_mkyoung(entry);
if (ptep_set_access_flags(vmf->vma, vmf->address, vmf->pte, entry,
                          vmf->flags & FAULT_FLAG_WRITE)) {
        update_mmu_cache_range(vmf, vmf->vma, vmf->address,
                          vmf->pte, 1);
```

/ fs / namei.c

```c
int do_rmdir(int dfd, struct filename *name)
{
        int error;
        struct dentry *dentry;
        struct path path;
        struct qstr last;
        int type;
        unsigned int lookup_flags = 0;
retry:
        error = filename_parentat(dfd, name, lookup_flags, &path, &last, &type);
        if (error)
                goto exit1;
```

```
inode_lock_nested(path.dentry->d_inode, I_MUTEX_PARENT);
dentry = lookup_one_qstr_excl(&last, path.dentry, lookup_flags);
error = PTR_ERR(dentry);
if (IS_ERR(dentry))
        goto exit3;
if (!dentry->d_inode) {
        error = -ENOENT;
        goto exit4;
}
error = security_path_rmdir(&path, dentry);
if (error)
        goto exit4;
error = vfs_rmdir(mnt_idmap(path.mnt), path.dentry->d_inode, dentry);

dput(dentry);

inode_unlock(path.dentry->d_inode);
```

Terminology

- Spinlock?
- Mutex?
- Monitor?
- Semaphore?
- Deadlock?
- Futex?

Review: this is a race condition without the lock

- Thread #1

  lock(L)

  x := x + 1

  unlock(L)

  Lock L

  Move x into Register

  Add 1 to Register

  Move Register into x

  Unlock L

- Thread #2

  lock(L)

  x := x + 1

  unlock(L)

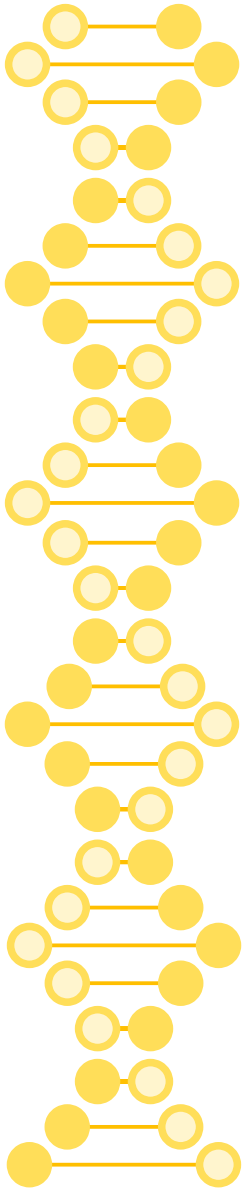
  Lock L

  Move x into Register

  Add 1 to Register

  Move Register into x

  Unlock L

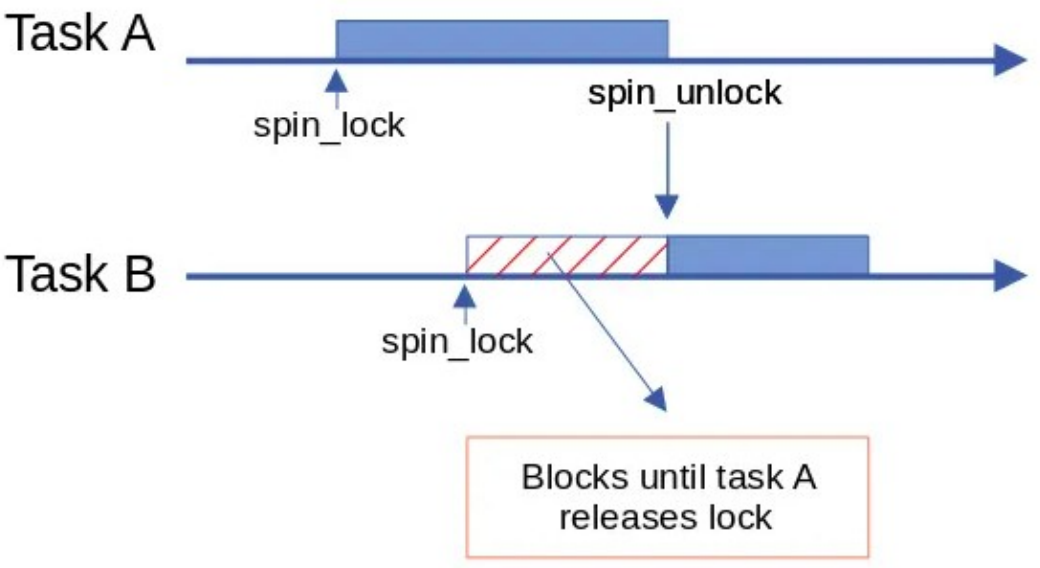


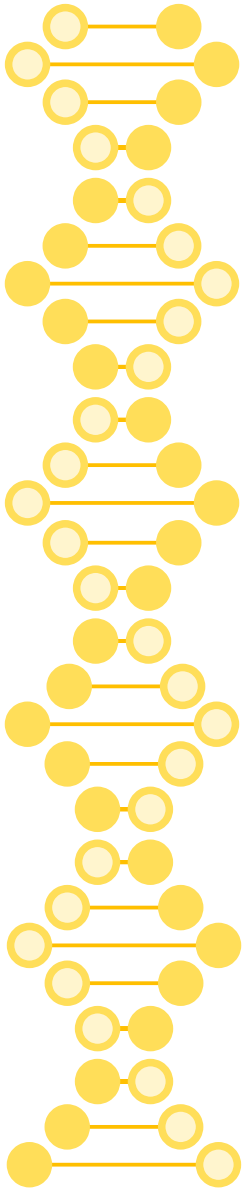
Terminology: the code between the lock and unlock is called the *critical section*.

https://deepdives.medium.com/kernel-locking-deep-dive-into-spinlocks-part-1-bcdc46ee8df6

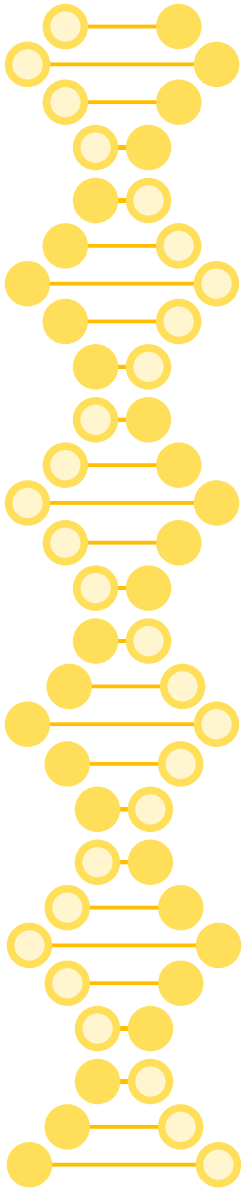
What are Spinlocks ?

Spinlock is probably the 'simplest' locking primitive that protects critical sections of code. This is conceptually how it works :



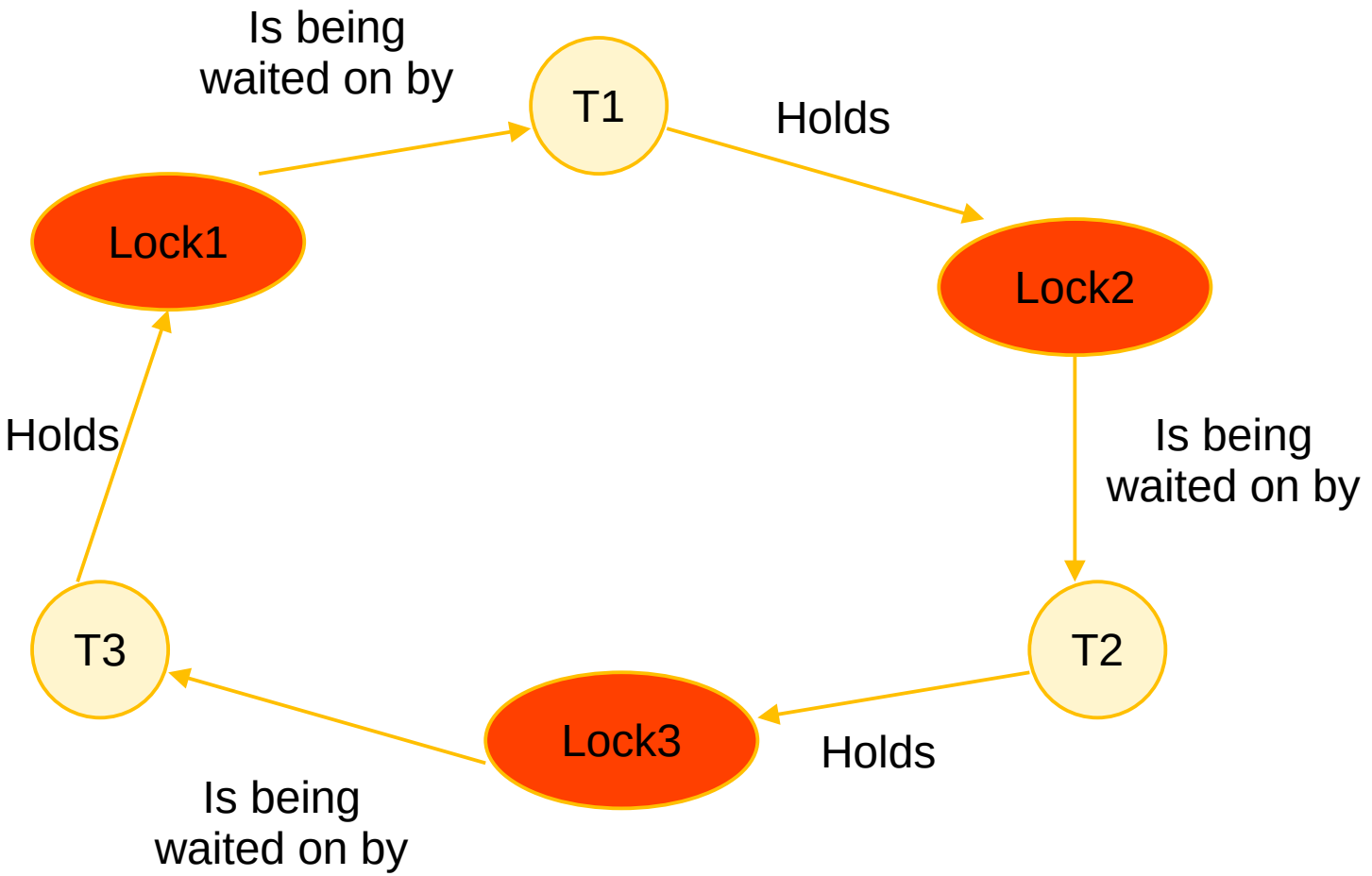


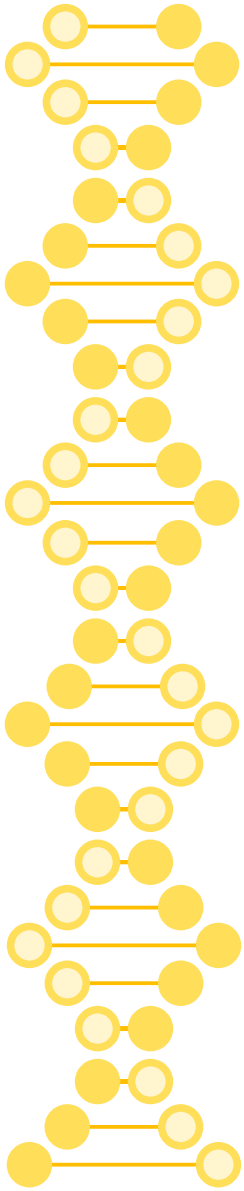
First we’ll talk about deadlocks because we need that terminology for semaphores, then we’ll talk about semaphores so we can see what what a “binary semaphore” is, then we’ll get into mutex, futex, *etc.* ...



Deadlock conditions

- All four conditions must be met for deadlock to occur, *i.e.*, if you break any of these you have mitigated deadlocks
  - Mutual exclusion (exclusive access to resources)
  - Hold-and-wait (hold resources while obtaining others)
  - No preemption (can’t take resources away from threads)
  - Circular wait (circular chain of threads waiting on resources)


Is being
waited on by
T1
Holds
Lock1
Lock2
Holds
Is being
waited on by
T3
T2
Lock3
Holds
Is being
waited on by



Break circular wait

- Always grab locks in the same order
  - Programming discipline, no OS support needed
  - *E.g.*, always grab Lock1 before Lock2

Breaking hold-and-wait

- Grab all locks at the same time, atomically, by defining a global lock
  - *e.g.*:

```
Lock(GlobalLock);
Lock(Lock1);
Lock(Lock2);
Lock(Lock3);
Unlock(GlobalLock);
```

  - Not good for parallelism

Breaking no preemption

```
1    top:
2         lock(L1);
3         if( tryLock(L2) == -1 ){
4                   unlock(L1);
5                   goto top;
6         }
```

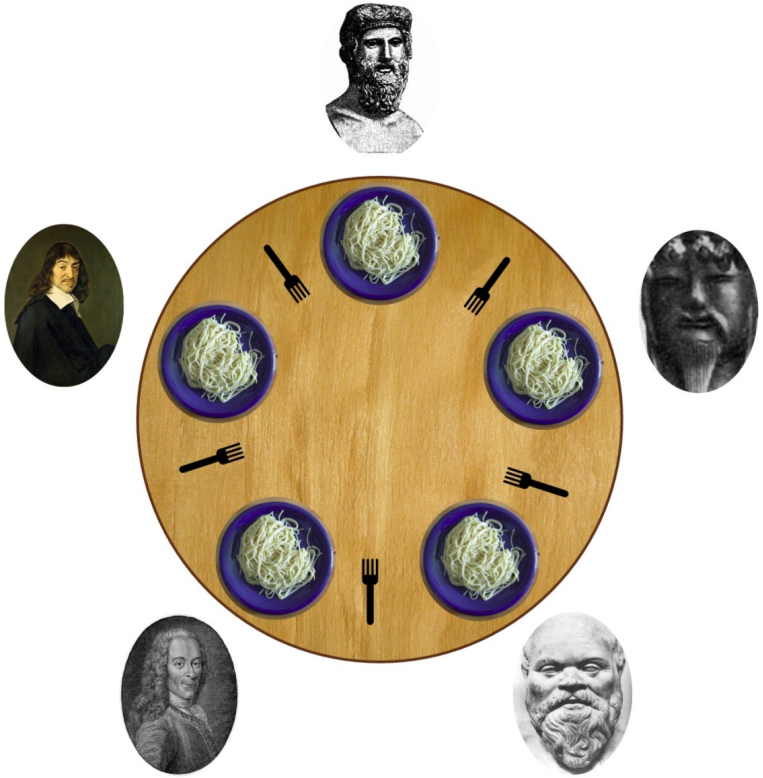
- Problem: livelock, where two processes keep looping through this if the lock order is reversed
- Solution: random delay

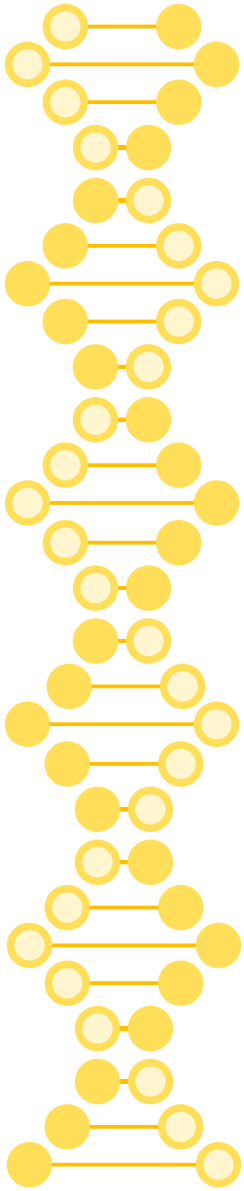
Breaking mutual exclusion

```
1   int CompareAndSwap(int *address, int expected, int new){
2       if(*address == expected){
3               *address = new;
4               return 1; // success
5       }
6       return 0;
7   }
```

```
1    void insert(int value) {
2        node_t *n = malloc(sizeof(node_t));
3        assert(n != NULL);
4        n->value = value;
5        do {
6                n->next = head;
7        } while (CompareAndSwap(&head, n->next, n));
8    }
```

https://en.wikipedia.org/wiki/Dining_philosophers_problem





Requirements

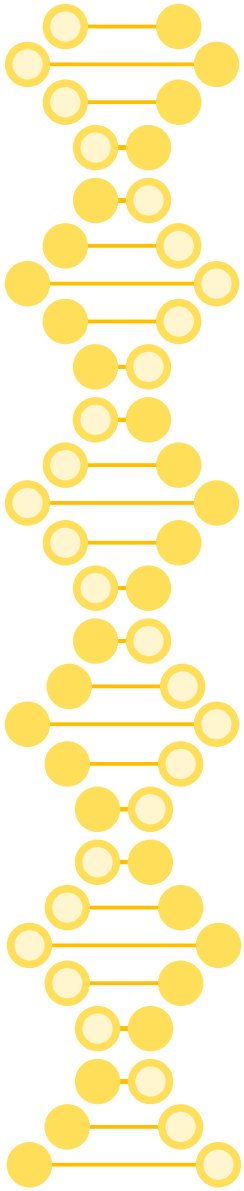
- No deadlocks
- No starvation
- High degree of parallelism

Semaphores

- Invented by Edsger Dijkstra in 1962 or 1963
- https://en.wikipedia.org/wiki/Semaphore_(programming)

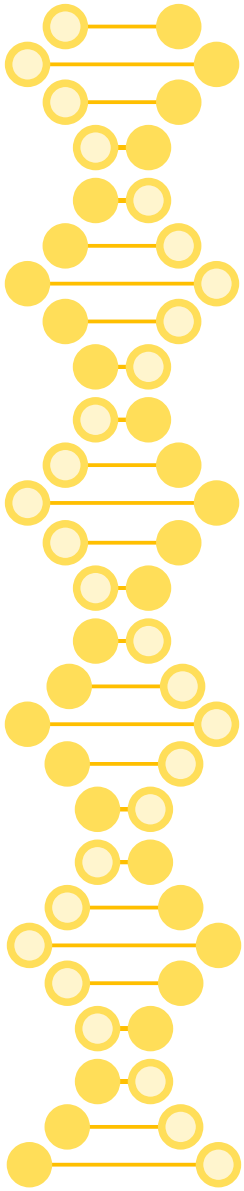
https://en.wikipedia.org/wiki/Electrologica_X8#/media/File:Electrologica_X8.jpg

https://en.wikipedia.org/wiki/Semaphore_(programming)#/media/File:Rail-semaphore-signal-Dave-F.jpg



Semaphore operations

- wait
  - Also known as
    - P
    - proberen
    - prolaag
    - down
    - acquire
- signal
  - Also known as
    - V
    - verhogen
    - vrijgave
    - up
    - release



Things we can do with semaphores

- Locks
  - *a.k.a.* binary semaphores
- Producer-consumer
  - uses binary and counting semaphores
- Dining philosophers solution

Atomic operations

wait (/down)
(P for Dutch "proberen"=to test)

```
void wait ( semaphore & S ) {
    S.value = S.value - 1;
    if (S.value < 0) {
        add this process to S.L;
        block;  //puts to sleep until another
                // process wakes it
    } // end if
} // end wait
```

signal (/up)
(V for Dutch "verhogen"=to increment)

```
void signal ( semaphore & S ) {
    S.value = S.value + 1;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P) ;  //unblocks a sleeping
                     // process
    } // end if
} // end signal
```

https://www.cs.uni.edu/~fienup/courses/copy-of-operating-systems/lecture-notes/notes98f-7.lwp/odyframe.htm

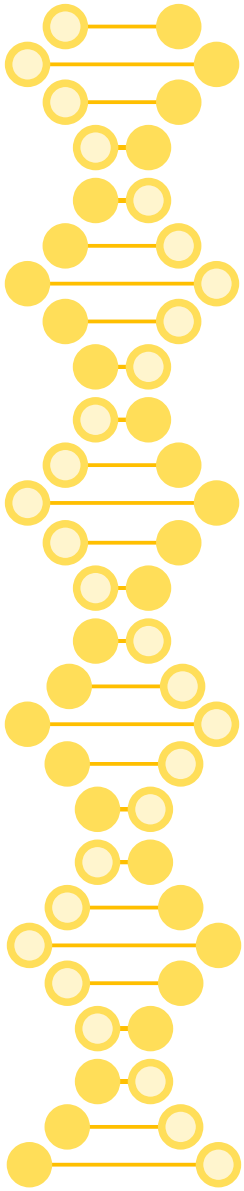
But how to make those atomic?

```
spin_lock:
    mov %rax, %rdx           # Store current value in RDX
    lock cmpxchg %rcx, (%rdi) # cmpxchg updates rax if it fails
    jne spin_lock            # Loop if ZF is not set (comparison failed)

    ret
```

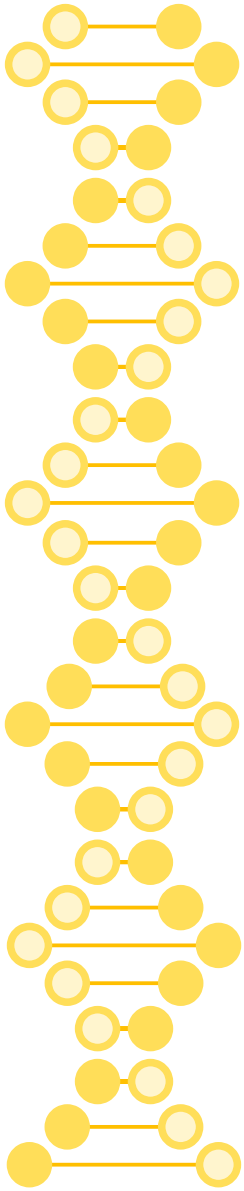
```
// The entirety of the below is executed ATOMICALLY
TYPE cmpxchg(TYPE* object, TYPE expected, TYPE desired) {
  TYPE actual = *object;
  if (actual == expected)
   *object = desired;
  return actual;
}
```

http://www0.cs.ucl.ac.uk/staff/B.Karp/3007/s2018/lectures/3007-lecture17-threads-x86-64.pdf



Producer-Consumer Problem

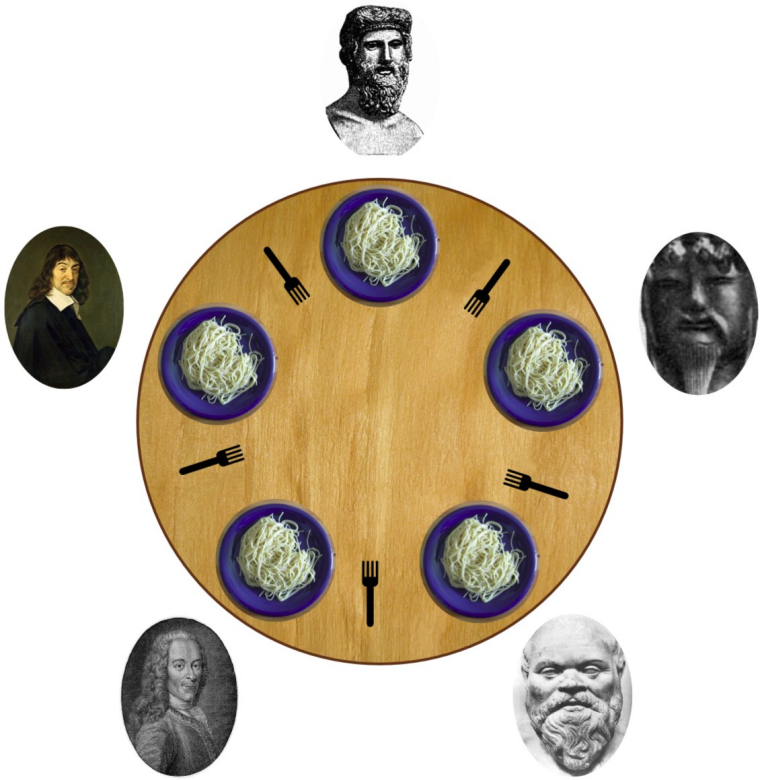
- Producer produces items
- Consumer consumes them
- Can have multiple producers and consumers running in parallel
- Requirements:
  - Concurrency (if there's work to do and a thread to do it, they should do it...)
  - No race conditions



```
produce:
    P(emptyCount)
    P(useQueue)
    putItemIntoQueue(item)
    V(useQueue)
    V(fullCount)
```

```
consume:
    P(fullCount)
    P(useQueue)
    item ← getItemFromQueue()
    V(useQueue)
    V(emptyCount)
```

https://en.wikipedia.org/wiki/Dining_philosophers_problem



```
1    void getforks() {
2    sem_wait(forks[left(p)]);
3    sem_wait(forks[right(p)]);
4    }
5
6    void putforks() {
7    sem_post(forks[left(p)]);
8    sem_post(forks[right(p)]);
9    }
```

The `getforks()` and `putforks()` Routines (Broken Solution)

https://www.cs.unm.edu/~crandall/operatingsystems20/slides/30-Concurrency-Bugs-Non-Deadlock.pdf

```
1    void getforks() {
2    sem_wait(forks[left(p)]);
3    sem_wait(forks[right(p)]);
4    }
5
6    void putforks() {
7    sem_post(forks[left(p)]);
8    sem_post(forks[right(p)]);
9    }
```

Deadlock!

The `getforks()` and `putforks()` Routines (Broken Solution)

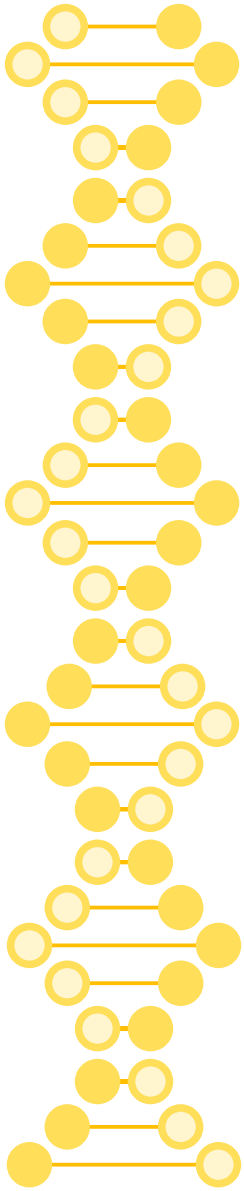
https://www.cs.unm.edu/~crandall/operatingsystems20/slides/30-Concurrency-Bugs-Non-Deadlock.pdf

```
1    void getforks() {
2    if (p == 4) {
3         sem_wait(forks[right(p)]);
4         sem_wait(forks[left(p)]);
5    } else {
6         sem_wait(forks[left(p)]);
7         sem_wait(forks[right(p)]);
8    }
9    }
```

https://www.cs.unm.edu/~crandall/operatingsystems20/slides/30-Concurrency-Bugs-Non-Deadlock.pdf

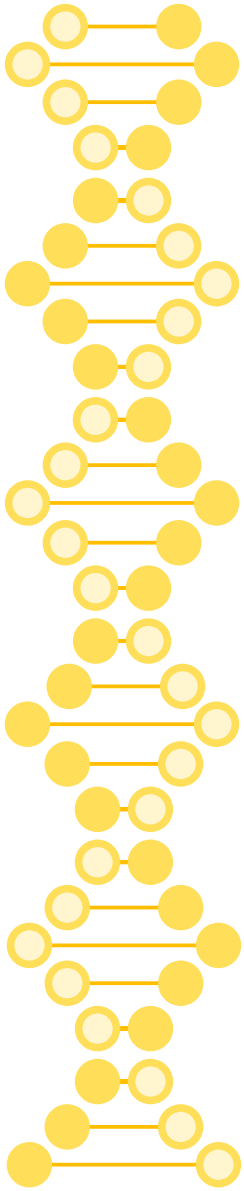
https://en.wikipedia.org/wiki/Lock_(computer_science)

- “While a binary semaphore may be colloquially referred to as a mutex, a true mutex has a more specific use-case and definition, in that only the task that locked the mutex is supposed to unlock it.”
- Basic problem with sempahores: you have no idea which thread is holding which resource
- “a true mutex has a more specific use-case and definition, in that only the task that locked the mutex is supposed to unlock it”
  - Implies OS support, or some type of runtime environment + memory safety
- If you wrap a mutex in an object-like programming construct you can call it a monitor
  - Ada, C#, Java, Go, Mesa, Python, ...



Problems with semaphores

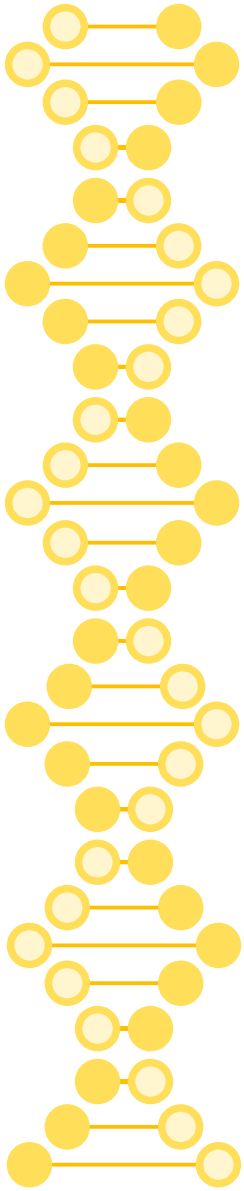
- Priority inversion (*vs.* OS can do priority inheritance)
- Premature task termination (*vs.* OS can release mutexes)
- Termination deadlock (*vs.* OS can release mutexes)
- Recursion deadlock (*vs.* mutexes can be reentrant)
- Accidental release (*vs.* OS can raise an error)



https://yarchive.net/comp/linux/semaphores.html

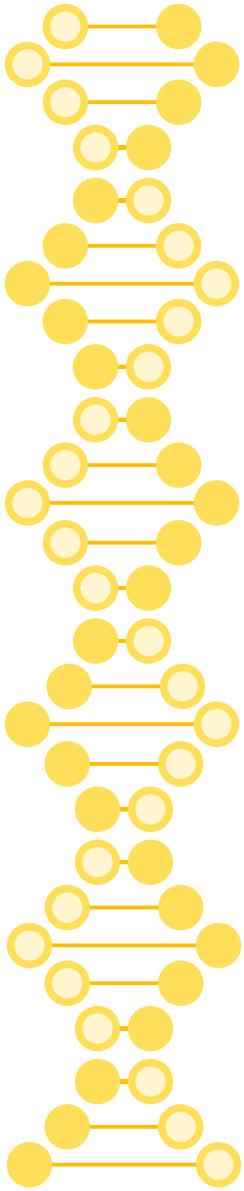
However, almost all practical use of semaphores is a special case where the counter is initialized to 1, and where they are used as simple mutual exclusion with only one user allowed in the critical region. Such a semaphore is often called a "mutex" semaphore for MUTual EXclusion.

Potentially useful, as I said, but the common case (and the only case currently in use in the Linux kernel - even though the implementation definitely can handle the general case) is certainly the mutex one.



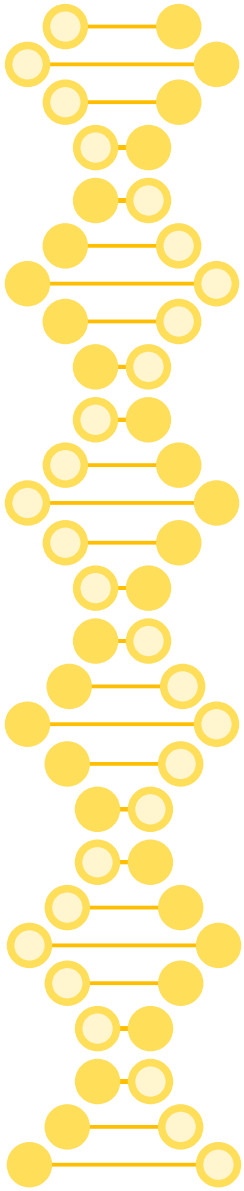
Linux kernel

- Spin locks when going to sleep is not an option
  - Low-level code
- Mutexes most of the rest of the time
- Semaphores in specific use cases, *e.g.*, USB gadget throttling
  - Counting semaphore
  - Still need to sleep (can’t use a spinlock)
  - Signaling (different task or interrupt handler can `up()` than the one that `down()`ed



Linux kernel mutexes

- Three-path implementation
  - Fast path – grab lock through atomic operations if possible
  - Midpath – optimistic spinlock (for just a little bit)
  - Slow path – go into a wait queue
- Priority inheritance
- Only the task that locked the mutex can unlock it, and they can’t lock it recursively



Same concepts apply in user space...

Need OS support

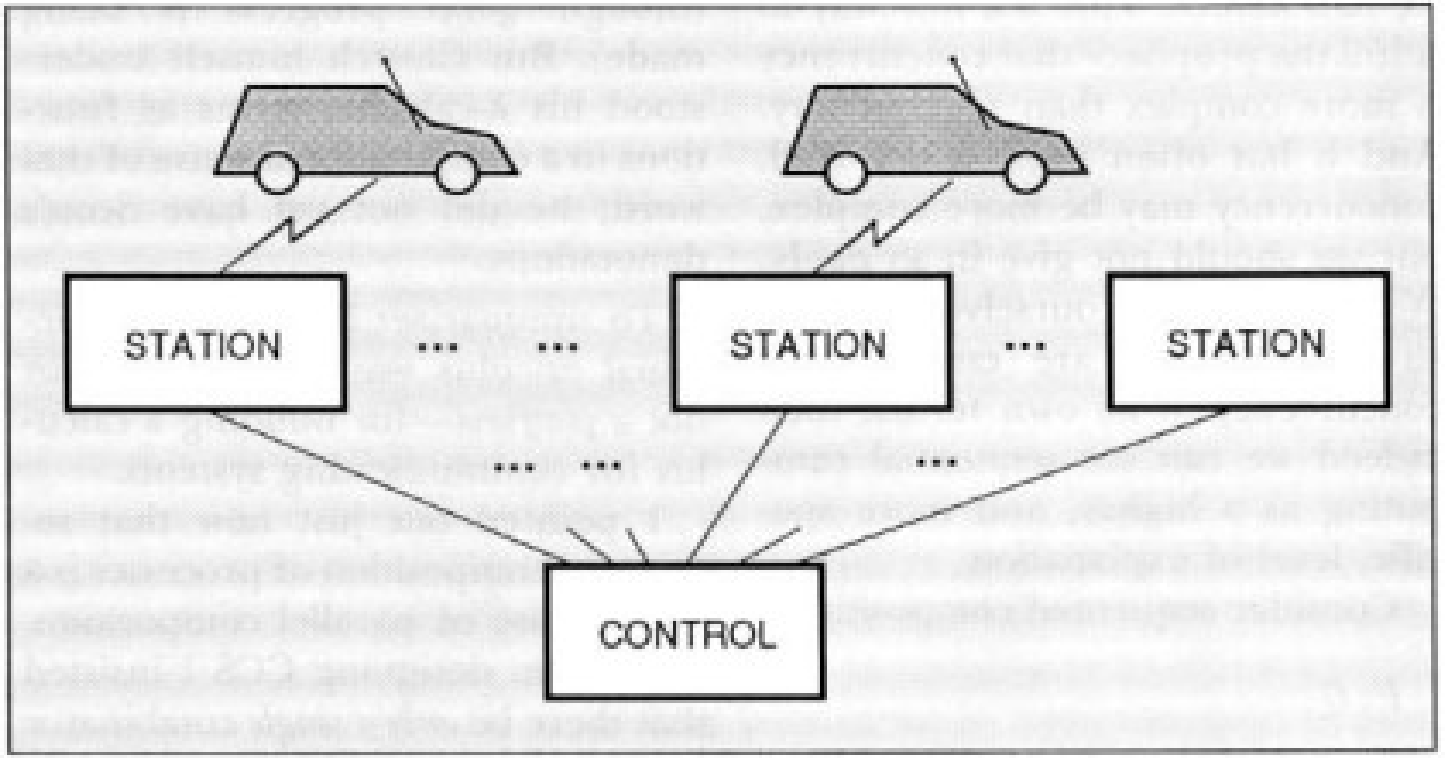
- Spinning to wait for a lock uses up 100% of a CPU when you're scheduled
- Do this instead...

```
1   void init() {
2       flag = 0;
3   }
4
5   void lock() {
6       while (TestAndSet(&flag, 1) == 1)
7           yield(); // give up the CPU
8   }
9
10  void unlock() {
11      flag = 0;
12  }
```

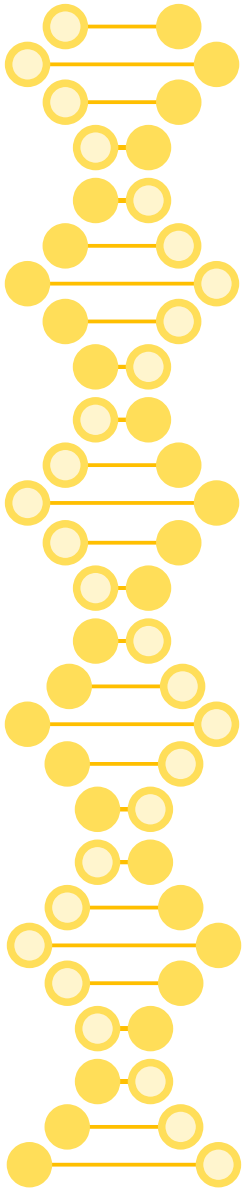
Linux's futex (similar to `setpark`, `park`, and `unpark` on Solaris)

- `futex_wait(address, expected)`
  - Put the calling thread to sleep
  - If the value at `address` is not equal to `expected`, the call returns immediately.
- `futex_wake(address)`
  - Wake one thread that is waiting on the queue.

Can we use semaphores, mutexes, *etc.* for this?

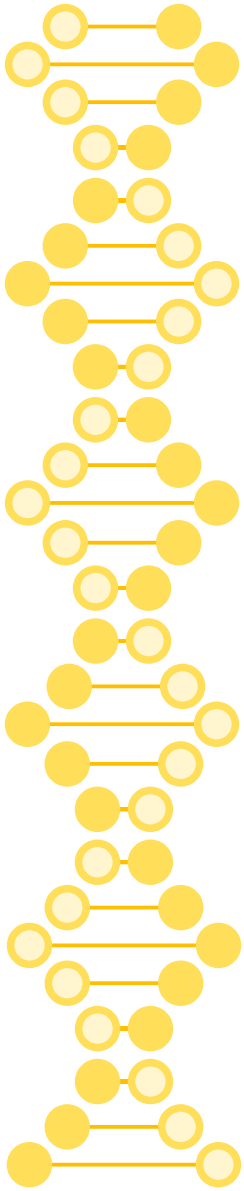


https://dl.acm.org/doi/pdf/10.1145/151233.151240



Coming up...

- poll(), select(), and epoll()
  - Event-based and asynchronous I/O
- Message passing
- Remote Procedure Calls



Source: Patrick Bridges' slides…

https://www.cs.unm.edu/~crandall/operatingsystems20/slides/31-Concurrency-Bugs-Deadlock.pdf

https://www.cs.unm.edu/~crandall/operatingsystems20/slides/26-Concurrency-Critical-Sections-2.pdf