Betriebssysteme

10. Tutorium - Synchronization und Deadlocks

Peter Bohner

16. Januar 2025

ITEC - Operating Systems Group

Synchronization Primitives

Synchronization Primitives

There are different kinds of synchronization primitives

Which ones do you know?

Spinlock

- `lock` / `unlock`
- Busy-waiting and atomic instructions (e.g. compare-and-set)

Synchronization Primitives

There are different kinds of synchronization primitives

Which ones do you know?

Spinlock

- `lock` / `unlock`
- Busy-waiting and atomic instructions (e.g. compare-and-set)
- Recommended for *short* critical sections as it wastes CPU time

Synchronization Primitives

There are different kinds of synchronization primitives

Which ones do you know?

Spinlock

- `lock` / `unlock`
- Busy-waiting and atomic instructions (e.g. compare-and-set)
- Recommended for *short* critical sections as it wastes CPU time
- Preemption wastes more resources (threads can't make progress)

Synchronization Primitives

There are different kinds of synchronization primitives

Which ones do you know?

Spinlock

- `lock` / `unlock`
- Busy-waiting and atomic instructions (e.g. compare-and-set)
- Recommended for *short* critical sections as it wastes CPU time
- Preemption wastes more resources (threads can't make progress)
- $\Rightarrow$ Mostly used in the kernel without interrupts

Synchronization Primitives

Semaphore

Synchronization Primitives

Semaphore

- `wait(sem)` (also called *acquire*) / `signal(sem)` (Also called *release*/*post*)

Synchronization Primitives

Semaphore

- `wait(sem)` (also called *acquire*) / `signal(sem)` (Also called *release*/*post*)
- Has an internal counter that is decremented (wait) or incremented (signal)

Synchronization Primitives

Semaphore

- `wait(sem)` (also called *acquire*) / `signal(sem)` (Also called *release*/*post*)
- Has an internal counter that is decremented (wait) or incremented (signal)
- Blocks if you try to decrement it below 0

Synchronization Primitives

Semaphore

- `wait(sem)` (also called *acquire*) / `signal(sem)` (Also called *release*/*post*)
- Has an internal counter that is decremented (wait) or incremented (signal)
- Blocks if you try to decrement it below 0
- Can be used to implement bounded buffers

Synchronization Primitives

Semaphore

- `wait(sem)` (also called *acquire*) / `signal(sem)` (Also called *release*/*post*)
- Has an internal counter that is decremented (wait) or incremented (signal)
- Blocks if you try to decrement it below 0
- Can be used to implement bounded buffers

Mutex

Synchronization Primitives

Semaphore

- `wait(sem)` (also called *acquire*) / `signal(sem)` (Also called *release*/*post*)
- Has an internal counter that is decremented (wait) or incremented (signal)
- Blocks if you try to decrement it below 0
- Can be used to implement bounded buffers

Mutex (Binary Semaphore)

Synchronization Primitives

Semaphore

- `wait(sem)` (also called *acquire*) / `signal(sem)` (Also called *release*/*post*)
- Has an internal counter that is decremented (wait) or incremented (signal)
- Blocks if you try to decrement it below 0
- Can be used to implement bounded buffers

Mutex (Binary Semaphore)

- `lock(m)`, `unlock(m)`

Synchronization Primitives

Semaphore

- `wait(sem)` (also called *acquire*) / `signal(sem)` (Also called *release*/*post*)
- Has an internal counter that is decremented (wait) or incremented (signal)
- Blocks if you try to decrement it below 0
- Can be used to implement bounded buffers

Mutex (Binary Semaphore)

- `lock(m)`, `unlock(m)`
- Or a Semaphore with values 0 and 1

Synchronization Primitives

Condition Variables

```
void consume() {
  lock(l);
  while(queue.size == 0) {
    unlock(l);
    sleep(); lock(l);
  }
  queue.poll(); unlock(l); signal();
}
void produce() {
  lock(l);
  while(queue.size == MAX_SIZE) {
    unlock(l);
    sleep(); lock(l);
  }
  queue.add(X); unlock(l); signal();
}
```

This code can incorrectly sleep a consumer/producer. How?

Synchronization Primitives

Condition Variables

```
void consume() {
  lock(l);
  while(queue.size == 0) {
    unlock(l);
    sleep(); lock(l);
  }
  queue.poll(); unlock(l); signal();
}
void produce() {
  lock(l);
  while(queue.size == MAX_SIZE) {
    unlock(l);
    sleep(); lock(l);
  }
  queue.add(X); unlock(l); signal();
}
```

This code can incorrectly sleep a consumer/producer. How? *Lost wakeup problem*

Condition Variables

```
void consume() {
  lock(l);
  while(queue.size == 0) {
    // unlocks and sleeps atomically. Relocks when waking up
    wait(cond_filled, l);
  }
  queue.poll(); signal(cond_empty); unlock(l);
}
void produce() {
  lock(l);
  while(queue.size == MAX_SIZE) {
    // unlocks and sleeps atomically. Relocks when waking up
    wait(cond_empty, l);
  }
  queue.add(X); signal(cond_filled); unlock(l);
}
```

Now no wakeup is lost :)

Deadlocks

Deadlocks

What is that? Do you know an example?

Deadlocks

What is that? Do you know an example?

- Several processes or activities can not make progress, as they are waiting for resources held by each other

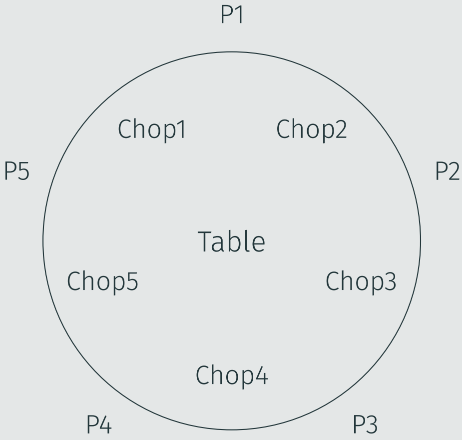
Deadlocks

What is that? Do you know an example?

- Several processes or activities can not make progress, as they are waiting for resources held by each other
- Examples: 4-way intersection, *Dining Philosophers*

Dining Philosophers

The Problem

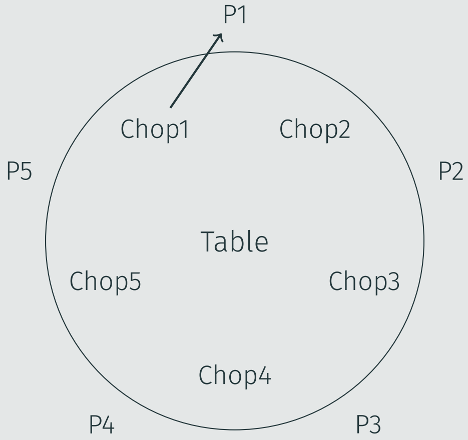


Philosophers (P) want to eat, but to do that they need *two* Chopsticks (Chop)!

How can this deadlock?

Dining Philosophers

The Problem

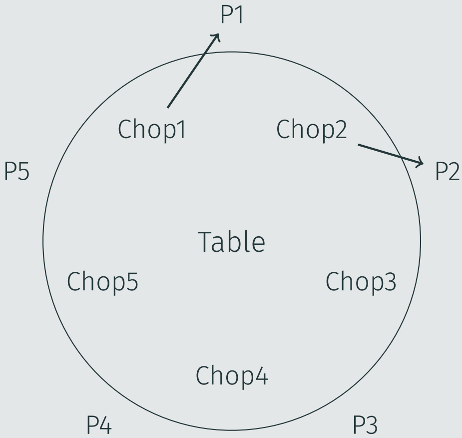


Philosophers (P) want to eat, but to do that they need *two* Chopsticks (Chop)!

How can this deadlock?

Dining Philosophers

The Problem

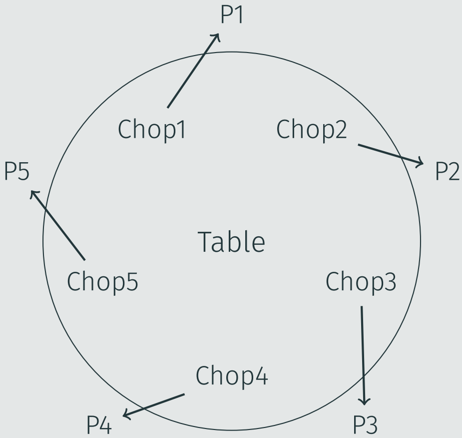


Philosophers (P) want to eat, but to do that they need *two* Chopsticks (Chop)!

How can this deadlock?

Dining Philosophers

The Problem

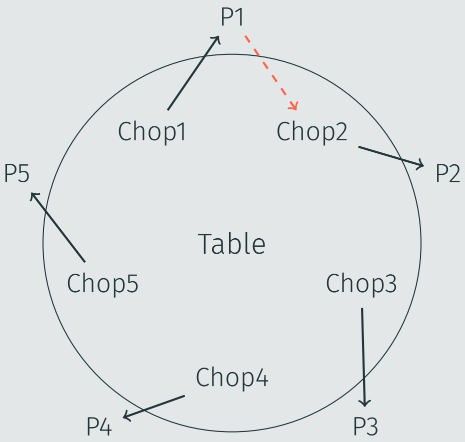


Philosophers (P) want to eat, but to do that they need *two* Chopsticks (Chop)!

How can this deadlock?

Dining Philosophers

The Problem

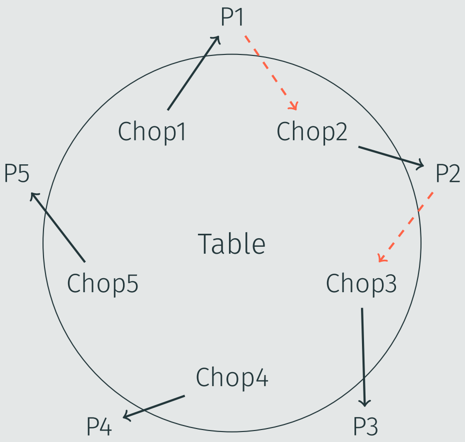


Philosophers (P) want to eat, but to do that they need *two* Chopsticks (Chop)!

How can this deadlock?

Dining Philosophers

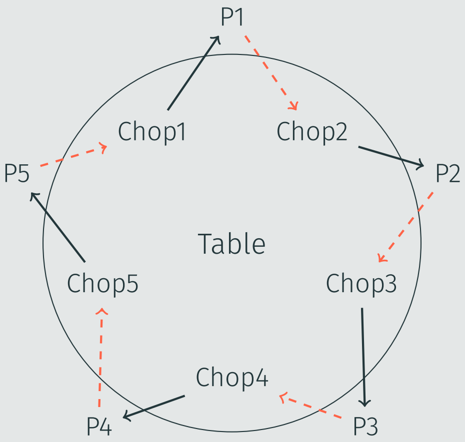
The Problem



Philosophers (P) want to eat, but to do that they need *two* Chopsticks (Chop)!

How can this deadlock?

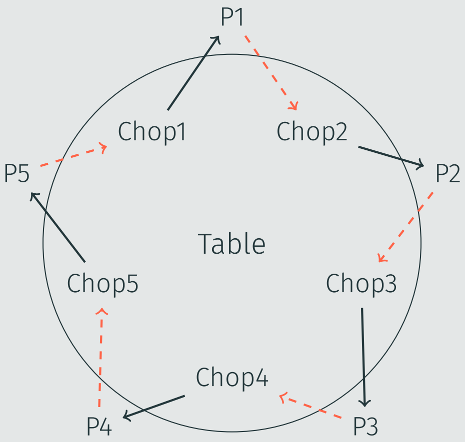
The Problem



Philosophers (P) want to eat, but to do that they need *two* Chopsticks (Chop)!

How can this deadlock?

The Problem



Philosophers (P) want to eat, but to do that they need *two* Chopsticks (Chop)!

How can this deadlock?

Why did that happen? What fateful circumstances lead to this starvation?

The Four ~~Horsemen of the Apocalypse~~ *Coffman Conditions*

The Four ~~Horsemen of the Apocalypse~~ *Coffman Conditions*

Mutual Exclusion

The Four ~~Horsemen of the Apocalypse~~ *Coffman Conditions*

Mutual Exclusion

Resources can not be shared between processes

The Four ~~Horsemen of the Apocalypse~~ *Coffman Conditions*

Mutual Exclusion

Resources can not be shared between processes

Hold and wait

A process already holding resources can acquire more

The Four ~~Horsemen of the Apocalypse~~ *Coffman Conditions*

Mutual Exclusion

Resources can not be shared between processes

Hold and wait

A process already holding resources can acquire more

No Preemption

Resources can not be forcibly taken away from processes

The Four ~~Horsemen of the Apocalypse~~ *Coffman Conditions*

Mutual Exclusion

Resources can not be shared between processes

Hold and wait

A process already holding resources can acquire more

No Preemption

Resources can not be forcibly taken away from processes

Circular Wait

There exists a set of Processes $P_0, P_1, \ldots P_n$ where P_0 is waiting for a resource held by P_1. P_1 is waiting for a resource held by P_2, ...and P_n is waiting for a resource held by P_1

The Four ~~Horsemen of the Apocalypse~~ *Coffman Conditions*

Mutual Exclusion

Resources can not be shared between processes

Hold and wait

A process already holding resources can acquire more

No Preemption

Resources can not be forcibly taken away from processes

Circular Wait

There exists a set of Processes $P_0, P_1, \ldots P_n$ where P_0 is waiting for a resource held by P_1. P_1 is waiting for a resource held by P_2, ...and P_n is waiting for a resource held by P_1

Note

These conditions *are not independent*! (e.g. Circular Wait $\Rightarrow$ Hold And Wait)

Finding a deadlock

Code

```
Spinlock s1,s2, s3 = FREE;
int counter = 0;
void Thread1() {
  if(counter == 0) {
    lock(s1);
    counter++;
    unlock(s1);
  }
  lock(s2);
  lock(s3);
  // update some more data
  unlock(s3);
  unlock(s2);
}
void Thread2() {
  lock(s3);
  counter++;
  // update some data
  if(counter == 2) {
    lock(s2);
    // update some more data
    unlock(s2);
  }
  lock(s1);
  // update even more data
  unlock(s3);
  unlock(s1);
}
```

Deadlock Prevention and Avoidance

Deadlock Prevention

Deadlock Prevention

Make a deadlock *impossible*!

Deadlock Prevention

Make a deadlock *impossible*! How?

Deadlock Prevention and Avoidance

Deadlock Prevention

Make a deadlock *impossible*! How? Break ≥ 1 of the four necessary conditions

Deadlock Prevention and Avoidance

Deadlock Prevention

Make a deadlock *impossible*! How? Break ≥ 1 of the four necessary conditions

Deadlock Avoidance

Deadlock Prevention and Avoidance

Deadlock Prevention

Make a deadlock *impossible*! How? Break ≥ 1 of the four necessary conditions

Deadlock Avoidance

- Deadlocks are still possible
- The resource allocator knows what resources are used by the processes
- The resource allocator denies requests that *might* lead to a deadlock

Deadlock Prevention - Slay The Horses

How can you negate *Mutual Exclusion*?

Deadlock Prevention - Slay The Horses

How can you negate *Mutual Exclusion*?

Sometimes: Spooling

Deadlock Prevention - Slay The Horses

How can you negate *Mutual Exclusion*?

Sometimes: Spooling

Like a Printer

- You send a job
- It is executed

$\Rightarrow$ Only the executor has access to the resource

Negate *Hold And Wait*

Deadlock Prevention - Slay The Horses

How can you negate *Mutual Exclusion*?

Sometimes: Spooling

Like a Printer

- You send a job
- It is executed

$\Rightarrow$ Only the executor has access to the resource

Negate *Hold And Wait*

Allocate resources atomically: All you will need or nothing
$\Rightarrow$ Once you have resources, you can no longer request new ones

Deadlock Prevention - Slay The Horses

How can you negate *No Preemption*?

Deadlock Prevention - Slay The Horses

How can you negate *No Preemption*?

Allow Preemption! Normally done by *multiplexing* resources (how RAM or CPU time is handled).

Not always possible

Negate *Circular Wait*

Deadlock Prevention - Slay The Horses

How can you negate *No Preemption*?

Allow Preemption! Normally done by *multiplexing* resources (how RAM or CPU time is handled).

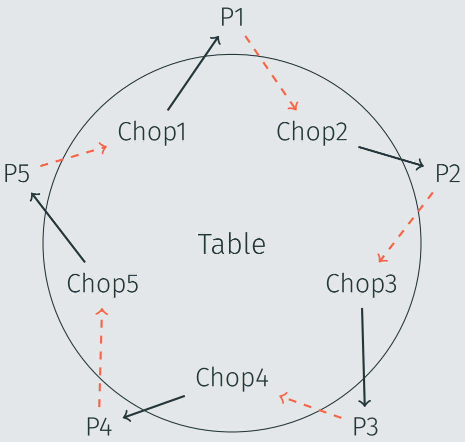
Not always possible

Negate *Circular Wait*

Order resources and only allocate in the *same* order, everywhere.

Commonly used (and also in the current exercise (not anymore :() :)

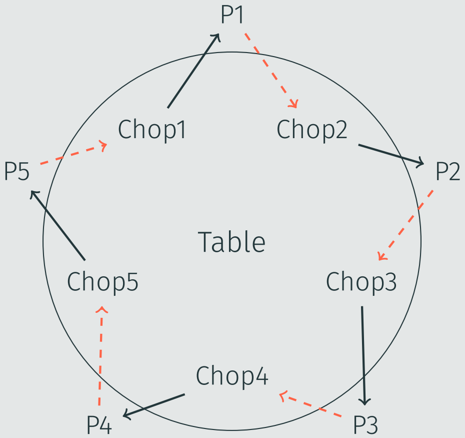
The Problem



What kind of vertices and edges are in this graph?

Dining Philosophers

The Problem

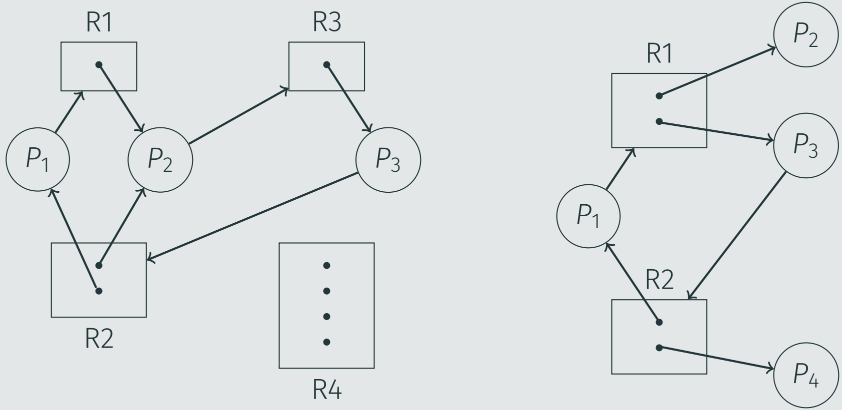


What kind of vertices and edges are in this graph?

How can you detect a deadlock in there?

Resource Allocation Graphs

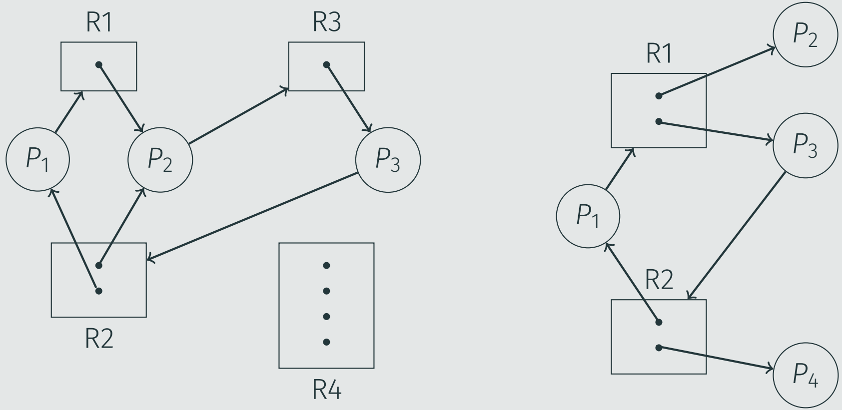
Some examples



Is there a deadlock in one of the graphs?

Resource Allocation Graphs

Some examples



Is there a deadlock in one of the graphs?

Yes, in the left. Right has a cycle *but no deadlock*.

Cycle $\equiv$ Deadlock only holds if you have *one* instance of each resource

Also nice

Deadlock Empire

`https://deadlockempire.github.io/`

Kernel Synchronization

How could you achieve mutual exclusion on Single-Core systems?

How could you achieve mutual exclusion on Single-Core systems?

Masking interrupts!

How could you achieve mutual exclusion on Single-Core systems?

Masking interrupts! Why?

How could you achieve mutual exclusion on Single-Core systems?

Masking interrupts! Why? Only one thread can run at a time, disabling interrupts prevents preemption (and other interrupt handlers)

Kernel Synchronization

How could you achieve mutual exclusion on Single-Core systems?

Masking interrupts! Why? Only one thread can run at a time, disabling interrupts prevents preemption (and other interrupt handlers)

Does this work on Multi-Core systems?

How could you achieve mutual exclusion on Single-Core systems?

Masking interrupts! Why? Only one thread can run at a time, disabling interrupts prevents preemption (and other interrupt handlers)

Does this work on Multi-Core systems?

Nope! Masking only affects the current CPU.
Additionally,

Kernel Synchronization

How could you achieve mutual exclusion on Single-Core systems?

Masking interrupts! Why? Only one thread can run at a time, disabling interrupts prevents preemption (and other interrupt handlers)

Does this work on Multi-Core systems?

Nope! Masking only affects the current CPU.
Additionally, another core could be in the same routine and access the same data

Mutual Exclusion In The Kernel

How would you solve that problem in the kernel?

Mutual Exclusion In The Kernel

How would you solve that problem in the kernel?

Real locks

Mutual Exclusion In The Kernel

How would you solve that problem in the kernel?

Real locks

The Big Kernel Lock™

Have a big lock *for the whole kernel*. Implications?

Mutual Exclusion In The Kernel

How would you solve that problem in the kernel?

Real locks

The Big Kernel Lock™

Have a big lock *for the whole kernel*. Implications? The kernel effectively serialises access

$\Rightarrow$ Can't make use of your processors if you have many syscalls

Mutual Exclusion In The Kernel

How would you solve that problem in the kernel?

Real locks

The Big Kernel Lock™

Have a big lock *for the whole kernel*. Implications? The kernel effectively serialises access

$\Rightarrow$ Can't make use of your processors if you have many syscalls

> *This removes the implementation of the big kernel lock, at last. A lot of people have worked on this in the past, I so the credit for this patch should be with everyone who participated in the hunt.* (Commit message)

Mutual Exclusion In The Kernel

Fine Grained Locking

Mutual Exclusion In The Kernel

Fine Grained Locking

+ Only lock areas that are *relevant* for the *current* task

$\Rightarrow$ Have many small locks

Mutual Exclusion In The Kernel

Fine Grained Locking

- \+ Only lock areas that are *relevant* for the *current* task

$\Rightarrow$ Have many small locks

- \- Complex and error prone

Mutual Exclusion In The Kernel

Fine Grained Locking

+ Only lock areas that are *relevant* for the *current* task

$\Rightarrow$ Have many small locks

- Complex and error prone

Remember Spinlocks and Interrupt handlers?

Without disabling interrupts there is a problem:

Fine Grained Locking

+ Only lock areas that are *relevant* for the *current* task

⇒ Have many small locks

- Complex and error prone

Remember Spinlocks and Interrupt handlers?

Without disabling interrupts there is a problem: *Lockholder Preemption*

Mutual Exclusion In The Kernel

Fine Grained Locking

+ Only lock areas that are *relevant* for the *current* task

$\Rightarrow$ Have many small locks

- Complex and error prone

Remember Spinlocks and Interrupt handlers?

Without disabling interrupts there is a problem: *Lockholder Preemption*

1. Thread enters spinlock

Mutual Exclusion In The Kernel

Fine Grained Locking

+ Only lock areas that are *relevant* for the *current* task

$\Rightarrow$ Have many small locks

- Complex and error prone

Remember Spinlocks and Interrupt handlers?

Without disabling interrupts there is a problem: *Lockholder Preemption*

1. Thread enters spinlock
2. Thread gets pre-empted by an interrupt handler

Mutual Exclusion In The Kernel

Fine Grained Locking

+ Only lock areas that are *relevant* for the *current* task

$\Rightarrow$ Have many small locks

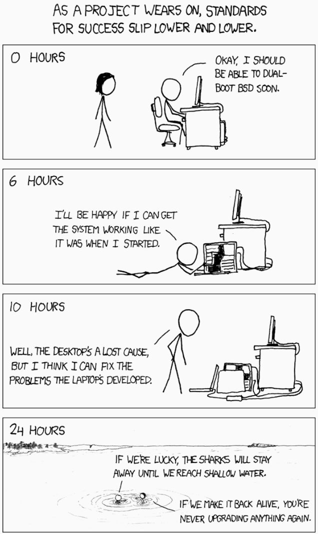
- Complex and error prone

Remember Spinlocks and Interrupt handlers?

Without disabling interrupts there is a problem: *Lockholder Preemption*

1. Thread enters spinlock
2. Thread gets pre-empted by an interrupt handler
3. Interrupt handler needs the same lock $\Rightarrow$ Can never acquire it!

$\Rightarrow$ You might still need to disable interrupts for those



XKCD 349 - Success

F R A G E N?

Bis nächste Woche :)