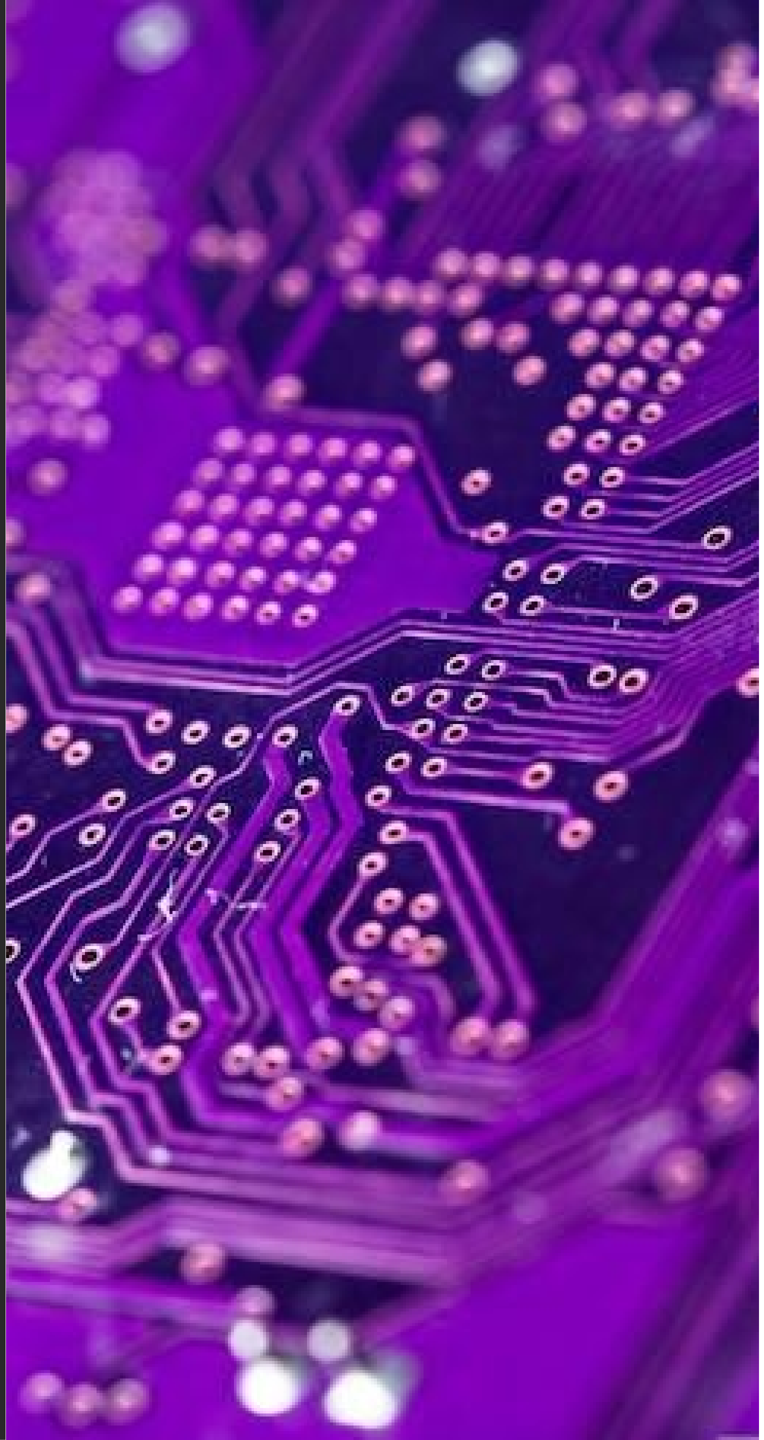
Synchronization primitives

Overview

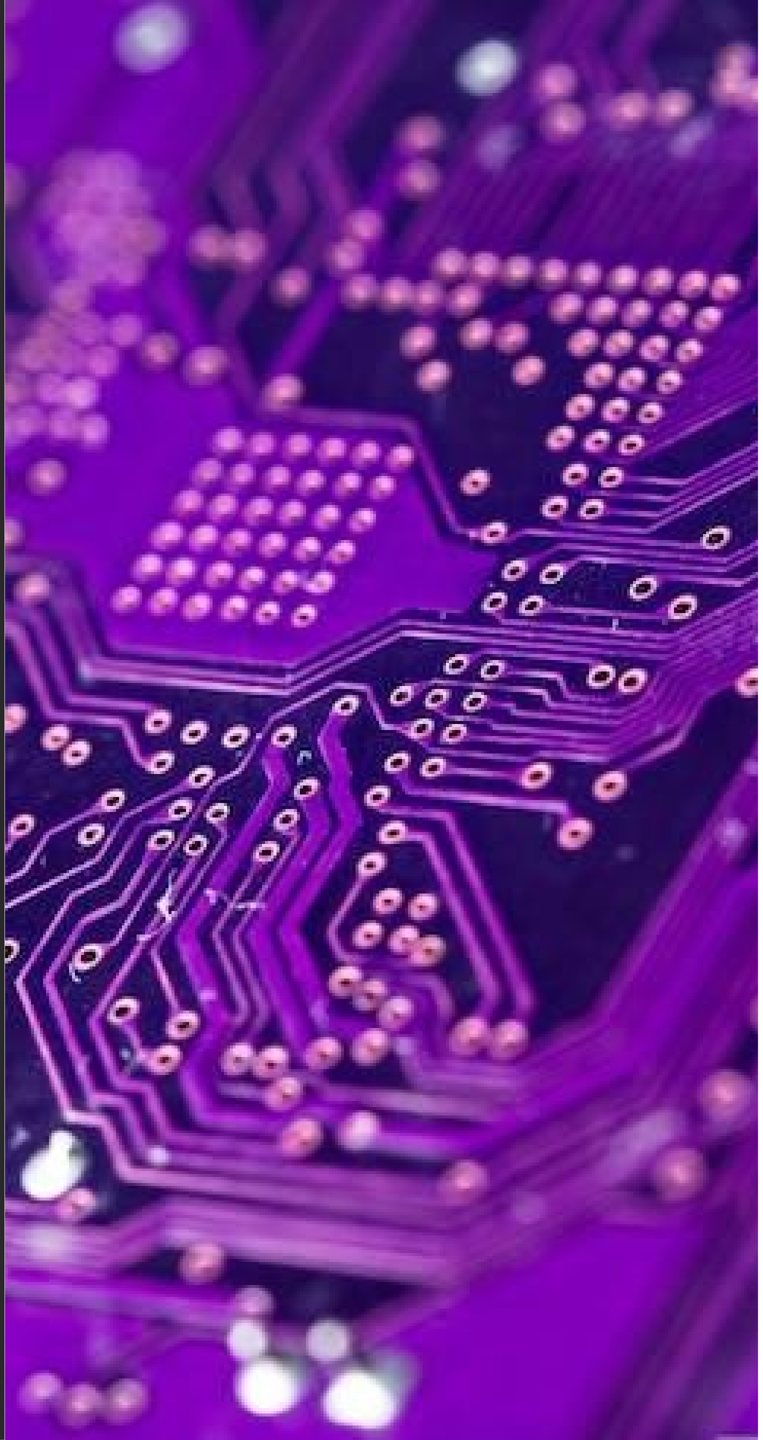
- Introduction
- What is concurrency?
- Race conditions and deadlocks
- Synchronization mechanisms on Linux
- When to use them
- When NOT to use them



Introduction to synchronization

Context: Early days of Computing

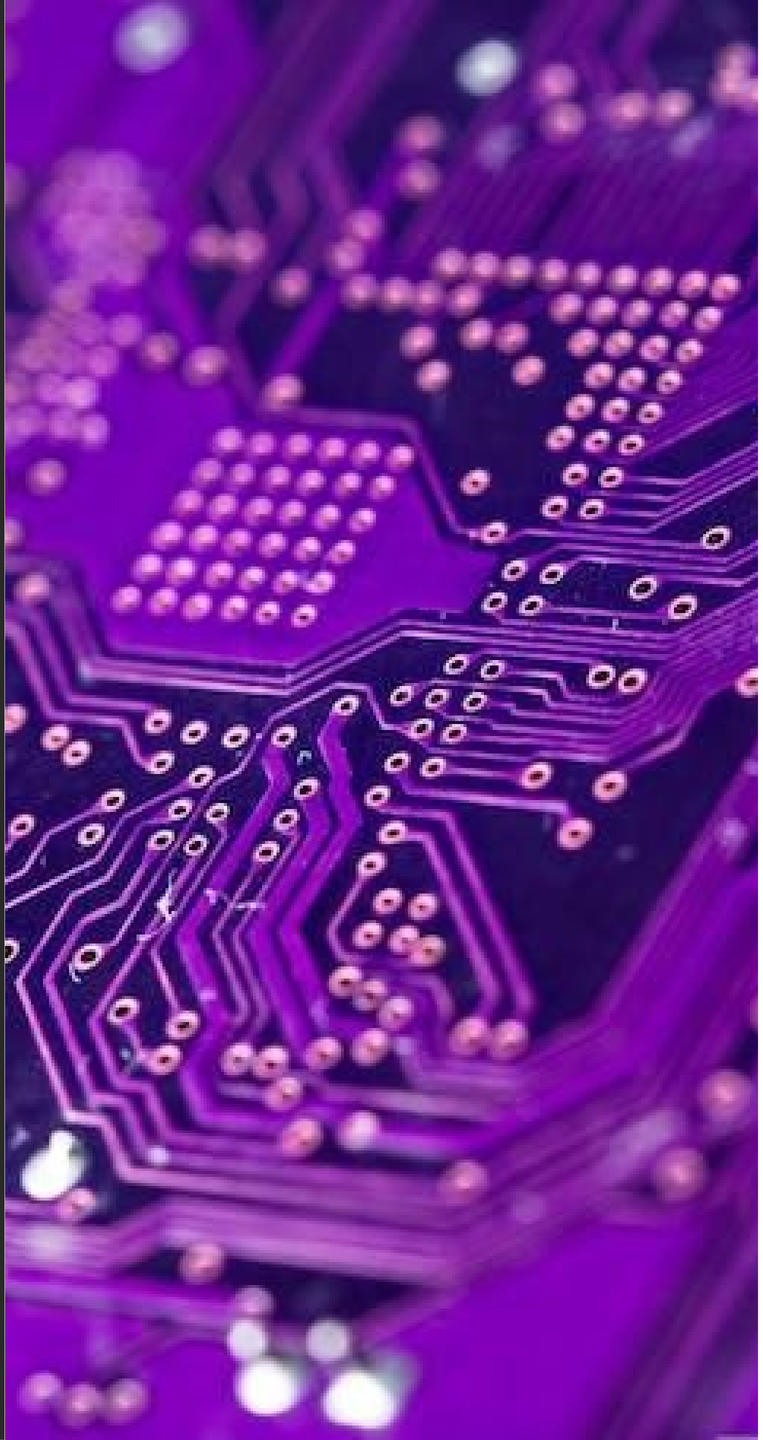
- Programming was easier
  - Computers had a single CPU and a single thread of execution
  - There was a single program running at a time
- We have a complete different scenario now
  - Hundreds of CPUs, cores (or both)
  - CPUs able to run different instructions simultaneously
  - OS’es juggling thousands of processes/threads and users at the same time.



Kernel perspective

Concurrency within the kernel

- Kernel code can also be executed concurrently
    - Even within the same CPU. Concurrency can happen with a single CPU.
- Different levels of concurrency within the kernel which may contain critical sections
    - Interrupt context
    - Preemption
    - Shared Resources



What should be protected against concurrent access?

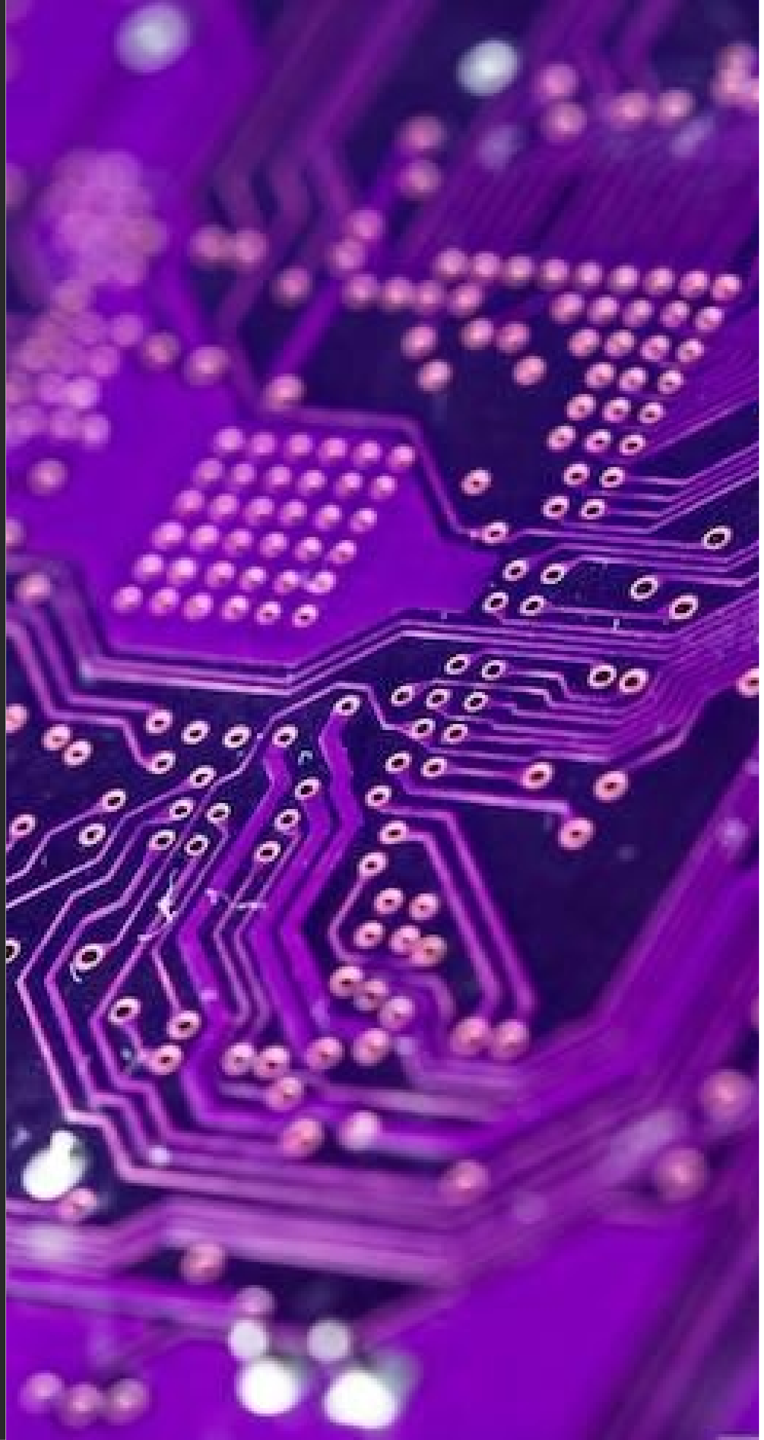
Locks exist to protect data, not code

- Always keep that in mind…
  - Locks must be used to protect data structure from concurrent access, not to protect your code.
- Look at a data structure and think what should be protected there.
- Code-centered locking design always end up in disasters sometime in the future.
  - Search for how long it took to get rid of kernel’s BKL

https://kernelnewbies.org/BigKernelLock

Concurrency within the kernel

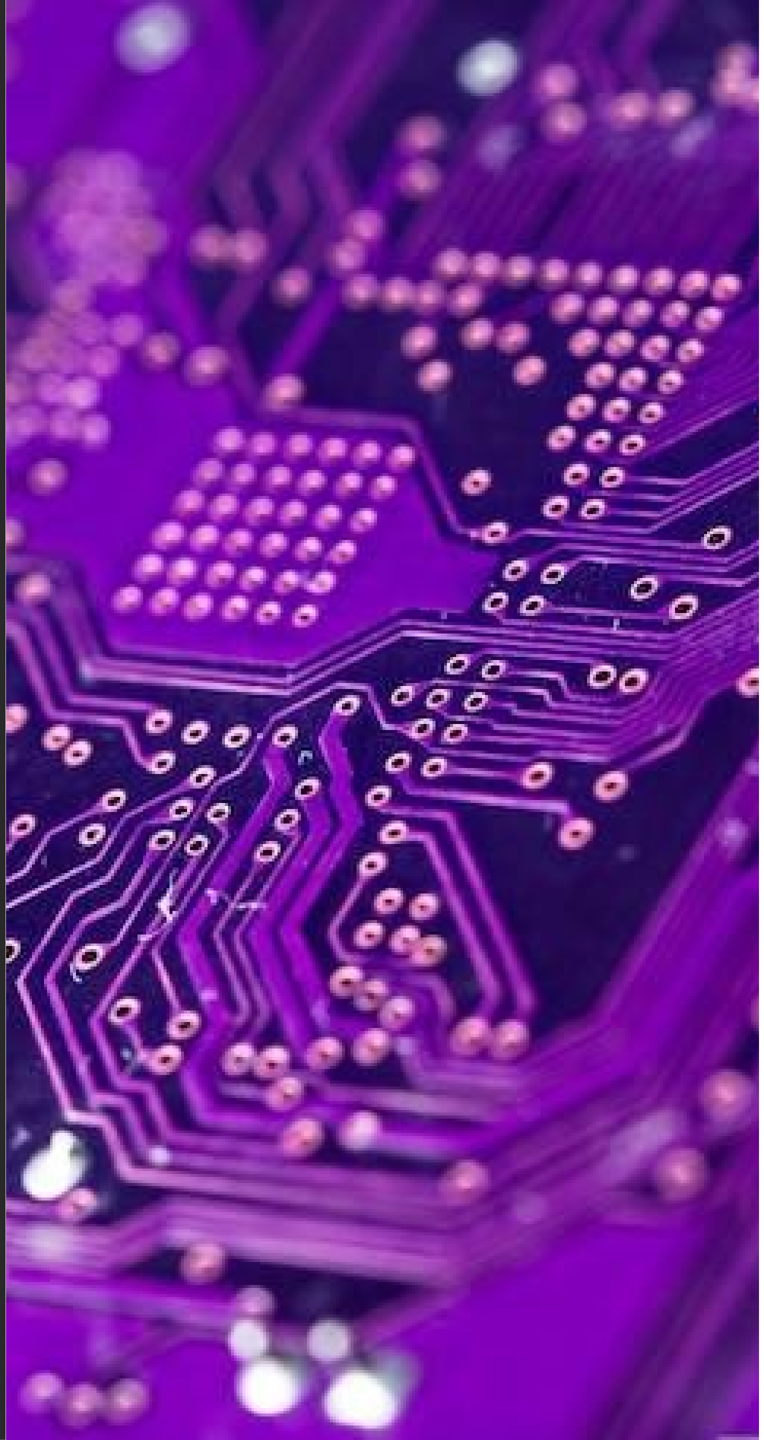
- Any data that can be accessed by more than one thread
  - Keep in mind that even a single CPU can concurrently access the same code (thanks to preemption and interrupts)
- Ask yourself
  - If the code sleeps while accessing data, can the new scheduled code access the same data?
  - If the code gets interrupted by an IRQ… Can the IRQ handler access the very same data?



Race conditions

The most annoying of all bugs

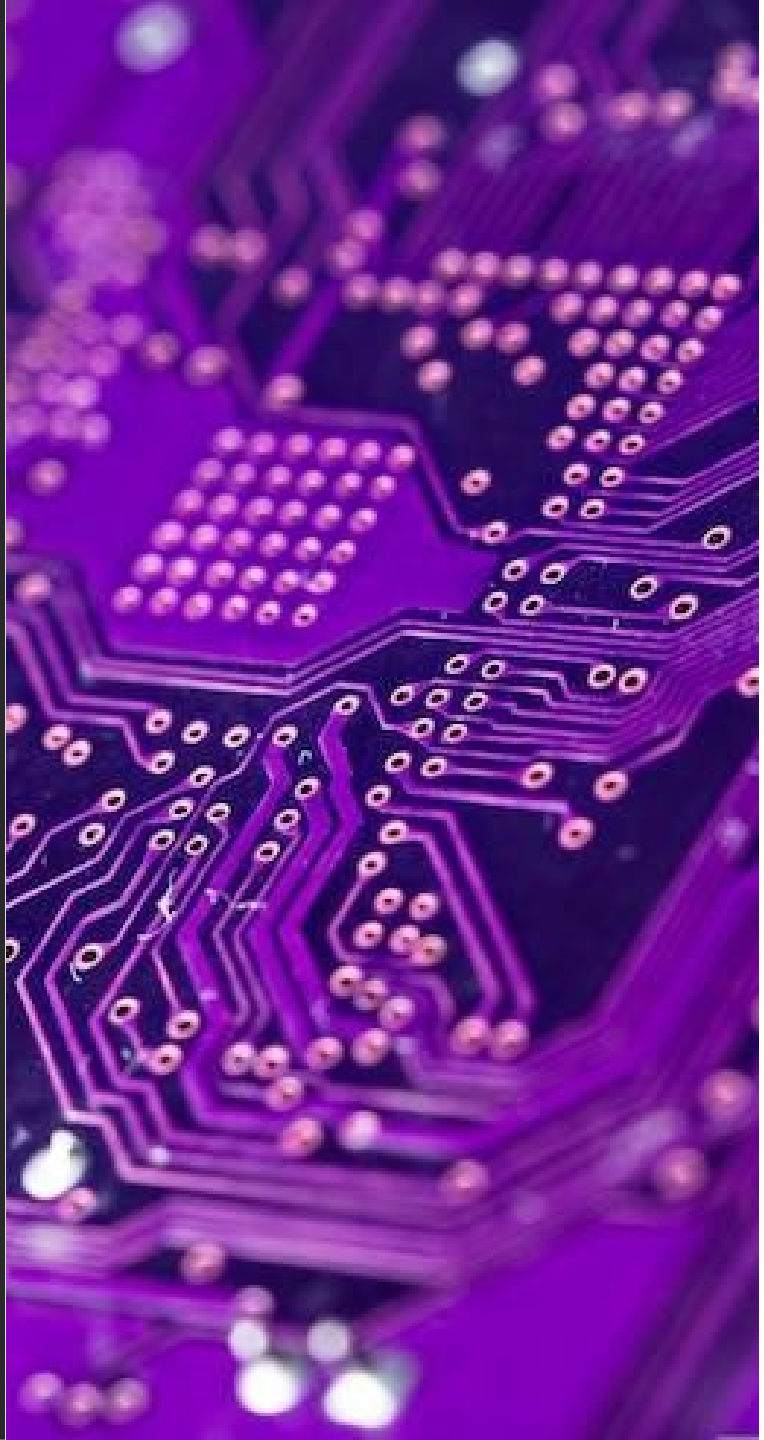
- Caused when two or more threads concurrently access the same data structure and at least one is modifying it.
- Race conditions might be extremely difficult to find
- They are hard to reproduce, as they are time dependent.
  - More often than not, adding instrumentation will hide the bug



Deadlocks

What are deadlocks?

- One or more threads attempt to lock a specific resource that is already held
  - For some reason (that we shall see), this held resource can never be released by the current holder.
  - The waiting thread will never make progress



Lock contention

Resources serialization

- Serialization caused by locking, may have a significant impact on performance.
- Consider lock “granularity”
  - How much data does a specific lock protect?
  - Coarse locks VS. Fine grained locks

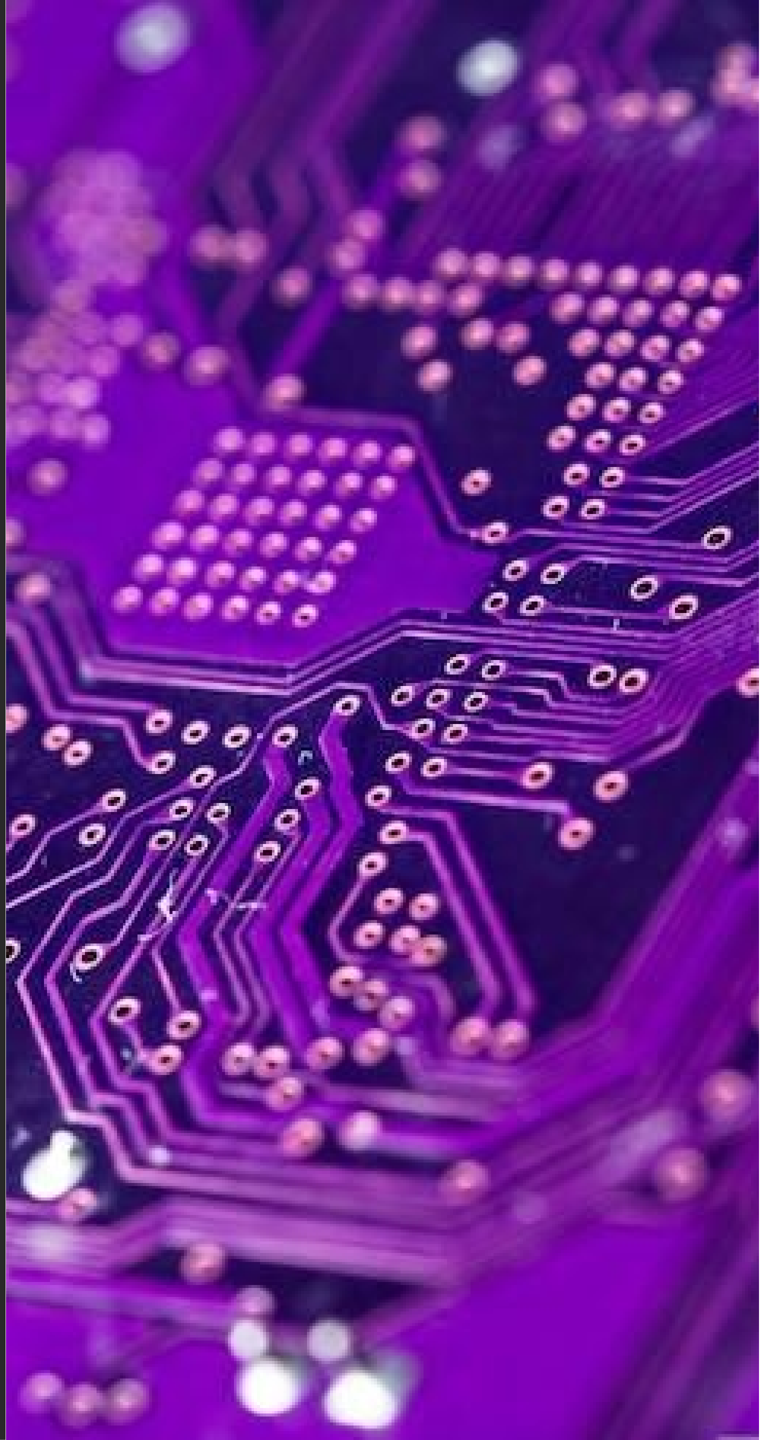
task_struct as example

- task_struct
  - What would happen if the whole task_struct was protected by a single lock?
  - How many locks are used within the task struct?

Careless scalability

- Fine grained locking reduces contention, but…
  - It does also add a lot of overhead.
  - Adds complexity
- Consider what kind of system that software will run.
- Extra locking overhead may kill small systems performance

Red Hat



Instruction ordering and memory barriers

Instruction ordering

- Compilers and processors are free to reorder instructions
  - Including load and store memory instructions
- Because sometimes instructions order are important, we must be able to control it.
  - We must be able to guarantee that a specific read happens before another, or
  - That a write appears before any subsequent read
- Compilers and CPUs able to reorder operations, provide machine instructions to enforce ordering requirements, aka **barriers**

Ordering example

- Let’s get a couple instructions:

  a=1;

  b=2;

- Nothing prevents the compiler or the CPU to process the second instruction first.
  - Compiler may statically reorder it within the object code
  - The CPU however, could dynamically reorder it by fetching and dispatching them in different order.

When reordering may happen

- When there is no clear relationship between both instructions.
- These instructions would not be reordered:
  - a = 1;
  - b = a;
- The compiler and the CPU though, doesn’t know about the code in different contexts.
- It’s our job to tell both about the specific ordering.

Architecture dependency, yet again

- Memory barriers and compiler directives are architecture dependent
  - Intel as example, never performs out-of-order store operations.
- We must not make any assumptions on which hardware our code will be running.
  - Unless of course, you are writing architecture-specific code.
- But.... There is yet another problem...

Compiler optimizations

- The following code:

while (tmp = a)

call_function(tmp);

- If the compiler can prove the variable ‘a’ is always zero, it may optimize to:

do {} while (0);

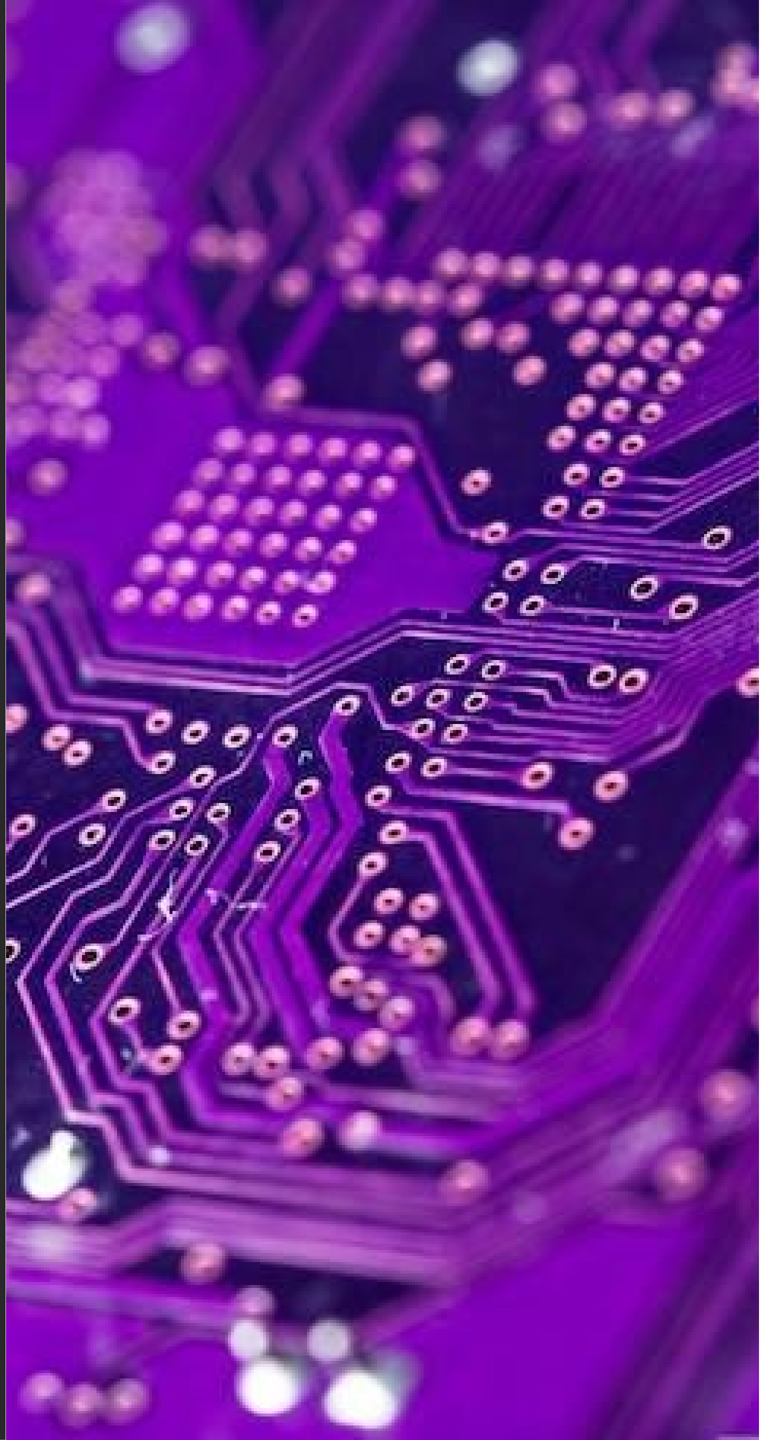
- Giving the compiler is not context aware.
  - What would happen if “a’ variable is shared and is actually updated from a different context?

Compiler optimizations #2

- If you are lucky, you will likely spend hours trying to understand why the kernel is crashing.
  - If you are not, you'll spend months trying to understand why it is misbehaving once in a while.
- And in such cases, we must explicitly tell the compiler that it should read variable 'a' every loop interaction.

What to take away from all of this?

- Be aware not only of how the CPUs will execute the code, but also
- How the compiler will treat such code.
  - Which optimizations it may do to the code and what consequences it will have.
- This is a place where learning ASM really pay dividends
  - Understanding how the generated assembly relates to the code you wrote is a great way to spot any unwanted optimizations.



Synchronization within Linux

CPU memory barriers

- Macros used to manipulate CPU memory barriers (Run-time barriers)
  - rmb() - Read memory barrier
  - wmb() - Write memory barrier
  - mb() - RW memory barrier
- These macros guarantee ordering of load/store **instructions**
  - Any load/store instruction coded before the barrier, will be executed before any instruction coded after the barrier.

Compiler barriers

- barrier()
  - Explicitly tell the compiler to not move memory accesses across the barrier, enforcing memory access ordering.
- READ_ONCE() and WRITE_ONCE()
  - It tells the compiler it must re-read/re-write the variable each time it is called.
  - while (tmp = **READ_ONCE(a)**) { do_something(tmp) };
- Please don't use ***volatile*** type class (with some rare exceptions)
  - It is rarely acceptable in Linux kernel and its use is almost never correct.

https://www.kernel.org/doc/html/latest/process/volatile-considered-harmful.html

Atomic operations

- A collection of instructions that execute atomically
- Architecture specific implementation
- Linux provides two types of atomic operations
  - Integer-based
  - Bitwise
- Linux provides a special data type for atomic operations
  - **atomic_t**

Red Hat

Atomic operations #2

- Fastest synchronization method, introducing no overhead compared to locking.
- No need to implement locking to protect small portions of data, like integers or single-bit changes
- Many locking primitives end up relying on atomic operations
- Usually are implemented as inline functions with inline assembly.
- It’s a no-brainer for some architectures

atomic_t data type

- Having a specific type guarantees type check, so atomic functions only accept atomic_t types.
- Prevents somebody using atomic data types with non-atomic functions.
- The atomic_t, prevents the compiler to do some ‘clever optimizations’ on these types.
- Prevent ourselves to use atomic types on non atomic operations
    - atomic_t VAR = 10;

64-bit atomic operations

- atomic_t variables are ALWAYS 32 bits
- Another type - atomic64_t can be used for 64-bit atomic operations
- Most operations available on 32-bit atomics are also provided in their 64-bit form.
- atomic64_t **IS NOT PORTABLE**
  - Because this, it’s mostly used on architecture-specific code.

Atomic bitwise operations

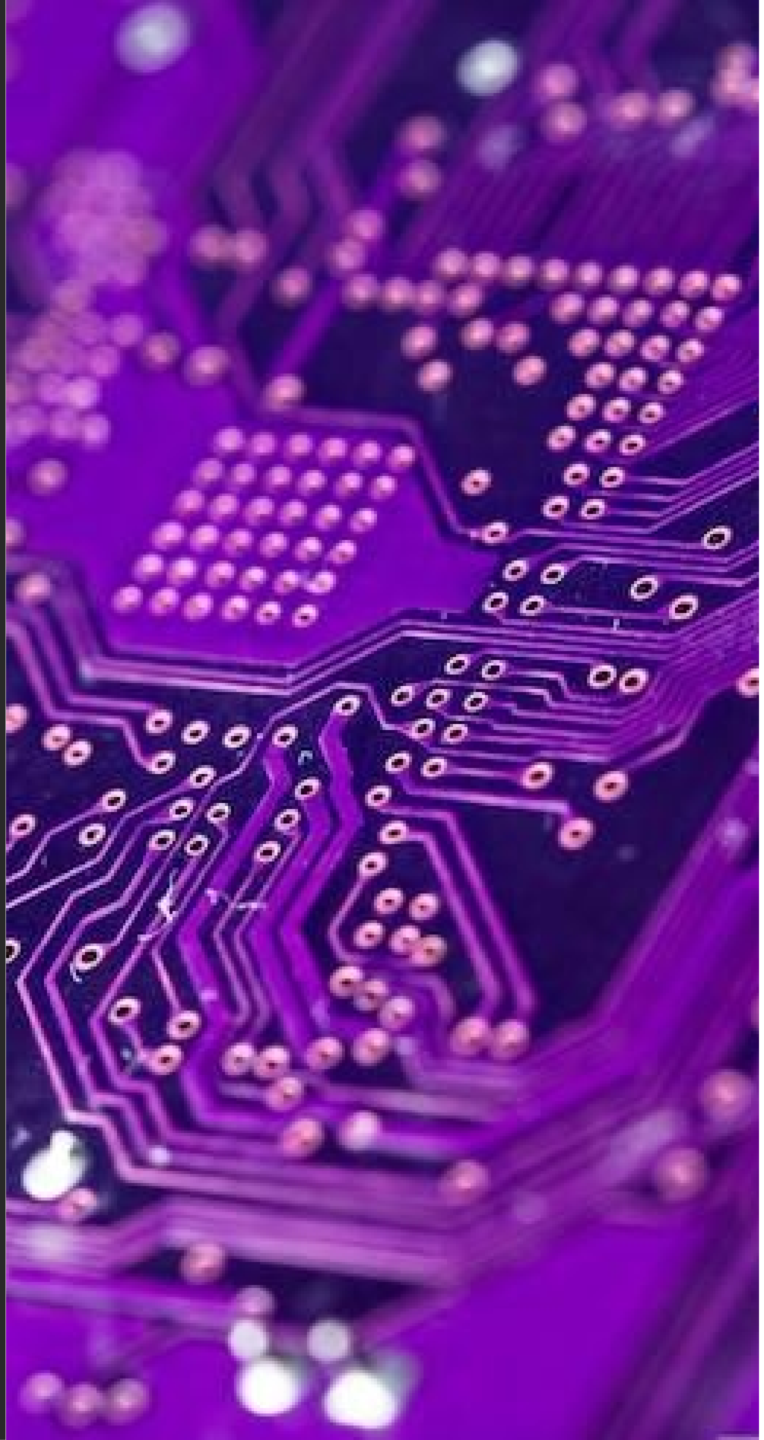
- Atomic single-bit data manipulation
- Also architecture-specific
- Operations are performed on **word-size** generic memory addresses
  - We simply pass to those operations a bit number and a memory address. (0 being the least significant bit).
- Linux provides a few functions to search for the first bit set/unset in a data type
  - find_first_bit() - find_first_zero_bit

Show time

- Atomic operations
- bitwise operations
- __ffs() and ffz()

Per-CPU data API

- Allocation
  - DEFINE_PER_CPU(), DECLARE_PER_CPU() - compile time
  - alloc_percpu(), __aloc_percpu, free_percpu() - runtime
- Access the variables:
  - get_gpu_var(),put_cpu_var() - Also disable/enable preemption
- Accessing other CPU's data:
  - per_cpu() - This doesn't handle preemption enable/disable
    - By accessing another CPU's data, synchronization is still required



When simplicity is not enough…

SpinLocks

- Most common lock used in Linux
- Can be held by a SINGLE thread of execution
- A thread attempting to acquire an already contended lock will “spin” waiting the lock to become available.
- Only lock type allowed in interrupt context
- Architecture and SMP dependent
- Provide “special APIs for interrupt context” - irqsave/irqrestore
    - disable local interrupts

Read-Writer spinlocks

- Lock acquisition can be split into Readers and Writers
- Reading doesn’t require mutual exclusion
- Splitting the usage of data structures between reader and writer paths (producer/consumer), we allow concurrent read access.
- Readers can’t be upgraded
- RW spinlocks favor readers over writers
    - Be careful to not starve the writers

Semaphores

- “Sleeping locks” - Once a task attempts to acquire an already locked semaphore, the task is put to sleep on a wait queue
- When the lock is released, the next task in the list will be awaken and then will grab the lock.
- Better CPU utilization but greater overhead
- Better suited for locks held for long periods of time
- Can’t be used in interrupt context
- Allow simultaneous holders

Reader-writer semaphores

- Semaphores also provide a reader-writer version
- RW semaphores are **ALWAYS** mutual exclusion writers.
  - Only a single writer at a time
  - But can have multiple readers.
- RW semaphores only allow waiters to be in **UNINTERRUPTIBLE_SLEEP**
- As with RW spinlocks, if you have no clear separation between read and write paths, don't use them

Mutexes

- Provides mutual exclusion and works similarly to a binary semaphore
- Provides a simpler interface and less overhead
- Impose several constraints on its usage, making it simpler to use
  - Only one task can hold a mutex at a time
  - Mutexes must be locked/unlocked in the same context
    - This is one specific usage for semaphores
  - Not allowed in interrupt context either
- Mutexes must be managed only through the APIs.

Red Hat

Mutexes #2

- Mutexes have a special debugging mode
  - Big help to look for constraints violations
- Semaphore vs. Mutexes
  - Similar, but mutexes are faster and with less overhead
  - Mutexes are simpler to use, so prefer them in lieu of semaphores.
    - Unless one of its constraints prevents you from using it.
- Spinlocks vs Mutexes - same semaphores rules applies

Completion variables

- Easy way to synchronize two tasks within kernel when:
  - one task needs to signal another that an event occurred.
- One task waits for the completion variable while another does some work.
  - Once the work is completed, the task uses the completion variable to wake up the waiting task(s)
- Similar to semaphores, but provides a simpler solution to the same problem.

Sequential locks

- Mechanism to read and write shared data
- Lockless readers
  - If inconsistency is found, the reader should retry reading the data
- Works great for data that is rarely written
- It works by maintaining a sequence counter, updated when the data in question is written to

Red Hat

Sequential locks - Writers

- Increment the sequence counter at the start and end of the critical section.
  - After starting the critical section, the seqcount is odd, indicating to readers there is an update in progress
  - Once the write is finished, the seqcount becomes even again, letting readers know no more write is happening.

Sequential locks - Readers

- The sequence number is read before any attempt to read the data
  - If the seqcount is even, the reader knows no write is happening.
- The reader must make a copy of the data to somewhere outside the critical section.
- At the end, the reader must read the seqcount again, and compare with the initial value.
  - If the count is the same, we know that the data is consistent.
  - If not, we need to retry the read

Red Hat

Sequential locks - serialization

- While readers are lockless, the same isn’t true for writers.
  - We must protect against multiple writers somehow
  - The writers must also be non-preemptible
- The seqlock api provides a few mechanisms to make this easier.
- seqlocks can be used in irq contexts, as long we properly handle interrupts and preemption disabling, and use the correct locks to protect against mutual writers.

Sequential locks - conclusion

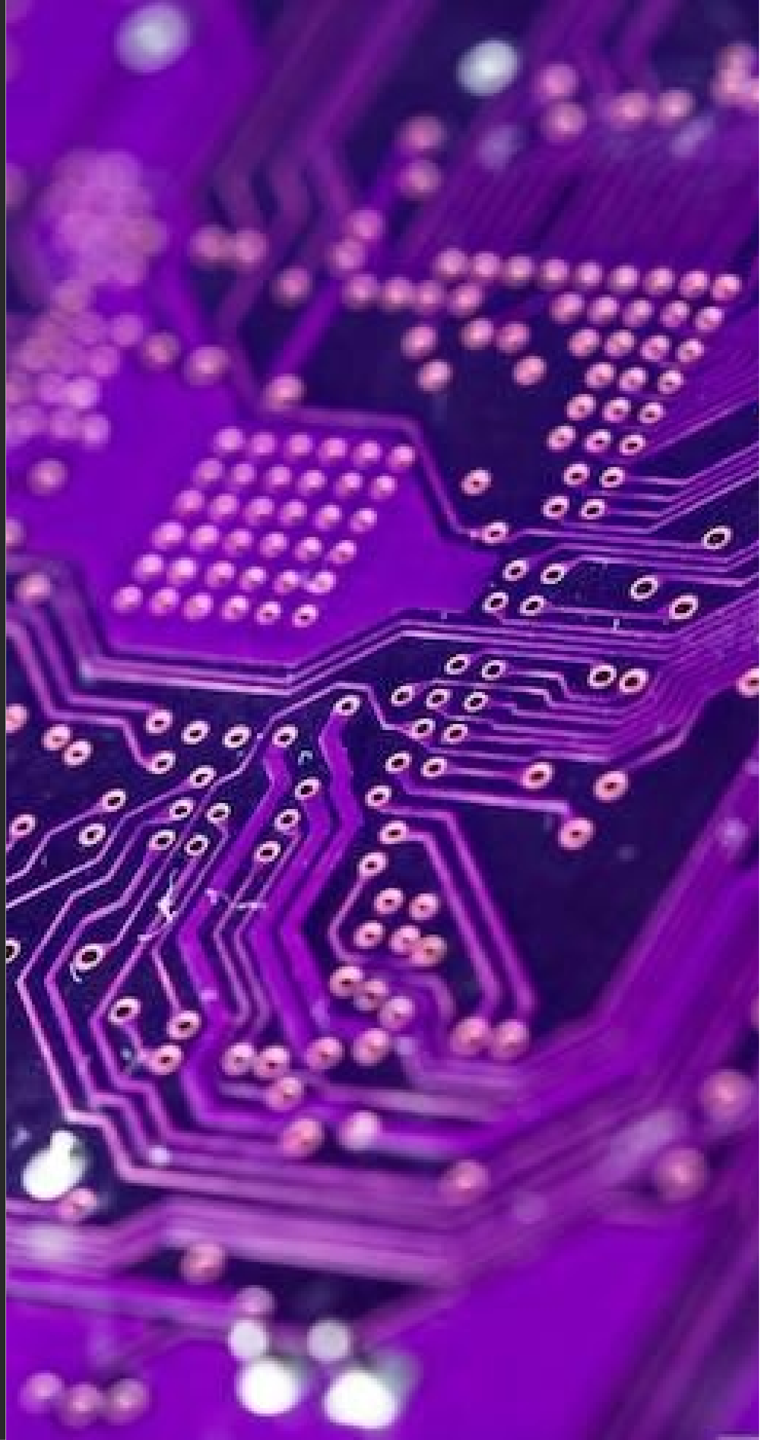
- Seq locks provide a scalable and lightweight lock mechanism for scenarios with read-most data.
- Writers are prioritized, so we must ensure we have few of them, otherwise, readers will keep retrying indefinitely.

Preemption

- Kernel is preemptive
  - A task in kernel space can stop running any time in lieu of a higher priority kernel task.
  - This new task, can actually access the same critical section being accessed by the preempted task.
- Spinlocks already solve this problem as they mark such regions non-preemptive.
  - So, why we need mechanisms to explicitly disable preemption?

Preemption #2

- Some situations require no locks, and spinlocks would add unneeded overhead.
- per-CPU data for example
  - Can be accessed only by a single CPU, so, no lock is needed.
  - But a task can be preempted and another scheduled on the same cpu
- We solve this problem by simply disabling preemption on that CPU
- Preemptions can be nested.



Read-Copy-Update or simply RCU

What is the RCU mechanism?

- Yet another synchronization mechanism
- Many readers and many *writers* (not really but close to it) are allowed to proceed concurrently
- Split updates into “removal” and “reclamation phases.
- RCUs maintain multiple ‘versions’ of the data, and guarantee they are not freed until all readers are done.

RCU reader side

- Reader implementation is really simple
  - No need to acquire any locks
  - No atomic instructions needed
  - No shared memory writes needed
- By not needing any of these expensive operations, RCU is extremely fast on read-mostly scenarios
- No locks == No deadlocks

RCU reader constraints

- As with spinlocks:
  - RCU readers can’t block
  - They can’t context switch
- Only dynamically allocated data can be protected
  - RCU works on the data address pointers

RCU writer side

- Split into “removal” and “reclamation” phases.
- When a task wants to update RCU protected data, it must (removal):
  - Read the data
  - Make a copy of the data and update it
  - Update the data pointer to point to this new updated version
- Free the old data at some point (reclamation)
  - Must not start until no readers hold a reference to it anymore

RCU writer side #2

- Writers still need to synchronize with each other somehow
  - Like using atomic operations, barriers, spinlocks(), etc
  - The data pointers update still must be atomic
- Enforcing memory access order is still required
  - We must ensure the new pointers are seen only the data has been modified

RCU writer side #3

- We are not done yet:
  - Old data, may still be being referenced
  - We must free the old data at some point
  - And here comes the beauty of RCUs
  - synchronize_rcu() / call_rcu()

Tracking usage and freeing old data

- According to RCU constraints, all readers must “unlock” the data before any context switch
  - no blocking, no user-mode switch, no idle loop
- So, we know that:
  - Once a CPU has gone through a quiescent state, that specific CPU is no longer within the RCU protected region.
- Once all CPUs have gone through a quiescent state, the old data can safely be freed.

RCU usage example

- Lockless iteration over system's processes
  - **task_struct**->tasks field is used to link all the processes
    - can be traversed in parallel to any updates to the list

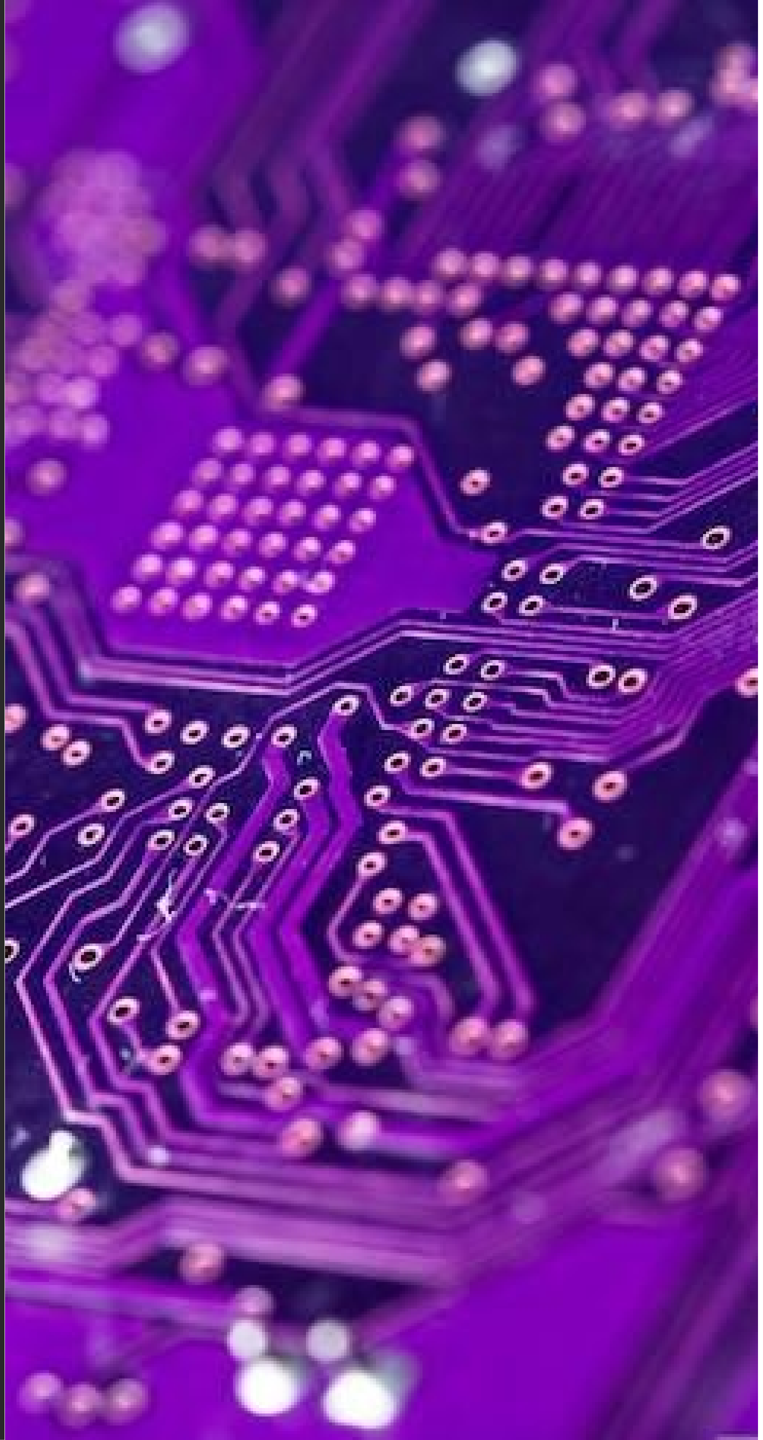
```
rcu_read_lock();
	for_each_process(p) {
		/* do something with p */
	}
rcu_read_unlock();
```

```
write_lock(&tasklist_lock);
list_del_rcu(&p->tasks);
write_unlock(&tasklist_lock);
call_rcu(&p->rcu, delayed_put_task_struct);
```

RCU’s grace period

- The time between the pointer to a data object is replaced, and the stale data is freed, is called the “grace period”
- The writers call to ***call_rcu()*** function which queue a RCU callback for invocation when this grace period expires
  - We can synchronously free some data, by explicitly waiting for a grace period to expire, with ***synchronize_rcu()*** which end up calling call_rcu().
- The RCU mechanism is responsible for controlling the grace periods, and it does so by polling the CPUs

Red Hat



What next? Lockdep, Preemptible RCUs, RT-kernel

Thank you

Red Hat is the world’s leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make
Red Hat a trusted adviser to the Fortune 500.

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

twitter.com/RedHat

Red Hat