

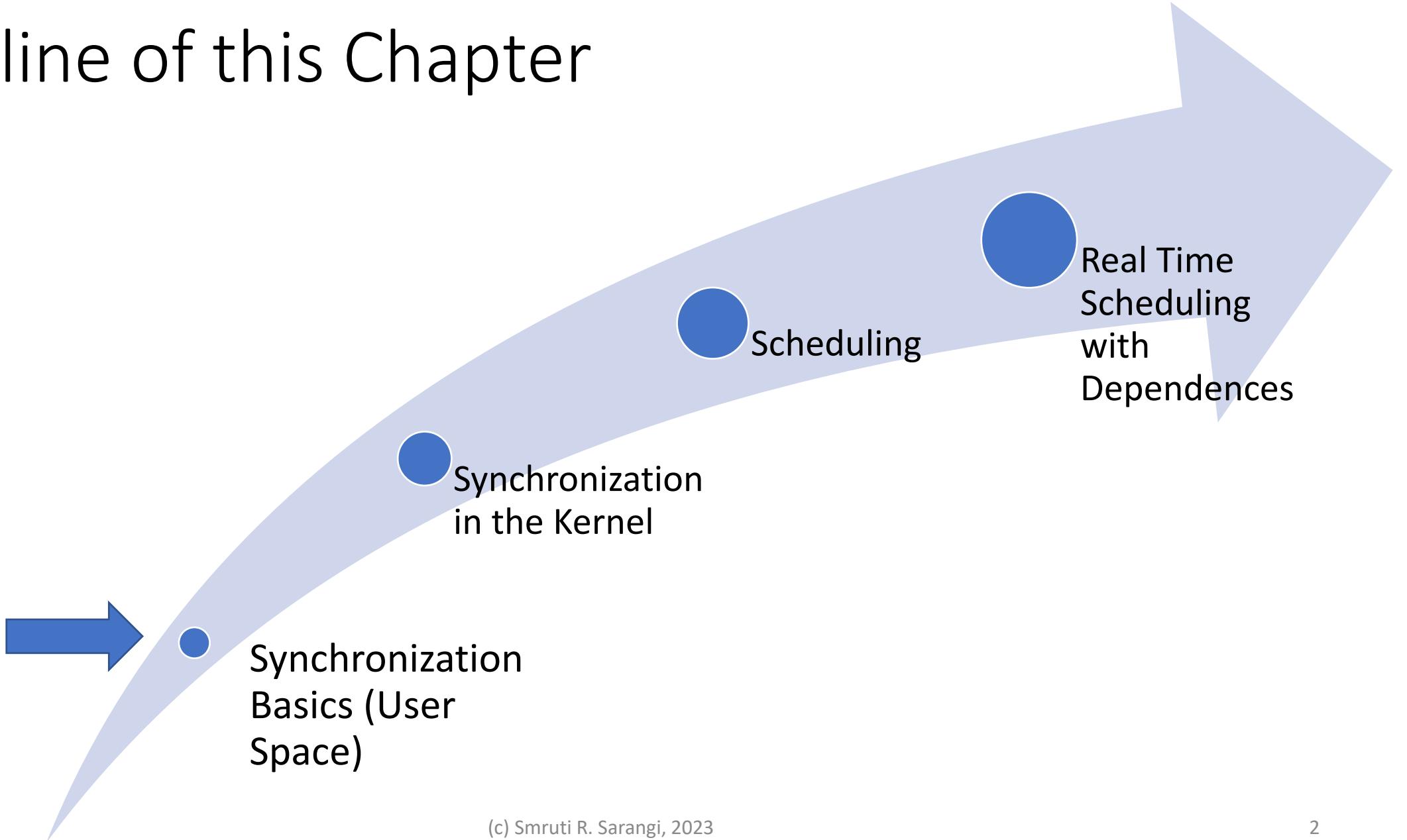


Chapter 5:
Synchronization
and Scheduling

Smruti R Sarangi

IIT Delhi

Outline of this Chapter



Consider a Multicore CPU

Do something as simple as



count++

Assume there are multiple threads.



Will this work correctly?

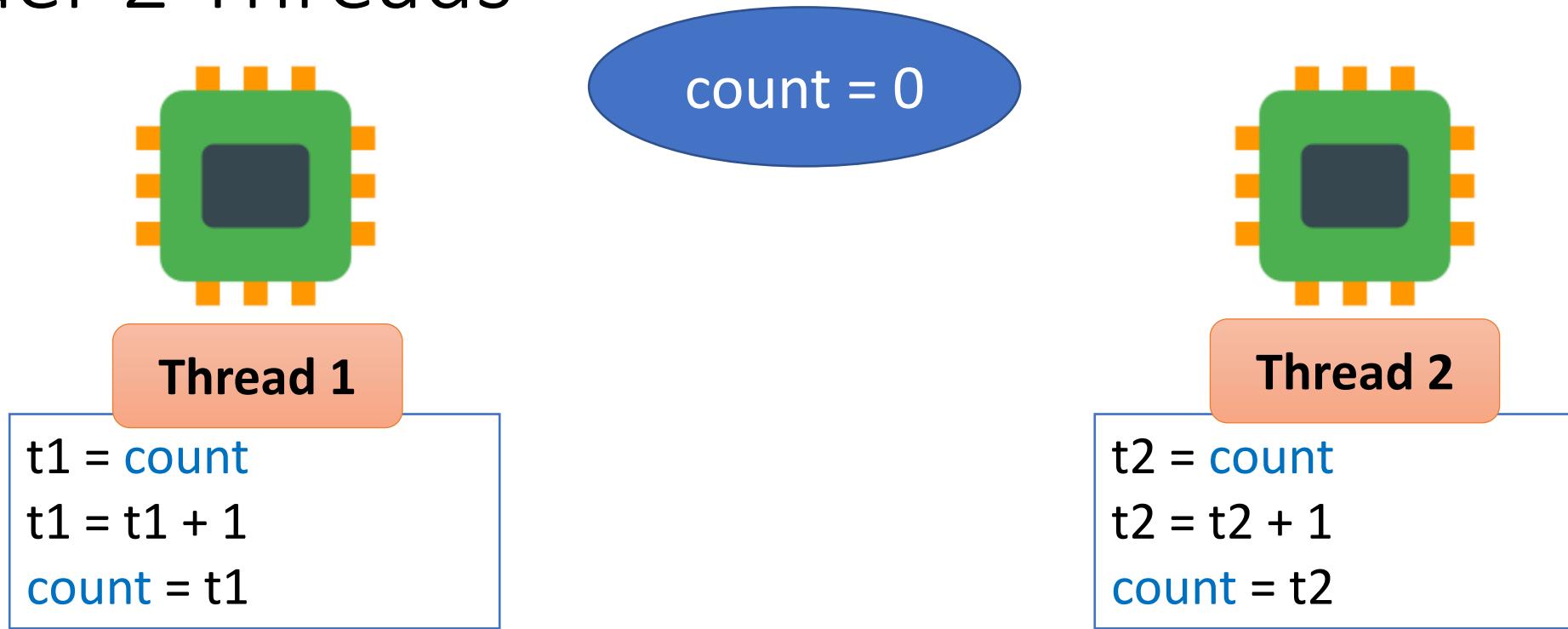


No!

```
t1 = count  
t1 = t1 + 1  
count = t1
```

Each line corresponds to one line of assembly code. `count` is a global variable stored in memory and `t1` is a register.

Consider 2 Threads

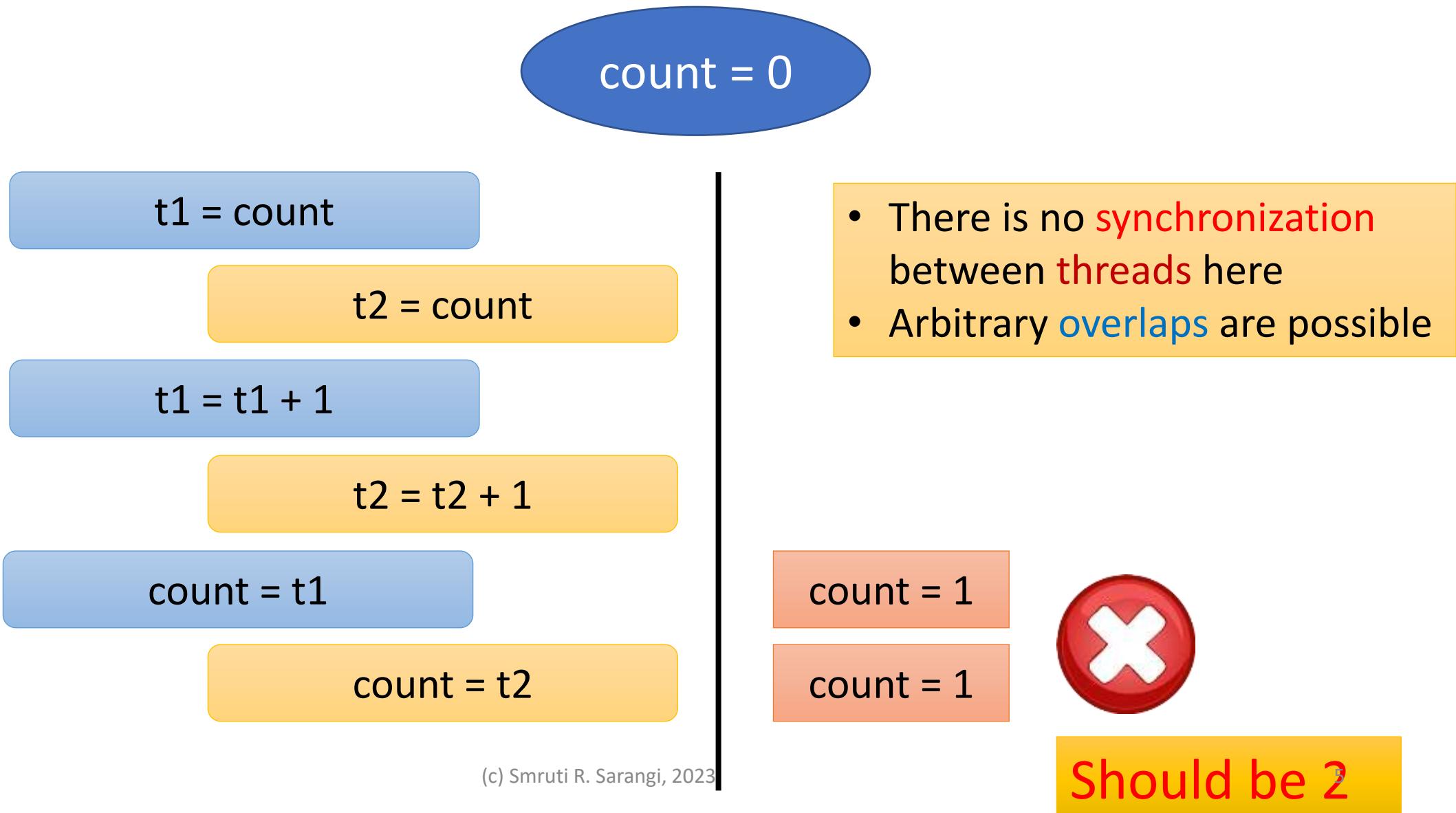


The delay between any two **instructions** can be **indefinite**



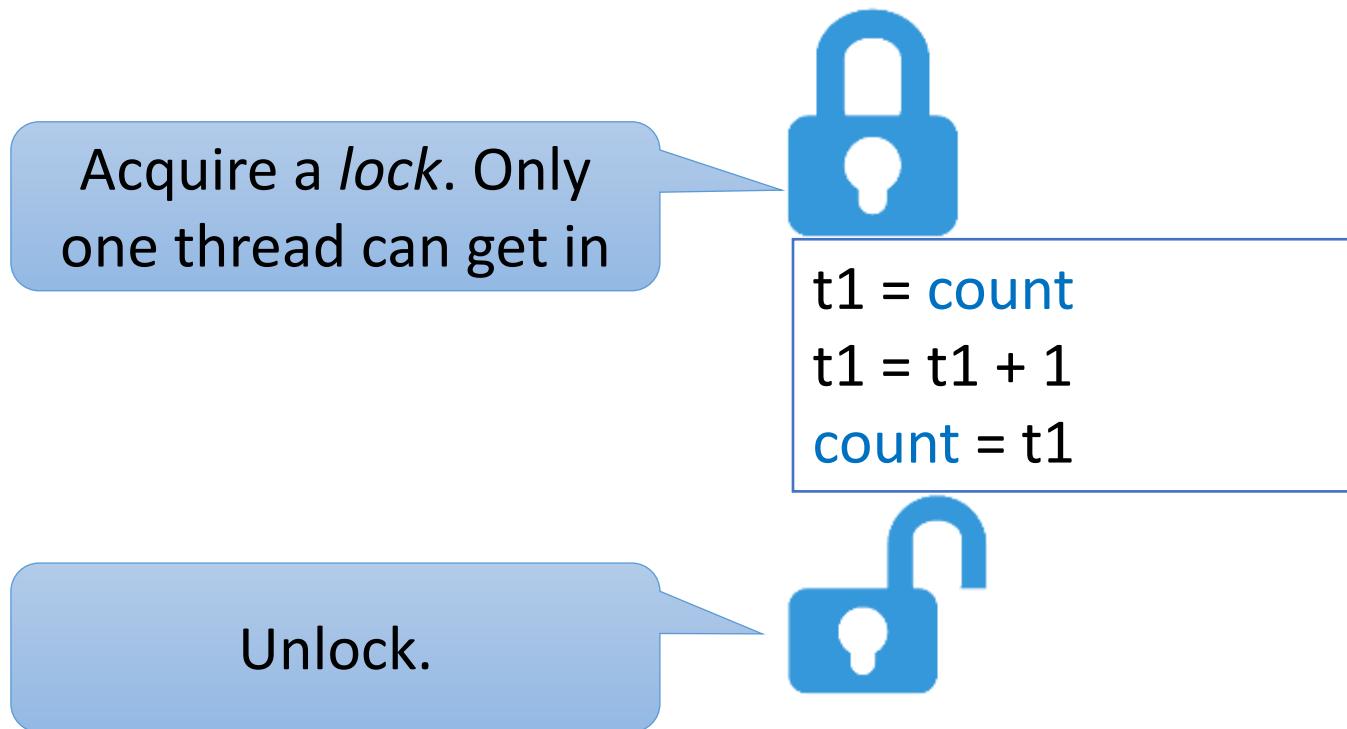
What can this **cause**?

Problematic Executions

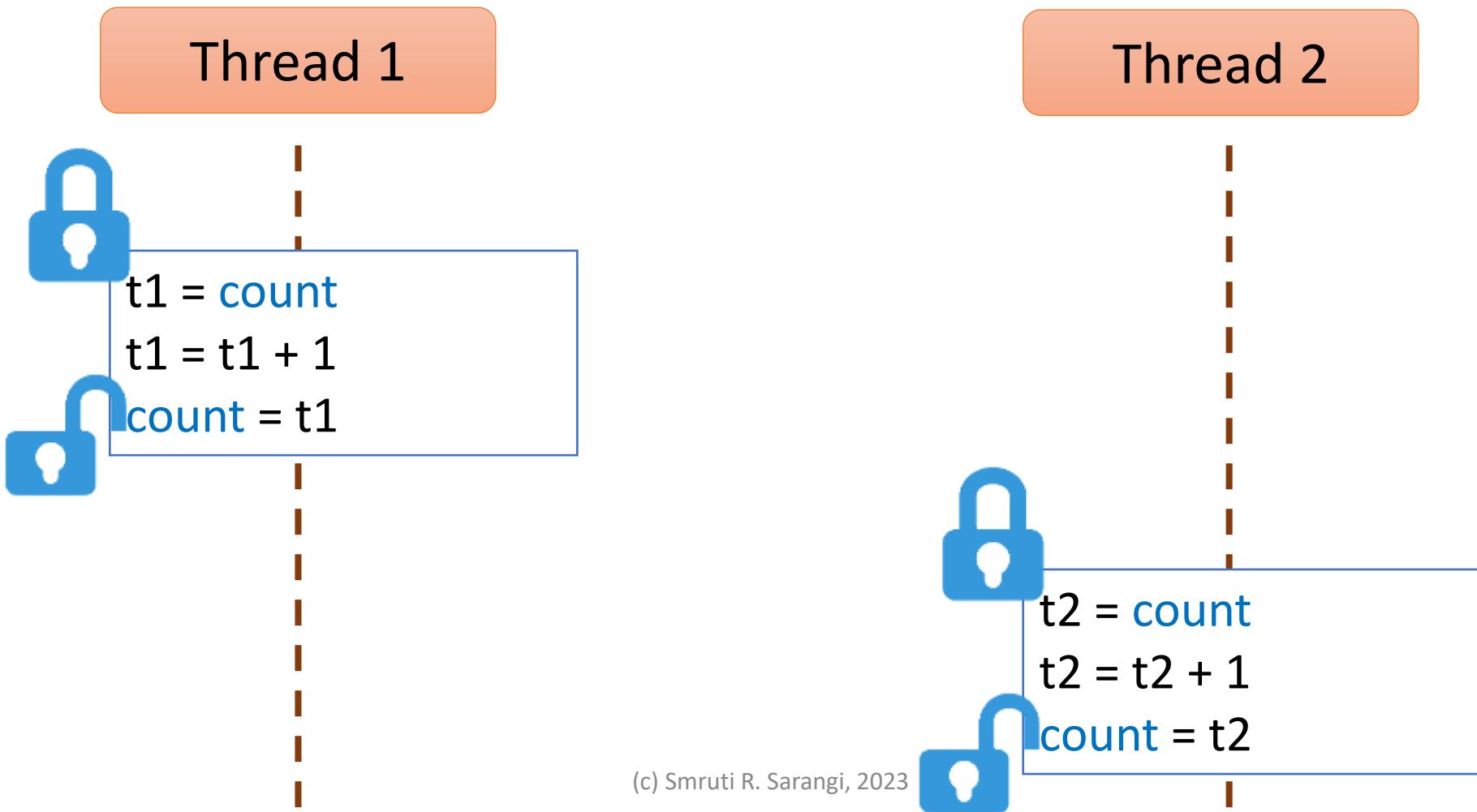


Concurrently incrementing *count* is hard

- What is the **main** issue?
- The **instruction** sequences are **overlapping**.
- We need **synchronization**.



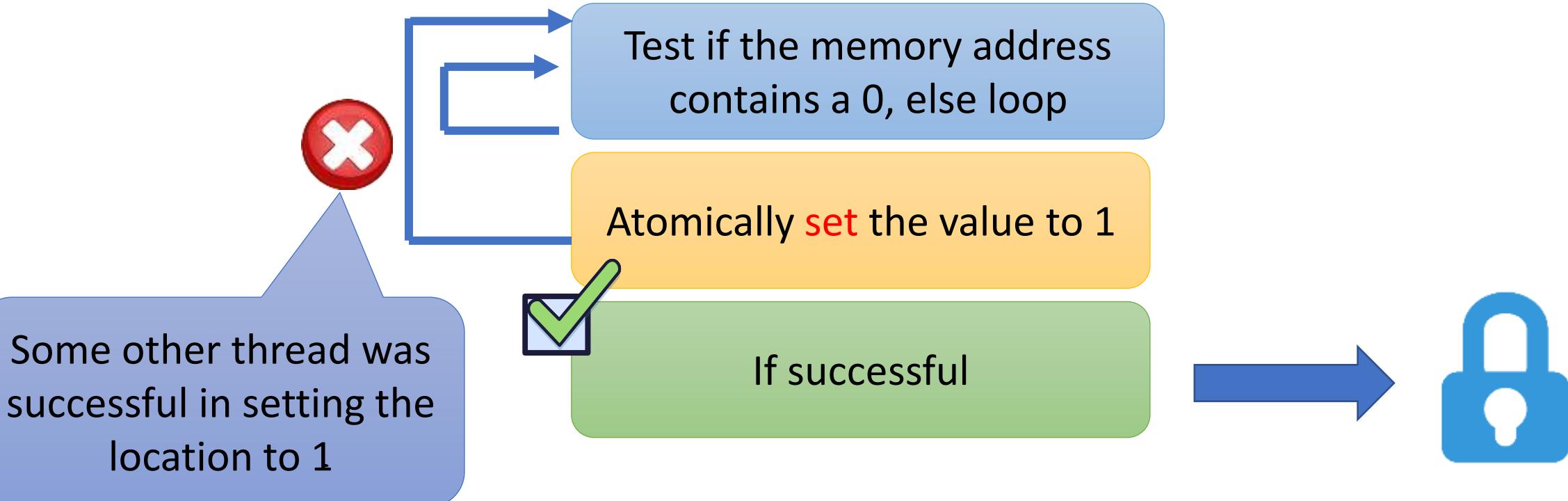
What will the code look like with locks ...



What is a lock anyway?

- Implementation of a **lock**

You lock a memory address A



Unlocking

Set the contents of the location to 1



Critical section



Notion of Data Races

- Why does this piece of **code** not work in the first place.
- We have a **data race**

Conflicting accesses by two threads

- At least one of the threads writes to the shared variable

Concurrent accesses

- Both the accesses are happening roughly at the **same time**. There is a more precise **definition** that uses the notion of **happens-before** relationships and lock-unlock relationships. Let A and B be memory access events. A **happens before** B , if A is a memory access in a critical section, then its lock is released and reacquired by the critical section that B is a part of.

Properly Labeled Programs

- Ensure that all your shared variables (shared across threads) are always accessed in the context of a **critical section** and there are no **data races**.



This program has a **data race**.

Lock (X)

Access locations A, B,
and C

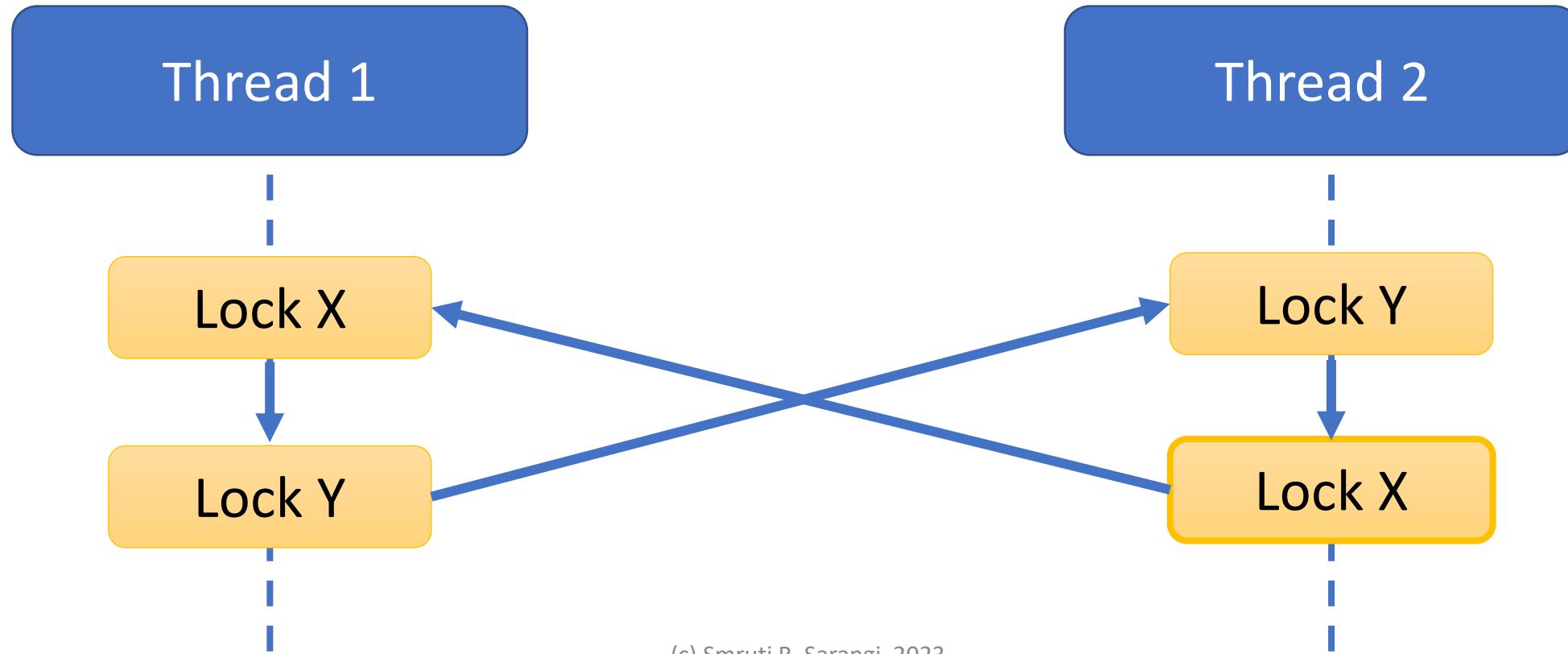
Lock (Y)

Access locations D, E,
and C



A **shared** variable always has to be **protected** by the same **lock**.

Locks can lead to deadlocks



Four Conditions for a Deadlock

Hold and Wait

- A **thread** is holding on to a **lock** and waiting for the other one.

No preemption

- The **lock** cannot be taken away from a **thread**.

Mutual exclusion

- A **lock** can be held by only a **single thread**.

Circular wait

Solutions for Preventing Deadlocks

Hold and Wait

- If a thread is waiting for a resources for too **long**, release the **resources** that it holds.



No preemption

- Resources can be **taken** away from threads by the OS.



Mutual exclusion

- This is a basic property of a lock. This cannot be tampered with.

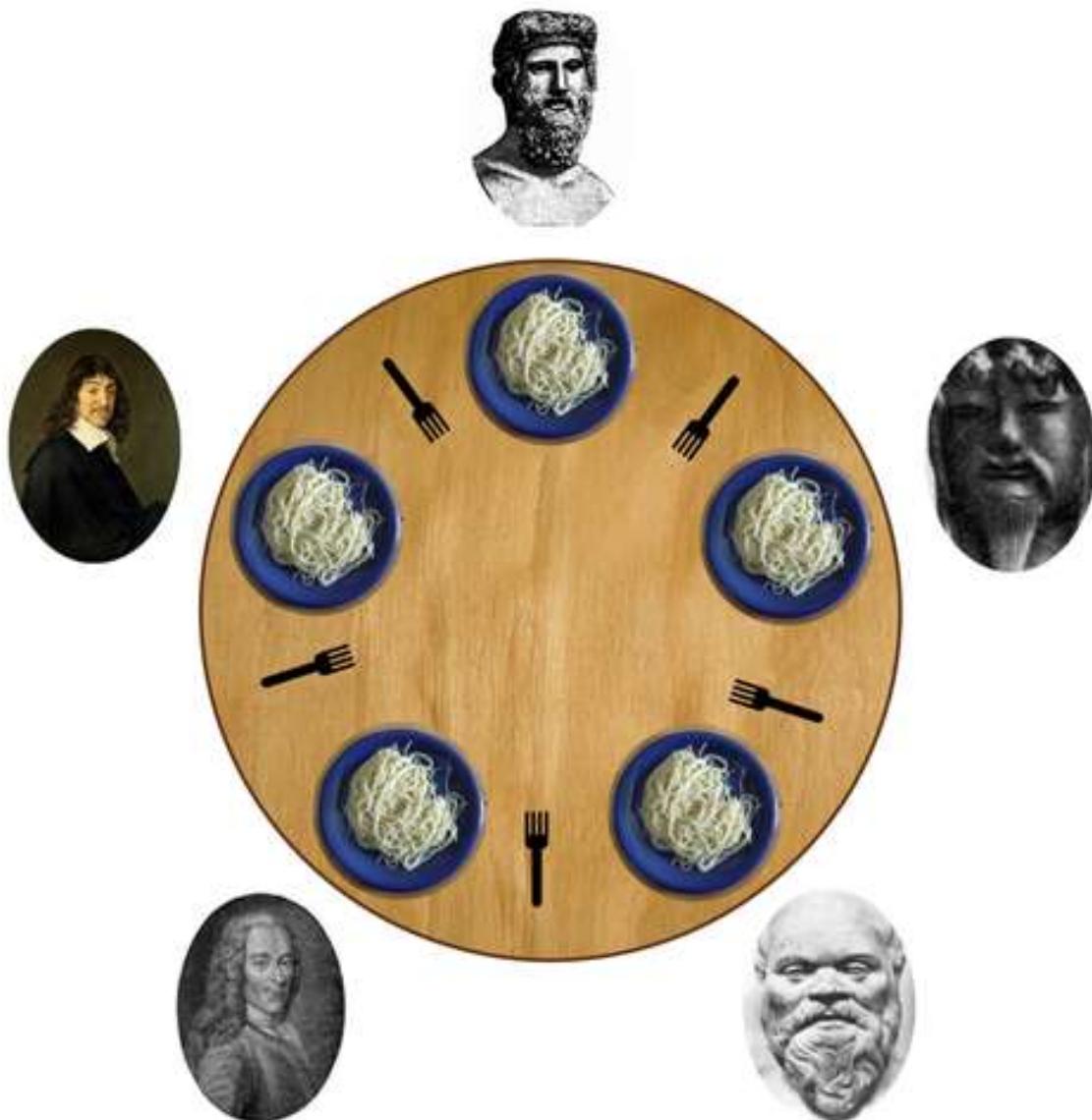


Circular wait

- Acquire locks in a given **order**



Dining Philosopher's Problem



A philosopher needs to pick up both the forks to eat. He can pick up only one fork at a time.



Avoid deadlocks.

Livelocks, Deadlocks, and Starvation

Deadlocks

- All the threads are **waiting** on each other.

Starvation

- A thread **waits** indefinitely to get **access** to the resource. Starvation freedom implies deadlock freedom (not the other way).

Livelocks

- Threads keep **executing** instructions but nothing productive gets **done**.

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
int counter;

void* func(void *arg)
{
    int *ptr = (int *) arg;
    printf("Thread %d \n", *ptr);
}
```

gcc <prog_name> -lpthread

```
int main(void)
{
    int errcode, i = 0; int *ptr;
    for (i=0; i < 2; i++)
    {
        ptr = (int *) malloc (sizeof(int));
        *ptr = i;
        errcode = pthread_create(&(tid[i]), NULL,
                                &func, ptr);
        if (errcode)
            printf("Error in creating pthreads \n");
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
}
```

More complicated version of this example

- The return value of the *func* function can be collected by the *pthread_join* call
- The second argument is a pointer to a pointer.
- It is populated with the pointer that was returned by the *func* function



pexaug.c

The same code with locks for incrementing a *shared variable*

```
int count = 0;
pthread_mutex_t cntlock;

void* func(void *arg)
{
    pthread_mutex_lock(&cntlock);
    count++;
    pthread_mutex_unlock(&cntlock);
}

int main () {
    retval = pthread_mutex_init (&cntlock, NULL);
    ...
    ...
    printf ("The final value of counter is %d \n", counter);
}
```



Code without locks (lock-free programming)

```
#include <stdatomic.h>

atomic_int count = 0;

void * fetch_and_increment (void *arg) {
    atomic_fetch_add (&count, 1);
}
```



Also known as non-blocking algorithms

Same Example with Compare_and_Exchange

Increment using the `compare_and_swap` primitive

```
atomic_int count = 0;
#define CAS atomic_compare_exchange_strong

void* fetch_and_increment (void *arg){
    int oldval, newval;
    do {
        oldval = atomic_load (&count);
        newval = oldval + 1;
        printf ("old = %d, new = %d \n", oldval, newval);
        while (! CAS (&counter, &oldval, newval)) {}
    }
```

Output



```
• old = 0, new = 1  
• old = 0, new = 1  
• old = 1, new = 2  
• The final value of count is 2
```



In this case, there is the possibility of many failed attempts at performing the CAS.



If threads are allowed to call this function repeatedly, then it is possible that a few threads may starve forever.

Theory of Non-Blocking Algorithms

Correctness

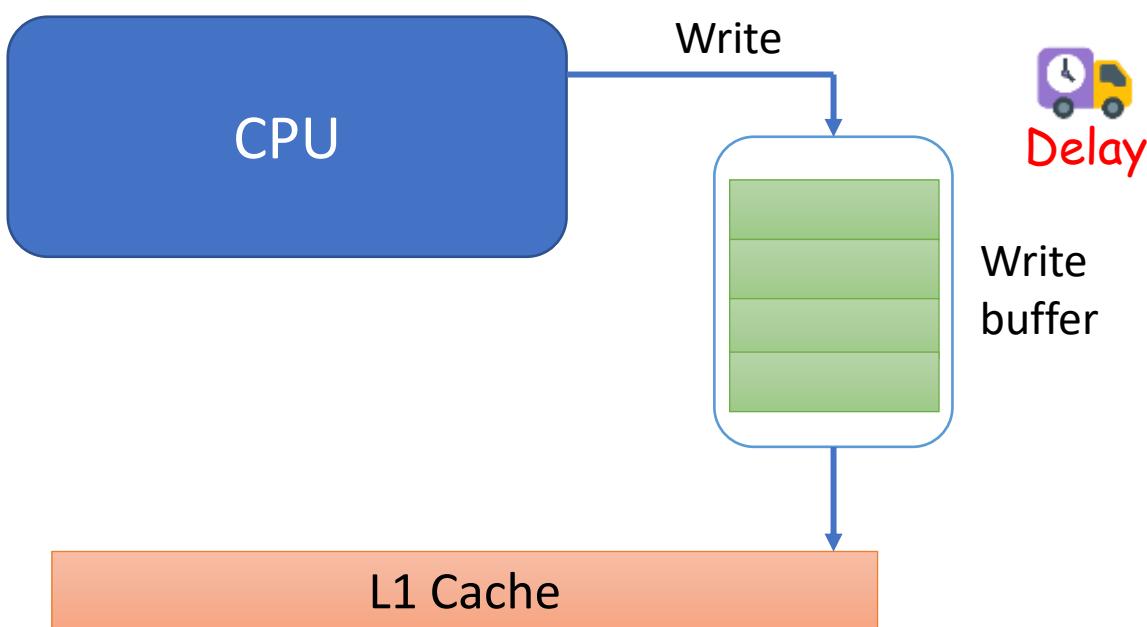
- Every read or write **operation** has the following attributes:
 - **start time**
 - **end time**
 - **completion time**

In the case of a read operation

1. The **start time** is when the load operation enters the pipeline
2. The **end time** is when the data (to be read) reaches the load-store unit
3. The **completion time** is when the data is read (at the cache or memory storage unit)

$\text{start time} < \text{completion time} < \text{end time}$

What about Write Operations?



Start time → Time at which the store operation enters the pipeline

End time → Time when the store operation is sent to the memory system (at this point the instruction also leaves the pipeline)

Completion time → Time at which the write takes effect (could be **delayed**)



start time < end time < completion time

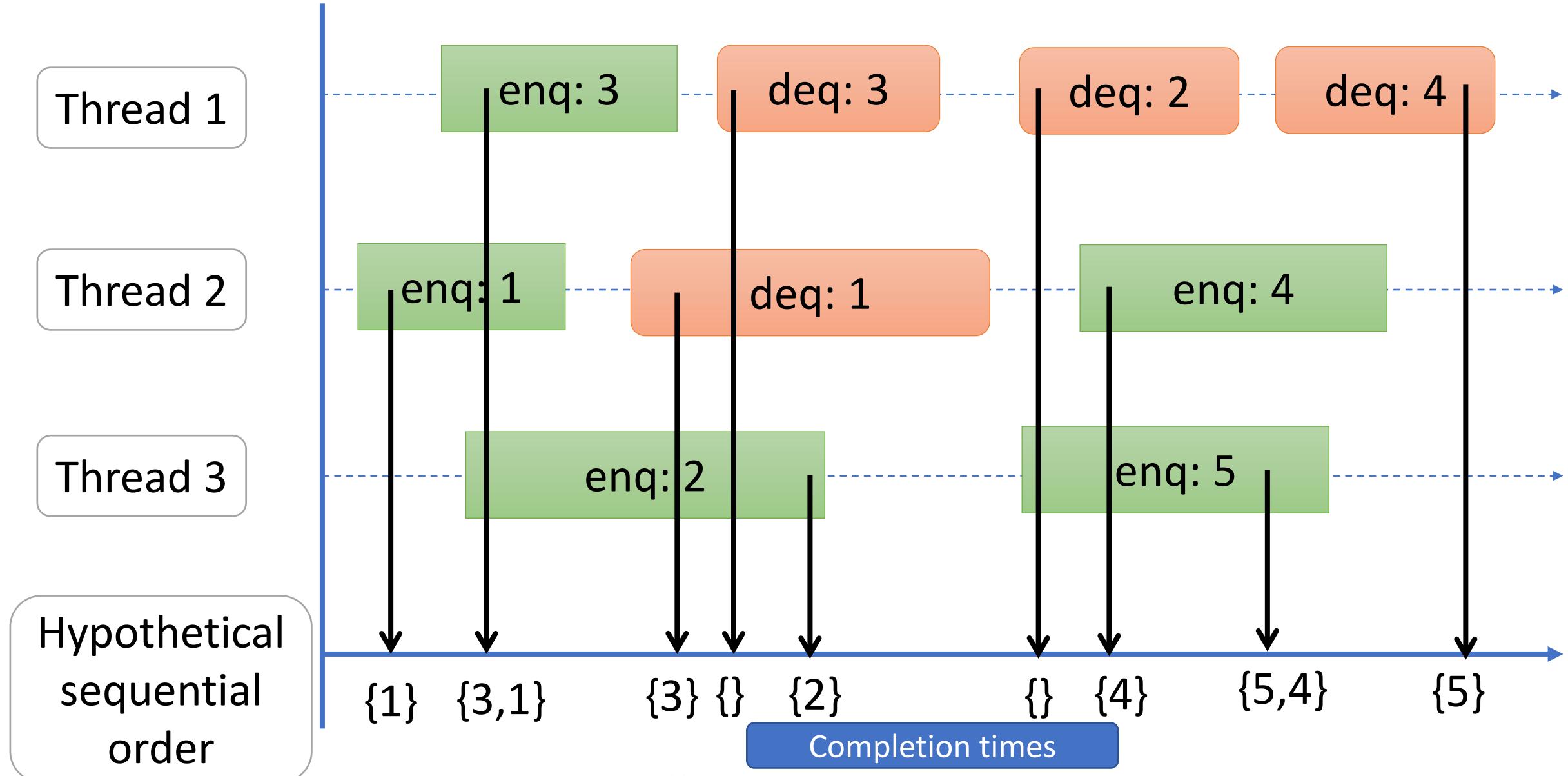
Let us generalize ...

- Reads and writes are the **simplest** of operations
- We can define many concurrent objects that can be accessed simultaneously by many threads:
 - concurrent queue
 - concurrent stack
 - concurrent linked list
 -
- Every operation has a **start time**, **end time** and a **completion time**
- An **atomic operation** appears to complete **instantaneously** at the completion time.
- The completion time itself may not be known.
- Whether the actual operation completed at that time or not is **irrelevant**.



We just need to show that it theoretically exists.

An execution involving a concurrent queue



Parallel Execution \equiv Sequential Execution

- The example on the previous slide shows a **parallel execution**
- If we arrange the operations by their **completion** times, we have a sequential execution
- We say that both are **equivalent** (\equiv)
- The sequential execution must be:
 - **Legal** → Respect the **semantics** of the object (in this case a queue)
 - Sequential semantics of a queue → FIFO order

Notion of Memory Models

Linearizable

For every operation, the completion time is between the start time and end time.

Sequentially Consistent

Per-thread order is **respected**. If *operations* A and B are issued by the same thread and A ends before B starts in the parallel execution, then A appears before B in the equivalent sequential order. The execution is **atomic**.

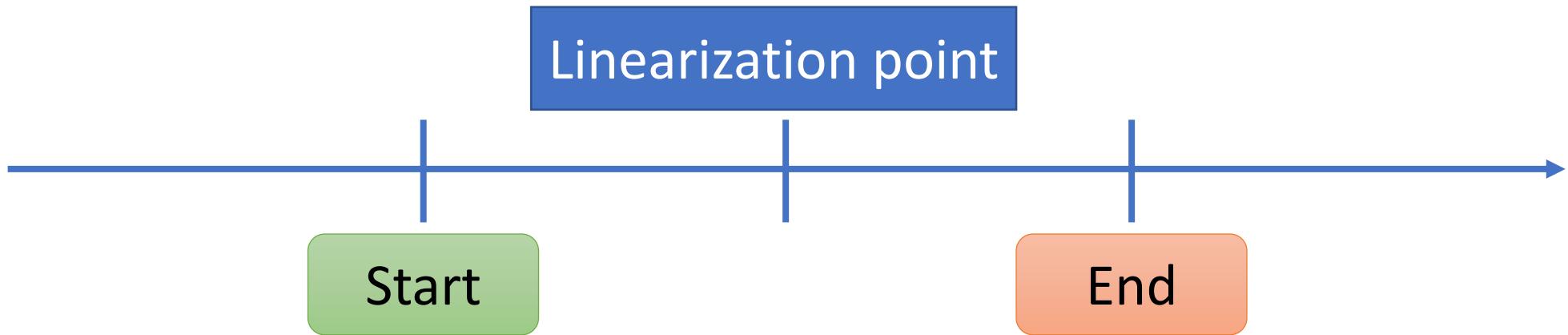
Weakly Consistent

Operations issued by the same thread to different addresses can be reordered in the equivalent sequential order. There can be different **reordering rules** and **atomicity** can also be relaxed.

Linearizability

- The operation needs to be **atomic**
 - It needs to “**appear**” to execute **instantaneously** (single instant of time)
 - Either the entire **operation** executes, or it appears to not have **executed** at all
- **Atomicity** vs **Linearizability**
 - Atomicity basically says that the **entire operation** (like enqueue/dequeue) appears to **execute** at any one instant of time
 - However, it does not say that this **execution point** needs to lie between the **start** and **end** of the operation
 - This is where we have **linearizability** – a **stronger** criterion
 - It says that if the end time of an **operation** is before the start time of another operation, then their **execution (linearization) points** follow the same order

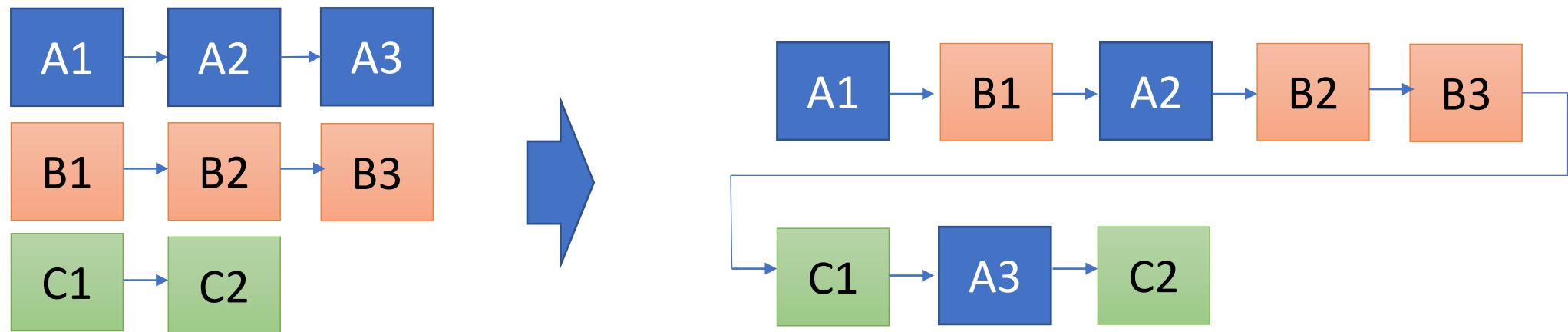
Another Look at Linearizability



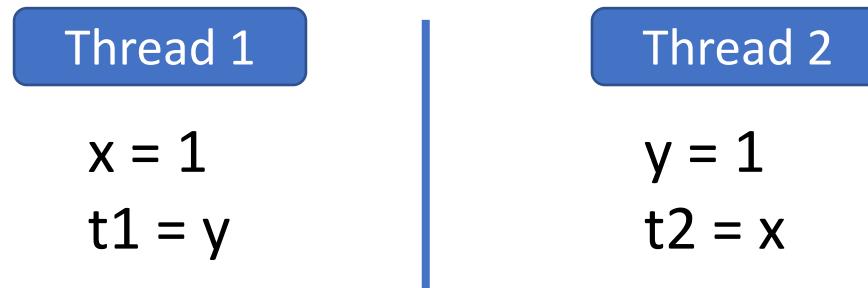
- The **linearization** point needs to lie between the **start** and **end**
- It corresponds to a specific **instruction** within the **function**
- Proving that a certain instruction is a **linearization** point is a very well established area of **research**, and it is often quite **difficult** to do so
- This makes it quite difficult to write such **programs**

Sequential Consistency

- The start and end times are not known to us
- Nor are they **relevant here**.
- What we have instead are **sequences** of operations (initiated by each thread)
- Arrange them in a global sequential order such that the outcomes are **legal**.



Some Concepts w.r.t. Memory Models



- Consider two **threads** that access two **global** variables: x and y
 - They are **initialized** to 0
 - $t1$ and $t2$ are temporary variables stored in **registers**
- ?
- Is the **outcome** $\langle t1, t2 \rangle = \langle 0, 0 \rangle$ possible?

If you assume **sequential** consistency or linearizability.

No!

Non-Sequentially Consistent Machines

Owing to **performance** reasons, all modern machines will allow this **behavior**. Their **memory models** are not **sequentially** consistent.



How do you write parallel code then?



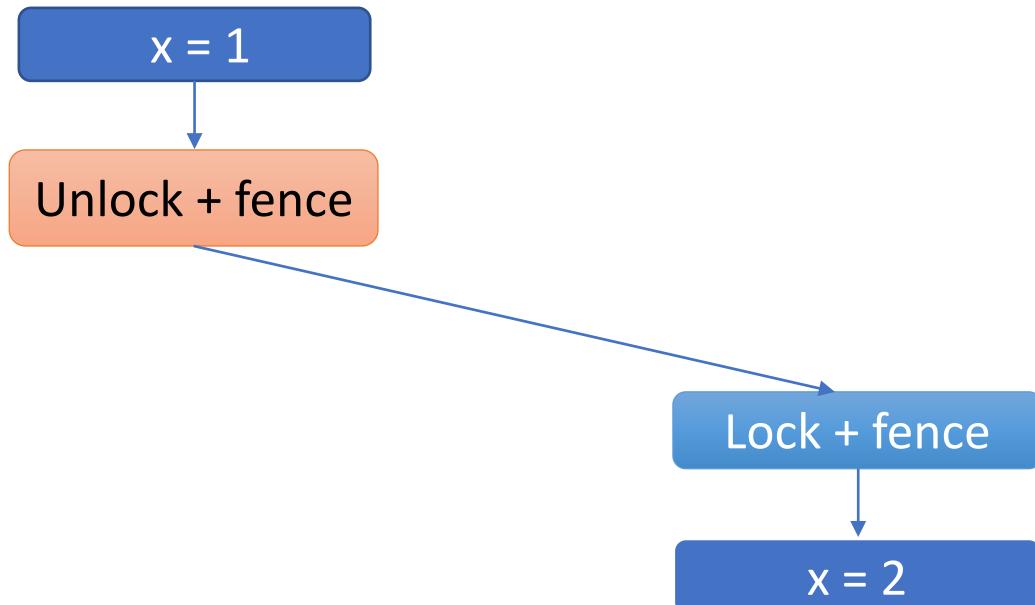
Theorem: Data-race-free programs have SC executions on all machines

- **Accesses** to global atomic variables can have **data races** → concurrent and conflicting accesses. **Avoid** data races. How? Use **fences** and **access shared variables in critical sections**
- A **fence** or a **memory barrier** instruction **forces** sequentiality (synch. between accesses)
 - All instructions **before** it in program order need to complete before it completes.
 - Before it completes, no instruction after it (in program order) can complete
- For all **accesses** to regular (non-atomic) **variables**, just wrap them in **critical sections**. A **lock** or **unlock** operation (around a critical section) needs to have a **fence**.



Data Races, Fences and Locks

- Locks use atomic operations
- Many atomic operations have built-in fences
- We can also add fences explicitly



The lock-unlock operation and the fences ensure an ordering between conflicting accesses to the same shared variable. This eliminates data races.

Enjoy SC ...



Progress Guarantees

Obstruction-free vs Lock-free vs Wait-free

- Assuming a **thread** that is holding a **lock** goes off to **sleep** or is swapped out
- Then this means that the rest of the **threads** will not be able to make any progress
- This is **unfair**.
- We thus need algorithms with better **progress** guarantees

Obstruction-free



- If $n-1$ threads stop, then the n^{th} thread can finish in a bounded number of steps

Lock-free



- At any point of time, there is at least one thread that is going to complete in a bounded number of steps.

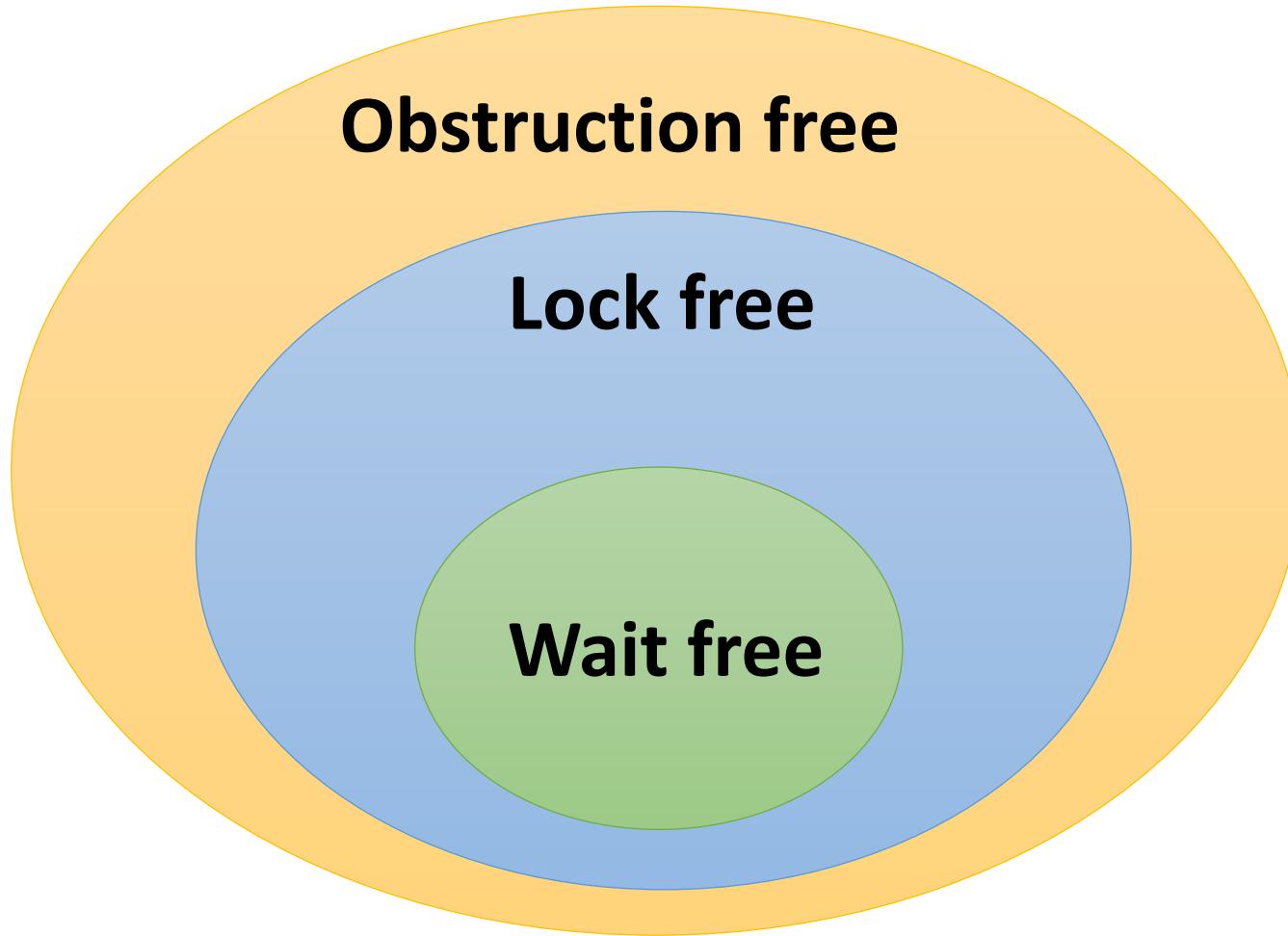
Wait-free

- All threads complete within a bounded number of steps.



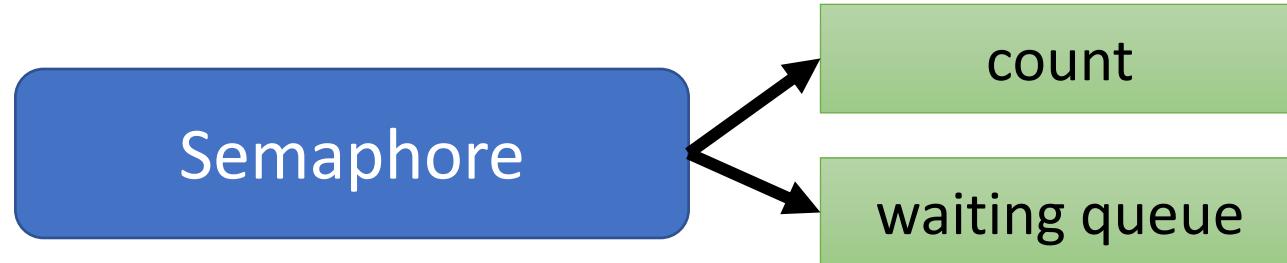
What does all of this mean?

- In an **obstruction-free** algorithm, no thread can hold a **lock** and go off to sleep
 - It should be possible to **finish** one thread's operation by putting the rest of the **threads** to **sleep** all the time
 - Using **locks** is thus precluded unless locks can be **released** on a context switch
- Lock-free algorithms are more **difficult** to write
 - They are **fast** in practice
 - They guarantee system wide **progress**, but individual threads can starve
- Use wait-free algorithms
 - They are slower than lock-free **algorithms** in practice but have much **stronger** progress guarantees.



Semaphores and Reader-Writer Locks

Semaphores



- A semaphore maintains a **count**
- Plus, a kernel-level **wait queue**

Execute atomically

sem_wait

```
if (count == 0)  
    wait();  
else  
    count--;
```

Semaphores – II

sem_post

```
if ( (count == 0) && process_waiting())
    wake_from_wait_queue();
else
    count ++;
```

- A **binary** semaphore (Boolean count) is equivalent to a lock
- A **non-binary** semaphore uses a count larger than 1
- Quite useful while **implementing** a **bounded queue**

Reader-Writer Lock

Key Idea: Either one thread **writes** or multiple threads **read**



Have two **modes**:

1. A **write mode** that allows a single **writer**
2. A **read mode** that allows multiple **readers** but no writer

Switch between them

Read and Write Locks

Basic Idea:

1

The `enqueue` and `dequeue` function need the `read_write lock`: `rwlock`

`Writing` always involves reading

2

For acquiring the `read` lock, we need to ensure that there are no current `writers`. Hence, acquire `rwlock` if there are no other `readers`

3

Update the number of `readers`, by locking the read lock

```
void get_write_lock(){
    LOCK(__rwlock);
}

void release_write_lock(){
    UNLOCK(__rwlock);
}

void get_read_lock(){
    LOCK(__rdlock);
    if (readers == 0)
        LOCK(__rwlock);
    readers++;
    UNLOCK (__rdlock);
}
```

```
void release_read_lock(){
    LOCK(__rdlock);
    readers--;
    if (readers == 0)
        UNLOCK (__rwlock);
    UNLOCK(__rdlock);
}
```

Fairness is an issue

Queues

Case of Bounded Queues



- The **bounded** queue is an important data structure
- It is used in a large number of **kernel** subsystems
- **Device drivers** use such structures a lot for **communicating** with other processes and devices
- There are multiple **producer** threads and multiple **consumer** threads
- This needs to be a concurrent queue.

Codes

E.g. 1

Bounded wait-free queue - single producer and single consumer



wfq.c

E.g. 2

Bounded queue with locks - pthread mutexes



wfbusylock.c

E.g. 3

Bounded queue with semaphores



wfsem.c

E.g. 4

Bounded queue with semaphores - no busy waiting



wfsem-ii.c

E.g. 5

Bounded queue with semaphores and reader-writer locks



rwlock.c



Atomics

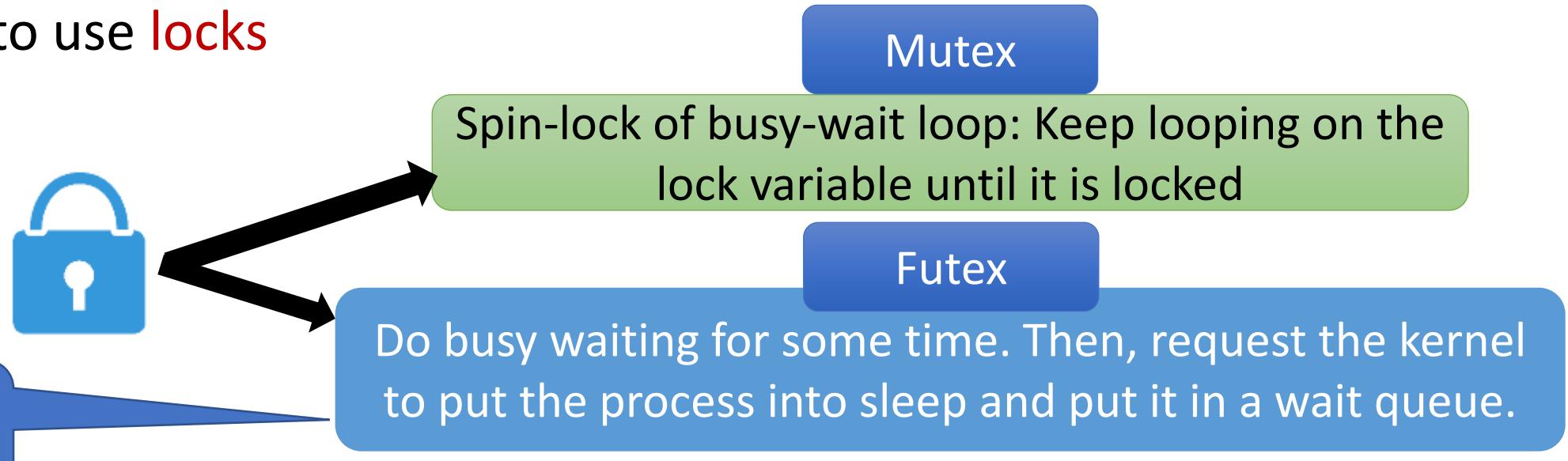
Example 1

- <stdatomic.h> contains a **library** of atomic integers and characters
- *atomic_load* and *atomic_store* operations are used to read and write these atomic variables
- A few atomic operations are supported:
 - *atomic_fetch_add*
 - *atomic_flag_test_and_set*
 - *atomic_compare_exchange_strong*
- For the sake of **simplicity**, let us assume that **atomic** operations are **globally** (sequentially) ordered.
 - We cannot say the same about regular **memory** accesses
- This is the crux of **lock-free** programming

Example 2

Wait-free Queue → Queue with Locks

- Wait-free queue → It is easy to construct one for a single enqueuer and dequeuer. One thread enqueues and the other dequeues.
- However, it is quite hard to design a wait-free algorithm with multiple enqueuers and dequeuers
- Easier to use locks



Queue with Reader-Writer Lock

- Let us now add a new function.

```
int peak() {  
    /* This is the read function */  
    int val = (head == tail)? -1 : queue[head];  
    printf ("Queue head = %d\n",val);  
    return val;  
}
```

Just read the head of the queue



If we have multiple **readers** at a time, they can be **allowed** to read the **head** of the **queue** concurrently.

Use a reader-writer lock

Other Kinds of Synchronization

Condition Variables

- **Semaphores** are a generic OS mechanism
- ***pthreads*** have a lightweight version – **condition variables**

```
pthread_mutex_t mlock;  
pthread_cond_t cond;  
pthread_cond_init (&cond, NULL);
```

A conditional variable is always associated with a mutex

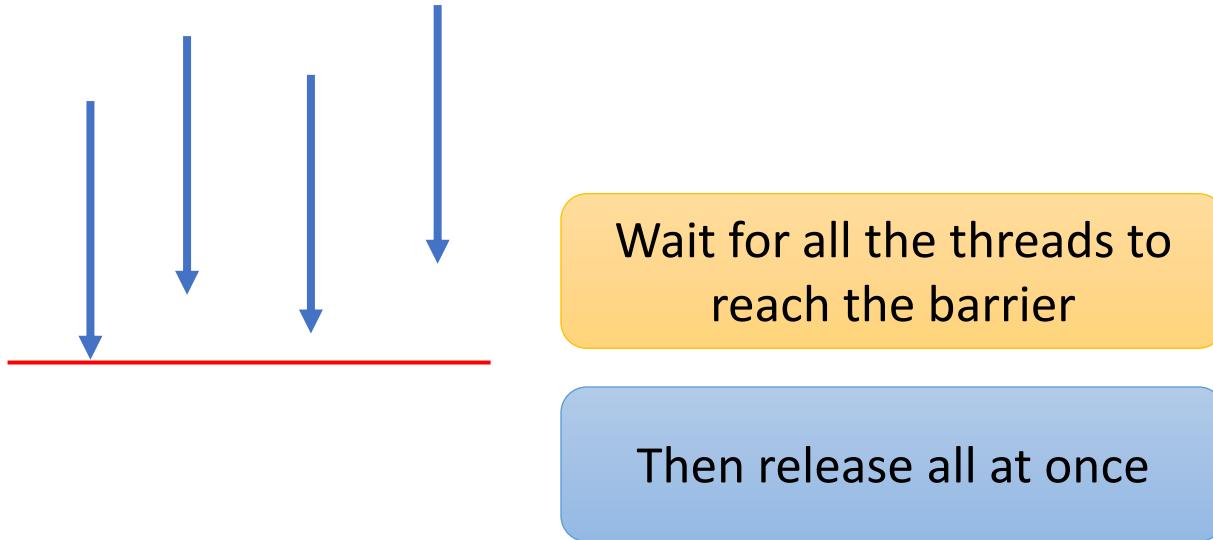
```
/* wait on the condition variable*/  
pthread_mutex_lock (&mlock);  
pthread_cond_wait (&cond, &mlock);  
pthread_mutex_unlock (&mlock);
```

The *wait* call releases the mutex while waiting and reacquires it once woken up

```
pthread_mutex_lock (&mlock);  
pthread_cond_signal (&cond);  
pthread_mutex_unlock (&mlock);
```

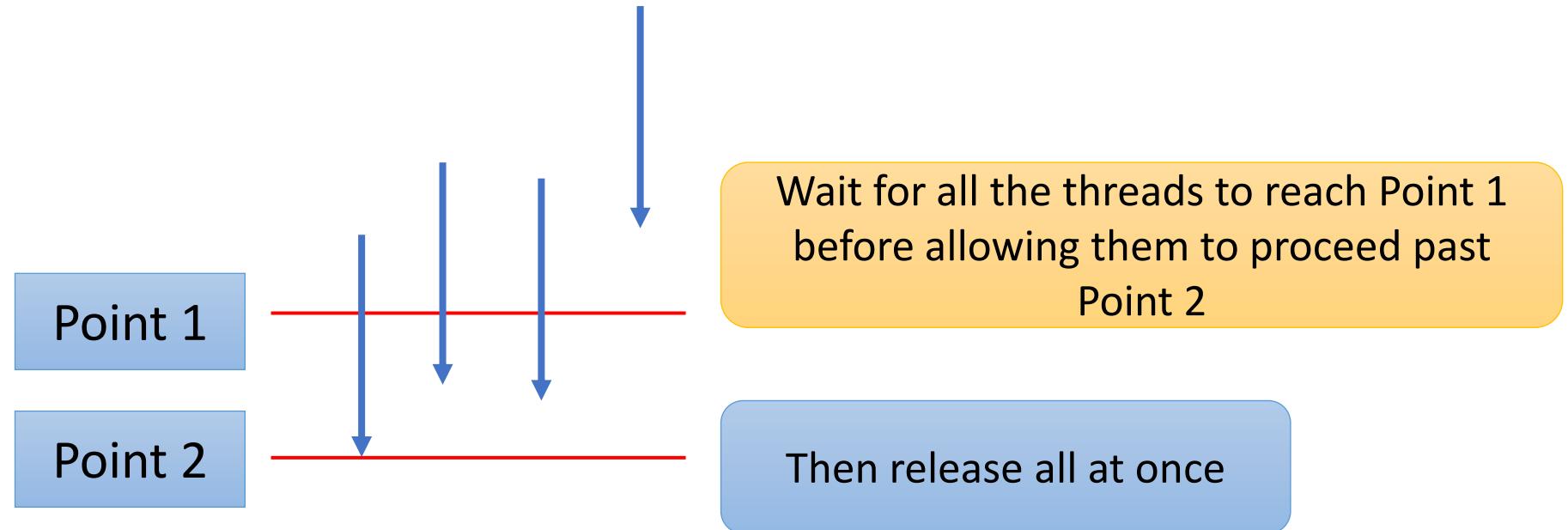
The *pthread_cond_signal* and *pthread_cond_broadcast* functions wake up waiting threads

Barriers



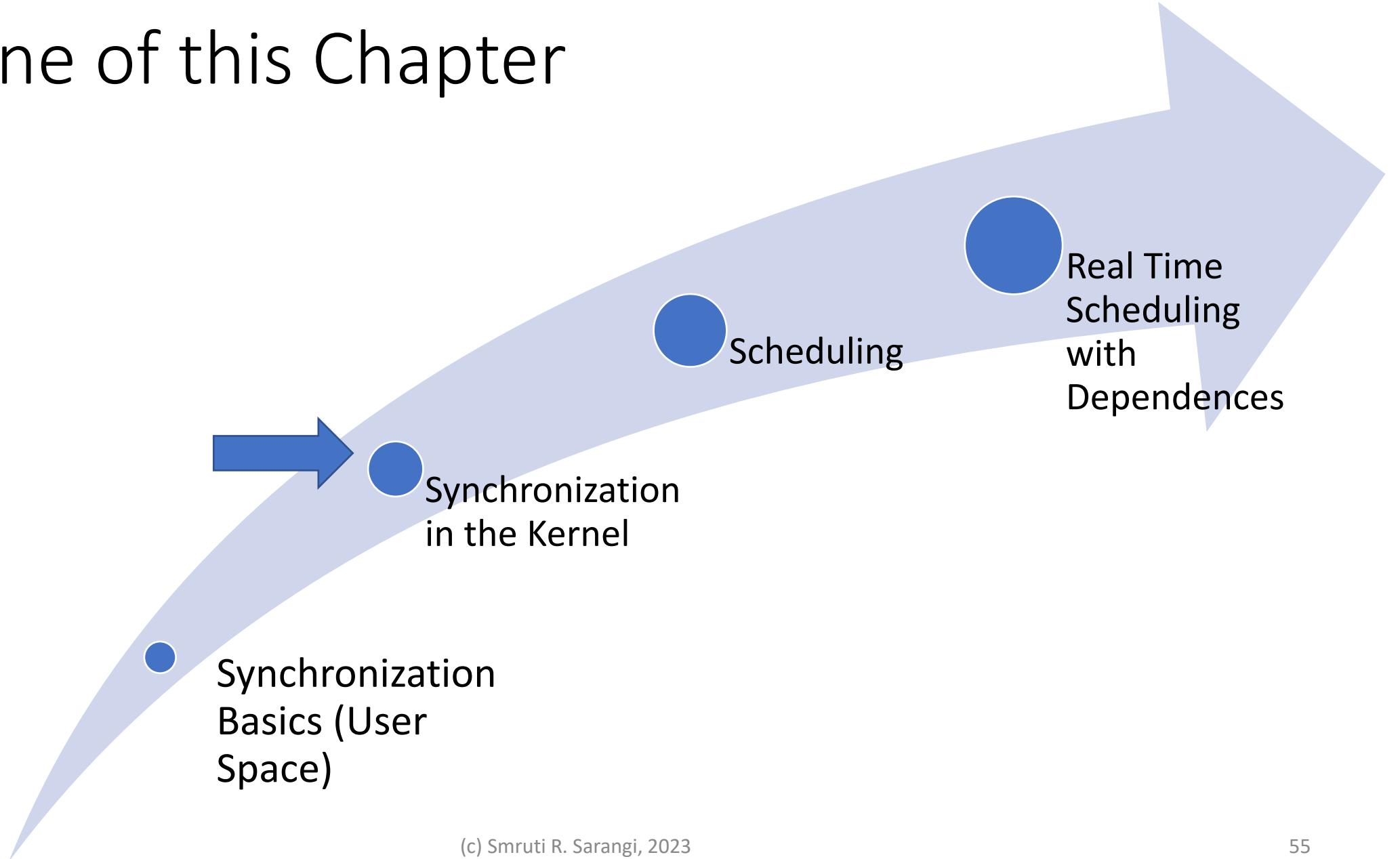
- A *barrier* acts as a synchronization point
- All the **threads** wait at the barrier for the **slowest** thread to arrive
- Then all of them are released and **begin** to **execute**

Phasers



- A *phaser* is a more **complex** synchronization point
- All the **threads** wait at Point 2 till the **slowest** thread reaches Point 1
- Then all of them are **released** and can proceed past Point 2

Outline of this Chapter



Questions regarding Concurrency in the Kernel



- If we have **multiple** threads in the kernel (as is normally the case), then a **data structure** can be accessed in parallel
- We need to support **concurrency**
 - Either use **locks**
 - Some variant of **lock-free programming**



- The other source of **concurrency** is the interrupt and scheduling process
- Kernel code can be **interrupted** in the middle. Then, the **scheduler** can **schedule** other threads, which can modify the state of the data structure.

Spinlocks

- It is the most basic **type** of lock
- They run a “busy wait” lock. Unlike other locks, the **thread** does not go off to **sleep**.
- They can be used in the **interrupt** context where **blocking** is not permitted
- After acquiring a spinlock, the **kernel** turns off pre-emption
- Otherwise, a new **thread** can run and that can try to acquire the lock. It will **block** yet the system will not make progress.
- The spinlock can also be visualized as a **lock** on the CPU. We don’t want to migrate the thread to a **different** CPU.

How to enable/disable pre-emption?

- The kernel maintains a per CPU variable that indicates whether pre-emption is allowed or not
 - If it is 0, pre-emption is allowed
 - Else, it is disallowed



```
#define preempt_disable() \
do { \
    preempt_count_inc(); \
    barrier(); \
} while (0)
```

Stop memory
reordering

/include/linux/preempt.h

Not the same barrier
we studied before. A
different one

Enable Preemption



```
#define preempt_enable() \
do { \
    barrier(); \
    if (unlikely(preempt_count_dec_and_test())) \
        __preempt_schedule(); \
} while (0)
```

- If the **count** reaches 0
 - Run the *schedule* function

Spinlocks



/include/linux/spinlock.h



/include/linux/spinlock_types_raw.h

```
typedef struct raw_spinlock {  
    arch_spinlock_t raw_lock;  
#ifdef CONFIG_DEBUG_LOCK_ALLOC  
    struct lockdep_map dep_map;  
#endif  
} raw_spinlock_t;
```

- The *raw_spinlock_t* type is just a wrapper on the architecture defined spin lock → This structure **wraps** an atomic/volatile **integer**
- It also has a *lockdep_map* structure to find deadlocks

Inner Workings of the Spinlock

Ticket lock



/include/asm-generic/spin-lock.h



```
void arch_spin_lock(arch_spinlock_t *lock)
{
    u32 val = atomic_fetch_add (1<<16, lock);
    u16 ticket = val >> 16; /* upper 16 bits of lock */

    if (ticket == (u16) val) /* Ticket id == ticket next in line */
        return;

    atomic_cond_read_acquire(lock, ticket == (u16)VAL);
    smp_mb(); /* barrier instruction*/
}
```

Ticket id

To be served

The spin lock function

The actual spin loop (keeping the **memory model** into account)



```
#define smp_cond_load_relaxed(ptr, cond_expr) ({\n    typeof(ptr) __PTR = (ptr);\n    __unqual_scalar_typeof(*ptr) VAL;\n    for (;;) {\n        VAL = READ_ONCE(*__PTR);\n        if (cond_expr)\n            break;\n        cpu_relax(); /* insert a delay*/\n    }\n    (typeof(*ptr)) VAL;\n})
```



The *unlock* Function

Ticket id

To be served



```
void arch_spin_unlock(arch_spinlock_t *lock)
{
    u16 *ptr = (u16 *)lock + IS_ENABLED(CONFIG_CPU_BIG_ENDIAN);
    u32 val = atomic_read(lock);

    smp_store_release(ptr, (u16)val + 1); /* store following release semantics */
}
```

- Increment the “*to be served*” part
- The *acquire* and *release* keywords are the same as the ones used in the Release Consistency (RC) literature
 - All *memory* operations after the acquire can *complete* only *after* it completes
 - The release needs to *complete* only after all the memory operations *before* it have completed

A Fast Path: Trylock function



```
static __always_inline bool arch_spin_trylock(arch_spinlock_t *lock)
{
    u32 old = atomic_read(lock);

    if ((old >> 16) != (old & 0xffff))
        return false;

    return atomic_try_cmpxchg(lock, &old, old + (1<<16));
}
```

The lock is held

Attempt to get the lock quickly

The **fast** path/ slow path approach is a **standard** technique. If the lock is **unlikely** to be held, attempt the **fast** path, otherwise **fall back** to the **slow(er)** path.

Spin locks and Interrupts

- **Interrupts** can still be processed on the **CPU** that is waiting on a spin lock
- What if the **interrupt handler** tries to acquire the same **spinlock**
 - We have a **deadlock**
- Hence, we use the “**_irq**” variant of the **function**. It enables and disables **interrupts** in the critical section.
 - *raw_spin_lock_irq* and *raw_spin_unlock_irq*
- Sadly *raw_spin_unlock_irq* will **enable** all interrupts (not in your best interest)
- Hence, use the **_irqsave** and **_irqrestore** versions that **save** and **restore** the interrupt state.

Kernel Mutexes



/include/linux/mutex.h

mutex_lock(struct mutex *lock)
mutex_unlock(struct mutex *lock)
mutex_trylock(struct mutex *lock)

- A **spinlock** is held by a CPU, a **mutex** is held by a task
- The task can go off to **sleep**



```
struct mutex {  
    atomic_long_t      owner;  
    raw_spinlock_t    wait_lock;  
    struct list_head  wait_list;  
  
    #ifdef CONFIG_DEBUG_LOCK_ALLOC  
    struct lockdep_map dep_map;  
    #endif  
};
```

struct task_struct*

The spin lock protecting the
wait_list

Contending processes

Mutex Locking



/kernel/locking/mutex.c



```
void mutex_lock(struct mutex *lock)
{
    might_sleep(); /* prints a stacktrace if called in
an atomic context (sleeping not allowed */

    if (!__mutex_trylock_fast(lock)) /* cmpxchg on owner */
        __mutex_lock_slowpath(lock);
}
```

- Supports the slow path/ fast path **concept**
- **Fast path:** Compare and exchange (acquire) as we have seen **earlier**
- **Slow path:** We need to try to **acquire** the spinlock. However, if that does not seem to be possible, make the process **sleep**. Lock the task in the **UNINTERRUPTIBLE** state.

Slow Path

- First, attempt a **fast path** lock
 - *cpxchg* on the *owner* field
 - If that works, well and good
- Acquire a **lock on the *wait_list***
 - Lock *wait_lock* using a **spinlock**
 - Insert the current task to the *wait_list*
 - Set the **state** of the current task to a sleeping state: UNINTERRUPTIBLE or INTERRUPTIBLE (in some cases)
- Call the **scheduler** to schedule ready processes



Different Kinds of Locks in the Kernel

- Queued Spinlock or MCS lock (`qspinlock.c`)
 - Create a linked list of **nodes**
 - Insert the given **node** (consisting of the current task) atomically to the end of **this list**
 - Uses complex **lock-free** programming
 - It is far more scalable than a regular **spinlock**.
 - It eliminates cache line **bouncing**: When we try to read/write the **lock** a copy of it needs to be brought into the **local cache**. When another core needs it, the entire cache line's contents needs to be **transferred** to it.
- `osq_lock.c` → A **variant** of the MCS lock (useful with mutexes)
- `qrwlock.c` → A reader writer lock that gives priority to **readers**



/kernel/locking/semaphore.c

Semaphores in the Kernel



```
struct semaphore {  
    raw_spinlock_t lock;  
    unsigned int count;  
    struct list_head wait_list;  
};
```



/include/linux/
semaphore.h

Waiting list of
processes

- There are two key **functions**: *up* and *down*
- Very similar to user-level **semaphores**
- For *down*, set the current state to **UNINTERRUPTIBLE** and call the scheduler
- For *up*, set the current state to **RUNNING** and call the scheduler

The Lockdep Mechanism

The *lockdep* mechanism



/kernel/locking/lockdep.c



- It is the **kernel** lock validator
- There could be different **issues** with **locks**
- We might have **circular** waiting that leads to deadlocks
- The base function that is called whenever a lock is acquired is ***lock_acquire*** in **lockdep.c**
- This makes a call to **lock_acquire** that does the bulk of the processing
 - It verifies that the **lock depth** (number of locks that the current task has acquired) is below a **maximum** number.
 - Next, it needs to **validate** the chain of locks that have been acquired including the lock that is about to be acquired. Makes a call to ***validate_chain***

Validate spin
locks

Validate
mutexes

Validate reader
writer locks

validate_chain

<https://www.kernel.org/doc/Documentation/locking/lockdep-design.rst>

- We define four **types** of **states**: softirq-safe, softirq-unsafe, hardirq-safe, hardirq-unsafe
- A **softirq-safe** lock means it was acquired in a **softirq** context. At that point, irqs would have been **disabled**.
- On the other hand, a **softirq-unsafe** lock would have been acquired with irq interrupts turned **on**. This potentially allows the interrupt **handler** to try to reacquire the lock.
- A **softirq-unsafe** is also **hardirq-unsafe**
- First, check for **deadlocks**. Cannot have the $A \rightarrow B$, and $B \rightarrow A$ problem (trivial deadlock). A and B are locks. \rightarrow is a **path**
- No path can contain a **hardirq-unsafe** and then a **hardirq-safe** lock. The latter may have interrupted the former. May lead to lock **inversion**. Same for softirq locks.

Lock inversion problem

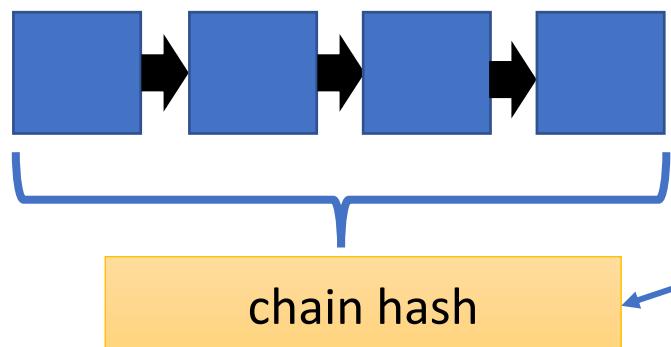




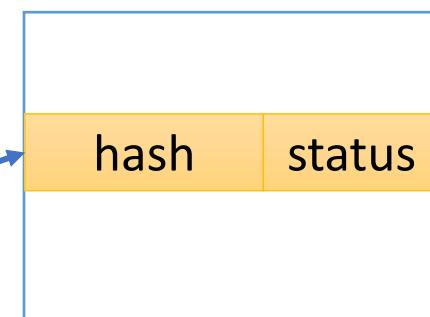
Why validate_chain not validate_graph?

- Find the **cycle** in a graph using **breadth first search (algorithm)** used in lockdep.c requires $O(V+E)$ steps
- This is very **slow**. 

Consider a chain of locks



A long sequence of **lock operations** possibly acquired by **different threads**.



Acts as a cache for chains of locks. No need to find cycles again and again

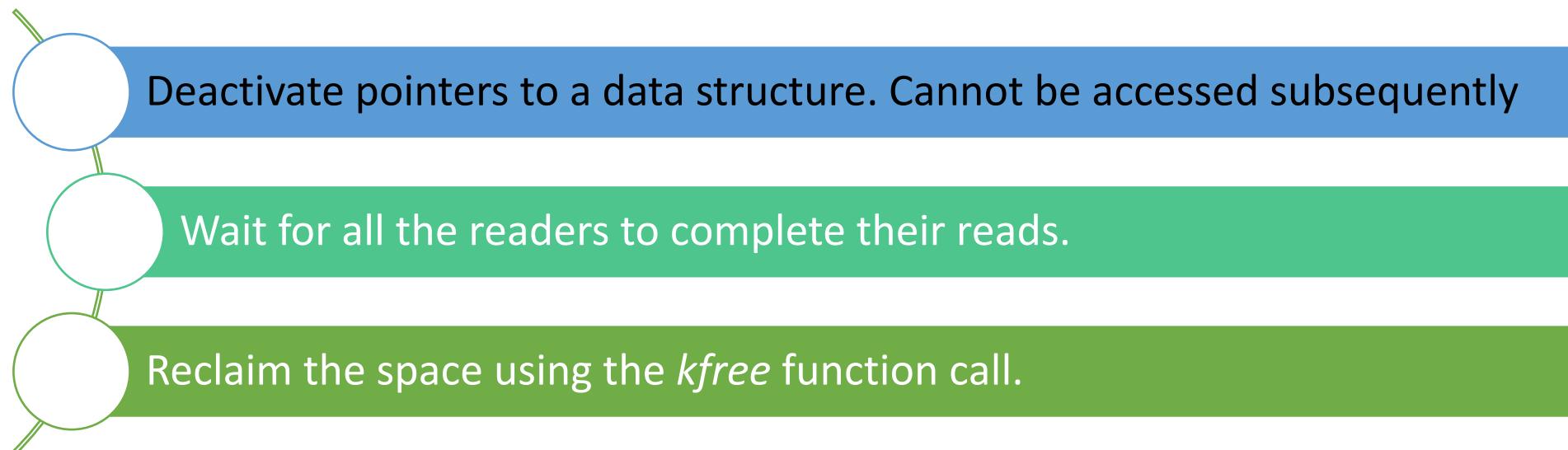
The RCU Mechanism

The RCU Mechanism

A smart reader-writer lock



- Allocating and **freeing** data structures in the kernel is a tricky issue
- Especially, when **multiple** threads are involved.
- This is where the **read-copy-update (RCU)** mechanism is handy



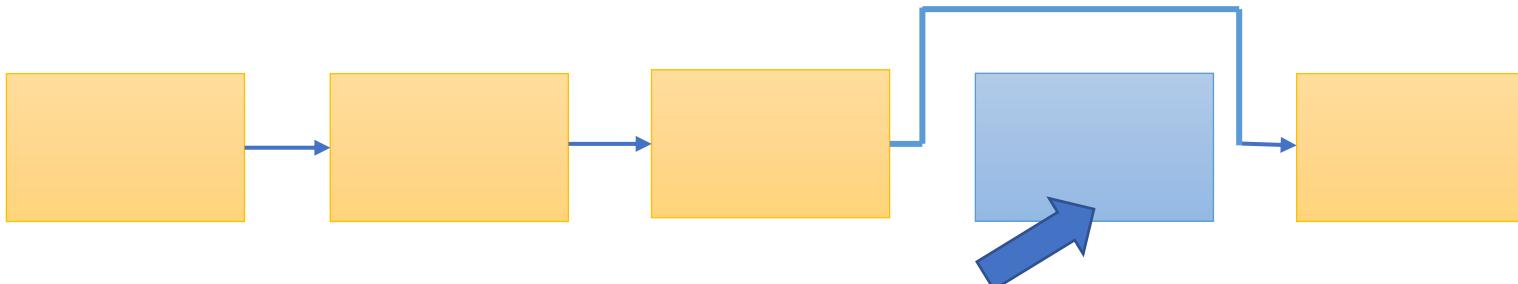
Example of a Linked List

1



Delete this node

2



Readers may be reading this

Synchronize
and reclaim
the space

3



Final state

The Key Idea



/include/linux/rcupdate.h



Decouple the **write(update)**, **read**, and memory **reclamation** steps.

Call	Explanation	Action
rcu_read_lock()	Enter a read-side critical section	Disable preemptions
rcu_read_unlock()	Exit a read-side critical section	Enable preemptions
 synchronize_rcu()	Marks the end of the reading phase	Wait until all readers finish
rcu_assign_pointer()	Assign a value to an RCU-protected pointer	Assignment + checks + memory barrier
rcu_dereference()	Read the value of an RCU-protected pointer	Read → memory barrier

Using an RCU

```
rcu_read_lock();
list_for_each_entry_rcu(p, head, list) {
    t1 = p->a;
    t2 = p->b;
}
rcu_read_unlock();
```

Reads the list between
RCU calls

```
list_replace_rcu(&p->list, &q->list);
synchronize_rcu();
kfree(p);
```

Write to the list,
synchronize, and then free

Blocking and non-blocking versions
(with callbacks) are available for the
synchronize_rcu operation

Assign an RCU-protected pointer



/include/linux/rcupdate.h



```
#define rcu_assign_pointer(p, v)
do {
    uintptr_t _r_a_p_v = (uintptr_t)(v);
    /* do some checking */
    if (__builtin_constant_p(v) && (_r_a_p_v) == (uintptr_t)NULL)
        WRITE_ONCE((p), (typeof(p))(_r_a_p_v));
    else
        smp_store_release(&p, RCU_INITIALIZER((typeof(p))_r_a_p_v));
} while (0)
```

- This is a **regular** assignment to a pointer + some checks + memory barrier

Example of *rcu_assign_pointer*



/include/linux/rculist.h

```
static inline void list_replace_rcu(struct list_head *old,
        struct list_head *new)
{
    new->next = old->next;
    new->prev = old->prev;
    rcu_assign_pointer(list_next_rcu(new->prev), new); /* new->prev->next = new */
    new->next->prev = new;
    old->prev = LIST_POISON2;
}
```

Get the value of an RCU-protected Pointer



```
#define __rcu_dereference_check(p, local, c, space) \
({ \
    typeof(*p) *local = (typeof(*p) *__force) READ_ONCE(p); \
    rcu_check_sparse(p, space); \
    ((typeof(*p) __force __kernel *)(local)); \
})
```



- Do some checks, read the value of the pointer, typecast and return
- READ_ONCE includes a compile-time memory barrier

Example of *rcu_dereference*



```
#define __hlist_for_each_rcu(pos, head) \
    for (pos = rcu_dereference(hlist_first_rcu(head)); \
          pos; \
          pos = rcu_dereference(hlist_next_rcu(pos)))
```

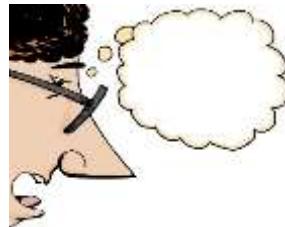
An *hlist* is a non-circular doubly-linked list

Mystery behind `synchronize_rcu`

- `rcu_read_lock` and `rcu_read_unlock` pretty much do nothing other than `disabling` and `enabling` pre-emption



How does `synchronize_rcu` know that all readers are done?



```
foreach_cpu (cpu)
    run_curr_task_on_cpu (cpu);
```

Insight: If you are able to `run` a `thread` on a CPU, then it means that it is in a `preemptable state`. This means that it has `exited` the “read critical section” → The `reader` is not `active` anymore → The `pointer` can be `reclaimed`

Too simple and too slow

RCU in all its glory ...



<https://docs.kernel.org/RCU/index.html>

RCU has two implementations

Tree RCU (/kernel/rcu/tree.c)

Tiny RCU (/kernel/rcu/tiny.c)



Tree RCU is the default implementation for **kernels** in **servers**

Grace periods in RCUs

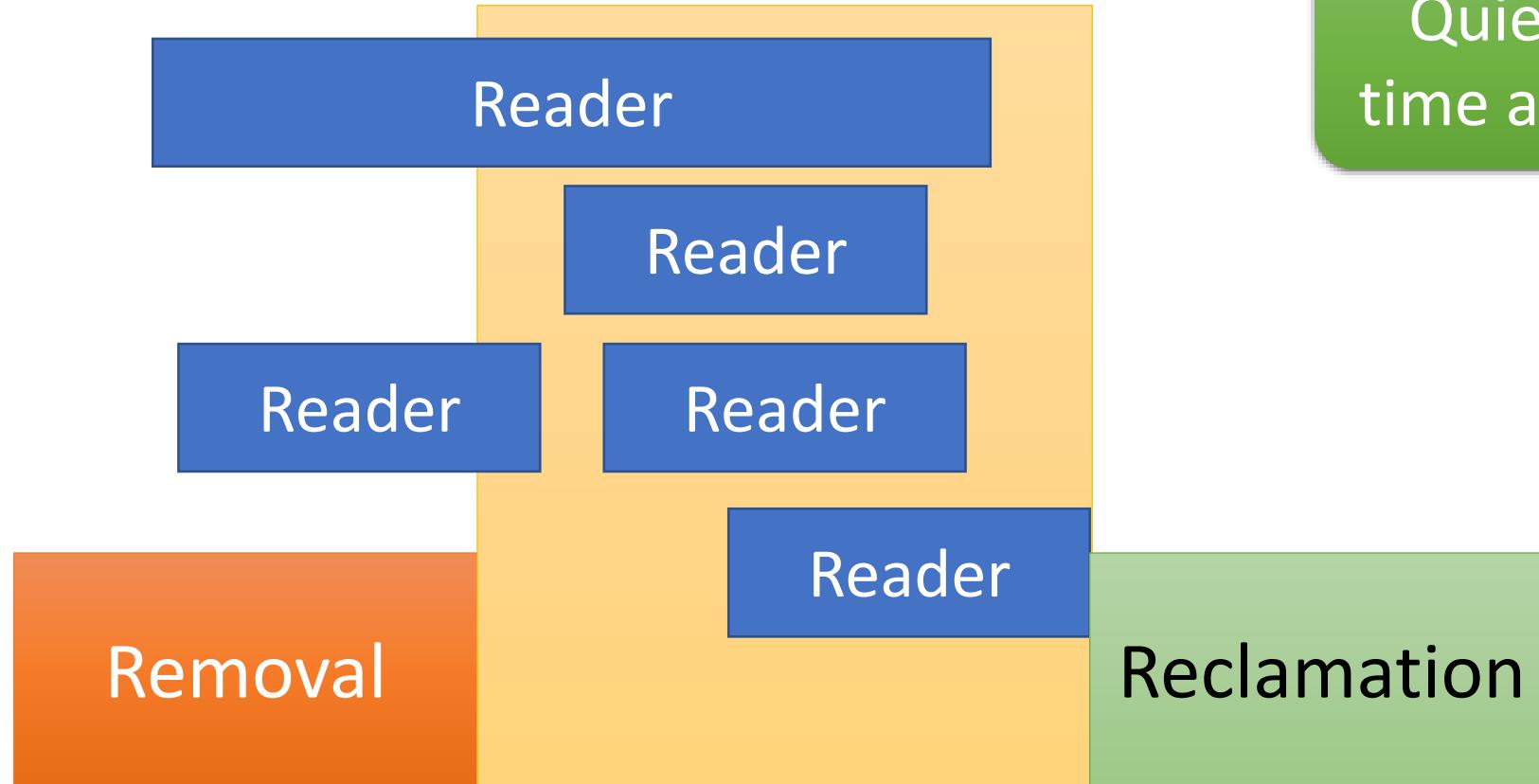
It is the **time** between the **read+update** and the **reclamation**. It must be **long** enough for all the **readers** to have dropped the references to the object.

When does the “Grace Period” end?

- Four conditions
 - When a thread **blocks** (you know it is out of the read lock-unlock phase)
 - **Switches** to user-mode execution
 - Enters an **idle** loop
 - Switch the **context** to run a kernel thread
- There are **preemptible** variants of RCU
 - However, they do not **rely** on indirect mechanisms
 - They require **read_lock** operations to **increment** per-CPU counters
- **synchronize_rcu** provides a **strict** ordering between threads. **Acts** like a **barrier** (waits for all readers to finish) and a barrier (a memory **barrier** where all the **writes** that happen in the **read block** are visible later)

Removal and Reclamation

<https://lwn.net/Articles/305782/>



Quiescent state: A point of time after the read block ends



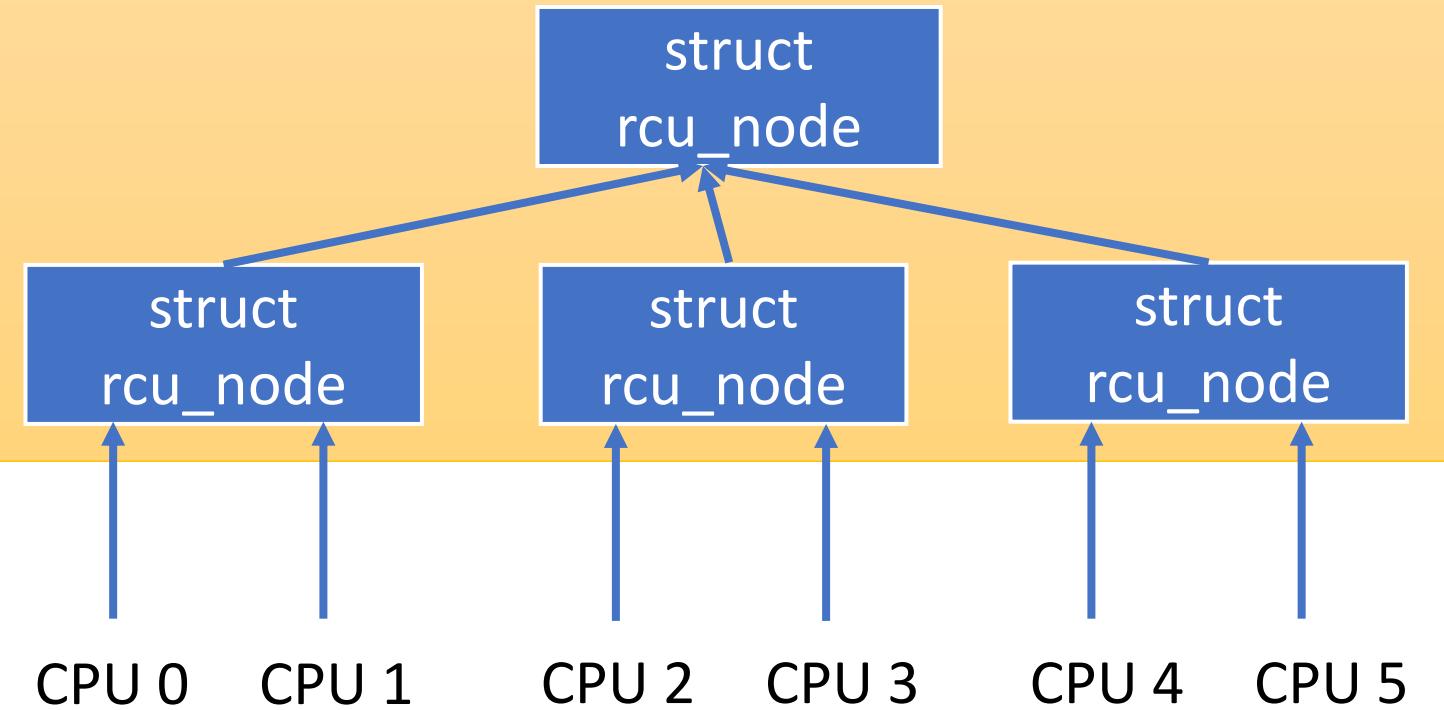
Let's say that a CPU sets/resets a bit at the end of a grace period

Tree RCU



Manage these bits as an Emde Boas tree

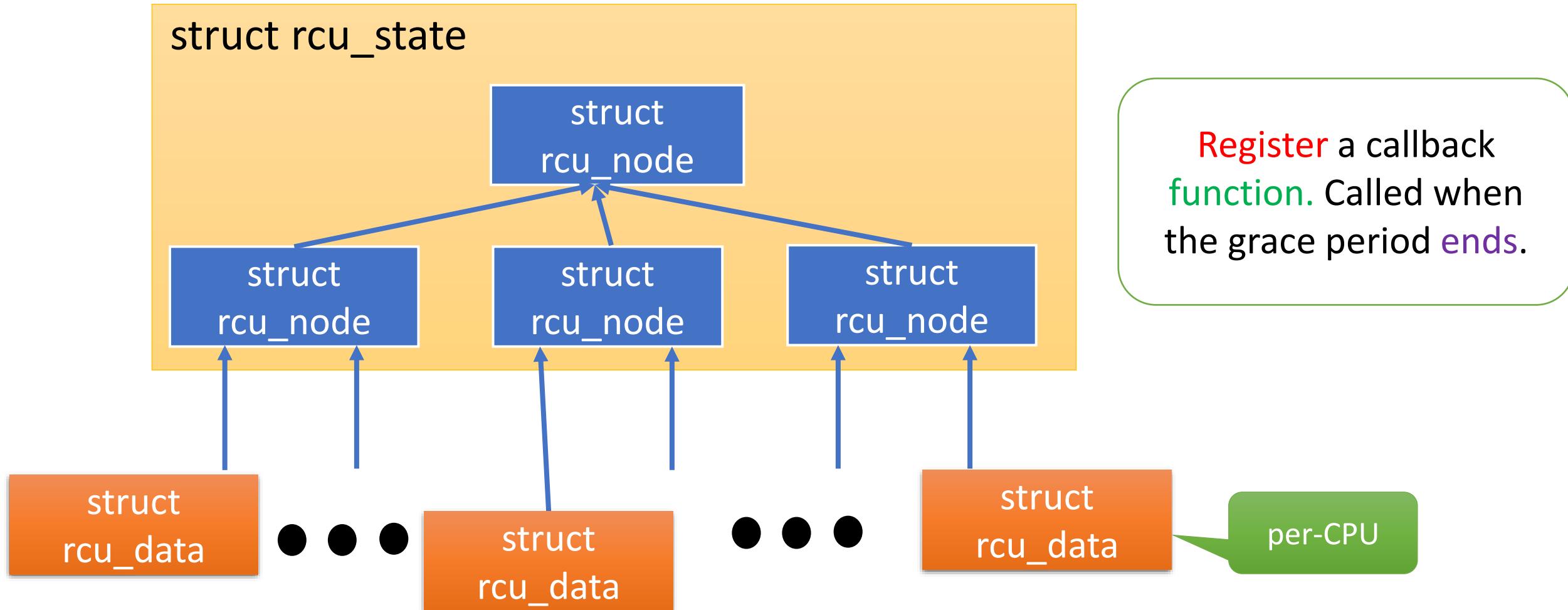
`struct rcu_state`



Underlying storage structure is an array in the *rcu_state* structure. Allows efficient storage and traversal.

RCUs and callbacks

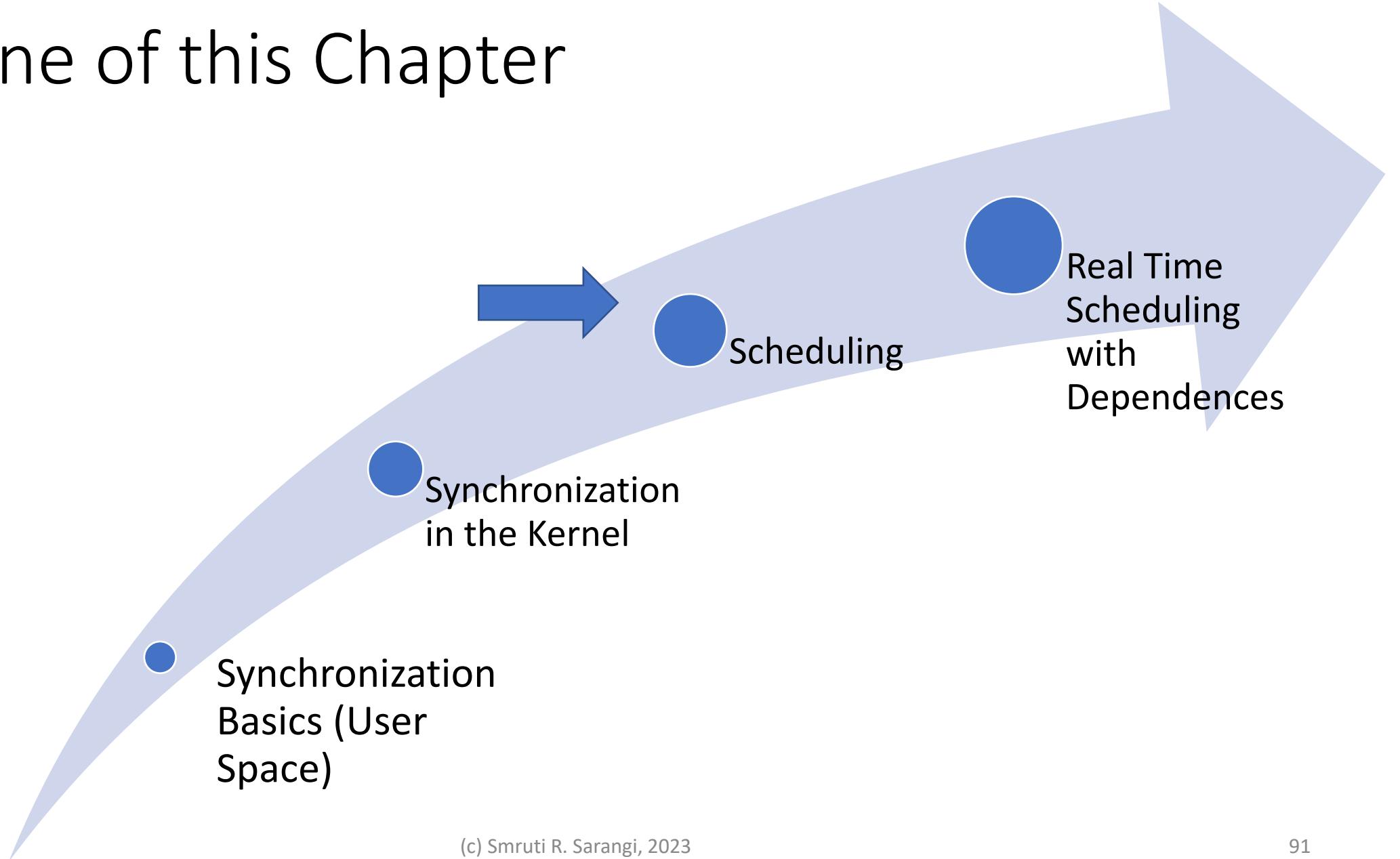
- When all the CPUs **pass** through a quiescent state, the grace period **ends**



Expedited Grace Periods

- Assume that a CPU is not entering a quiescent state
 - A quiescent state can be forced by sending an IPI to it
- Non-preemptable read block
 - The IPI will be serviced after the read block has finished. Its interrupt handler runs in the quiescent state.
- If the read block is preemptable (real-time kernel), then register a callback function that is called once the target CPU reaches a quiescent state
- Preemptible RCU-protected regions cause problems
 - We need to add reference counters and state variables to the `rcu_read_lock` and `rcu_read_unlock` functions. They don't remain ZERO overhead anymore.

Outline of this Chapter



The Problem of Scheduling

Set of tasks/jobs model

Simple variant

Single CPU



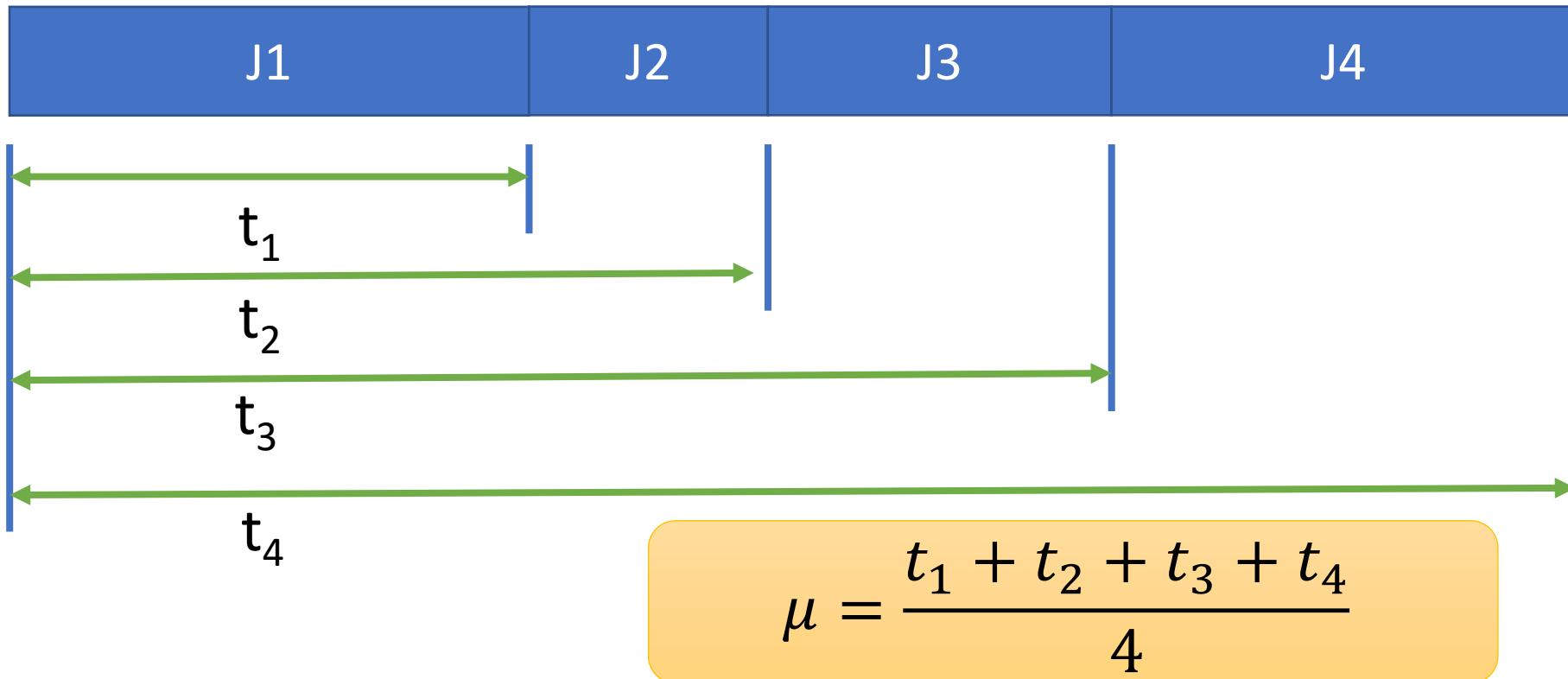
Assume that there is no preemption



A bunch of **tasks** that arrive at the beginning. Each one has a fixed **duration**, which is known in advance (somewhat **impractical** assumption).

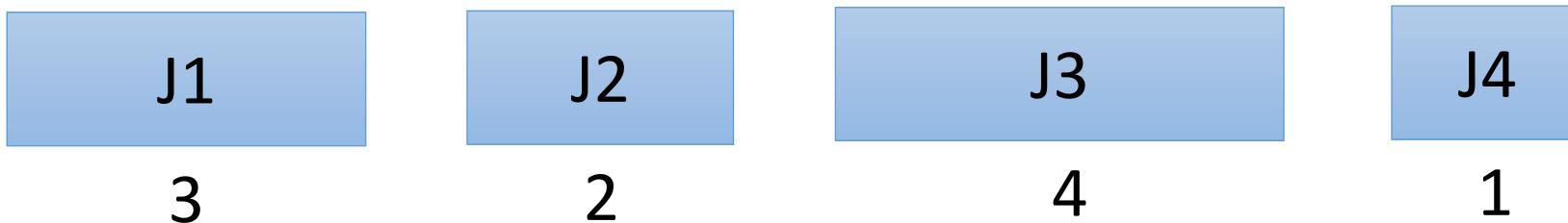
Find different ways of sequencing them.

Metric: Minimize the Mean Completion Time

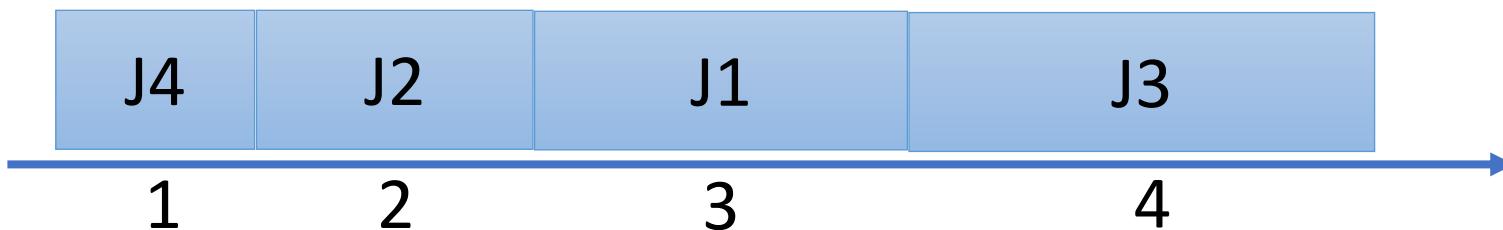


- In this case the **completion** times are t_1 , t_2 , t_3 , and t_4
- It is a more **convenient** tool to create a **formal** model

Example



- Shortest job first (SJF) algorithm



A More Formal Way of Thinking

KSW Model

$\alpha|\beta|\gamma$

The Karger-Stein-Wein Model (KSW model) [1]

α : There are n jobs, $1 \dots n$. The processing time of the j^{th} job is p_j . This parameter characterizes the machine: #cores.

β : Constraints: pre-emption, arrival (release) times of jobs, dependence between jobs, deadlines

γ : Optimality criteria: Mean completion time, Max. completion time, weighted completion time

Class of Scheduling Problems

- For different **values** of $\alpha|\beta|\gamma$, we need to have different **scheduling** algorithms
- Some of them are **optimal** with respect to different **optimality** criteria
- In general in **scheduling** problems:
 - We either have an optimal algorithm. The metric of **interest** is minimized.
 - OR the setting is **NP-complete**
- For NP-complete scheduling problems
 - Need **effective** heuristics.

Mostly either very easy or NP-Complete





How do you prove that a scheduling algorithm A is optimal?

Set up the
problem

Assume it is **not**



Then there must be another **optimal** algorithm A'

Let it take the same **decisions** till a certain point

Prove by
contradiction

Find the first point of **divergence**



Prove that an algorithm A'' is **better** than A' because of the **decision** that it took.

Hence, A' cannot be **optimal**.

Proof that SJF is Optimal for 1|| ΣC_j

- For the **Algorithm A'** (supposedly **optimal**) there must be a **pair** of jobs
 - j and k such that
 - j immediately **precedes** k
 - $p_j > p_k$ (this will not **happen** in algorithm A)
 - p_j **started** at time t
 - Now **exchange** them (result of algorithm A'')
 - Contribution to the completion time of j and k in A': $(t+p_j) + (t+p_j+p_k)$
 - Contribution to the completion time of j and k in A'': $(t+p_k) + (t+p_j+p_k)$
 - Given that $p_k < p_j$, A'' has a **lower** completion time
 - Thus A' cannot be **optimal**.



Contradiction

Weighted Jobs

- Every job has a **weight** w_j
- Let us schedule it in a **non-increasing** order of w_j/p_j
- If $\forall j, w_j = 1$, this is the SJF **algorithm**
- Then for the following this algorithm is **optimal**.

$$1 \mid \mid \sum w_j C_j$$



Same exchange-based argument

Lateness and the Earliest Due Date (EDD) Algorithm/ Earliest Deadline First (EDF) Algorithm

- Assume that jobs are associated with deadlines and distinct arrival times
- Let us define lateness as $\langle \text{completion time} \rangle - \langle \text{deadline} \rangle$
- Let us now solve the problem:

$$1|r_i, dl_i, pmtn|L_{max}$$

- EDF algorithm: Order jobs by non-decreasing deadlines
- Schedule in that order; break ties arbitrarily.
- It is an exact algorithm for this problem



Same exchange-based argument

Preemption and Release Dates

$$1|r_i, pmtn|\Sigma C_i$$

- Jobs arrive at **different** times. Job i arrives at time r_i .
- Preemption is **allowed**
- The most optimal algorithm is **SRTF** → Shortest Remaining Time First algorithm
- Same **exchange-based** argument

Some NP-Complete and Impossibility Results

- The ability to pre-empt jobs when all the jobs are released at $t=0$ does not lead to better schedules ($\sum C_i$ or L_{max}).
- NP-complete Problems
 - $1|r_i|\sum C_i$
 - $1|r_i|L_{max}$
 - $1|r_i, pmtn|\sum w_i C_i$

Look up other variants



What about the fact that it is impractical?

You normally don't know how long a job will take to execute.



However, you can approximate.



- You just need to **predict** CPU bursts. Let t_n be the size of the n^{th} burst. **Predict** it using a time-series **model**.

$$t_n = \alpha t_{n-1} + \beta t_{n-2} + \gamma t_{n-3}$$

Other Algorithms

FIFO

Maintain a **queue** of all processes

At every point, choose the **process** at the **head** of the queue

Problems with FIFO

A long job might delay a lot of small **jobs**: convoy effect

Fairness is an issue

Round robin

Run each job for one **time quanta**

Then **preempt** the task and run a **new** job in round robin order

Priority-based Scheduling

- At any **point** of time, the **highest-priority** task should be running
- If a task with a higher priority arrives, then **swap** out the currently running task
- We can think of **SRTF** and **FIFO** as subsets of this class
 - For SRTF, the priority is the **reciprocal** of the remaining time
 - For FIFO, the priority is the **reciprocal** of the job arrival time
- Recall Linux **priorities** ...



Priority Classes

- Linux has 140 different **priority** levels. It is the job of the **system** administrator and OS to assign **priorities** to running tasks
- The sys admin. can also change the priorities **dynamically**
- Other OSes have different **priority** classes
 - Real time
 - Interactive
 - Regular
 - Batch
- Windows gives a 3X **priority boost** to **foreground** processes

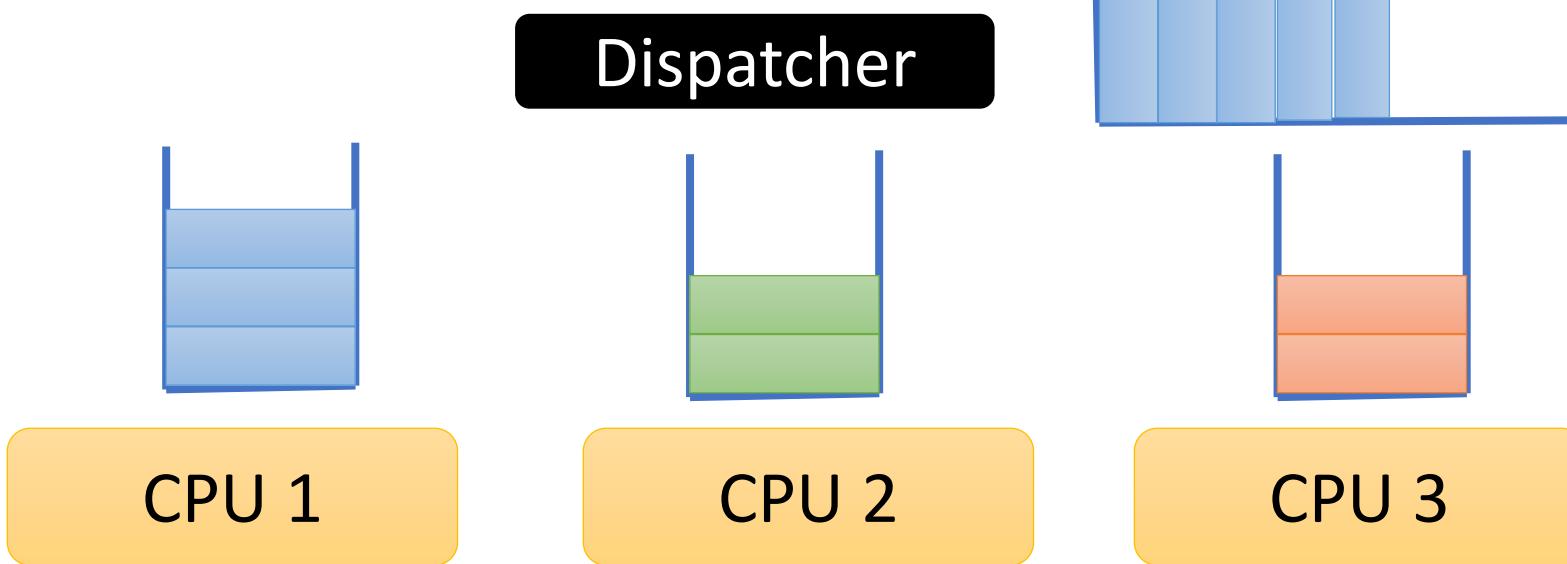
Queue-based Scheduling

Multi-level Feedback Queue



- Different **queues** have different **priorities**. They could have different scheduling **policies** as well.
- Choose the queue with the **highest** priority that is **non-empty**
- Schedule a **process** from it
- Move **processes** between queues based on factors such as interactivity and age

Multi-Processor Scheduling



- Different CPUs **maintain** their queues.
- The dispatcher also has a central **queue**. It picks a task from the queue and sends it to a **CPU-specific** queue.

Some Theoretical Definitions

- The main aim is to minimize the makespan: C_{max}
- Pre-emptible variants are easier to schedule. Non-preemptive versions are mostly NP-complete

Examples

$P|pmtn|C_{max}$

Evenly divide the work among the CPUs

$P||C_{max}$

NP-Hard

The Bin Packing and Partition Problems

- These are some **common** NP-complete problems that are used to prove the NP **hardness** of scheduling problems

Bin Packing



- We have a **finite** number of bins
- Each bin has a **fixed** capacity
- There are n **items**, where each item's **size** is s_i
- We need to pack **items** into bins without exceeding their **capacity**

Each item is a job and each bin is a CPU

yes!

OR

No!

Partition Problem

- Consider a set S that contains a set of numbers
- Find a subset whose sum is equal to a given number T (if it exists)
- This is an NP-complete problem



In a non-preemptive setting, this indicates that it is computationally hard to schedule exactly a certain amount of work on a given CPU.

Heuristics for Multiprocessor Scheduling

List Scheduling

1. Maintain a list of ready jobs
2. Order them according to some priority scheme
3. When a CPU is free, it searches the list and finds a job to execute (it never returns empty handed if the list is non-empty)
 1. It preferably picks the job with the highest priority.
 2. However, this is not strictly necessary
 3. There is thus no idleness
4. The machines are identical (in terms of performance)
5. The scheduling is non-preemptive

Different List Scheduling Strategies

- Order them in **descending** order of processing times
- Find the **critical path** (longest path) in the graph of jobs. Choose **jobs** on the longest path first.
- Order the jobs in **descending** order of **out-degree**



It is often **near-optimal**. The computed **makespan** is often less than 2 times the optimal **makespan**.

Few Important Results in List Scheduling

- Assume, n jobs, m CPUs
- Scheduling problem: $P \parallel C_{\max}$
- **Minimize** the makespan
- We are interested in the ratio: $R = \frac{C_{\max}}{C_{opt}}$

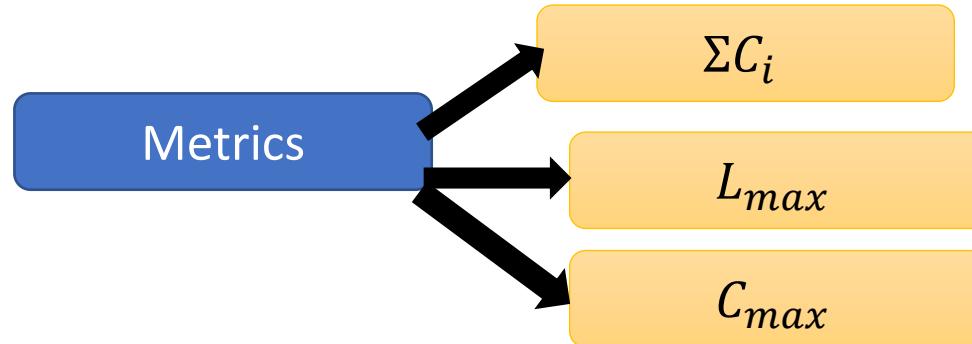
Regardless of the priorities, $R \leq 2 - \frac{1}{m}$

If we sort the jobs in descending order of execution times,

$$R \leq \frac{4}{3} - \frac{1}{3m}$$

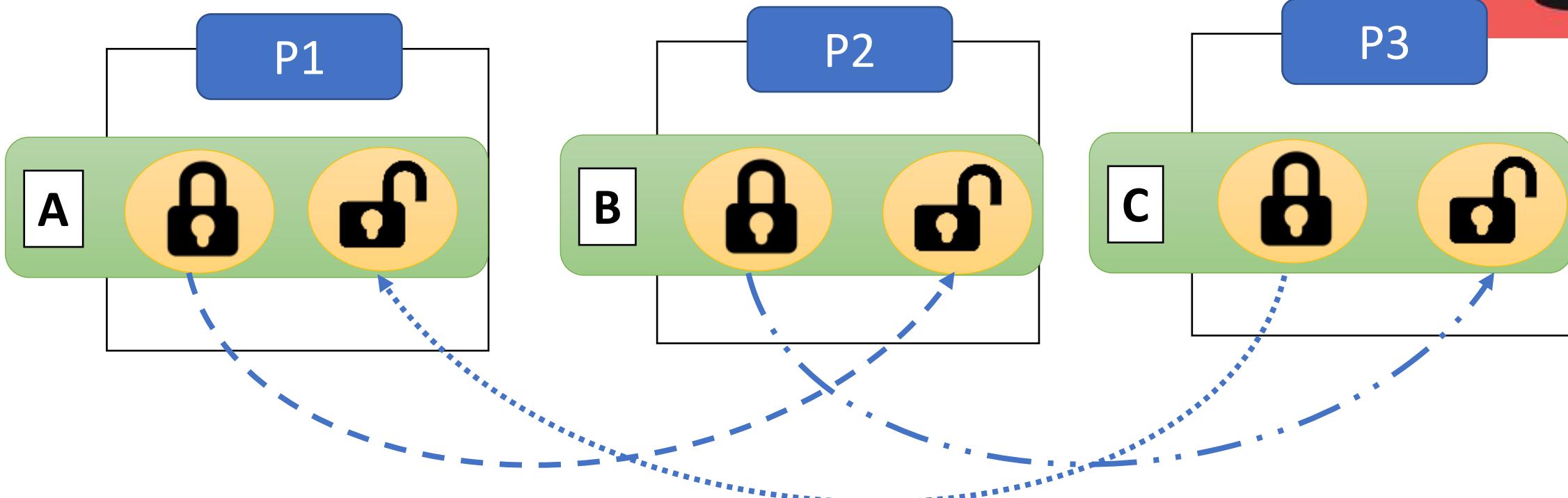
Banker's Algorithm to Create Schedules and Avoid Deadlocks

- We now have a complex schedule with the following attributes:
 - Task **arrival** times
 - Preemption or no preemption
 - **Deadlines**
 - **Priorities**
 - **Dependences**: pre-defined or dynamic (created by lock usage)
 - CPU **affinities**



What about the **correctness** of the **schedule**? Need to avoid **deadlocks**.
Checking for **circular** waits is not enough when there are **multiple** copies
of each resource.

Circular Dependencies aren't a Problem ...



Data Structures in the Banker's Algorithm

n processes

m types of resources

Data structures	Remarks
avlbl [0 ... (m-1)]	$\text{avlbl}[i] = \# \text{copies of the resource } i$
max [0 ... (n-1)][0 ... (m-1)]	Process i can request at the most $\text{max}[i][j]$ copies of resource j (<i>maximum allocation</i>)
acq [0 ... (n-1)][0 ... (m-1)]	Process i has acquired $\text{acq}[i][j]$ copies of resource j (<i>resources acquired</i>)
need [0 ... (n-1)][0 ... (m-1)]	Process i may request $\text{need}[i][j]$ copies of resource j . $\text{acq} + \text{need} = \text{max}$

Check if the system is in a **safe** state

Complexity: $O(mn^2)$



Initialize

`cur_cnt = avlbl`

`done[i] = false (0 ... n-1)`

Find an i
such that

`(done[i] == false) && (need[i][...] ≤ cur_cnt)`

requirement ≤
availability

If such an i **cannot** be found, jump to the last step

Update

`cur_cnt += acq[i][...]`

`done[i] = true`

Assume the process finishes and releases its
resources. Increment `cur_cnt` accordingly.
Elementwise array addition.

Safety
check

If $\forall i$, `done[i] == true` → **Safe**

Else **Unsafe**

All the requests can
be satisfied

Resource Request Algorithm

Initialize

The $req[]$ array is an **array** of resource requests for array i . If $req[j] = k$, then it means that process i needs k **copies** of **resource** j .

Basic check

If $req > need[i][...]$, then the **requirements** cannot be satisfied.
If $req > avlbl$, **wait** until resources are available

Dummy Allocation

$avlbl = avlbl - req$

$acq[i][...] = acq[i][...] + req$

$need[i][...] = need[i][...] - req$

Then check if the state is safe or not. If it is not safe, undo the current allocation and wait

reqs is an $n \times m$ matrix that stores resource requests (for each process)

Deadlock Detection Algorithm

Initialize $cur_cnt = avlbl$
 $\forall i, \text{done}[i] = \text{false}$ if $acq[i][\dots] \neq 0$, else true



Find an i $(\text{done}[i] == \text{false}) \&\& (\text{reqs}[i][\dots] \leq cur_cnt)$
such that If such an i cannot be found, jump to the last step

Can satisfy

Update $cur_cnt = cur_cnt + acq[i][\dots]$
 $\text{done}[i] = \text{true}$

Deadlock If $\forall i, \text{done}[i] == \text{true} \rightarrow \text{NO deadlock}$
check Else **Deadlock**

What happens if you find a deadlock?



Don't allow a deadlock to perform by delaying a request or kill one of the processes involved in a circular wait.

- Don't enter an **unsafe** state
- If we have entered an **unsafe** state, then perform a **deadlock** check when the system does not appear to make any **progress**
- Kill one of the **processes** in the deadlock. Choose a process that has not **executed** for a very long time.

Scheduling in Linux

The *schedule* function



/kernel/sched/core.c

<https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1740572.html>



```
void schedule(void)
{
    struct task_struct *tsk = current;
    sched_submit_work(tsk);
    do {
        preempt_disable();
        __schedule(SM_NONE);
        sched_preempt_enable_no_resched();
    } while (need_resched());
    sched_update_worker(tsk);
}
```

To avoid deadlocks, dispatch work to other kernel threads

Update the status of the worker threads.

The main scheduler function: _schedule

- There are **several** ways that one can enter the *schedule* function
 - A **blocking** call to a mutex, semaphore, etc.
 - The **TIF_NEED_RESCHED** **flag** is set while returning from an interrupt or system call.
 - One process **pre-empts** another one
- Every CPU has a **runqueue**, which is the main data structure that queues tasks
- **struct rq**  /kernel/sched/sched.h
- **Encapsulates** three different **run queues** (one for each scheduler): completely fair scheduler, real time, deadline driven

Notion of Scheduling Classes

Descending order of priority

- **STOP TASK** → Stops everything and runs (like a kernel panic)
- **DL** (Deadline Scheduling)
 - Tasks that need to finish before a deadline.
 - Audio and video encoding
- **RT** (Real Time)
 - SoftIRQ threads
 - Custom software with mission critical requirements
- **Fair** (CFS – completely fair scheduler)
 - Default
- **Idle**
 - Idle process (nothing running basically)

`struct sched_class`

- Defines a bunch of **function** pointers
 - All **scheduling classes** need to provide an implementation for these functions
 - **Examples**
 - `enqueue_task`
 - `dequeue_task`
 - `pick_task`
 - `pick_next_task`
 - `migrate_task_rq`
 - `update_curr`
- Object-oriented languages define child classes.
 - This is C's way of doing the **same**.
 - Just call the **function** pointer

Each Scheduler Class corresponds to a Scheduler

- **Mapping** between *sched_classes* and *files* that implement the corresponding *schedulers*.
- All expose the **same** API



Scheduler	File
Stop Task Scheduler	stop_task.c
Deadline Scheduler	deadline.c
Real-Time Scheduler	rt.c
Completely Fair Scheduler (CFS)	cfs.c
Idle	idle.c

The runqueue



```
struct rq {  
    raw_spinlock_t      __lock;  
  
    unsigned int        nr_running;  
    u64                nr_switches;  
    int                cpu;  
  
    struct cfs_rq       cfs;  
    struct rt_rq        rt;  
    struct dl_rq        dl;  
  
    struct task_struct * __rcu  *curr;  
    struct task_struct *           *idle;  
    struct mm_struct          *prev_mm;  
  
    struct task_struct *core_pick;  
};
```

Lock the runqueue for all operations

Basic stats about the CPU

All three runqueues: one for each scheduling class

Pointers to the current task, idle task, and mm_struct

The task selected for running

schedule → *pick_next_task* → *pick_next_task*

Regular non-real time processes

- If the currently **executing** task is in the CFS class or above and there are **no** tasks in any other **higher** classes
 - Find the next **fair** task (as per CFS)
- Else, iterate through the **classes** in descending order
 - Find the first **eligible** task and **schedule**.
- **Select** the current core
- **Add** to the runqueue
- **Run** the scheduling algorithm of the appropriate scheduling class
- **Effect** context switches (back to user mode or another task)

Relevant fields in *task_struct*



```
struct sched_entity           se;
struct sched_rt_entity        rt;
struct sched_dl_entity        dl;
const struct sched_class     *sched_class;

struct rb_node;
unsigned long                 core_node;
                           core_cookie;
```

Scheduling statistics

Node in an RB tree

Uniquely identifies a
group of tasks

Completely Fair Scheduler



```
struct sched_entity {  
    struct load_weight          load;      /* load balancing*/  
    struct rb_node               run_node;  
  
    u64                          exec_start;  
    u64                          sum_exec_runtime;  
    u64                          vruntime;  
    u64                          prev_sum_exec_runtime;  
    u64                          nr_migrations;  
    struct cfs_rq                *cfs_rq;  
    struct sched_avg              avg;      /* statistics */  
};
```

Key concept: Virtual runtime of a process.
Lower it is, higher the priority



pick_next_task_fair (for CFS Scheduling)

- Update the **current** task
 - **Update** the scheduling statistics
 - Update the virtual runtime of the task
 - Consider the current **execution interval** (**current time** – when the task started **executing last**) (= δ)
 - Let it be **delta** (δ)

$$\delta_{vruntime} = \delta * \frac{weight(nice=0)}{weight(nice)}$$

→ $curr \rightarrow vruntime += \delta_{vruntime}$

}

The increment to the virtual runtime is a function of the priority

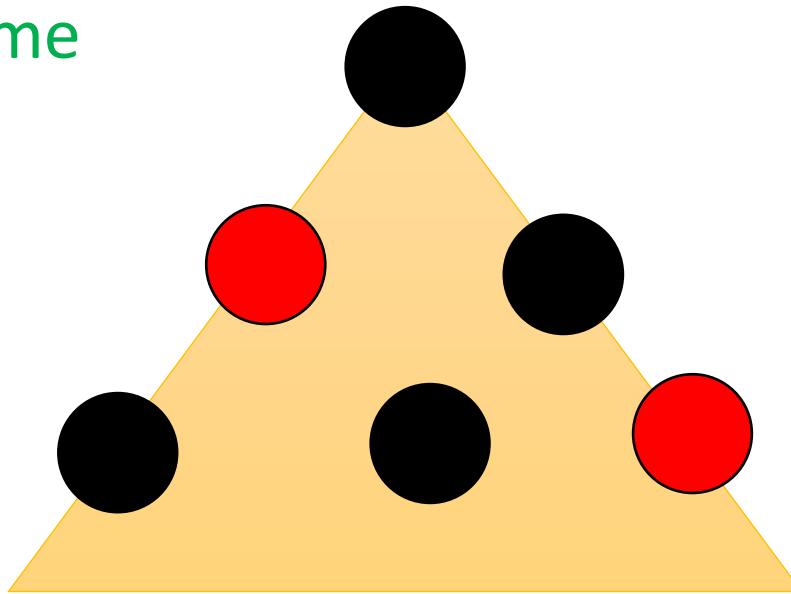
Priority (nice value) → Weight

```
const int sched_prio_to_weight[40] = {  
    /* -20 */     88761,      71755,      56483,      46273,      36291,  
    /* -15 */     29154,      23254,      18705,      14949,      11916,  
    /* -10 */     9548,       7620,       6100,       4904,       3906,  
    /* -5 */      3121,       2501,       1991,       1586,       1277,  
    /* 0 */       1024,       820,        655,        526,        423,  
    /* 5 */       335,        272,        215,        172,        137,  
    /* 10 */      110,         87,         70,         56,         45,  
    /* 15 */      36,          29,          23,          18,          15,  
};
```

Next Steps ...

- Update the `vruntime`

Reduce the virtual
executed time of
the entire `cfs_rq`



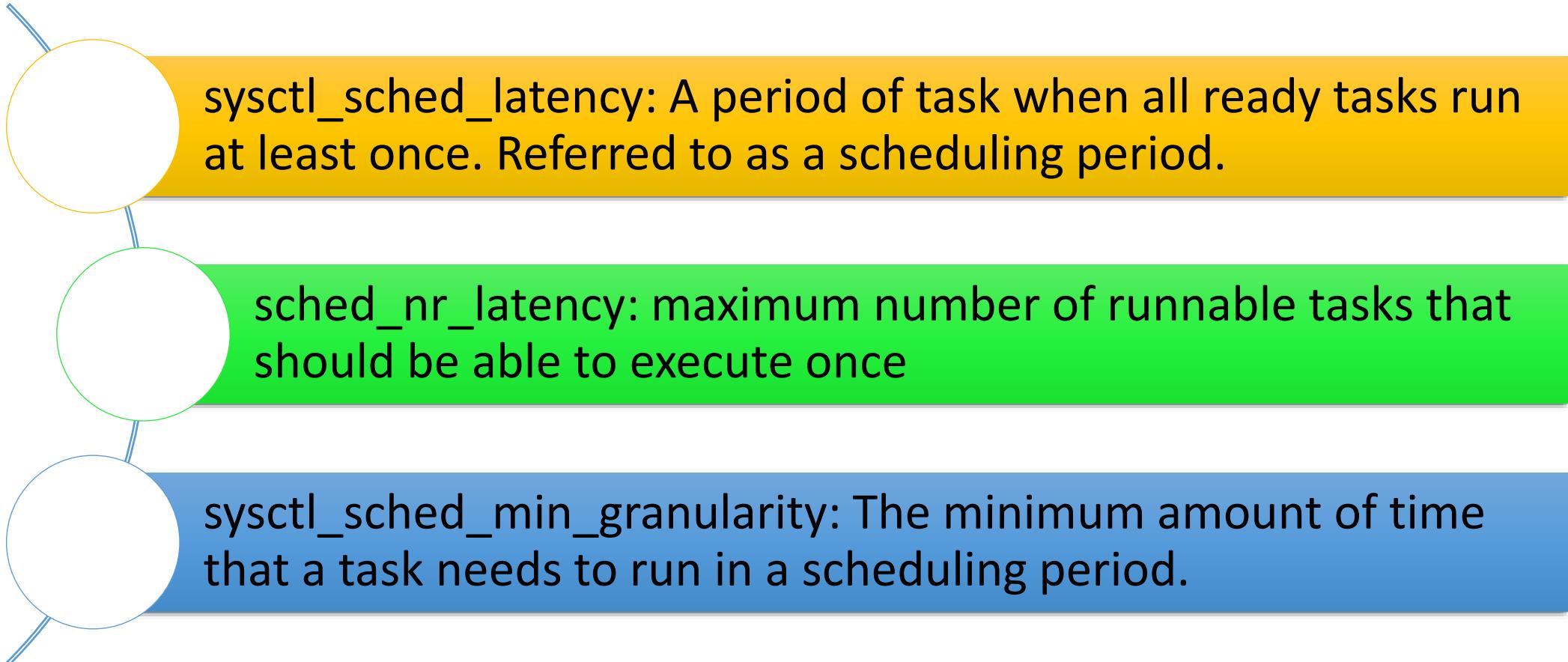
Red-black tree indexed
by the *vruntime*

- A **red**-black tree stores a **list** of tasks indexed by the **vruntime**.
- The **leftmost** **entry** stores the task with the **least** **vruntime**
- Store the current **minimum** in `cfs_rq->vruntime` (`u64`)

`pick_next_entity`

- This is called by *`pick_next_task_fair`*
- **Default:** Choose the **leftmost** entity that has the least vruntime
- **Otherwise:** If the **current** task has executed for more than a **quantum**, then try a pre-emption
 - Choose the **second** leftmost entity
- Each group of **processes** is treated as a single unit for the purposes of allocation of vruntime

Fairness Aspects of CFS



More about the scheduling slice

```
u64 __sched_period(unsigned long nr_running)
{
    if (unlikely(nr_running > sched_nr_latency))
        return nr_running * sysctl_sched_min_granularity;
    else
        return sysctl_sched_latency;
}
```

$$\text{sched_nr_latency} = \frac{\text{sysctl_sched_latency}}{\text{sysctl_sched_min_granularity}}$$

Every task gets the minimum

$$\text{Scheduling Slice} = \text{sched}_{\text{period}} * \frac{\text{weight of the task}}{\text{Sum of weights}}$$

Updating vruntime



The *vruntime* will keep on increasing. Wouldn't that heavily prioritize new tasks?



1. The *cfs_rq* maintains a variable called **min_vruntime** (lowest *vruntime* of all processes)
2. Let *se* be the *sched_entity* that is being enqueued to the *run queue*
3. If an old task is being restored or a new task is being added, then set:
 $se->vruntime += cfs_rq->min_vruntime$
4. This ensures that other tasks have a **fair** chance of getting scheduled
5. Always ensure that all *vruntimes* monotonically increase
(in the *cfs_rq* and *sched_entity* structures)
6. When a child process is created, it inherits the *vruntime* of the parent
7. Upon a task migration or block/unblock, subtract *cfs_rq->vruntime* from the source rq and add the corresponding number for the destination rq (while enqueueing back again in the same or a different rq)



Load Averages and Decaying



Computing the **load average** is important for taking **decisions** such as allocating CPUs and **modulating** the CPU frequency.

Solution:

- Divide the **timeline** into 1 ms intervals
- Let the **intervals** be p_0, p_1, p_2, \dots
- Let u_i denote the **fraction** of time p_i was runnable

$$Load_{avg} = u_0 + u_1 * y + u_2 * y^2 + \dots$$

$$y^{32} = 0.5$$



Deadline Scheduler

- We have an analogous **function**: *pick_next_task_dl*
- **Maintain** a **red-black tree**
 - The task with the **earliest** (smallest) **deadline** is the leftmost
 - Maintain *sched_dl_entity* structures **indexed** by their deadline

```
#define rb_entry(ptr, type, member) container_of(ptr, type, member)
```

Find the *sched_dl_entity* for a given *rb_node* using this **macro** and **return** it. Find the *task_struct* from *sched_dl_entity* using the same **method**.



Realtime Scheduler

- Maintain a queue for each priority level
- Maintain a bit vector: one bit for each queue.
- Find the highest priority queue that has an entry.
- Pick the first task and run it.
- Two policies: FIFO and RR

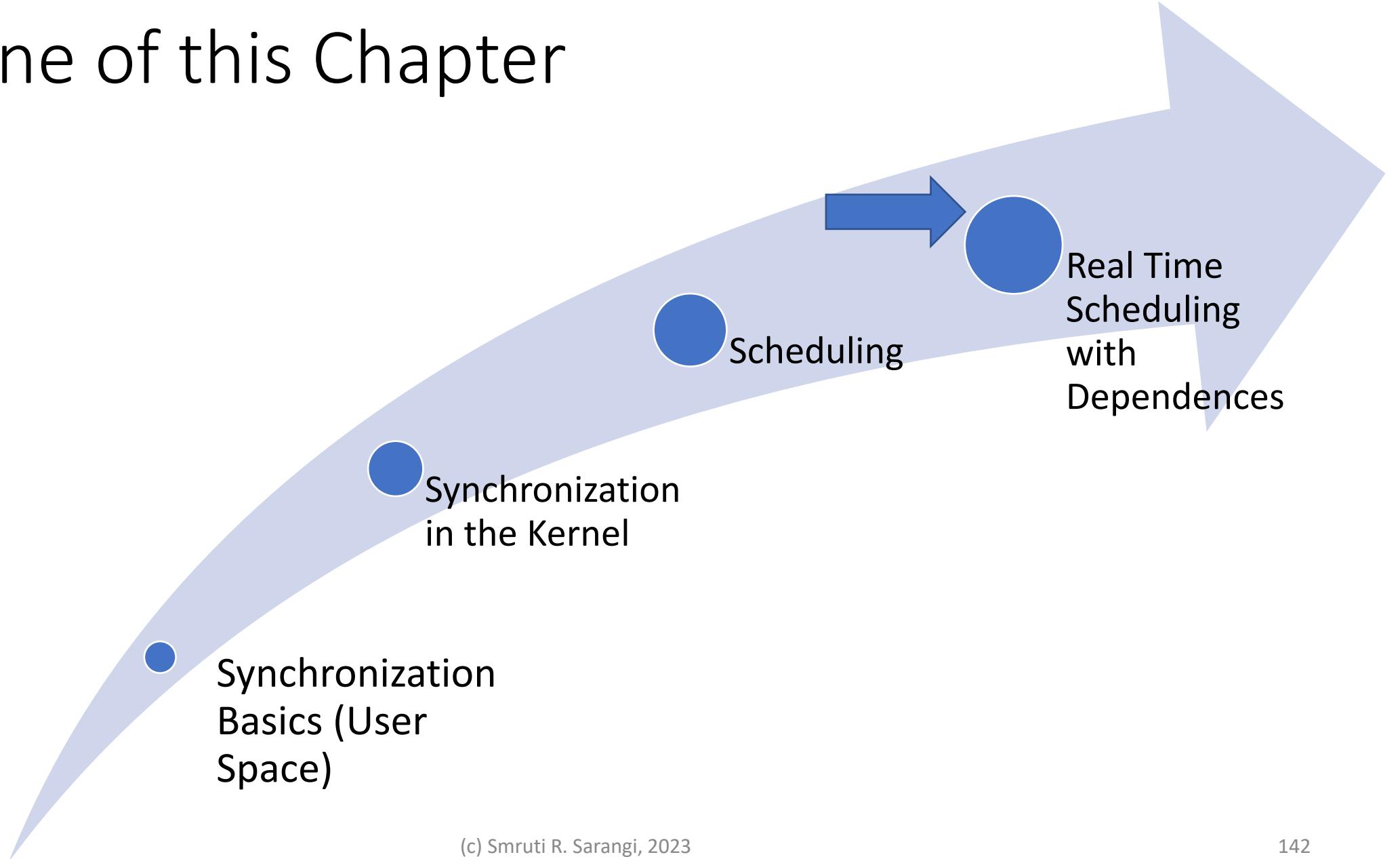
FIFO

- Tasks are queued in FIFO order
- We scan every queue in FIFO order
- The FIFO ordering is always maintained

RR

- If a task has executed for more than its time slice
- Put it at the end of its queue
- Find the next task and mark it ready. Set the re-schedule flag to true.

Outline of this Chapter



Real-Time Systems

Every task has an associated deadline

Soft real-time system

- If a task misses its **deadline**, its utility reduces, does not become zero.

Firm real-time system

- Missing a few deadlines is **acceptable**. Even though the utility is **zero**, we will not have a total failure.

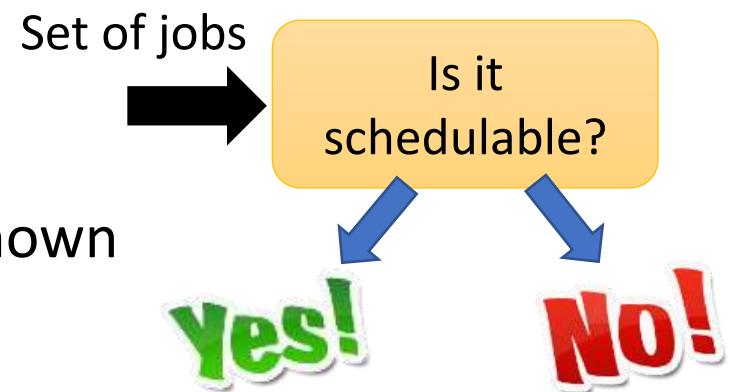
Hard real-time system

- No task can **miss** its deadline

The Notion of Schedulability

- In real-time systems such as **rockets** or **avionics**

- The **tasks** are known
- They are often **periodic**
- The **maximum** processing time is also (more or less) known
- The **deadlines** are known



This makes the **process** of scheduling easier and it is also easier to provide guarantees.



A set of **jobs** is schedulable with an **algorithm A**, if no job misses its **deadline** in the final **schedule**.

EDF (Earliest Deadline First)

Uniprocessors

- We already know that

Priority $\propto 1 / \langle\text{time till deadline}\rangle$



EDF minimizes L_{max}

- This means that if we have a set of **tasks** and associated **deadlines**, if it is **schedulable**, we have $L_{max} \leq 0$
- Furthermore, if it is **schedulable**, EDF will always be able to **find** a feasible schedule.



If pre-emption is enabled, EDF is optimal, and furthermore it provides the test of schedulability

A Stronger Result for EDF on Uniprocessor Machines and Periodic Jobs

Zero overheads for preemption and scheduling

- Let the **duration** of a job be d_i ,
- Let its **period** be P_i
- Let its **deadline** be same as the **period** (needs to **complete** before the next **instance** of the job arrives).

$$U = \sum_{i=1}^n \frac{d_i}{P_i}$$

- If $U \leq 1$, we can always find a **feasible** schedule for it using EDF



The scheduler computes U and if $U \leq 1$, it reports that the set of jobs are not schedulable. The same algorithm is followed if new tasks arrive in the middle of the execution.

Rate-monotonic Scheduling (RMS Scheduling) [2]

- In EDF, priorities are decided dynamically.
 - If a new job arrives with an earlier deadline, then it gets a higher priority.
- Consider a system with periodic jobs where
 - All the priorities are assigned statically (at start-up time)
 - The priority is inversely proportional to the job's period
- A periodic task i has two attributes: P_i (time period) and d_i (execution duration)
 - For each periodic task, the deadline is the time period

$$U = \sum_{i=1}^n \frac{d_i}{P_i}$$

Total utilization

Liu and Layland '73

A set of n jobs is RMS-schedulable if $U \leq n(2^{\frac{1}{n}} - 1)$

For $n=2$, $U \approx 0.83$

For $n = \infty$, $U \approx 0.69$
($\ln 2$)

More about RMS [3]

- The Liu-Layland bound is a very **conservative** bound. Often, we can do **better**.
- It can be proven that the **worst** case occurs when all the **tasks** are in phase (**start** together).
- **Lehoczky Test**
 - Assume the **worst** case (zero phasings)
 - Check whether every **task** is meeting its first **deadline**
 - Then, the we have **schedulability** (even if U exceeds the Liu-Layland bound)

Arrange the tasks
in descending
order of RMS
priority

$$W_i(t) = \sum_{j=1}^i d_j \lceil \frac{t}{P_j} \rceil$$

Find the number of periods within a time interval and account for all the tasks' execution. This is the total load of the first i tasks.

$$L_i(t) = \frac{W_i(t)}{t} \rightarrow L_i = \min_{\{0 < t \leq P_i\}} L_i(t)$$

i^{th} task is schedulable if and only if $L_i \leq 1$

$$L = \max_{\{1 \leq i \leq n\}} L_i \leq 1$$

Necessary and sufficient condition

All tasks are schedulable

DMS: Deadline Monotonic Scheduling [4]

RMS

deadline = period

DMS

deadline \leq period

$$\text{Priority} \propto \frac{1}{\text{deadline}}$$

Every task should be schedulable

$$\frac{I_i}{D_i} + \frac{d_i}{D_i} \leq 1$$

1

D_i is the deadline for task i

2

I_i is the interference that it suffers because of other tasks

Not enough

Exact Algorithm

```
foreach task  $\tau_i$ 
     $t = \sum_{j=1}^i d_j$ 
    cont = true
    while (cont)
        if  $\left( \frac{I_i(t)}{t} + \frac{d_i}{t} \leq 1 \right)$  cont = false
        else  $t = I_i(t) + d_i$ 
        if  $(t > D_i)$  return false;
    }
return true;
```

Interference

$$I_i(t) = \sum_{j=1}^{i-1} \left\lceil \frac{t}{P_j} \right\rceil * d_j$$

Schedulable

Iterate

Not schedulable

Priority Inheritance Protocol (PIP) [5]

- What happens if a **low-priority** process **holds** a resource, and this **stops** high priority processes from making progress? Priority inversion
- A high priority process could be **blocked** for a long time if a medium priority process decides to **run**.

Solution:

Raise the priority of the lock-holding low-priority task to that of the high-priority task (until it **holds** the resource).

Chain blocking



There is a problem if a **high-priority** process needs many **resources** and this **process** has to be carried out for every resource.

Highest Locker Protocol (HLP)

- Let us assume some **facts** are known **a priori**
 - Let $\text{ceil}(\text{resource})$ be the **priority** of the highest-priority process that can possibly **acquire** the resource.
 - Once a process **acquires** a resource, raise its **priority** to $\text{ceil}(\text{resource})$
- The moment a process **acquires** a resource, it cannot be blocked any more by a **low-priority** process.



We cannot have any **deadlocks**.



Quite unfair for **intermediate** priority tasks that can't **run** now.

Priority Ceiling Protocol (PCP)

- Each resource has a **highest ceiling priority**
- **Don't** allocate a **resource**, even if it is free 
- System ceiling (CSC) = **max (ceilings of all resources that are in use)**
- **Resource** grant clause  Does not change the priority
 - Either a **task** holds a **resource** that set the current system ceiling
 - **OR** its **priority** needs to be higher than the **CSC** (also **set** the new CSC in this case)
- **Inheritance** clause Changes the priority
 - The process **holding** a resource inherits the priority of the blocked task, if its priority is lower

Revert the priority accordingly on a
resource release.

Brief Note about PCP

No deadlocks and chain blocking

- Once a task **acquires** a resource, it does not **undergo** any more priority inversions.
- This breaks the **symmetry** and thus **deadlocks** are not possible.

Controlled priority inheritance

- We look at the actual **pattern** of resource usage.
- We don't design a **protocol** assuming the theoretical **worst-case**.

Bibliography

- [1] Karger, David, Cliff Stein, and Joel Wein. "Scheduling algorithms." *Algorithms and Theory of Computation Handbook: special topics and techniques*. 2010. 20-20.
- [2] Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, Liu and Layland, Journal of the ACM, 1973
- [3] The Rate Monotonic Scheduling Algorithm: Event Characterization and Average Case Behavior, Lehoczky, Sha, and Ding. IEEE Real Time Systems Symposium, 1989
- [4] Deadline Monotonic Scheduling, Audsley, 1990
- [5] Real-Time Systems: Theory and Practice, Rajib Mall, 2006. Pearson Publishers.



srsarangi@cse.iitd.ac.in

t h a n k y o u

