CS 477
Advanced Operating System

Tutorial 04: Synchronization

Today's Tutorial

- Why synchronization?
- Race Condition Example
- 4 kernel synchronization mechanisms
  - Spinlock
  - RWLock
  - Seqlock
  - RCU

Why do we Need Synchronization?

- **Critical section** (also called critical region)
  - Code paths that access and manipulate shared data
  - Must execute **atomically** without interruption
  - Should not be executed in parallel -> must be always sequential!
- **Race condition**
  - Two threads concurrently executing the same critical region -> Bug!

Concurrency in the Kernel vs in User Space

- More scenarios in the kernel than userspace
- Similar to userspace:
  - **Symmetric Multiprocessing**: 2+ processors running at the same time contend on data
  - **Preemption**: One process modifying data is preempted by another modifying same data
- Unique to kernel:
  - **Interrupts:** async *non-sleepable* callback
  - **Softirq/tasklet**: deferred version of interrupt

Kernel Synchronization Examples

- Task Management: tasklist_lock (rwlock)
  - write_lock() held on fork() when adding created process to task list: https://elixir.bootlin.com/linux/v6.12.6/source/kernel/fork.c#L2522

- File System: file descriptor table (spinlock + RCU)
  - Read file from fdt w/rcu_read_lock(): https://elixir.bootlin.com/linux/v6.12.6/source/fs/file.c#L898
  - Add file to fdt w/rcu_read_lock() + spinlock(): https://elixir.bootlin.com/linux/v6.12.6/source/fs/file.c#L588

- Memory Management: mprotect (rwsem)
  - Acquire write lock on mm_struct when finding VMA: https://elixir.bootlin.com/linux/v6.12.6/source/mm/mprotect.c#L740

Experimental Setup

- Runs in the kernel using kernel modules (Tutorial 03)
  - On init, spawns experimental threads
  - On exit, signals to threads to report data
- 10 kernel threads
  - `num_readers` readers
  - `(10 - num_readers)` writers
  - Each perform `ops_per_thread` operations

```
#define MAX_CONTENDING_THREADS 10

struct task_struct
*threads[MAX_CONTENDING_THREADS];

module_param(ops_per_thread, int, 0664);
// default 1000000

module_param(num_readers, int, 0664); //
default 5
```

Race Condition

```
read_function() {
    unsigned int arr[ops_per_thread];
    int i = 0;
    while (i < ops_per_thread &&
    !kthread_should_stop()) {
        arr[i] = counter;
        i++;
     }
}
```

```
writer_function() {
    int i = 0;
    while (i < ops_per_thread &&
    !kthread_should_stop()) {
        counter++;
        i++;
     }
}
```

https://github.com/sidharth-sundar/advos-25-demo/blob/main/Week-4-Demo/race/sync_race.c

- Note: in the source code, `arr` is allocated on the heap w/kmalloc, not the stack. Variable length arrays are highly discouraged in the kernel as kernel thread stack size is small. The code in this slide is for legibility.

Spinlock Usage

- Most common mechanism to avoid race conditions
- If lock is held by another thread, current thread **spins** (repeatedly checks the lock) until it becomes available
- Code between spin_lock and spin_unlock is perceived as atomic to others spinning on same lock
- Not sleepable -> can be used in interrupt context (there are special APIs)

Spinlock

```
#include <linux/spinlock.h>
DEFINE_SPINLOCK(counter_lock);
```

```
read_function() {
    unsigned int arr[ops_per_thread];
    int i = 0;
    while (i < ops_per_thread &&
    !kthread_should_stop()) {
        spin_lock(&counter_lock);
        arr[i] = counter;
        spin_unlock(&counter_lock);
        i++;
     }
}
```

```
writer_function() {
    int i = 0;
    while (i < ops_per_thread &&
    !kthread_should_stop()) {
        spin_lock(&counter_lock);
        counter++;
        spin_unlock(&counter_lock);
        i++;
     }
}
```

https://github.com/sidharth-sundar/advos-25-demo/blob/main/Week-4-Demo/spinlock/sync_spinlock.c

RWLock Usage

- Useful if you can cleanly separate accesses into readers + writers
  - E.g. list search (read) vs insertion (update)
  - In our example, this is counter read (read) vs counter increment (update)
- Readers can concurrently access critical section
- Only one writer can access critical section at a time
- *Readers block writers, writers block both readers and writers*
- Linux's implementation is read-favored

RWLock

```
#include <linux/spinlock.h>
DEFINE_RWLOCK(counter_lock);
```

```
read_function() {
    unsigned int arr[ops_per_thread];
    int i = 0;
    while (i < ops_per_thread &&
    !kthread_should_stop()) {
        read_lock(&counter_lock);
        arr[i] = counter;
        read_unlock(&counter_lock);
        i++;
     }
}
```

```
writer_function() {
    int i = 0;
    while (i < ops_per_thread &&
    !kthread_should_stop()) {
        write_lock(&counter_lock);
        counter++;
        write_unlock(&counter_lock);
        i++;
     }
}
```

https://github.com/sidharth-sundar/advos-25-demo/blob/main/Week-4-Demo/rwspinlock/sync_rwspinlock.c

Sequential Locks (seqlock) Usage

- RWLock is a **pessimistic** mechanism for allowing concurrent readers/exclusive writers
  - Pessimistic: Ensure mutual exclusion **prior** to executing the critical section

- Seqlock allows for **optimistic readers**
  - Reader performs its computation and checks if it's allowed to continue **post-hoc**

- Writers block readers, but readers don't block writers

Seqlock Under The Hood

```
typedef struct {
    seqcount_spinlock_t seqcount; // equiv. to unsigned for our purposes
    spinlock_t lock;
} seqlock_t;
```

- Readers read seqcount at start of critical section
- Readers reread seqcount at end
  - if start != end, or start is odd, retry
    - Why retry on odd?
- Writers acquire spinlock and increment seqcount at start of critical section
- Writers increment seqcount again and release spinlock at end of critical section

Seqlock

```
#include <linux/seqlock.h>
DEFINE_SEQLOCK(counter_lock);
```

```
read_function() {
    unsigned int arr[ops_per_thread];
    int i = 0;
    while (i < ops_per_thread &&
    !kthread_should_stop()) {
        unsigned seq;
        do {
            seq = read_seqbegin(&counter_lock);
            arr[i] = counter;
        } while (read_seqretry(&counter_lock,
        seq));
       i++;
     }
}
```

```
writer_function() {
    int i = 0;
    while (i < ops_per_thread &&
    !kthread_should_stop()) {
        write_seqlock(&counter_lock);
        counter++;
        write_sequnlock(&counter_lock);
        i++;
     }

}
```

https://github.com/sidharth-sundar/advos-25-demo/blob/main/Week-4-Demo/seqlock/sync_seqlock.c

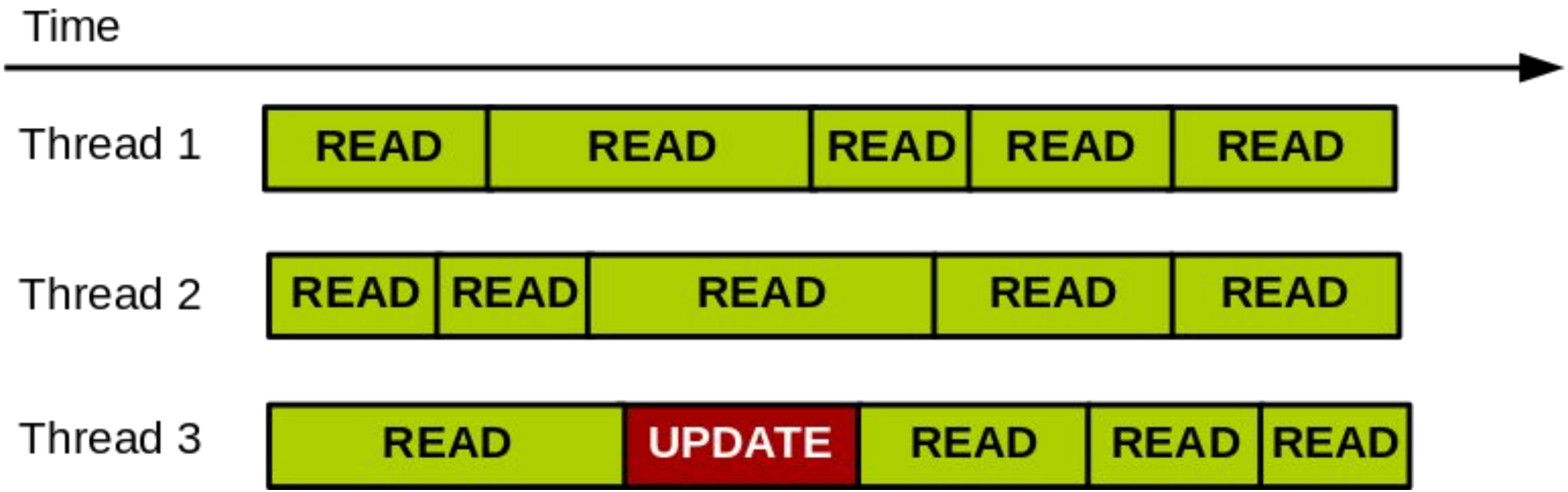
RCU Usage

- All prior locks guarantee **mutual exclusion**, strictly ordering reads w.r.t. writes

- RCU Checklist:
  1) **Readers can read stale data**
  2) Read-mostly data
  3) Low write performance is fine
  4) **Updates can be performed with a single atomic pointer swap**
     a) Individual struct *s
     b) Singly-linked list

- Used in page cache (xarrays), dentry cache (RCU walk), DNS Name Databases

RCU Diagram


Time
Thread 1
READ
READ
READ
READ
READ
Thread 2
READ
READ
READ
READ
READ
Thread 3
READ
UPDATE
READ
READ
READ

RCU Read Critical Section

```
#include <linux/rcupdate.h>
unsigned int *counter;
DEFINE_SPINLOCK(counter_lock);

read_function() {
    unsigned int arr[ops_per_thread];
    int i = 0;
    while (i < ops_per_thread && !kthread_should_stop()) {
        rcu_read_lock();
        arr[i] = *rcu_dereference(counter);
        rcu_read_unlock();
        i++;
     }
}
```

https://github.com/sidharth-sundar/advos-25-demo/blob/main/Week-4-Demo/rcu/sync_rcu.c

RCU Write Critical Section

```
writer_function() {
    int i = 0;
    while (i < ops_per_thread && !kthread_should_stop()) {
        unsigned int *new_counter = kmalloc(sizeof(*counter), GFP_KERNEL));
        spin_lock(&counter_lock);
        unsigned int *old_counter = rcu_dereference(counter);
        *new_counter = *old_counter + 1;
        rcu_assign_pointer(counter, new_counter);
        spin_unlock(&counter_lock);
        synchronize_rcu();
        kfree(old_counter);
        i++;
     }
}
```

https://github.com/sidharth-sundar/advos-25-demo/blob/main/Week-4-Demo/rcu/sync_rcu.c

Additional Reading/Work

- Good Readings/exercises
  - **Run the code on your own machine. Understand the APIs + different mechanisms' performance based on read/write load**
  - Lock types and their rules
  - What is RCU, Fundamentally
  - RCU Kernel Documentation
- Hard Readings
  - Preemptible RCU
  - RCU-walk and REF-walk
- Research Extensions of RCU
  - Read-Log-Update
  - Multi-Version Read-Log-Update
- Even more RCU resources
  - RCU Publications Google Doc (not mine)