

# LDD3 Ch. 5 Concurrency & Race Conditions

ECEN 427

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

# Learning Outcomes



- Explain **race conditions**, **critical sections** and **atomicity**, and why passing tests don't show code is race-free
- List the ways driver code can run concurrently with itself, and identify which shared state needs a lock
- Choose a mutex or a spinlock based on whether the code can sleep or runs in an ISR
- Protect state shared by a driver’s syscall functions and its ISR

# Dining Philosophers Problem



- Based on a 1965 exam exercise by Edsger Dijkstra

- Five philosophers sit at a round table, with one fork between each pair
- To eat, a philosopher needs both neighboring forks
- Philosophers eat, then put down their forks and think, then repeat

- Philosopher = thread
- Fork = a shared resource (variable / data)



# Example Code



- https://github.com/byu-cpe/ecen427_student/tree/main/examples/dining_philosophers

```
void pick_up(int fork, int id) {

  // wait until the fork is free
  while (fork_owner[fork] != FREE)
    ;

  fork_owner[fork] = id;    // take it

}
```

# Vocabulary



- **Race condition:** the result depends on how threads happen to interleave
- **Critical section:** code that touches shared state and must not overlap with other code touching it
- **Atomic:** happens as one indivisible step. "Check, then take" is not atomic.

# "But it worked when I tested it"



```
taskset -c 0 ./philosophers_unsync     # one core

taskset -c 0,1 ./philosophers_unsync   # two cores
```

- On one core, the race window is a couple of instructions, and the thread is almost never preempted inside it
- On two cores, threads really do run at the same time
- If your tests pass, that doesn't show the code is correct

# What is a lock?



- The bug: "is it free?" and "take it" are two separate steps
  - Another thread can get in between

- A **lock** (or **mutex**, short for *mutual exclusion*) combines them into one atomic step:
  - **lock:** if free, take it; otherwise wait until it is. Only one thread ever holds it.
  - **unlock:** release it, letting one waiting thread take it

```
pthread_mutex_t m;
pthread_mutex_init(&m, NULL);
```

Declaration / Initialization

```
pthread_mutex_lock(&m);
/* critical section */
pthread_mutex_unlock(&m);
```

Using the lock in each thread

# Fix: a lock per fork



```
int fork_owner[N];
pthread_mutex_t fork_lock[N];
```

```
void pick_up(int fork, int id) {
  while (!try_pick_up(fork, id))
    ; // fork is busy, try again
}
```

```
void put_down(int fork) {
  pthread_mutex_lock(&fork_lock[fork]);
  fork_owner[fork] = FREE;
  pthread_mutex_unlock(&fork_lock[fork]);
}
```

```
bool try_pick_up(int fork, int id) {
  bool got_it = false;

  pthread_mutex_lock(&fork_lock[fork]);
  if (fork_owner[fork] == FREE) {
    fork_owner[fork] = id;
    got_it = true;
  }
  pthread_mutex_unlock(&fork_lock[fork]);

  return got_it;
}
```

# Deadlock



- Suppose everyone picks up their left fork first
- Then everyone holds one fork and waits forever for the second

- Our code takes the **lower-numbered fork first**:

```
if (second < first) { first = second; second = id; }
```

- That's a global lock order to prevent deadlock

# Concurrency in the Kernel

# Why the kernel is worse



Your driver code can run concurrently with itself because of:

- Multiple processes calling write() at the same time
- Multiple threads sharing one file descriptor
- SMP: the PYNQ's Zynq has two Cortex-A9 cores
- Kernel preemption: the scheduler can switch processes in the middle of your driver code, not just between syscalls
- Interrupts: your ISR can run in the middle of anything

# LDD3's ground rules



1. Avoid shared state where you can (globals are the usual source)

2. Any state you do share must be protected so that only one thread touches it at a time

3. Don't make an object visible to the kernel (setting up character device, interrupt handler, etc.) until it is fully set up

# Lab 5: who touches what?



| Field | write() | ISR | read() | ioctl() |
|---|---|---|---|---|
| sound_file[] | writes | reads | | |
| num_samples | writes | reads | | |
| current_idx | writes | reads/writes | | |
| is_playing | writes | writes | reads | |
| is_looping | | reads | | writes |

# What we did in the lab



```
static ssize_t audio_write(...) {
  disable_intr();

  audio.num_samples = count / 4;
  audio.current_idx = 0;
  copy_from_user(sound_file,…,…);
  audio.is_playing = true;

  enable_intr();

  return count;
}
```

**Why this usually works in Lab 5:**

- On one core with one writer, it would be fully correct
- On the PYNQ's two cores, a race needs bad timing: the ISR has to be running on the other core at the moment write() starts
- Only one user space program writes to the device
- When it does fail, it’s usually subtle: a glitch at the start of a clip, which is easy to miss

# What we did in the lab

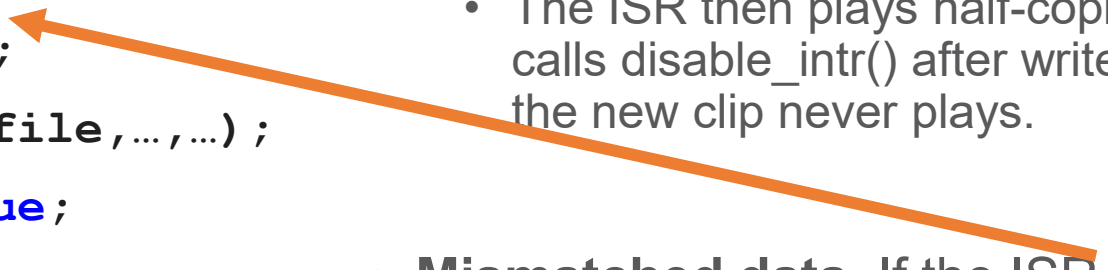


```
static ssize_t audio_write(...) {
  disable_intr();


  audio.num_samples = count / 4;
  audio.current_idx = 0;
  copy_from_user(sound_file,…,…);
  audio.is_playing = true;


  enable_intr();


  return count;
}
```



## What can go wrong?

- **The ISR is already running on the other core.**
  - disable_intr() stops *new* interrupts but doesn't wait for one in progress.
  - The ISR then plays half-copied data, or calls disable_intr() after write() re-enabled interrupts, and the new clip never plays.
- **Mismatched data.** If the ISR runs **here**, it will see the new sample size, but old idx. Could be a problem.
- **Two writers.** Two processes, or two threads sharing an `fd`, both `copy_from_user` into the same buffer.

# Mutexes (sleeping locks)



- A waiter sleeps while waiting for the lock
- Process can sleep while holding lock

- Never call this in an ISR
  (you should never sleep in an ISR)

```
struct mutex lock;
mutex_init(&lock);



mutex_lock(&lock);
/* critical section: may sleep */
mutex_unlock(&lock);
```

Textbook is out of date:

- LDD3 uses `struct semaphore`
- Modern kernels use `struct mutex`

# Spinlocks



- A waiter busy-waits for the lock
- Works in any context, including an ISR

- Rules:
  - **Never sleep while holding one**
    - copy_from_user
    - kmalloc(GFP_KERNEL)
    - mutex_lock
  - Hold it for as short a time as possible

```
spinlock_t lock;
spin_lock_init(&lock);

spin_lock(&lock);
/* critical section */
spin_unlock(&lock);
```

Waiting or holding turns off preemption (switching to another process), **but not interrupts**

- What if an interrupt handler wants a lock its own CPU already holds?

# Spinlocks + interrupts → spin_lock_irqsave



- Suppose a driver's write() and its ISR share one spinlock
  - write() on CPU 0 calls spin_lock → gets the lock
  - The device interrupts CPU 0 → CPU 0 jumps into the ISR
  - ISR calls spin_lock → lock is held, so it spins
  - The lock holder is write(), but it can't resume until the ISR returns
- **Deadlock:** CPU 0 spins forever, waiting on itself

- **Fix:** outside the ISR, take the lock with spin_lock_irqsave
  - Turns off interrupts *on this CPU* while you hold the lock
  - The ISR can still fire on another CPU; it just spins briefly (normal case)

- **Inside the ISR,** plain spin_lock is fine: interrupts are already off there

```
// track current interrupt state
unsigned long flags;


spin_lock_irqsave(&lock, flags);
...
spin_unlock_irqrestore(&lock, flags);
```

# Mutex vs Spinlock for Lab 5



| | Mutex | Spinlock |
|---|---|---|
| What does the waiter do? | sleeps | spins |
| Use in ISR? | no | yes |
| Can sleep while holding? | yes | no |

Lab 5 needs both:

- write() can go to sleep while accessing sound_file (in copy_from_user) → Mutex
- Need to lock other variables that ISR accesses, but ISR can't sleep → Spinlock

**Mutex:** for keeping two write() calls from overlapping

- sound_file

**Spinlock:** for sharing state between write(), read(), ioctl() and the ISR

- num_samples
- current_idx
- is_playing
- is_looping

# Lab 5: declare/init locks



```
struct audio_device {
  ...
  spinlock_t lock;            // protects num_samples, current_idx,
                              //   is_playing, is_looping

  struct mutex write_lock;  // one write() at a time, protects sound_file
};

// in probe: right after the memset, before cdev_add and request_irq
spin_lock_init(&audio.lock);
mutex_init(&audio.write_lock);
```

# Lab 5: write()



```
static ssize_t audio_write(struct file *fp, const char __user *buff, size_t count, loff_t *offp) {
  unsigned long flags;

  // one writer at a time
  mutex_lock(&audio.write_lock);

  // waits out a running ISR, then stop
  spin_lock_irqsave(&audio.lock, flags);
  audio.is_playing = false;
  disable_intr();
  spin_unlock_irqrestore(&audio.lock, flags);

  // may sleep: no spinlock held
  if (copy_from_user(...)) {
    mutex_unlock(&audio.write_lock);
    return -EFAULT;
  }

  // start new clip
  spin_lock_irqsave(&audio.lock, flags);
  audio.num_samples = count / 4;
  audio.current_idx = 0;
  audio.is_playing = true;
  enable_intr();
  spin_unlock_irqrestore(&audio.lock, flags);

  // next writer may enter
  mutex_unlock(&audio.write_lock);
  return count;
}
```

# Lab 5: ISR



```
irqreturn_t audio_isr(int irq, void *dev_id) {
  spin_lock(&audio.lock);

  ... existing body, with every `return` changed to `goto out` ...

out:
  spin_unlock(&audio.lock);
  return IRQ_HANDLED;
}
```

- In `read()` and `ioctl()`, take the same lock with `spin_lock_irqsave`
- In `read()`, copy `is_playing` to a local under the lock, then `copy_to_user` after unlocking
- Lock hold time is the whole ISR, so `write()` may spin for that long.

# Summary



- A race needs shared state plus two things that can run at once.
  - A driver has four: other callers, the other core, preemption, and its own ISR
- Passing tests don't prove there's no race. Reason about which code touches each piece of shared state
- Use a mutex if the code may sleep, and a spinlock if an ISR shares the data
- Never sleep while holding a spinlock. Outside the ISR, use spin_lock_irqsave
- Take locks in one fixed order everywhere to avoid deadlock (like the forks)