Locking

Concurrency Aspects of Project 2

- Proc is reading shared data
- New system calls are updating shared data
- Elevator scheduler is updating shared data
-
- Examples of race conditions
  - Passengers may appear on a floor at the same time the elevator does
  - The elevator might update it’s state in the middle of /proc/elevator being read
- Need to protect all shared data
- How do you guarantee correctness?
  - Lock access to the shared data

Global vs Local

- Global data is
  - Declared outside of the functions
  - Before any function that uses it
  - Often required for kernel programming
  - Very sensitive to concurrency issues
- Local data is
  - Declared within a functions
  - Sensitive to concurrency when
    - It depends on global data
    - Parallel access to the function is possible
  - Carefully consider whether they need to be synchronized

Synchronization Primitives

- Atomic functions
- Spin locks
- Semaphores
- Mutexes

- You can use any of these but I recommend mutexes

Mutexes

- MUTual Exclusion
- Based on semaphores
- States
  - Locked
  - Unlocked
- Only one thread may hold the lock at a given time

Declare and Initialize

```
#include <linux/mutex.h>
```

- Header file

```
struct mutex my_mutex
```

- Declaration

```
mutex_init(&my_mutex)
```

- Call before using to setup

```
mutex_destroy(&my_mutex)
```

- Call when done to cleanup

Locking and Unlocking

`mutex_lock(&my_mutex)`

- Waits indefinitely for the lock

`mutex_lock_interruptible(&my_mutex)`

- Locks so long as it is not interrupted by a signal
- Returns 0 if succeeded, <0 if interrupted
- Preferred as it helps keep the kernel from deadlocking

`mutex_unlock(&my_mutex)`

- Guarantees that the mutex is unlocked

Examples

- Example6
  - A thread and proc read both access shared state
  - Thread constantly updates state without locking
  - proc shows that values in array are inconsistent
- Example7
  - Same as example6 but uses locking
  - Shows values in array are always the same when read