Threads and Concurrency

ECE 373

Threads of Execution

- Thread – smallest unit of processing that can be scheduled by the OS
  - UNIX-like systems are process oriented
  - WinNT-like systems are thread oriented

- Single CPU core
  single line of instructions at any one time
- Multiple CPU cores
  multiple simultaneous threads of execution

Thread Sources

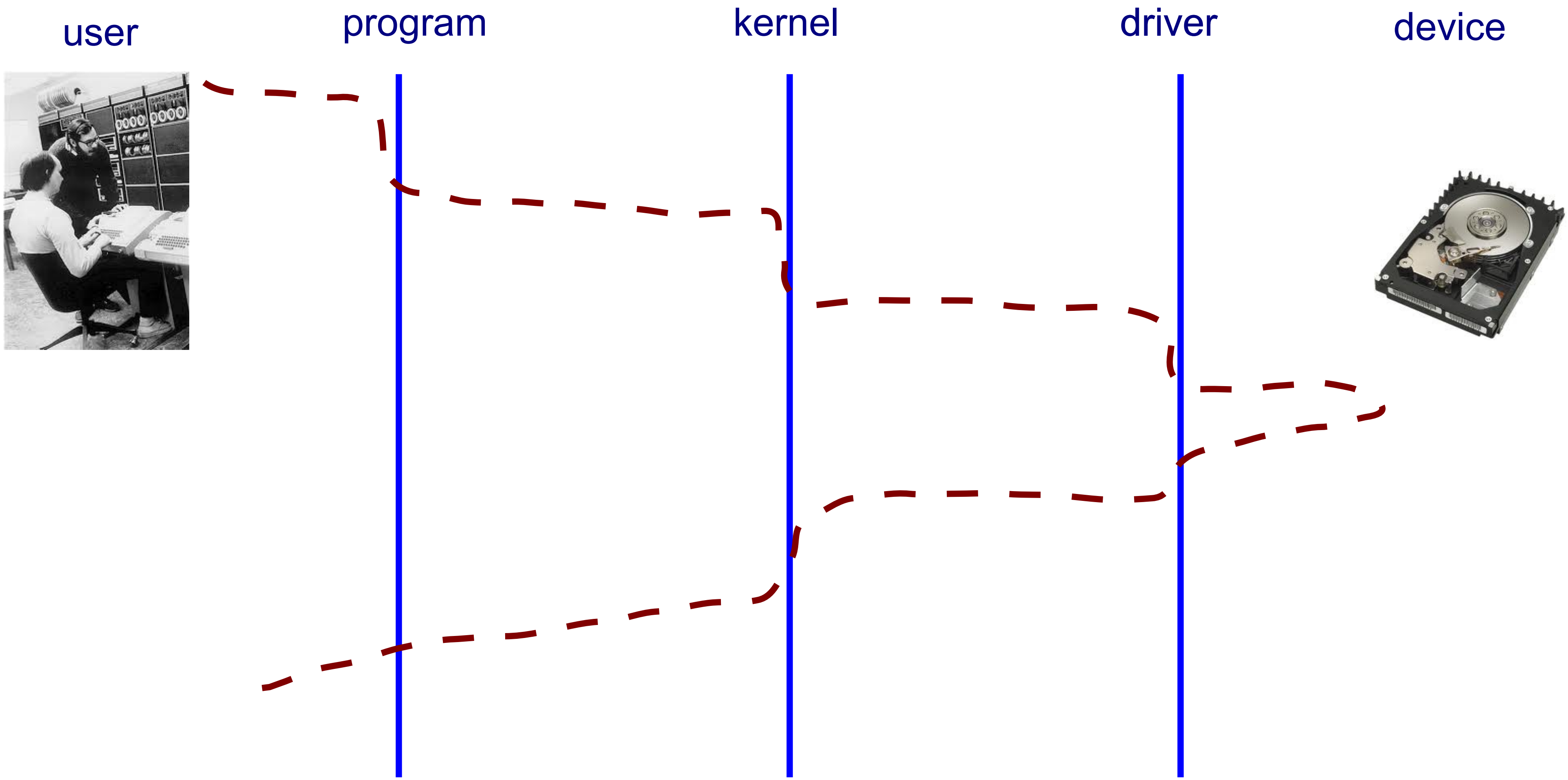
- Kernel threads
- User processes
- Workqueue threads
- Timer callbacks
- Interrupt handlers

Threads

- Kernel threads
  - Jobs the kernel itself is doing for internal [projects]
  - All have access to the same kernel data
- User context
  - Threads running user jobs, might be running kernel code to service system calls from user code
  - Collected into specific user processes, see only the individual process space data
- Interrupts
  - Not really full threads, but in the mix
- See ps -ef

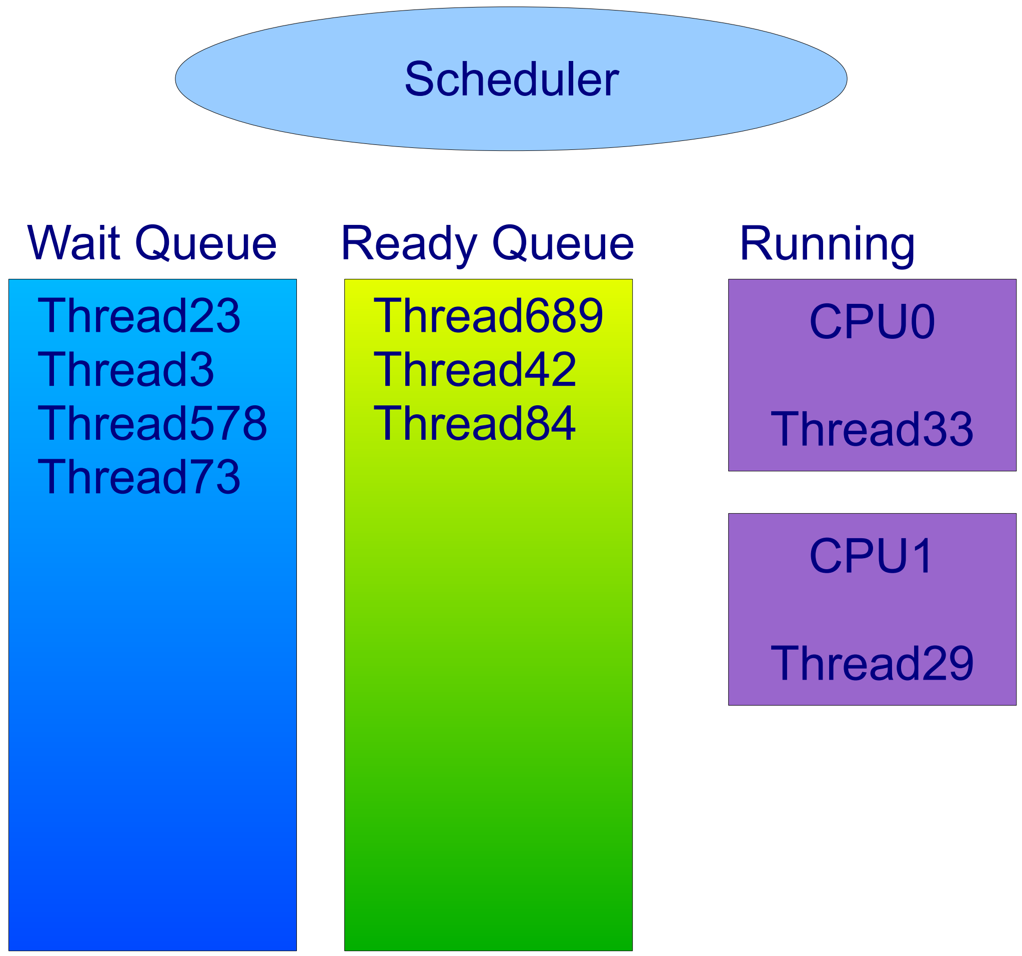
Linux process tree

Thread simple



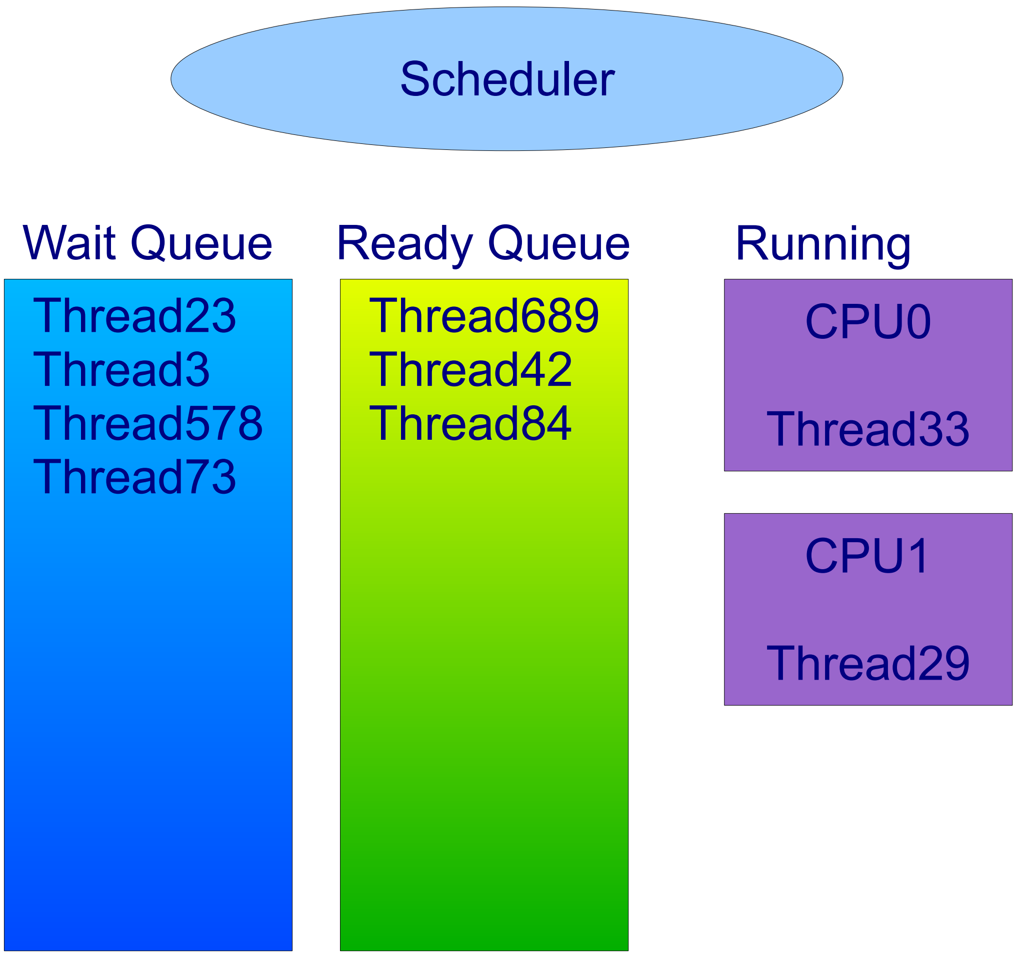
OS Scheduler

- Chooses which to process/thread to run next on which CPU
  - Ready queue
  - Wait queue
  - Thread priority
  - Time slice
  - Preemption
  - CPU core affinity
  - Etc



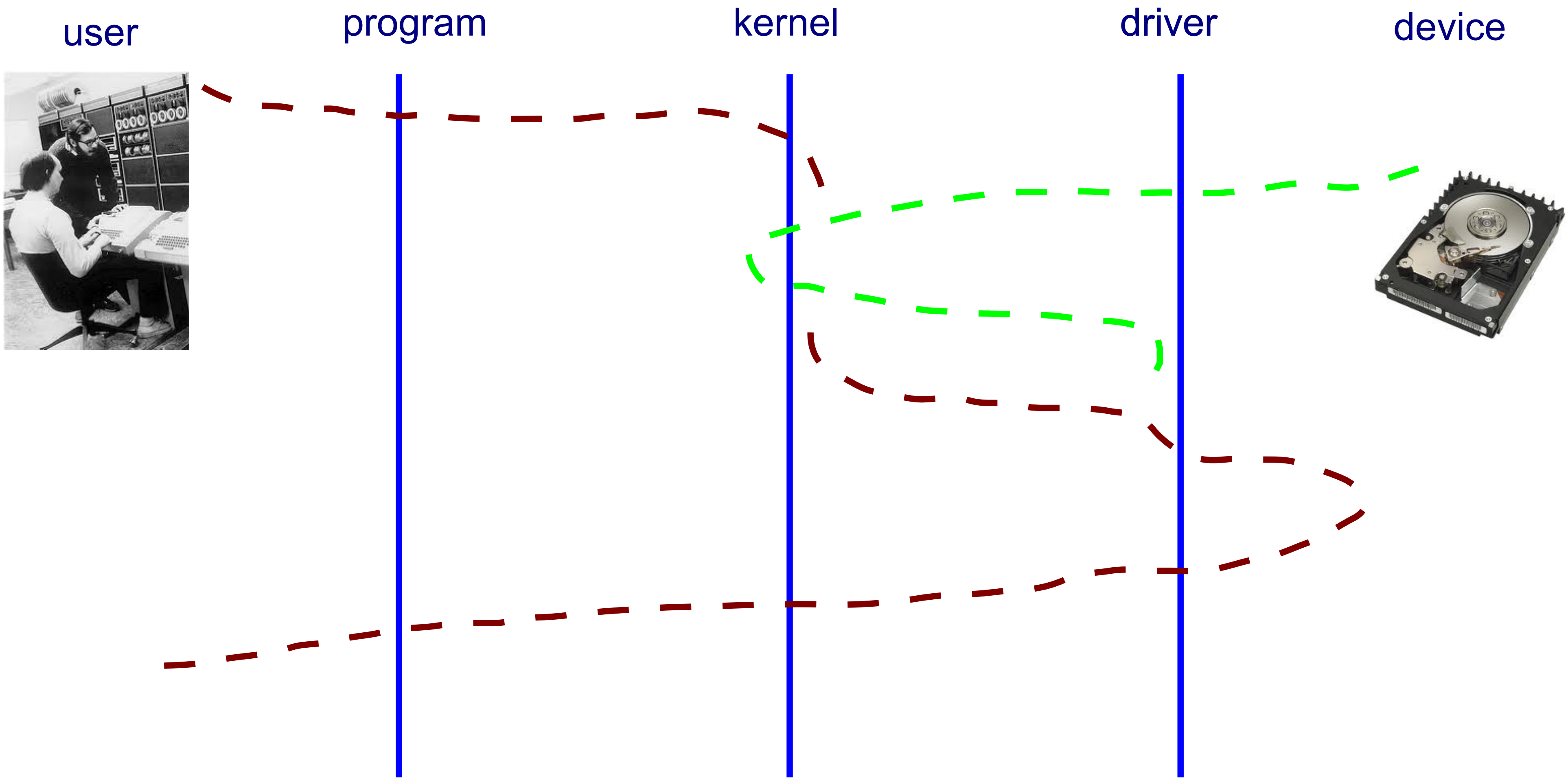
OS Scheduler

- Chooses which to process/thread to run next on which CPU
  - Ready queue
  - Wait queue
  - Thread priority
  - Time slice
  - Preemption
  - CPU core affinity
  - Etc



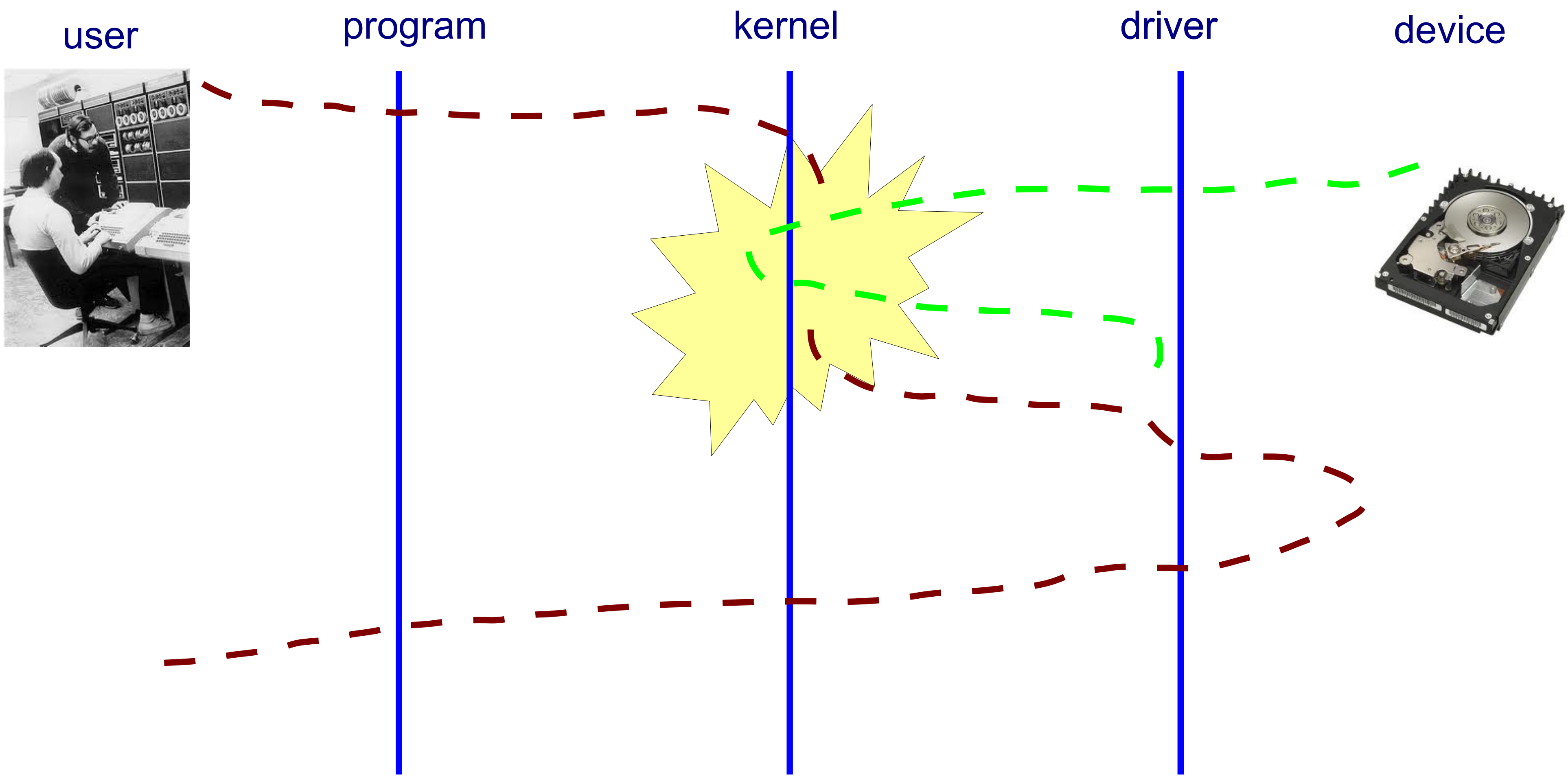
- Interrupts mess with scheduler plans

Thread, interrupted



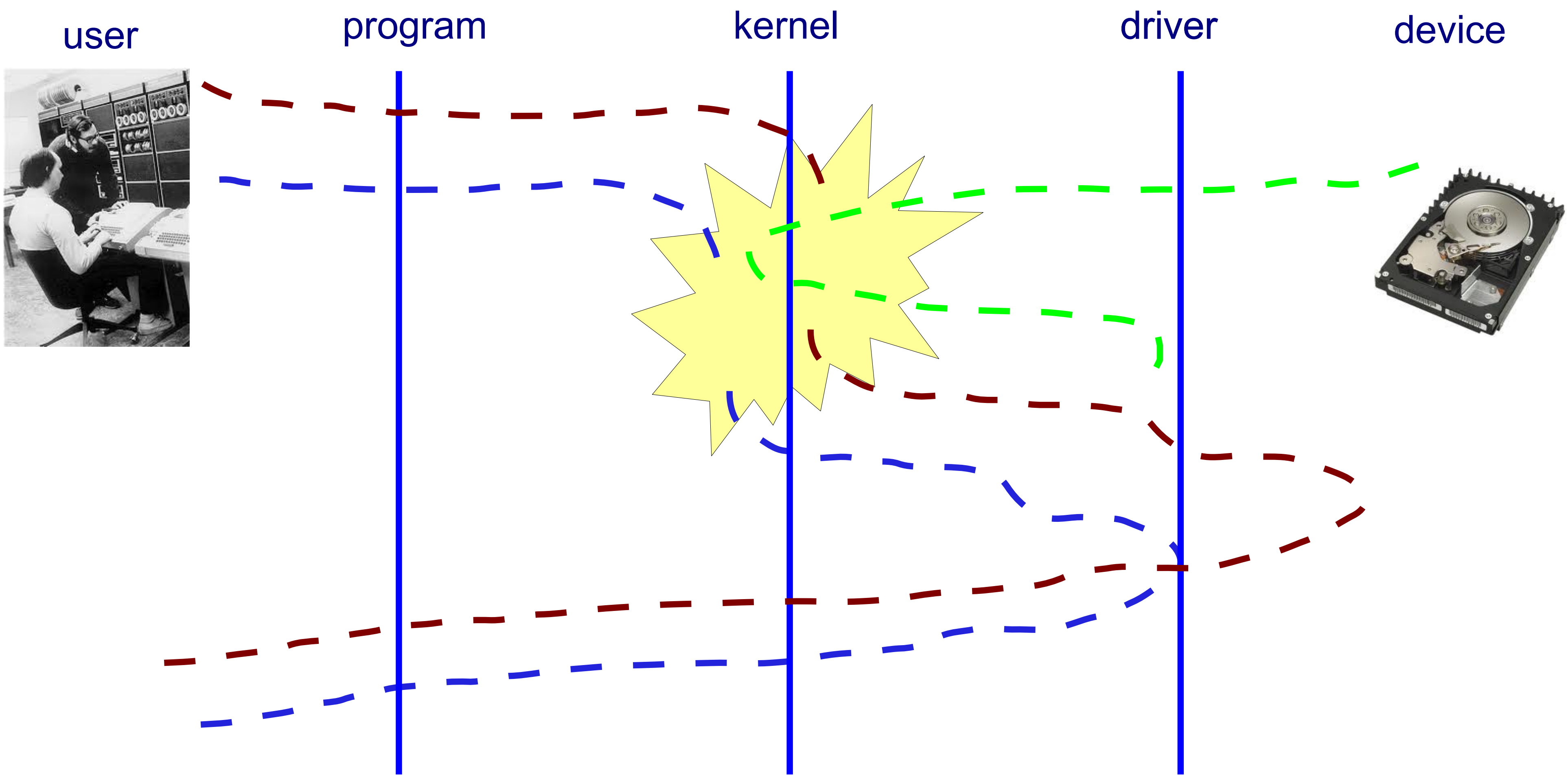
Concurrency and conflict

- Multiple threads could hit data at same time



Concurrency and conflict

- Multiple CPU threads could hit data at same time



Example: conflict

- User thread 1 asks for data
  - Driver requests data from HW, sleeps while waiting
- User thread 2 removes device
  - Driver removes data structures
- HW interrupt to finish data retrival
  - Driver interrupt handler tries to access removed data struct
  - Uh oh...

Order matters

Instance 1	**Instance 2**	**Value**
`read very_important_count`		5
`add 5 + 1 = 6`		6
`write very_important_count`		6
	`read very_important_count`	6
	`add 6 + 1 = 7`	7
	`write very_important_count`	7

Order matters

Instance 1	Instance 2	Value
`read very_important_count`		5
`add 5 + 1 = 6`		6
`write very_important_count`		6
	`read very_important_count`	6
	`add 6 + 1 = 7`	7
	`write very_important_count`	7

Instance 1	Instance 2	Value
`read very_important_count`		5
	`read very_important_count`	5
`add 5 + 1 = 6`		6
	`add 5 + 1 = 6`	6
`write very_important_count`		6
	`write very_important_count`	6

Order matters

Instance 1	Instance 2	Value
read very_important_count		5
add 5 + 1 = 6		6
write very_important_count		6
	read very_important_count	6
	Add 6 + 1 = 7	7
	write very_important_count	7

Instance 1	Instance 2	Value
read very_important_count		5
	read very_important_count	5
add 5 + 1 = 6		6
	Add 5 + 1 = 6	6
write very_important_count		6
	write very_important_count	6

Coordination needed

- Multiple threads could hit data at same time
- Tools:
  - Completions
  - Semaphore
  - Atomic action – increment, decrement
  - Mutex
  - Spin lock
  - RCU

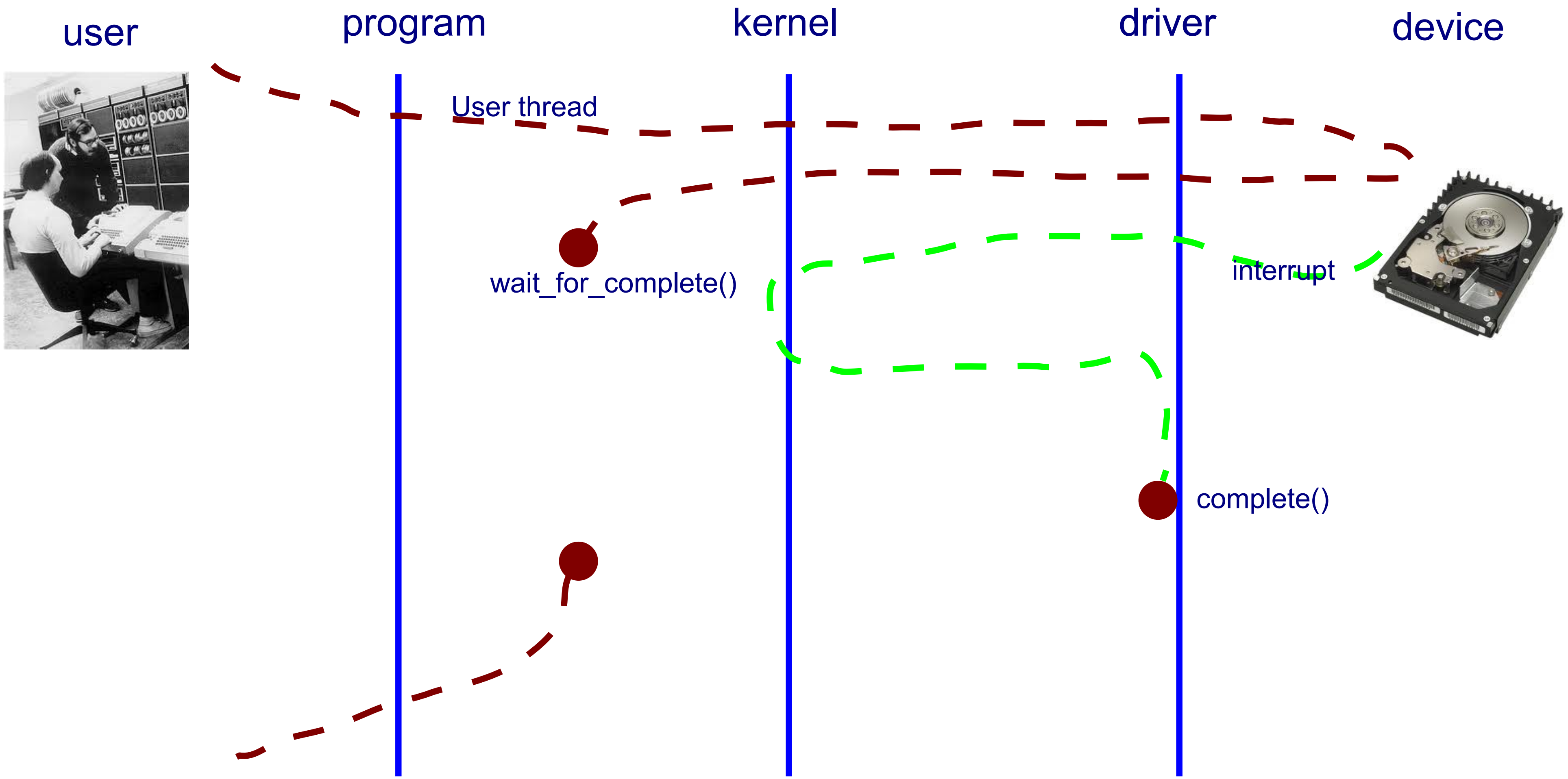
Completions

- Shared "flag" on which one thread can wait until another says 'go'.
- Handy when need to wait for unknown amount of time

```
<linux/completion.h>

DECLARE_COMPLETION(my_comp);

void wait_for_completion(struct completion *comp);

void complete(struct completion *comp);
```

- Example:
  - User thread requests data, calls wait_for_completion() to sleep until data ready
  - I/O event finishes, interrupt handler calls complete()
  - User thread gets to finish the read()
    - http://lwn.net/Articles/23993/

Waiting for the completion



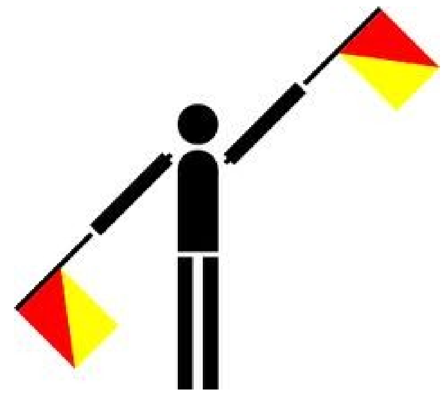
Atomic action

- CPU instructions for atomic increment, decrement, test_and_set
  - http://lxr.free-electrons.com/source/arch/x86/include/asm/atomic.h#L90
- All cores must coordinate CPU cache – expensive

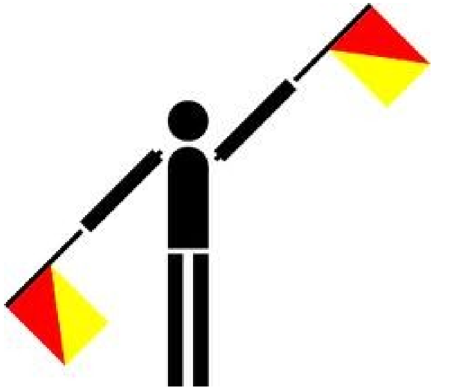
```
atomic_inc(x)
atomic_inc_and_test(x)
set_bit(n, *s)
clear_bit(n, *s)
test_bit(n, *s)
```

```
static inline void atomic_inc(atomic_t *v)
{
        asm volatile(LOCK_PREFIX "incl %0"
                     : "+m" (v->counter));
}
```

Semaphore



Semaphore



- Counter that many threads can inc/dec
  - Usually starts positive, each user decrements to start, inc when done
  - If zero, next process must wait
- Thread A might start operation, sleep, then thread B might finish
- Use atomic inc/dec to implement counter

```
struct semaphore sem;
sema_init(&sem, val);
down(&sem);
up(&sem);
```

Spinlock

Spinlock

- While not have lock, try again
  - Tight spin
  - Unlimited spin can "hang" thread, block other operations

```
while (test_and_set(3, &bit_string))
        /* tight loop */ ;
```

- Alternative is a sleep spin
  - Less CPU intense...

```
while (test_and_set(3, &bit_string))
        usleep(2);
```

  - ... but might miss a window of opportunity

- Linux:

```
spinlock_t slock;
spin_lock_init(&slock);
spin_lock(&slock);
spin_unlock(&slock);
spin_trylock(&slock);
```

Mutex

Mutex

- Mutual Exclusion
  - Everyone waits until the thread is done
  - Thread A gets lock, only thread A can release it
- Other threads will sleep while waiting for lock
- Good for blocking access to data, other resource
- Like spinlock, but more restrictive

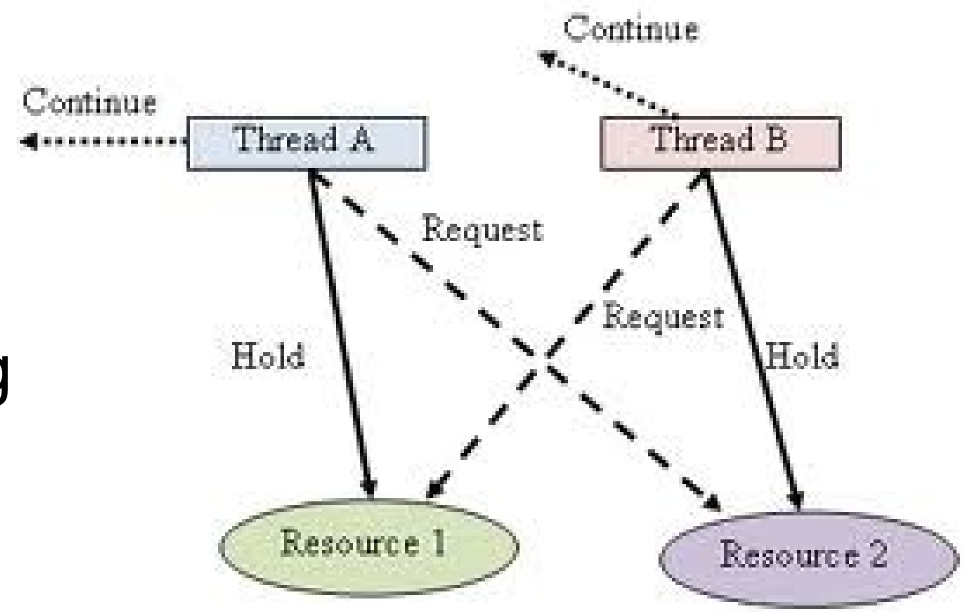
- Linux:

```
struct mutex mlock;
mutex_init(&mlock);
mutex_lock(&mlock);
mutex_unlock(&mlock);
```

Deadlock

Deadlock

- Deadlock possible with multiple locks
  - Process A gets lock 1, wants lock 2
  - Process B gets lock 2, wants lock 1



- Now what?
  - Linux does some checking

Watch out

- Blocking processes
  - Slowing other threads, whole system
  - Priority inversion – low prio thread holds lock, high prio thread can't continue
- Granularity
  - Lock smallest amount of code possible
- Balancing lock/unlock
- CPU communication overhead
- Hard to debug because of timing related

Reading

- LDD3 – Chapter 5: Concurrency
- LDD3 – Chapter 10: Interrupts
- ELDD – Chapter 2: Concurrency, pgs 39-48
- ELDD – Chapter 3: Kernel Facilities
- ELDD – Chapter 4: Interrupt Handling, pgs 92-103
- Linux src – ../Documentation/atomic_ops.txt