Concurrency

Concurrency

- **Classroom Activity!**
- Let's have a student come to the board and simulate this code:

```
while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Concurrency

- Next, let's add a 2nd student, running a 2nd CPU, accessing a different variable
  - Run in parallel

- Finally, let's have both students update the **same** variable

Atomicity

- Why did we get the wrong value for the variable?
  - Possible to interrupt the code between instructions
- An **atomic** operation is one that cannot be broken up; it either entirely happens, or doesn't happen at all
  - Single instructions are atomic
  - Sequences are not

Atomicity

Example!
`/tmp/critical.c`

- Why did we get the wrong value for the variable?
  - Possible to interrupt the code between instructions
- An **atomic** operation is one that cannot be broken up; it either entirely happens, or doesn’t happen at all
  - Single instructions are atomic
  - Sequences are not

In-Class Activity

Adding a busy variable:

```
while True:
    if busy == 0:
        busy = 1
        x += 1
        busy = 0
```

Does this solve the problem?

Atomicity

- The busy-flag still fails because there is a window of time between the read and the write:

```
while True:
    if busy == 0:
        DANGER HERE!
        busy = 1
        x += 1
        busy = 0
```

- Because the read and the write are not jointly atomic, we have a **race condition**

Races

- A **race condition** is when the outcome of a calculation depends on “accidental” (that is, unpredictable) details of how quickly it runs

- Remember: it is **impossible** to reliably predict your speed
  - Might be interrupted
  - Might be context-switched
  - Cache, paging, etc.

Races

- Races are **really bad!**

- Often will seem to work, but fail randomly
- Very difficult to replicate
- Very difficult to test your fix

- Conclusion: **prevent them before they happen!!!**

Critical Sections

- Let’s reconsider this code:

```
while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

- What do we need to **prevent,** in order the eliminate our race condition?
  - Need to forbid interrupting between load, store

Critical Sections

- A **critical section** is a portion of the program where interrupting it might cause a race

```
while True:              } Non-Critical
    load  x → $r1        }
    inc   $r1            } Critical Section
    store $r1 → x        }
```

Critical Sections

- We protect a critical section by indicating where the code “enters” and “leaves” it

```
while True:                   # not in CS
    enter_CS()
    load  x → $r1             # in CS
    inc   $r1                 # in CS
    store $r1 → x             # in CS
    leave_CS()
```

In-Class Activity

- Why does the CS start before the **`load`**, instead of after it?

```
while True:                # not in CS
    enter_CS()
    load  x → $r1          # in CS
    inc   $r1              # in CS
    store $r1 → x          # in CS
    leave_CS()
```

Critical Sections

- **Mutual exclusion** is the simplest way to protect critical sections
  - Somehow, make it impossible to be running more than one critical section at a time
  - Processes take turns, which one is in their CS
    - (Sometimes, *nobody* is in any CS)
- Note: this is a *goal,* not a *mechanism*

Critical Sections

- To provide mutual exclusion:
  - **`enter_CS()`**
    - Mark busy if first
    - Block if somebody is already in their CS
      - ***But how???***
  - **`leave_CS()`**
    - Mark not busy
    - Wake up one blocked process (if any)

Locks

- A **lock** is a simple, classic mechanism for providing mutual exclusion

  - The Problem: **Critical Section**
  - The Goal: **Mutual Exclusion**
  - The Mechanism: **Lock**

Locks

- A lock can only have one owner
- To become the owner, you “gain” (or “lock”) it
  - Will block if it is already owned
- To release ownership, you “release” (or “unlock”) it

Locks

```
while True:                 # not in CS
    gain(my_lock)
    load  x → $r1           # in CS
    inc   $r1               # in CS
    store $r1 → x           # in CS
    release(my_lock)
```

Locks

- **WARNING**
- Locks are only useful if you use them in all the right places. Locks don't truly protect data; they just block processes from running!
- If you forget to use `gain()` / `release()` on one CS, it will be a *danger* to all the other Cses (and vice-versa).

OS - History

Problem

- How to implement a lock? We can’t read a flag and then write it...

Insight

- A new type of assembly language instruction is needed

Atomic Instructions

- An **atomic read-modify-write instruction** is one that has the ability to perform all three operations in one atomic step; it can't be interrupted
- Essentially, it grabs a lock *in hardware* and then does:
  - Read
  - Perform a small calculation
  - Write

Atomic Instructions

- Many atomic instructions have been created
  - test and set
  - atomic inc / atomic dec
  - conditional swap
  - linked load / conditional store

- All are **slow**
- Always use normal instructions if possible

Atomic Instructions

- **test-and-set** (TAS) is one of the simplest atomic instructions
  - Reads a single variable (often, a single bit)
  - Sets it to 1
  - Returns ***old*** value to the user
  - Impossible for any other process to interrupt

Atomic Instructions

- Remember this old, broken code?

```
while True:
    if busy == 0:
        busy = 1
        x += 1
        busy = 0
```

- What if we used **test-and-set** to set our busy flag?

Atomic Instructions

```
while True:
    old_val = TAS(busy)
    if old_val == 0:
        x += 1
        busy = 0
```

- We always set **`busy`** to 1
- But if the old value was not zero, then this changed nothing
- We only increment x if we were the **first** to set **`busy`**

Locks

- We can use TAS to implement a lock gain:

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return
```

- The lock loops **forever,** trying to set the variable
- It keeps looping so long as somebody already owns the lock
- Called a “spin loop” or “spin lock”

Locks

- We can implement release:

```
func release(lock_var):
  lock_var = 0:
```

- Just set the lock_var back to 0
- This should allow some other process stuck in the spinlock to proceed

Race Example

Race Example

Process A

```
func main():
  while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Process B

```
func main():
  while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Process A	Shared Var:	Process B
`$r1 = ?`	`x = 0`	`$r1 = ?`

Race Example

Process A

```
func main():
  while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Process B

```
func main():
  while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Process A	Shared Var:	Process B
`$r1 = 0`	`x = 0`	`$r1 = ?`

Race Example

Process A

```
func main():
  while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Process B

```
func main():
  while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 0`	`$r1 = 0`

Race Example

Process A

```
func main():
  while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Process B

```
func main():
  while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 1`	`$r1 = 1`

Race Example

Process A

```
func main():
  while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Process B

```
func main():
  while True:
    load  x → $r1
    inc   $r1
    store $r1 → x
```

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 1`	`$r1 = 1`

x has been incremented ***twice*** but the value is still 1!

Lock Example

Lock Example

Process A	Shared Var:	Process B
`$r1 = ?`	`x = 0`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = ?`	`x = 0`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = ?`	`x = 0`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = ?`	`x = 0`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = ?`	`x = 0`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = ?`	`x = 0`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = ?`	`x = 0`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 0`	`x = 0`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
$r1 = 1	x = 0	$r1 = ?

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0;

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0;

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
$r1 = 1	x = 1	$r1 = ?

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
→   store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
→   if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 1`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
  → release(my_lock)
```

Process B

```
func gain(lock_var):
→ while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 1`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 1`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 1`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0;

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0;

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 1`	`$r1 = ?`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 1`	`$r1 = 1`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 1`	`$r1 = 2`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 2`	`$r1 = 2`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 2`	`$r1 = 2`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 2`	`$r1 = 2`

Process A

```
func gain(lock_var):
  while True:
→   if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

→ func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 2`	`$r1 = 2`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 2`	`$r1 = 2`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Lock Example

Process A	Shared Var:	Process B
`$r1 = 1`	`x = 2`	`$r1 = 2`

Process A

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Process B

```
func gain(lock_var):
  while True:
    if TAS(lock_var) == 0:
      return

func release(lock_var):
  lock_var = 0:

func main():
  while True:
    gain(my_lock)
    load  x → $r1
    inc   $r1
    store $r1 → x
    release(my_lock)
```

Locks

- A **spinlock** is a lock where **`gain()`** is implemented as a tight **`while`** loop

Locks

- A **spinlock** is a lock where **`gain()`** is implemented as a tight **`while`** loop

Example!

`/tmp/critical_lock.c`

Concurrency

Interrupts

Activity: Interrupts

Think, Pair, Share:

We would normally use a lock to protect a shared variable. What happens if a process owns the lock that is needs to modify this variable, then gets interrupted, but the interrupt handler also wants to gain this lock to change the variable?

Interrupts

- **Self-deadlock** is the condition when a process owns a given lock, but is also blocked, trying to gain the lock a second time
- Because the lock will never be released, the lock will never be gained
- Thus, we're stuck forever
- Interrupts can trigger self-deadlock
  - (Other things can cause it, too...)

Interrupts

- Kernel code (not user!) can **disable interrupts** at any time
  - No interrupts will fire
  - No interrupt handlers will run
  - External interrupts still happen, CPU remembers them
  - Interrupts fire **immediately** when user re-enables interrupts

Interrupts

- What are the tradeoffs of disabling interrupts?
  - Good: self-deadlock avoided
  - Bad: preemptive context switches never happen

Concurrency

Application

Concurrency

Three Classic forms of Concurrency

Multiprocessing

We won't be doing this in USLOSS

Time-Sharing

Will need to be considered

Interrupts

Exist in USLOSS!

Concurrency

- **Student Complaint:**
- If we are not running multiple CPUs, why did we learn about CSes, mutex, and locks?

Concurrency

- **Student Complaint:**
- If we are not running multiple CPUs, why did we learn about CSes, mutex, and locks?

- **Partial Answer:**
  Real OSes use it, important to understand
- **Better Answer:**
  It still *appears* to happen because of time slicing!

Concurrency

- **Remember:**
- OS presents a “virtual CPU” to each process
- Process has no idea when it runs, or when it is interrupted
- Even if we are time slicing on a *single CPU,* to the programs it seems like all are running in parallel
- Thus, **concurrency matters!**

Concurrency

- **Student Complaint:**
- Why don’t we just force all programs to be single-threaded, so that we can ignore concurrency?

Concurrency

- **Student Complaint:**
- Why don’t we just force all programs to be single-threaded, so that we can ignore concurrency?
- Even if the user processes are single-threaded, the *kernel never is!*
  - Many processes syscall into the kernel
  - Plus, have to deal with interrupts

Concurrency

- **Student Complaint:**
- But isn't the kernel "protected?" Why would concurrency be an issue?

Concurrency

- **Student Complaint:**
- But isn't the kernel "protected?" Why would concurrency be an issue?

- Kernel code can be time-sliced like any other process
- Also, can be interrupted at any time
  - (If ints not disabled)

Concurrency

- **Conclusion:**
- Kernel code must be treated as if it was the worlds most crazy-parallel program
  - Hundreds of threads
  - Locks absolutely necessary
- But wait...do we have a shortcut?

Preventing Concurrency

- In a multi-CPU OS, concurrency is **real**
  - Use locks to protect data
  - When self-deadlock is a worry, disable interrupts **before** you gain a lock
- But a **single-CPU** OS is simpler! If you disable interrupts:
  - Time-slicing doesnt happens
  - Interrupts can’t run

Preventing Concurrency

- Instead of gaining & releasing any locks, we will simply enable and disable interrupts!

```
while True:                     # not in CS
    old_psr = disable_ints()
    load  x → $r1               # in CS
    inc   $r1                   # in CS
    store $r1 → x               # in CS
    restore_ints(old_psr)
```