Concurrency

Three Classic forms of Concurrency

- Multiprocessing
- Time-sharing
  - Sometimes pre-emptive
  - Sometimes more controlled
- Interrupts

Concurrency

Theory

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

- Can we solve this with a flag?

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

- Discuss in groups: does this solve the problem?
- Then we'll simulate it

Atomicity

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

```
if busy == 0:
    --- DANGER! ---
    busy = 1
```

- 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 marking 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()
    ```
- Q: Why does the CS start before the `load`, instead of after it?
    ```

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.

Locks

Q: But how to implement a lock???

A: We use atomic read-modify-write instructions

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
  - Wide variety of types
  - All bad for performance
  - Use ordinary instructions whenever 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

```
while True:
    if TAS(some_lock) == 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"

Locks

- A **spinlock** is a lock where `gain()` is implemented as a tight `while` loop
  - In some implementations, CPU can be stuck forever
    - Common in the kernel
  - In others, the process eventually gives up and goes to sleep
    - Almost all user mode implementations

Concurrency

Interrupts

Interrupts

- Interrupts introduce a special type of concurrency
  - When a process is interrupted, the interrupt and the process are (roughly) parallel processes
  - Not symmetric, but the interrupt code can certainly screw up the program!
  - Worse: self-deadlock

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

- Inside any kernel code, if we plan to gain any spinlock **inside** an interrupt handler (say, to change some variables), then…
  - We must also use the spinlock **outside** the handler
  - But this would make us vulnerable to self-deadlock
  - We solve this by **disabling interrupts**

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?

  <u>Good:</u> self-deadlock impossible

  <u>Bad:</u> preemptive context switches never happen

Concurrency

Application

Concurrency

Three Classic forms of Concurrency

- Multiprocessing
  - *We won’t be doing this in USLOSS*
- Time-sharing
- Interrupts

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?

- 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?

- Kernel code can be time-sliced like any other process
- Also, can be interrupted at any time

Concurrency

Conclusion:

- Kernel code must be treated as if it was the worlds most crazily-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 spinlocks 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 never 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)
```