Concurrency & Synchronization

COMPSCI210 Recitation

25th Feb 2013

Vamsi Thummala

Slides adapted from Landon Cox

Midterm Review

http://www.cs.duke.edu/~chase/cps110-archive/midterm-210-13s1.pdf

- Please follow carefully the instructions listed on the exam

Midterm Solutions

http://www.cs.duke.edu/~chase/cps110-archive/midterm-210-13s1-sol.pdf

On Shell lab

Expected for write-up phase:

1. Read the handout. Read the material from OSTEP [1] on process creation and execution. Bryant and O'Hallaron can be a handy reference [2]
2. Read the **man pages** for fork(), exec (), wait (), dup2(), open(), read(), write(), and exit()
3. **Write small programs to experiment** with these system calls
4. Read the **man pages** for tcsetpgrp() and setpgid()
5. Read the code we provided for tcsetpgrp() and setpgid() and combine it with earlier programs you have written
6. Using the parser we gave, start writing single commands

Next lab: Elevator

- Start early!
- It is a group lab, but:
  - Recommend highly to start practicing synchronization problems individually
  - Coding practices may also differ among individuals but we follow common set of guidelines to write readable code
- Please follow lab guidelines
  - Only one submission from a group

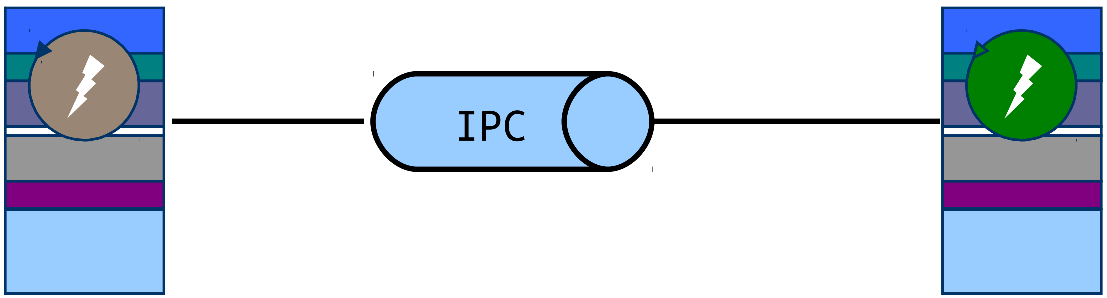
Goal: Next few weeks

- Master synchronization techniques
- Develop best practices for writing synchronization code
- Write solid concurrent code

So far: Multi-process concurrency

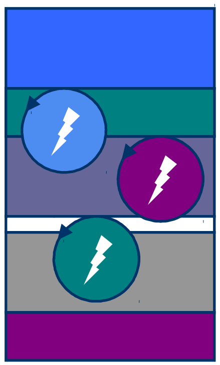
- Pipes: `ls | wc`

Process 1: ls **Process 2: wc**

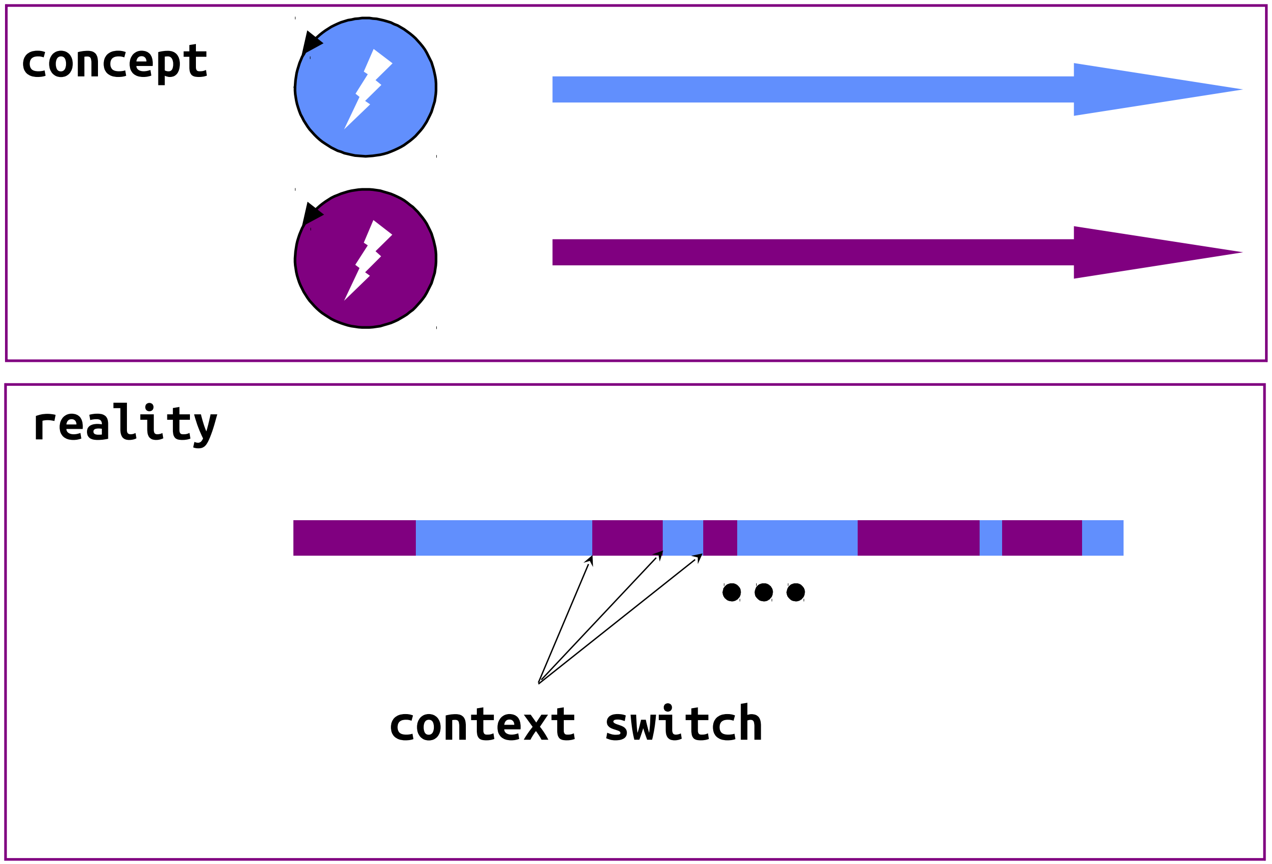


Now: Concurrency using threads

- Light weight
- Fast to create
- Shared address space
  - Each process can have multiple threads
- Each thread has its own:
  - SP
  - PC
  - Registers
- Everything else is shared
  - Heap
  - Code library
  - Common runtime



Two threads sharing a CPU



An Example: Shared account

- Two threads accessing account concurrently
  - Landon tries to withdraw money
  - Melissa tries to deposit money
- Initial balance = 100

Thread Landon:

```
balance = balance - 110;
printf("%d\n", balance)
```

Thread Melissa:

```
balance = balance * 2;
printf("%d\n", balance);
```

What is the final balance?

Debugging non-determinism

- Requires **worst-case** reasoning
  - Eliminate **all** ways for program to break
- Debugging is hard
  - Cannot test all possible interleavings
  - Bugs can only happen sometimes
- Heisenberg
  - Re-running the program may make the bug disappear
  - Doesn't mean that it still ain't there!

Constraining concurrency

- Synchronization
  - Control thread interleavings
- Some events are independent
  - No shared state
  - Relative (scheduling) order of these does not matter
- Other events are dependent
  - Order (of scheduling) can effect program output

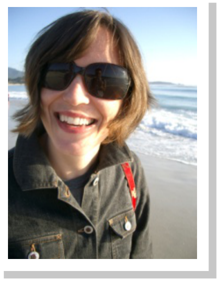
Goals of synchronization

- All interleavings must give correct result: deterministic concurrent program!
  - Works no matter how scheduling is done
  - How fast threads run
- Constrain as little as possible
  - Constraining slows program down
  - Creates complexity

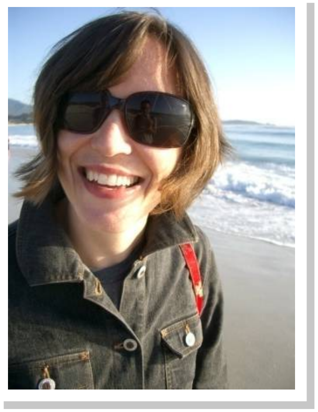
Work through example: "Too much milk"

- Rules
  - Quantity
    - The fridge must always be stocked with milk
    - Milk expires quickly, so never > 1 milk
  - Order
    - Landon and Melissa can come home at any time
    - If either see empty fridge, must buy milk
  - Code (no synchronization)

```
if (noMilk){
  buy milk;
}
```



“Too much milk” principals



Time		
3:00	Look in fridge (no milk)	
3:05	Go to grocery store	
3:10		Look in fridge (no milk)
3:15	Buy milk	
3:20		Go to grocery store
3:25	Arrive home, stock fridge	
3:30		Buy milk
3:35		Arrive home, stock fridge Too much milk!

What broke?

- Code worked sometimes, but not always
  - Code contained a race condition
  - Processor speed caused a incorrect result
- First type of synchronization
  - Mutual exclusion inside critical sections

Synchronization concepts

- Mutual exclusion
  - Ensure **only one** thread doing something at a time
    - E.g., one person shops at a time
  - Code blocks are atomic w.r.t each other
    - Threads cannot run atomic code blocks at the same time

Synchronization concepts

- Critical section
  - Code in critical section must run atomically w.r.t other critical section code
- If A and B are critical w.r.t each other:
  - A and B must mutually exclude each other
- Conflicting Code is often the same block
  - But executed by different threads
  - Reads/writes same data (e.g., fridge, screen)

Back to “Too much milk”

- What is a critical section?

```
if (noMilk){
  buy milk;
}
```

- Landon and Melissa's critical section must be atomic w.r.t each other

Attempt 1

- Atomic operations
  - Load: check note
  - Store: leave note

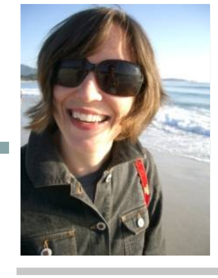
```
if (noMilk) {
  if (noNote){
    leave note;
    buy milk;
    remove note;
  }
}
```

Does this work?

1

```
if (noMilk) {
  if (noNote){
    leave note;
    buy milk;
    remove note;
  }
}
```

3



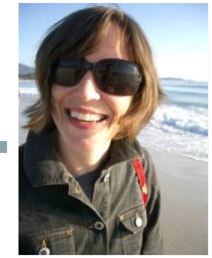
2

```
if (noMilk) {
  if (noNote){
    leave note;
    buy milk;
    remove note;
  }
}
```

4

Does this work?

```
if (noMilk) {
1  if (noNote){
3    leave note;
     buy milk;
     remove note;
   }
}
```



```
if (noMilk) {
2  if (noNote){
4    leave note;
     buy milk;
     remove note;
   }
}
```

Is this better than no synchronization at all?

What broke?

- Melissa's events can happen
  - After Landon checks for a note
  - Before Landon leaves a note

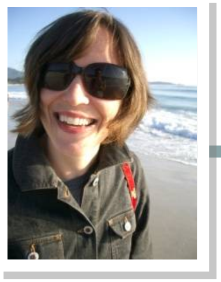
```
if (noMilk) {
  if (noNote){
    leave note;
    buy milk;
    remove note;
  }
}
```

Attempt 2

- Idea
  - Change the order of “leave note” and “check note”
  - Requires labeled notes (else you will see your note)

Does this work?

```
leave noteLandon
if (no noteMelissa){
  if (noMilk){
    buy milk;
  }
}
remove noteLandon
```



```
leave noteMelissa
if (no noteLandon){
  if (noMilk){
    buy milk;
  }
}
remove noteMelissa
```

Does this work?

```
leave noteLandon
if (no noteMelissa){
  if (noMilk){
    buy milk;
  }
}
remove noteLandon
```

```
leave noteMelissa
if (no noteLandon){
  if (noMilk){
    buy milk;
  }
}
remove noteMelissa
```

Nope! Illustration of “starvation”.

What about now?

```
while(noMilk) {
  leave noteLandon
  if (no noteMelissa){
    if (noMilk){
      buy milk;
    }
  }
}
remove noteLandon
```

```
while(noMilk) {
  leave noteMelissa
  if (no noteLandon){
    if (noMilk){
      buy milk;
    }
  }
}
remove noteMelissa
```

What about now?

```
while(noMilk) {
  leave noteLandon
  if (no noteMelissa){
    if (noMilk){
      buy milk;
    }
  }
}
remove noteLandon
```

```
while(noMilk) {
  leave noteMelissa
  if (no noteLandon){
     if (noMilk){
       buy milk;
     }
  }
}
remove noteMelissa
```

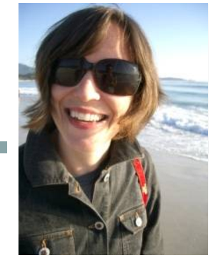
Nope! Same “starvation” problem as before.

Attempt 3

- We are getting closer
- Problem: Who should buy milk if both leave notes?
- Idea: Let Landon hang around to make sure job is done

Does this work?

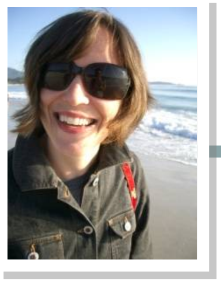
```
leave noteLandon
while (noteMelissa){
  do nothing
}
if (noMilk){
  buy milk;
}
remove noteLandon
```



```
leave noteMelissa
if (no noteLandon){
  if (noMilk){
    buy milk;
  }
}
remove noteMelissa
```

Does this work?

```
leave noteLandon
while (noteMelissa){
  do nothing
}
if (noMilk){
  buy milk;
}
remove noteLandon
```



```
leave noteMelissa
if (no noteLandon){
  if (noMilk){
    buy milk;
  }
}
remove noteMelissa
```

Yes, it does work! Can you show it?

Downside of the solution

- Complexity
  - Hard to convince yourself it works
- Asymmetry
  - Landon and Melissa runs different code
  - What about when number of threads > 2?
- Landon consumes CPU while waiting
  - Busy-waiting
  - However, only need atomic load/store

Raising the level of abstraction

- OS can provide better abstractions
- Locks
  - Also known as **mutexes**
  - Provides mutual exclusion
  - Prevents from entering critical section
- Lock operations
  - Lock aka Lock.acquire()
  - Unlock aka Lock.release()

Lock operations

- Lock: wait until lock is free, then acquire it

```
do {
  if (lock is free) {
    acquire lock
    break
  }
} while (1)
```

Must be atomic with respect to other threads calling this code

This is busy wait implementation. We will fix later!

“Too much milk”, Attempt 2

- Why doesn't the note work as lock?

```
if (noMilk) {
  if (noNote){
    leave note;
    buy milk;
    remove note;
  }
}
```

Block is not atomic. Must atomically
- check if lock is free
- grab it

Elements of locking

- The lock is initially free
- Threads acquire a lock before an action
- Threads release a lock when an action completes
- Operation lock() must wait when someone else has acquired the lock
- Key Idea:
  - All synchronization involves waiting!

“Too much milk”, Attempt 4 with locks

```
lock ()
if (noMilk) {
  buy milk
}
unlock ()
```

```
lock ()
if (noMilk) {
  buy milk
}
unlock ()
```

“Too much milk”, Attempt 4 with locks

```
lock ()
if (noMilk) {
  buy milk
}
unlock ()
```

```
lock ()
if (noMilk) {
  buy milk
}
unlock ()
```

Problem: Waiting for a lock while the other buys milk

“Too much milk”, Attempt 5 without waiting

```
lock ()
if (noNote && noMilk){
  leave note “at store”
  unlock ()
  buy milk
  lock ()
  remove note
  unlock ()
} else {
  unlock ()
}
```

```
lock ()
if (noNote && noMilk){
  leave note “at store”
  unlock ()
  buy milk
  lock ()
  remove note
  unlock ()
} else {
  unlock ()
}
```

"Too much milk", Attempt 5 without waiting

```
lock ()
if (noNote && noMilk){
  leave note "at store"
  unlock ()
  buy milk
  lock ()
  remove note
  unlock ()
} else {
  unlock ()
}
```

Not holding lock

```
lock ()
if (noNote && noMilk){
  leave note "at store"
  unlock ()
  buy milk
  lock ()
  remove note
  unlock ()
} else {
  unlock ()
}
```

Only hold lock while handling shared resource.

“Too much milk”, Attempt 6

```
lock ()
if (noMilk && noNote){
  leave note “at store”
  unlock ()
  buy milk
  stock fridge
  remove note
} else {
  unlock ()
}
```

```
lock ()
if (noMilk && noNote){
  leave note “at store”
  unlock ()
  buy milk
  stock fridge
  remove note
} else {
  unlock ()
}
```

Does this work?

"Too much milk", Java version

```
synchronized (obj){
  if (noMilk) {
    buy milk
  }
}
```

```
synchronized (obj){
  if (noMilk) {
    buy milk
  }
}
```

Every object is a lock

Using synchronized keyword:

Lock = {

Unlock = }

Modified "Too much milk"

- Landon and Melissa take turns to buy the milk
- Similar to a Ping-Pong problem
- How to satisfy the required constraint?
  - Strict alternative ordering

New synchronization concept: Monitors

- Mutual exclusion is necessary but not sufficient
- Still need ordering constraints
  - Often must wait for something to happen
  - And wake up
- Monitors
  - wait() on a conditional variable
  - signal() aka notify()
  - broadcast() aka notifyall()

Detour: Concurrency in Java

- http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- Two ways:
  - Implement a Runnable interface

```
public class HelloRunnable impelments Runnable {
  public void run() {}
}
```

  - Subclass Thread

```
public class HelloThread extends Thread {
  public void run() {}
}
```

Java Runnable Interface

```
// Custom thread class
public class MyThread
 Implements Runnable{
  public MyThread(..){
       …
  }
  // Override thread run method
  public void run() {
       // run the thread
      …
  }
}
```

```
// Client class
public class Client {
  public void action(..) {
   // Create thread
   Thread t1 = new Thread(new
                 myThread(..));
   // Start thread
   t1.start();
   Thread t2 = new Thread(new
                 myThread(..));
   t2.start();
  }
}
```

java.lang.Runnable

- java.lang.Thread → java.lang.Runnable
- run()
- start()
- interrupt()
- isAlive()
- join()
- sleep()
- yield()
- isInterrupted()
- currentThread()

Other abstractions support by a language/OS

- Software transaction memory
  - Transaction support within shared memory
  - Analogous to db transaction support
  - Active area of research
- Actors aka asynchronous communication via message passing
  - Erlang