Concurrency

COMPSCI210 Recitation

12 Oct 2012

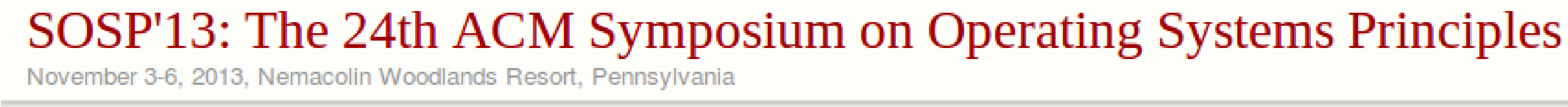
Vamsi Thummala

Venues for systems research

10th USENIX Symposium on Operating Systems Design and Implementation

OCTOBER 8—10, 2012
HOLLYWOOD, CA

sponsored by USENIX in cooperation with ACM SIGOPS



SOSP'13: The 24th ACM Symposium on Operating Systems Principles

November 3-6, 2013, Nemacolin Woodlands Resort, Pennsylvania

Comments on lab submissions so far..

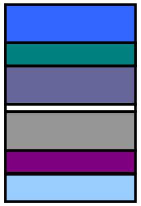
- Read handout, ask questions
- Think before you start coding
- Write readable code
- man/doc are your friends
- Please do not post your code on the web
- Please do not copy any code directly from the web or other sources
  - You can look, but write your own code
  - When in doubt, always ask first

We hear your feedback

- Next lab: Multi-threaded programming in Java
  - We provided only the interfaces
  - You can start from the scratch
  - Due on 26th Oct, 11:59pm
  - You can work in groups of 2 or at most 3.
- Exam FAQ
  - Check the course page

Processes: A Closer Look

virtual address space

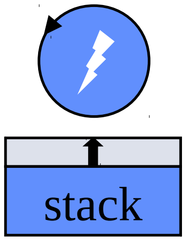


The address space is represented by page table, a set of translations to physical memory allocated from a kernel memory manager.

The kernel must initialize the process memory with the program image to run.

+

thread



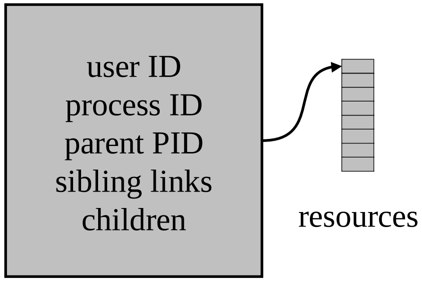
Each process has a thread bound to the VAS.

The thread has a saved user context as well as a system context.

The kernel can manipulate the contexts to start the thread running wherever it wants.

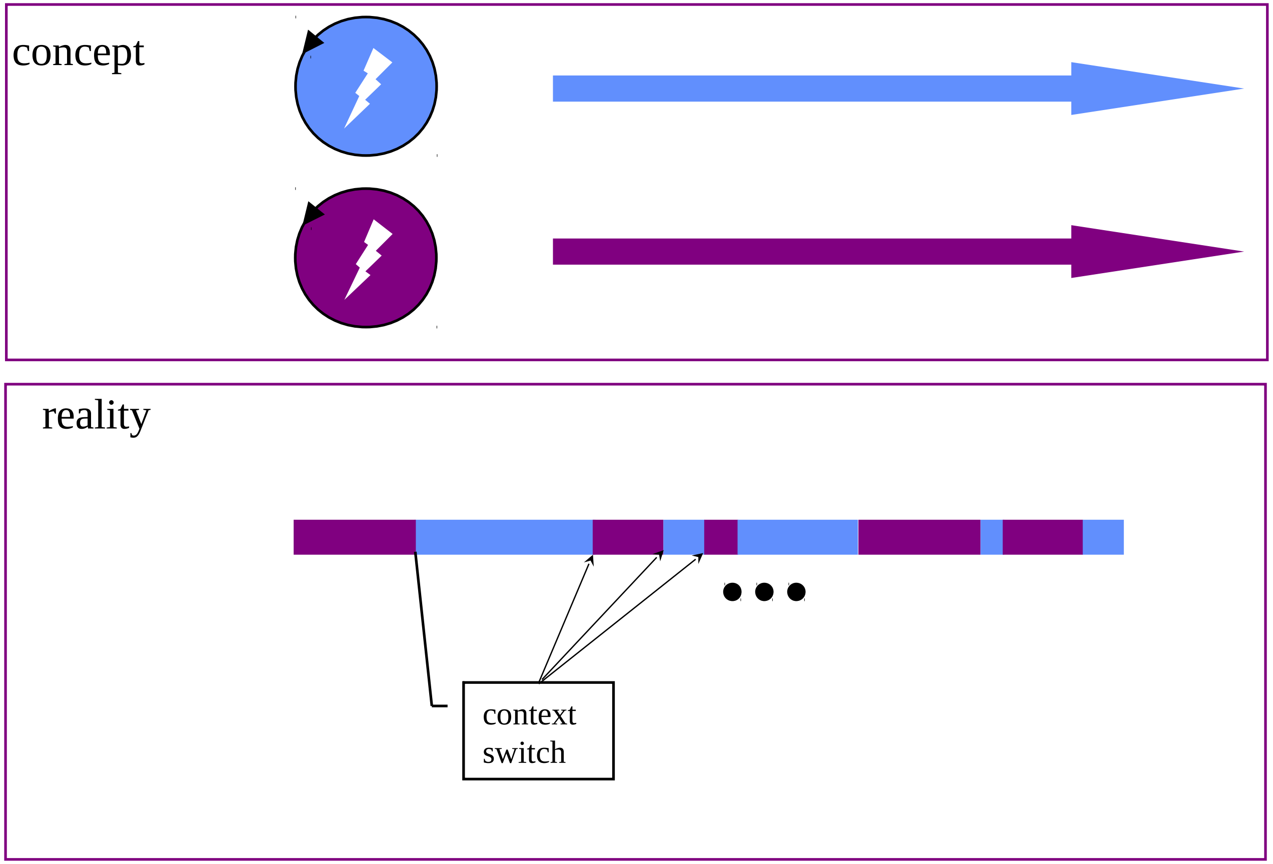
+

process descriptor (PCB)



Process state includes a file descriptor table, links to maintain the process tree, and a place to store the exit status.

Two threads sharing a CPU



Concurrency

- Having multiple threads active at one time
- Thread is the unit of concurrency
- Primary topics
  - How threads cooperate on a single task
  - How multiple threads can share the CPUs

An example

- Two threads (A and B)
  - A tries to increment i
  - B tries to decrement i

Thread A:

```
 i = 0;
 while (i < 10){
   i++;
 }
printf(“A done.”)
```

Thread B:

```
i = 0;
while (i > -10){
  i--;
}
printf(“B done.”)
```

Example continued ..

- Who wins?
- Does someone has to win?

Thread A:

```
i = 0;
while (i < 10){
 i++;
}
printf(“A done.”)
```

Thread B:

```
i = 0;
while (i > -10){
 i--;
}
printf(“B done.”)
```

Debugging non-determinism

- Requires **worst-case** reasoning
  - Eliminate **all** ways for program to break
- Debugging is hard
  - Can’t test all possible interleavings
  - Bugs may only happen sometimes
- **Heisenbug**
  - Re-running program may make the bug disappear
  - Doesn’t mean it isn’t still there!

Constraining concurrency

- **Synchronization**
  - Controlling thread interleavings
- Some events are independent
  - No shared state
  - Relative order of these events don’t matter
- Other events are dependent
  - Output of one can be input to another
  - Their order can affect program results

Goals of synchronization

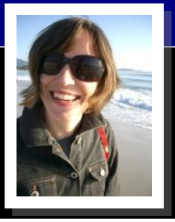
1. All interleavings must give correct result
   - Correct concurrent program
     - Works no matter how fast threads run
     - Important for your projects!
2. Constrain program as little as possible
   - Why?
     - Constraints slow program down
     - Constraints create complexity

“Too much milk” rules

- The fridge must be stocked with milk
  - Milk expires quickly, so never > 1 milk
- Landon and Melissa
  - Can come home at any time
  - If either sees an 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 incorrect result
- First type of synchronization
  - **Mutual exclusion**
  - **Critical sections**

Synchronization concepts

- **Mutual exclusion**
  - Ensure 1 thread doing something at a time
  - E.g. 1 person shops at a time
  - Code blocks are atomic w/re to each other
  - Threads can’t run code blocks at same time

Synchronization concepts

- **Critical section**
  - Code block must run atomically
    - w.r.t some piece of the code
- If A and B are critical w/re to each other
  - Threads mustn’t interleave code from A and B
    A and B mutually exclude each other
- Conflicting code is often same block
  - But executed by different threads
  - Reads/writes shared data (e.g. screen, fridge)

Back to “Too much milk”

- What is the critical section?

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

- Landon and Melissa’s critical sections
  - Must be atomic w/re to each other

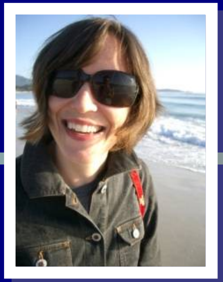
Solution 1 code

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

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

Does it work?

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

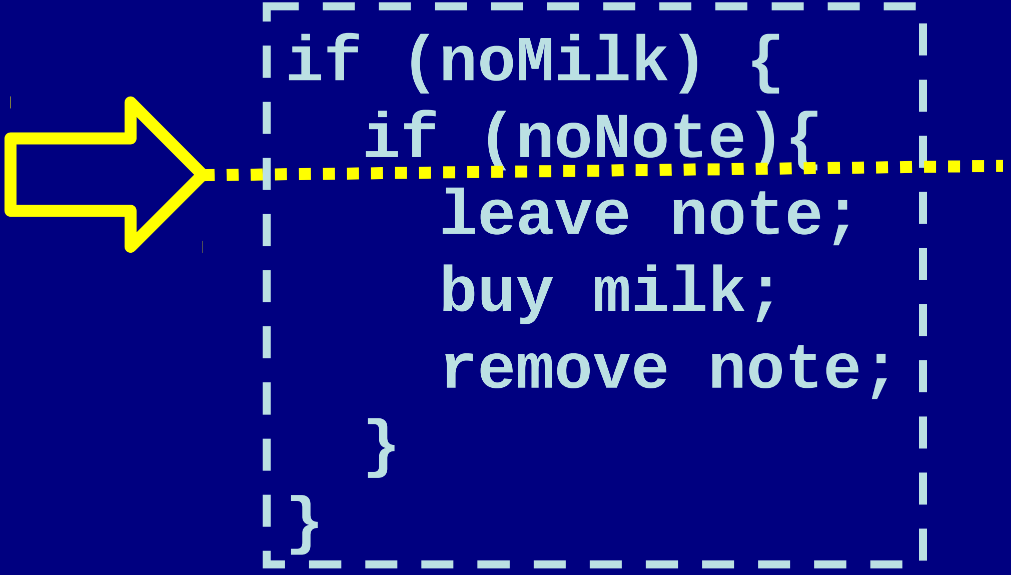


```
 if (noMilk) {
②  if (noNote){
      leave note;
  ④   buy milk;
      remove note;
    }
 }
```

Is this better than no synchronization at all?
What if "if" sections are switched?

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;
  }
}
```

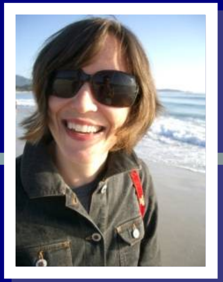
Next solution

Idea:

- Change the order of “leave note”, “check note”
- Requires labeled notes (else you’ll see your note)

Does it 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
}
```

Nope.
(Same starvation problem as before)

Next solution

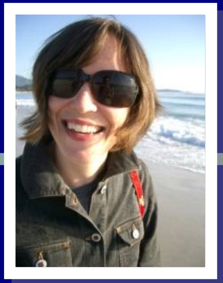
- We’re getting closer

- Problem
  - Who buys milk if both leave notes

- Solution
  - Let Landon hang around to make sure job is done

Does it 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?