Multi-core is a reality...

... but how do we write multi-core safe code?
Cache Coherence:
Necessary, but not Sufficient
Shared Memory Multiprocessor (SMP)

- Suppose CPU cores share physical address space
- Assume write-through caches (write-back is worse!)
What could possibly go wrong?

...  
Core0  Core1  ...  Core3  ...  

x++  

...  

read x  read x  ...  

...  

Interconnect  

...  

I/O
Cache coherence defined...

Informal: Reads return most recently written value

Formal: For concurrent processes $P_1$ and $P_2$

- $P$ writes $X$ before $P$ reads $X$ (with no intervening writes) $\Rightarrow$ read returns written value
- $P_1$ writes $X$ before $P_2$ reads $X$ $\Rightarrow$ read returns written value
- $P_1$ writes $X$ and $P_2$ writes $X$ $\Rightarrow$ all processors see writes in the same order
  - all see the same final value for $X$
Recall: **Snooping** for Hardware Cache Coherence

- All caches monitor bus and all other caches
- **Bus read**: respond if you have dirty data
- **Bus write**: update/invalidate your copy of data
- In reality: very complicated, lots of corner cases
Is cache coherence sufficient?

$P_1$

... 

$x = x + 1$

...

$P_2$

...

$x = x + 1$

...

→ happens even on single-core (context switches!)
Programs and Processes
How do we cope with lots of activity?

Simplicity? Separation into **processes**

Reliability? **Isolation** (e.g. virtual memory)

Speed? Program-level **parallelism** (e.g. during I/O)
**Process**

OS abstraction of a running computation
- The unit of execution
- The unit of scheduling
- Execution state
  + address space

From process perspective
- a virtual CPU
- some virtual memory
- a virtual keyboard, screen, ...

**Program**

“Blueprint” for a process
- Passive entity (bits on disk)
- Code + static data
Role of the OS

Context Switching
• Provides illusion that every process owns a CPU

Virtual Memory
• Provides illusion that process owns some memory

Device drivers & system calls
• Provides illusion that process owns a keyboard, ...

To do:

How to start a process?
How do processes communicate / coordinate?
Creating Processes:
Fork
Q: How to create a process? Double click?

After boot, OS starts the first process...

• e.g. *init*

...which in turn creates other processes

• parent / child → the **process tree**
Init is a special case. For others...

Q: How does parent process create child process?
A: **fork() system call**

- creates new address space  
  (Copy-On-Write duplicate of parent’s)
- creates new execution state in OS process table  
  (Exact copy of parent’s)
- returns child’s id to parent  
  (context[parent_id]->v0 = child_id)
- returns zero to child  
  (context[child_id]->v0 = 0)

Wait. what?

- int fork() returns TWICE!
main(int ac, char **av) {
    int x = getpid(); // get current process ID from OS
    char *hi = av[1]; // get greeting from command line
    printf("I'm process %d\n", x);
    int id = fork();
    if (id == 0)
        printf("%s from %d\n", hi, getpid());
    else
        printf("%s from %d, child is %d\n", hi, getpid(), id);
}
$ gcc -o strange strange.c
$ ./strange "Hi"
I'm process 23511
Hi from 23512
Hi from 23511, child is 23512
Parent can pass information to child
• In fact, *all parent data* is passed to child
• But isolated from then on…
  – C-O-W ensures they don’t see each other’s changes

Q: How to continue communicating?
A: Invent OS “IPC channels” : `send(msg)`, `recv()`, ...
A: Shared (Virtual) Memory!
• Before fork: allocate pages, mark as “shared”
• During fork: don’t set copy-on-write for these pages
• After fork: either can read/write
Processes and Threads
Parallel programming with processes:

- They share almost everything code, shared mem, open files, filesystem privileges, ...
- Pagetables will be *almost* identical
- Differences: PC, registers, stack

Recall: process = execution context + address space
Processes and Threads

Process

OS abstraction of a running computation
- The unit of execution
- The unit of scheduling
- Execution state
  + address space

From process perspective
- a virtual CPU
- some virtual memory
- a virtual keyboard, screen, ...

Thread

OS abstraction of a single thread of control
- The unit of scheduling
- Lives in one single process

From thread perspective
- one virtual CPU core on a virtual multi-core machine
Multithreaded Processes

**Single-threaded process**
- code
- data
- files
- registers
- stack

**Multithreaded process**
- code
- data
- files
- registers
- registers
- registers
- stack
- stack
- stack

Thread
#include <pthread.h>

int counter = 0;

void PrintHello(int arg) {
    printf("I’m thread %d, counter is %d\n", arg, counter++);
    ... do some work ...
    pthread_exit(NULL);
}

int main () {
    for (t = 0; t < 4; t++) {
        printf("in main: creating thread %d\n", t);
        pthread_create(NULL, NULL, PrintHello, t);
    }
    pthread_exit(NULL);
}
in main: creating thread 0
I’m thread 0, counter is 0
in main: creating thread 1
I’m thread 0, counter is 0
in main: creating thread 2
in main: creating thread 3
I’m thread 3, counter is 2
I’m thread 2, counter is 3

If processes?
Example: Apache web server
Each client request handled by a separate thread (in parallel)
• Some shared state: hit counter, ...

Thread 52
read hits
addi
write hits

Thread 205
read hits
addi
write hits

(look familiar?)

Timing-dependent failure $\Rightarrow$ race condition
• hard to reproduce $\Rightarrow$ hard to debug
Programming with threads

Within a thread: execution is sequential

Between threads?

- No ordering or timing guarantees
- Might even run on different cores at the same time

Problem: hard to program, hard to reason about

- Behavior can depend on subtle timing differences
- Bugs may be impossible to reproduce

Cache coherency isn’t sufficient...

Need explicit synchronization to make sense of concurrency!
Managing Concurrency

Races, Critical Sections, and Mutexes
Concurrency Goals

Liveness
• Make forward progress

Efficiency
• Make good use of resources

Fairness
• Fair allocation of resources between threads

Correctness
• Threads are isolated (except when they aren’t)
Race Condition

Timing-dependent error when accessing shared state

- Depends on scheduling happenstance
  ... i.e. who wins “race” to the store instruction?

Concurrent Program Correctness =
all possible schedules are safe

- Must consider every possible permutation
- In other words...
  ... the scheduler is your adversary
What if we can designate parts of the execution as critical sections

• Rule: only one thread can be “inside”

<table>
<thead>
<tr>
<th>Thread 52</th>
<th>Thread 205</th>
</tr>
</thead>
<tbody>
<tr>
<td>read hits</td>
<td>read hits</td>
</tr>
<tr>
<td>addi</td>
<td>addi</td>
</tr>
<tr>
<td>write hits</td>
<td>write hits</td>
</tr>
</tbody>
</table>
Q: How to implement critical section in code?
A: Lots of approaches....

**Disable interrupts?**

`CSEnter()` = disable interrupts (including clock)
`CSExit()` = re-enable interrupts

Works for many kernel data-structures
- but only within a single core: why?

Very bad idea for user code
  (important events are delayed... forever?)
Q: How to implement critical section in code?
A: Lots of approaches....

Modify OS scheduler?
CSEnter() = syscall to disable context switches
CSExit() = syscall to re-enable context switches

Doesn’t work if interrupts are part of the problem (e.g. won’t work for many kernel datastructures)

Usually a bad idea anyway (caller forgets to CSExit? Or waits a long time?)
Q: How to implement critical section in code?
A: Lots of approaches....

**Mutual Exclusion Lock (mutex)**

*acquire(m)*: wait till it becomes free, then take it
*release(m)*: free it

```c
apache_got_hit() {
    pthread_mutex_lock(m);
    hits = hits + 1;
    pthread_mutex_unlock(m)
}
```
Q: How to implement mutexes?
A: next time...