Parallel Programming and Synchronization

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

P&H Chapter 2.11

Multi-core is a reality…

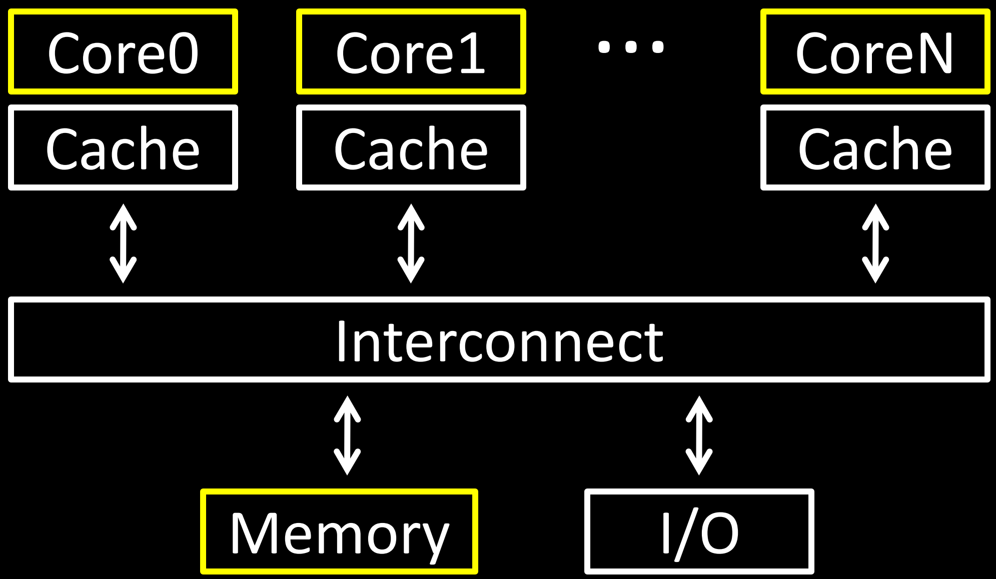
… but how do we write multi-core safe code?

Cache Coherence:

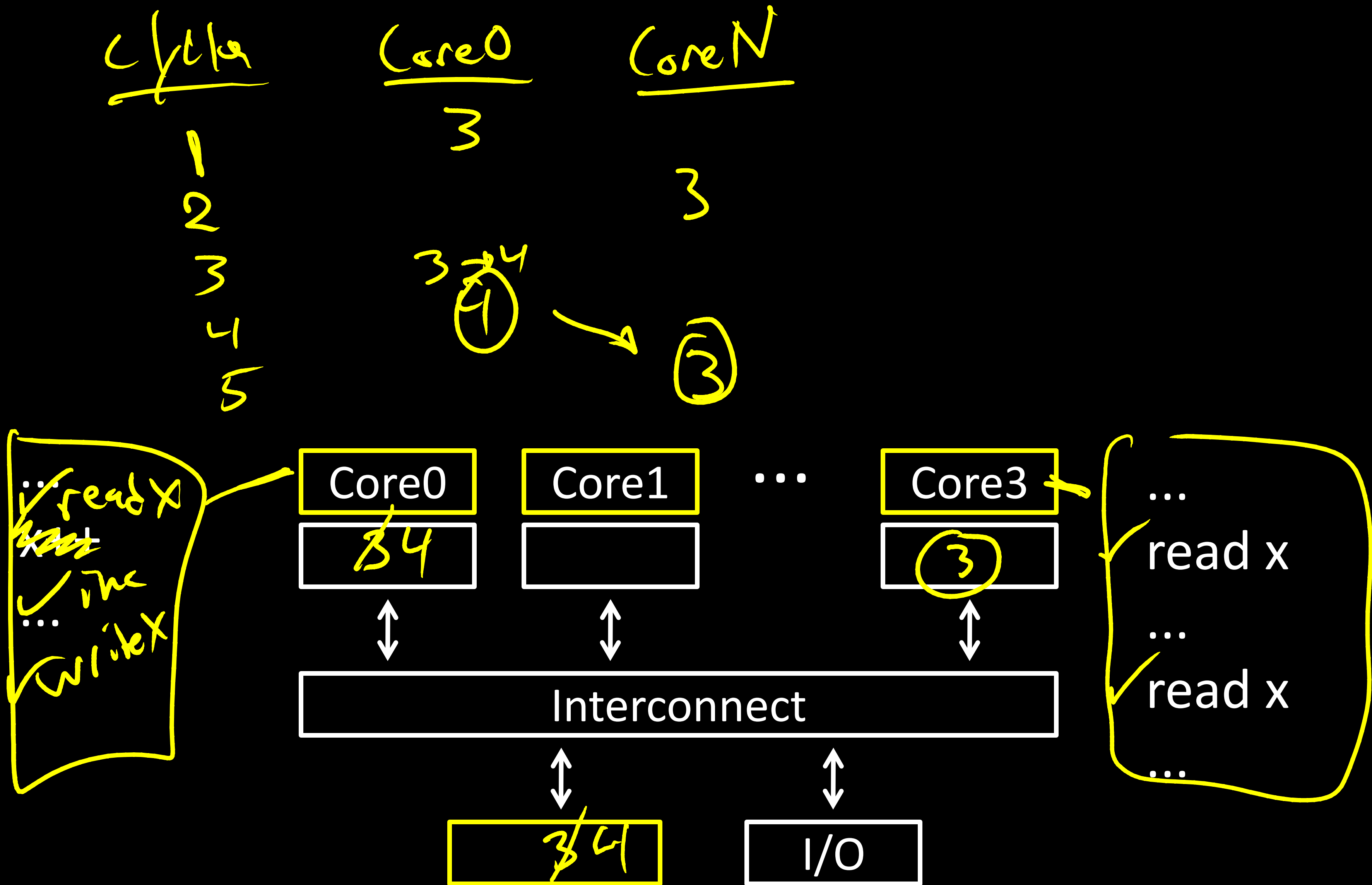
Necessary, but not Sufficient

Shared Memory Multiprocessor (SMP)

- Suppose CPU cores share physical address space
- Assume write-through caches (write-back is worse!)



What could possibly go wrong?



Cache coherence defined...

Informal: Reads return most recently written value

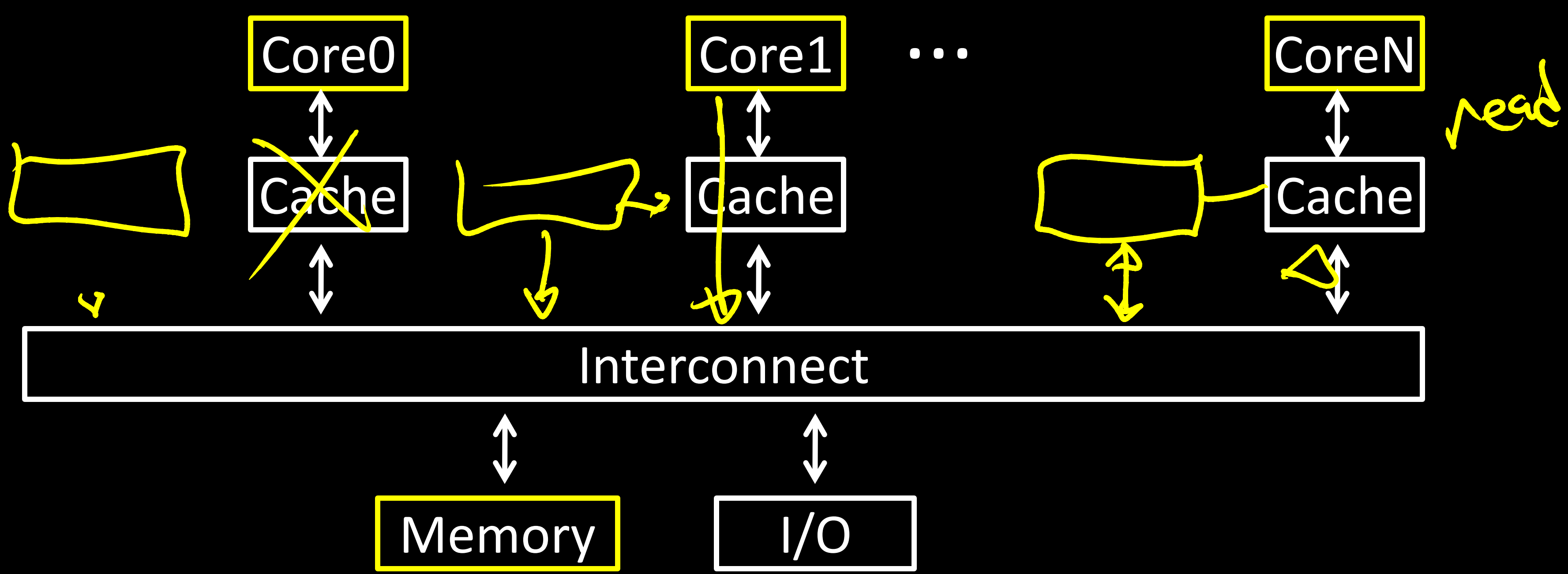
Formal: For concurrent processes P_1 and P_2

- P writes X before P reads X (with no intervening writes)
  ⇒ read returns written value
- P_1 writes X before P_2 reads X
  ⇒ read returns written value
- P_1 writes X and P_2 writes X
  ⇒ all processors see writes in the same order
  - all see the same final value for X

for some value of "before"

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



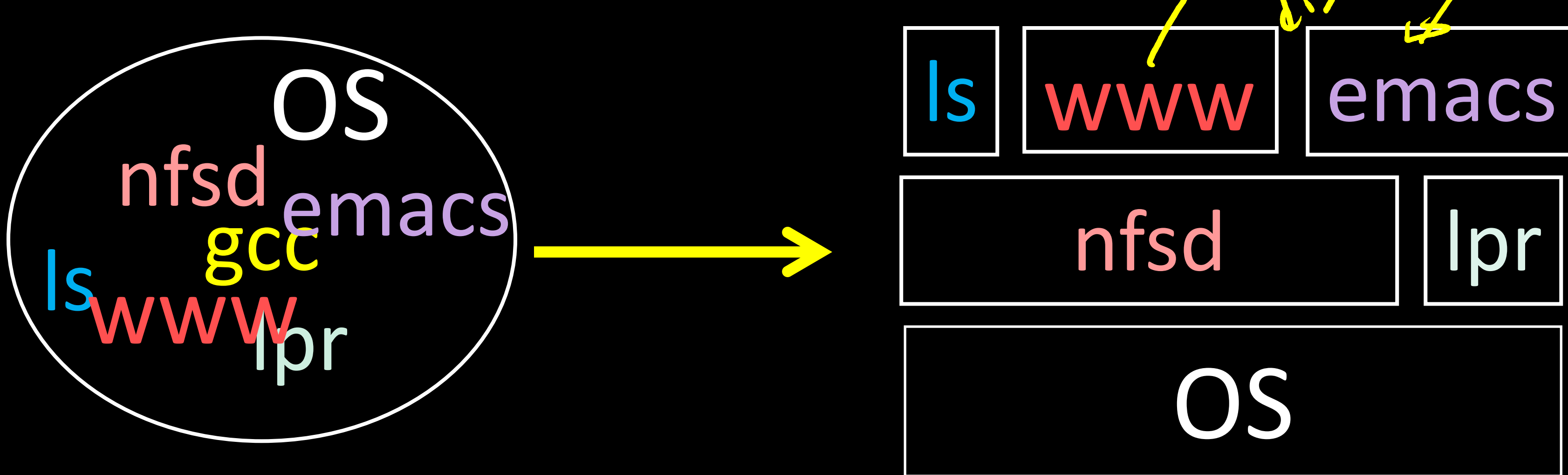
Is cache coherence enough?

P_1			P_2
…	read (1) 40	40 (3) read	…
x = x +1	inc (2) 41	41 (4) inc	x = x + 1
…	write (5) 41	41 (6) write	…

Single Core too!

Programs and Processes

How do we cope with lots of activity?



Simplicity? Separation into processes

Reliability? Isolation (think: VM)

Speed? Program-level <u>parallelism</u>

Process

OS abstraction of a running computation

- The unit of execution
- The unit of scheduling
- Execution state
  + address space } PTBR

From process perspective

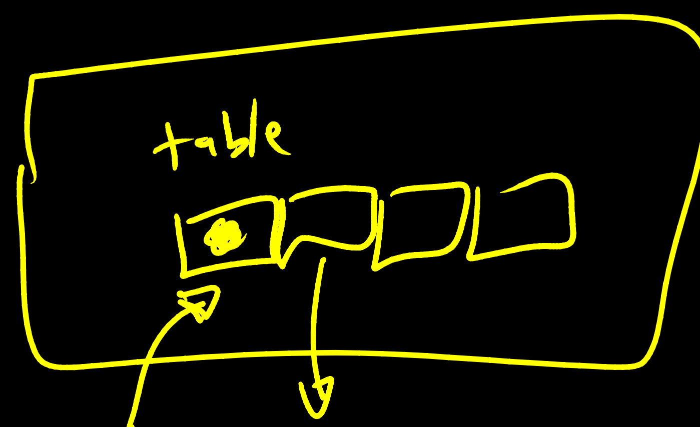
- a virtual CPU
- some virtual memory
- a virtual keyboard, screen, …

virtual machine

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*) …

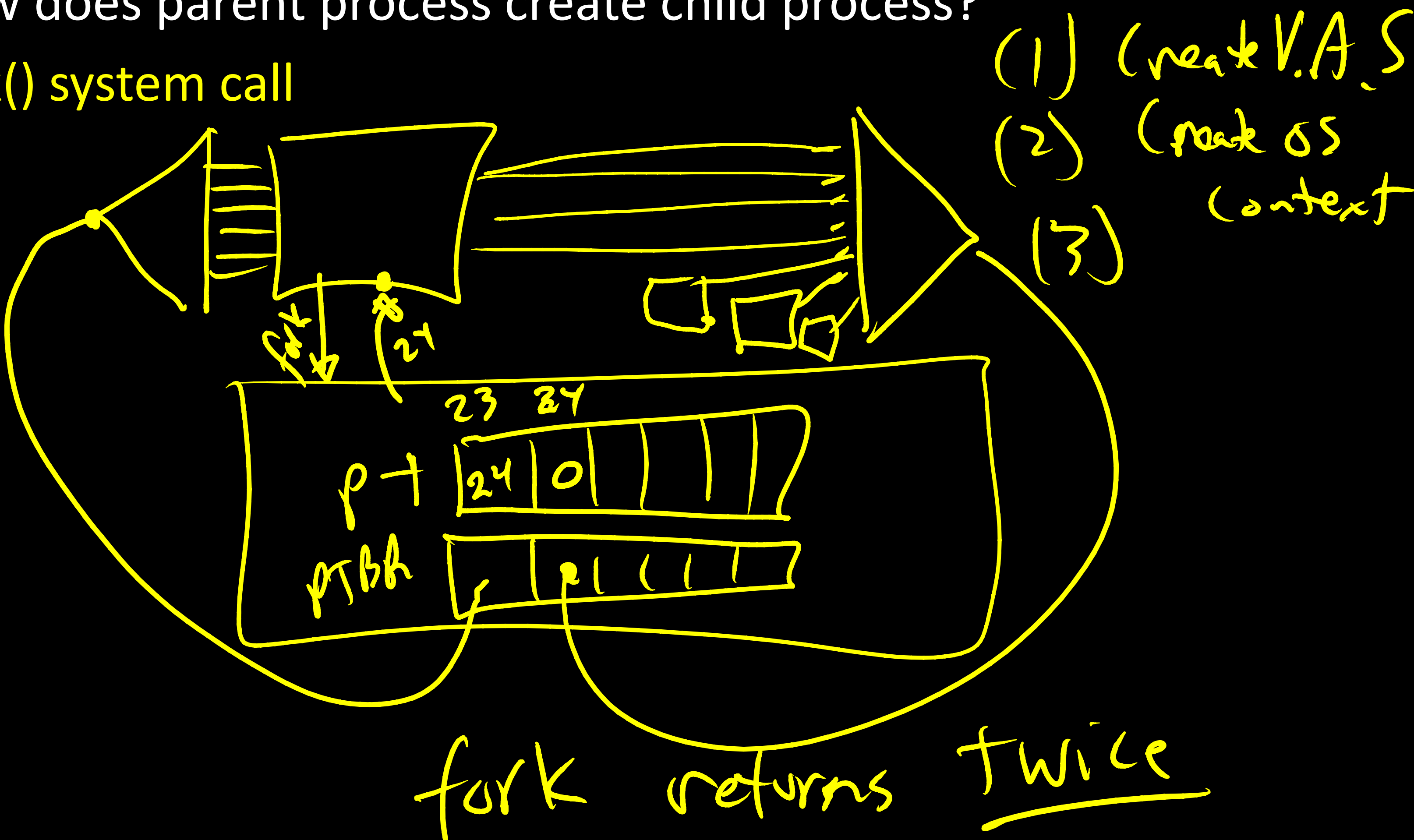
…which in turn creates other processes

- parent / child → the process tree

Init is a special case. For others…

Q: How does parent process create child process?

A: fork() system call



Wait. what? int fork() returns TWICE!

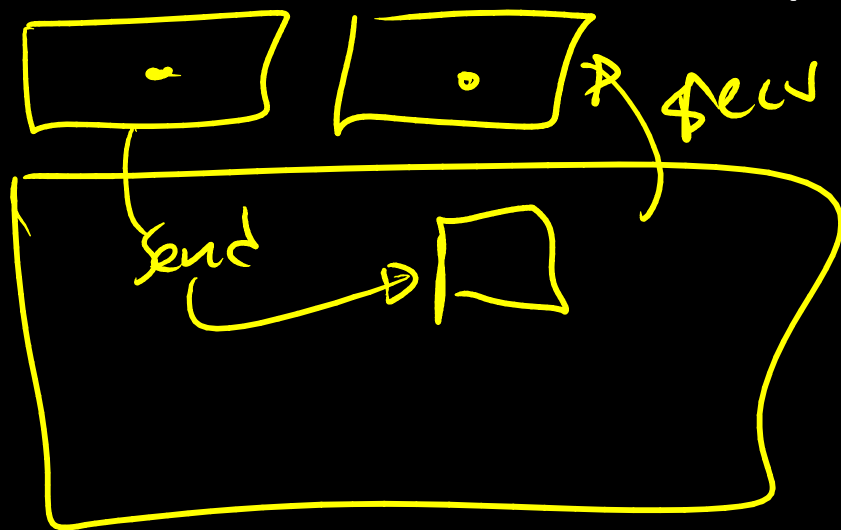
```
main(int ac, char **av) {
    int x = getpid(); // get current process ID from OS
    char *hi = av[1]; // get greeting from command line
    printf(“I’m process %d\n”, x);
    int id = fork();
    if (id == 0)
        printf(“%s from %d\n”, hi, getpid());
    else
        printf(“%s from %d, child 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 after (C-O-W ensures changes are invisible)

Q: How to continue communicating?

A: Invent OS “IPC channels” : send(msg), recv(), …



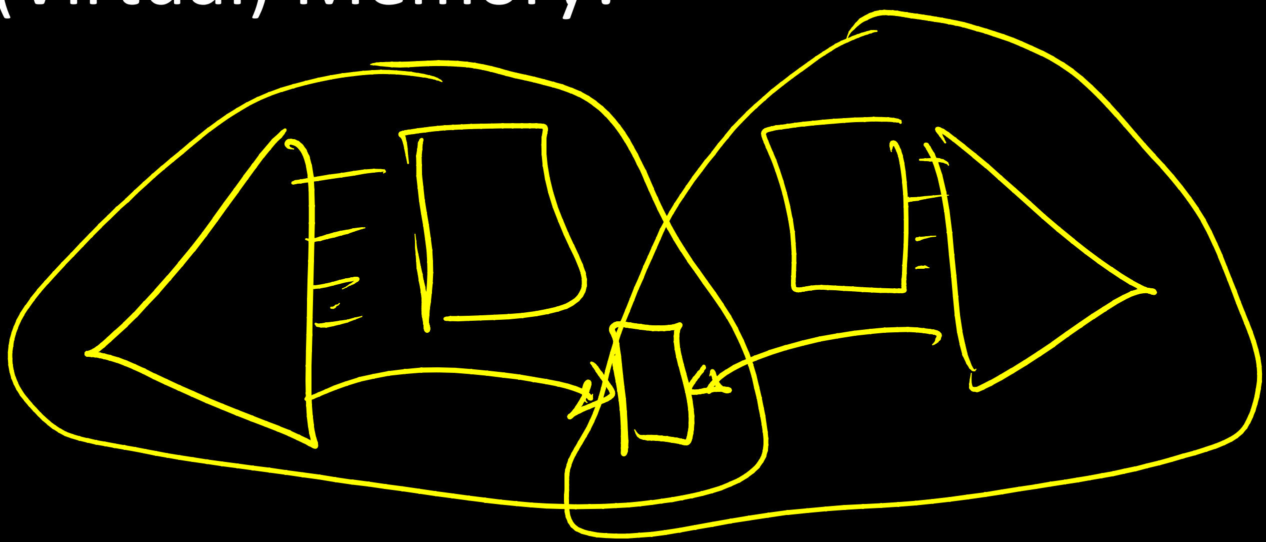
IPC
Sockets
RMI

Parent can pass information to child

- In fact, *all parent data* is passed to child
- But isolated after (C-O-W ensures changes are invisible)

Q: How to continue communicating?

A: Shared (Virtual) Memory!



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

expensive

Recall: process = execution context + address space

Thread

Process

Process

OS abstraction of a running computation

- The unit of execution
- ~~The unit of scheduling~~
- (Execution state or a few of them) + address space

From process perspective

- a virtual CPU multi-core
- some virtual memory
- a virtual keyboard, screen, …

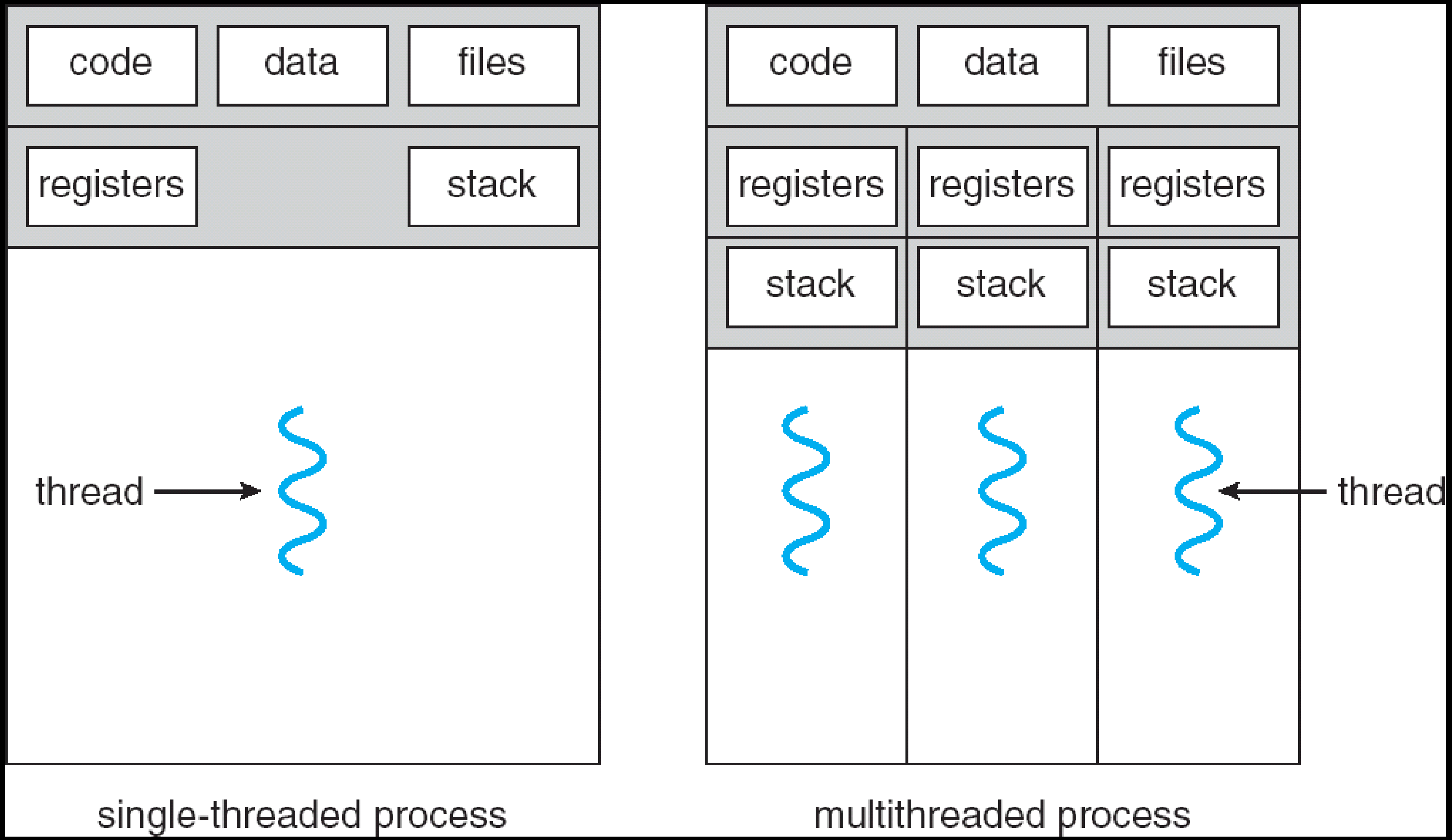
Thread

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


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

```
#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 1, counter is 1
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	Thread 205
read hits (1)	(3) read hits
addi (2)	(4) addi
write hits (5)	(6) write hits

(look familiar?)

Timing-dependent failure ⇒ race condition

- hard to reproduce ⇒ hard to debug

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"

Thread 52

```
read hits
addi
write hits
```

Thread 205

```
read hits
addi
write hits
```

Q: How to implement critical section in code?

A: Lots of approaches….

Disable interrupts?

CSEnter() = disable interrupts (including clock)

CSExit() = re-enable interrupts

```
read hits
addi
write hits
```

works but...

Kernel ... ok.

Works for some kernel data-structures

Very bad idea for user code

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

```
read hits
addi
write hits
```

Doesn't work if interrupts are part of the problem

Usually a bad idea anyway

Q: How to implement critical section in code?

A: Lots of approaches….

Mutual Exclusion Lock (mutex)

acquire(m): wait till it becomes free, then lock it

release(m): unlock it

```
apache_got_hit() {
    pthread_mutex_lock(m);
    hits = hits + 1;
    pthread_mutex_unlock(m)
}
```

Q: How to implement mutexes?