Synchronization

Hakim Weatherspoon

CS 3410, Spring 2013

Computer Science

Cornell University

P&H Chapter 2.11 and 5.8

Big Picture: Parallelism and Synchronization

How do I take advantage of multiple processors; *parallelism*?

How do I write (**correct**) parallel programs, *cache cohrency and synchronization*?

What primitives do I need to implement correct parallel programs?

Goals for Today

Understand Cache Coherency Problem

- Define Cache coherency problem

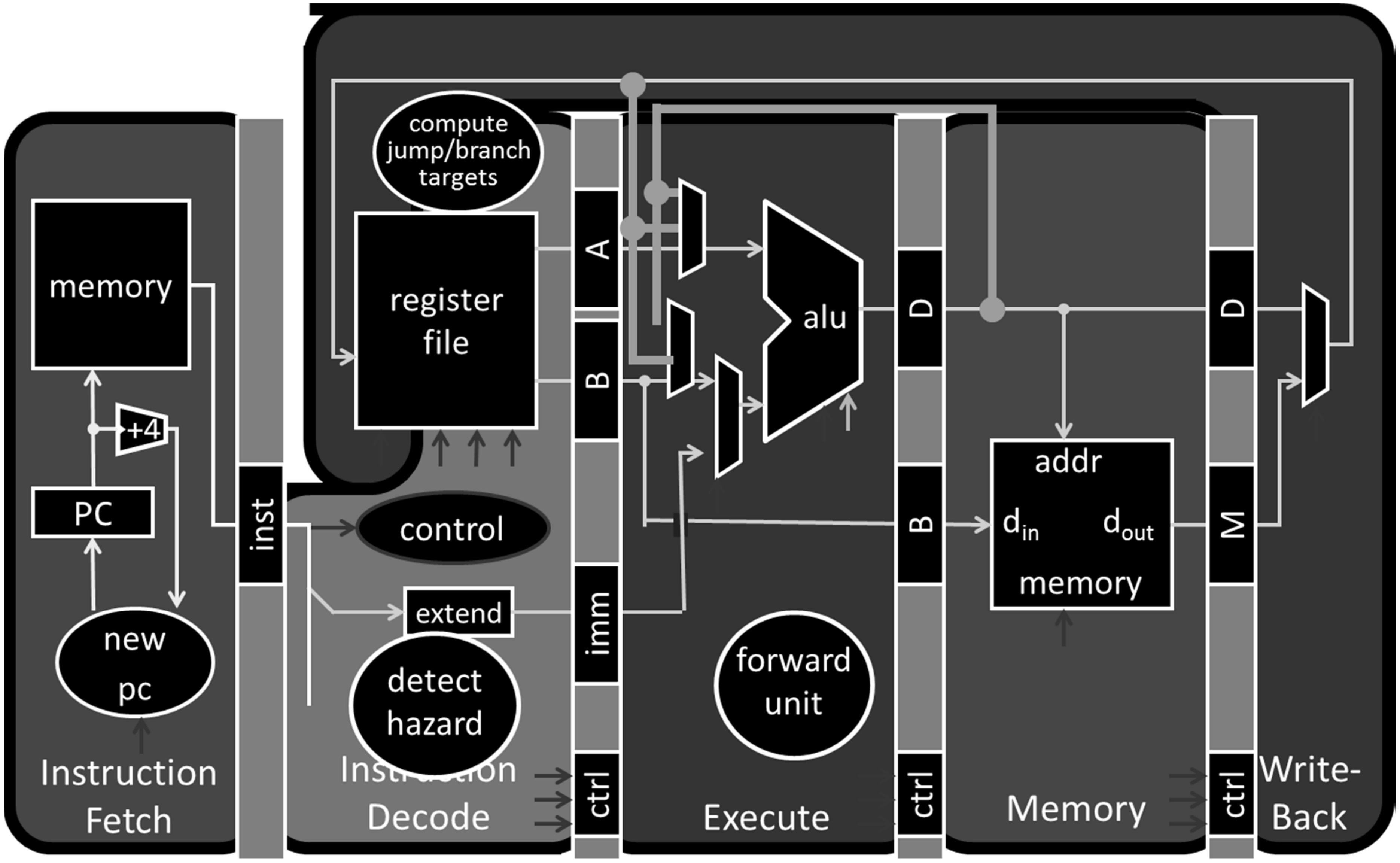
Synchronizing parallel programs

- Atomic Instructions
- HW support for synchronization

How to write parallel programs

- Threads and processes
- Critical sections, race conditions, and mutexes

Big Picture: Parallelism and Synchronization



Next Goal: Parallelism and Synchronization

Cache Coherency Problem: What happens when to two or more processors cache ***shared*** data?

Next Goal: Parallelism and Synchronization

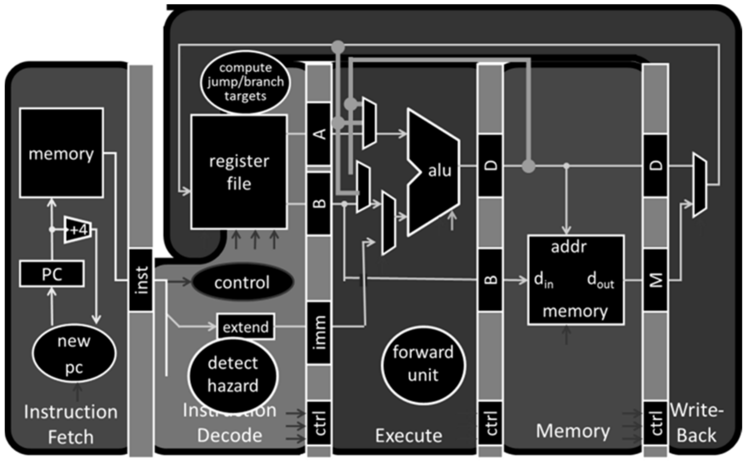
Cache Coherency Problem: What happens when to two or more processors cache *shared* data?

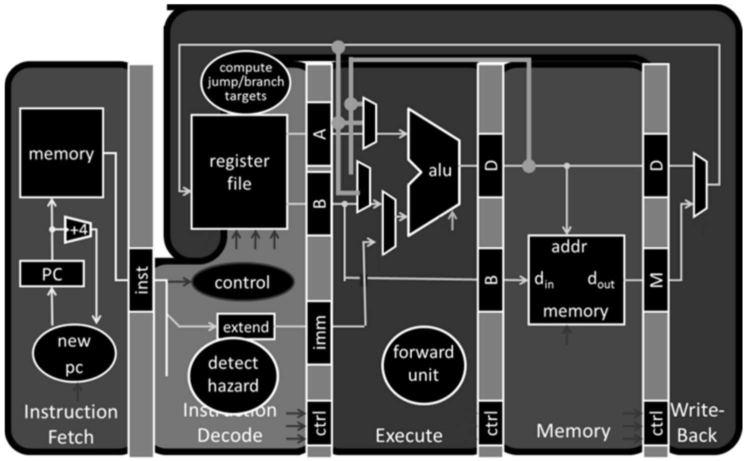
i.e. the view of memory held by two different processors is through their individual caches.

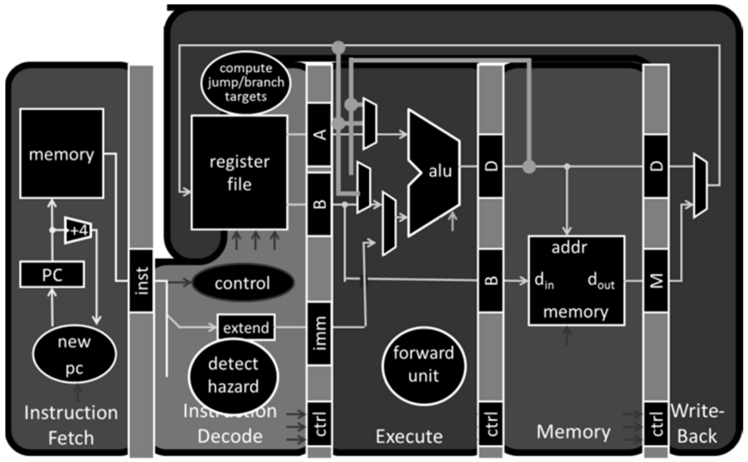
As a result, processors can see different (incoherent) values to the *same* memory location.

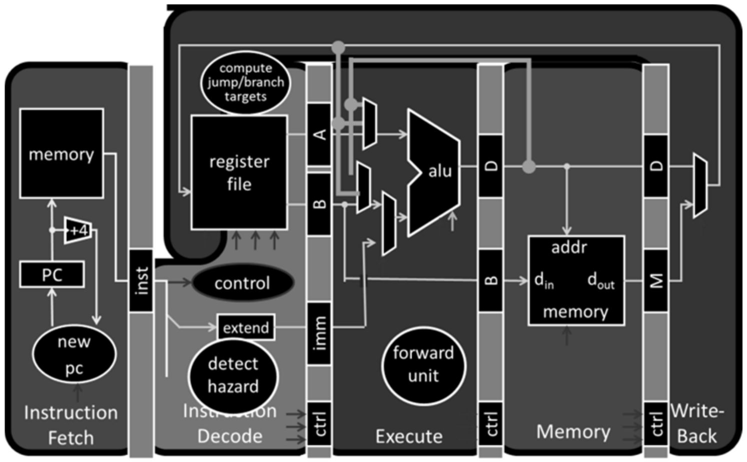
Big Picture: Parallelism and Synchronization

Each processor core has its own L1 cache



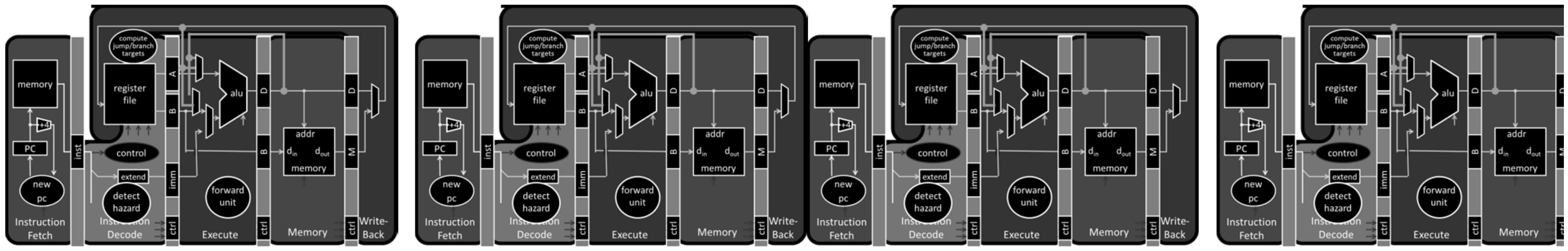






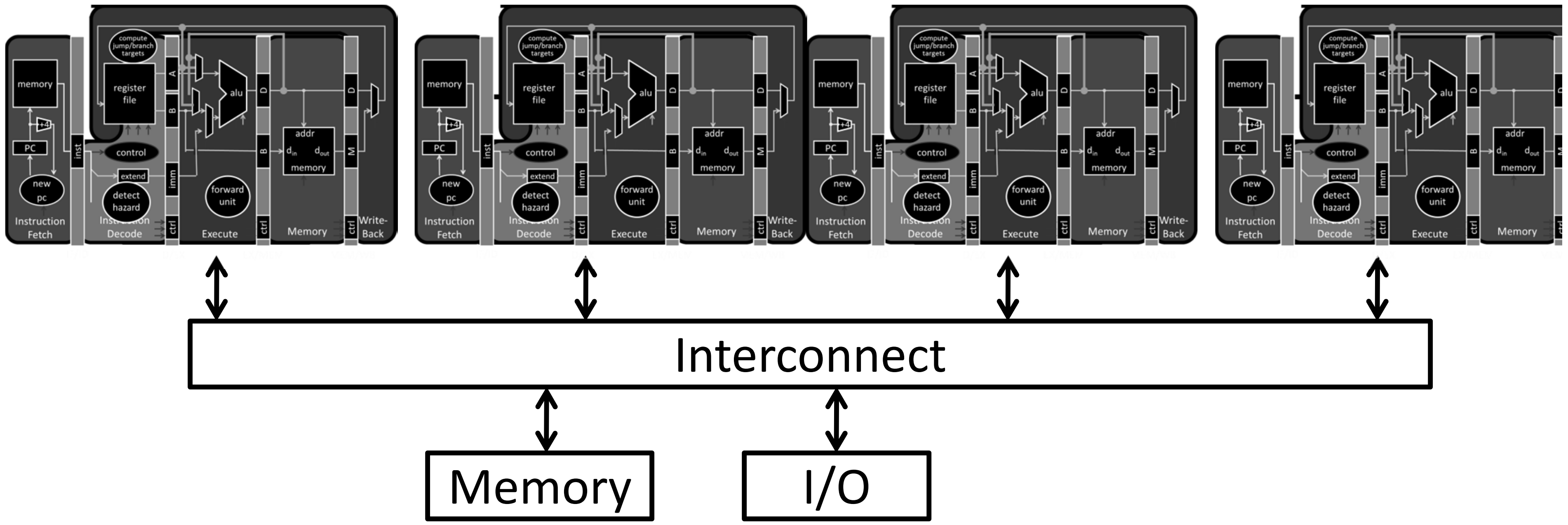
Big Picture: Parallelism and Synchronization

Each processor core has its own L1 cache



Big Picture: Parallelism and Synchronization

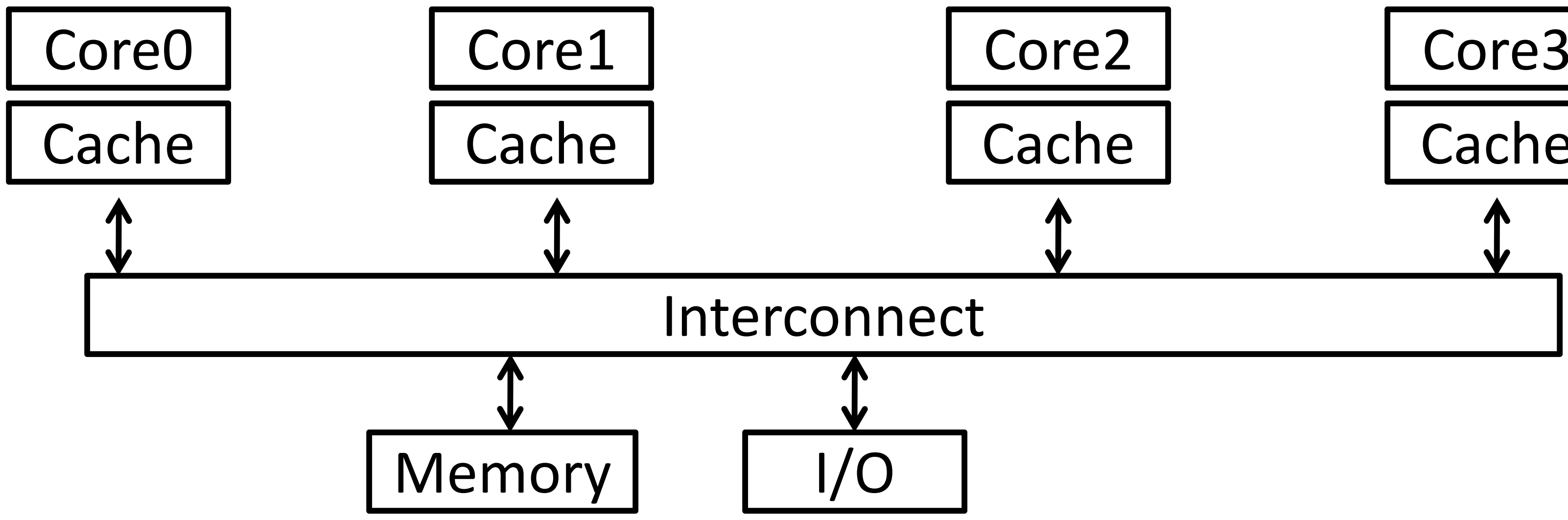
Each processor core has its own L1 cache



Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

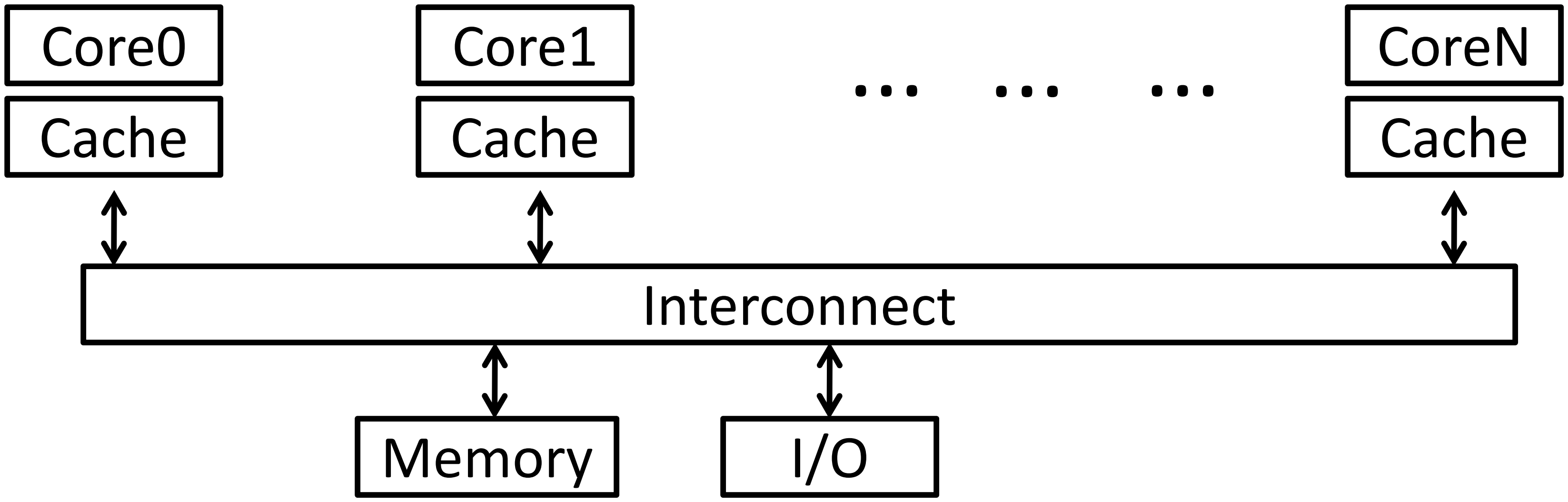
- Typical (today): 2 – 4 processor dies, 2 – 8 cores each
- HW provides *single physical address* space for all processors
- Assume physical addresses (ignore virtual memory)
- Assume uniform memory access (ignore NUMA)



Shared Memory Multiprocessors

Shared Memory Multiprocessor (SMP)

- Typical (today): 2 – 4 processor dies, 2 – 8 cores each
- HW provides *single physical address* space for all processors
- Assume physical addresses (ignore virtual memory)
- Assume uniform memory access (ignore NUMA)



Cache Coherency Problem

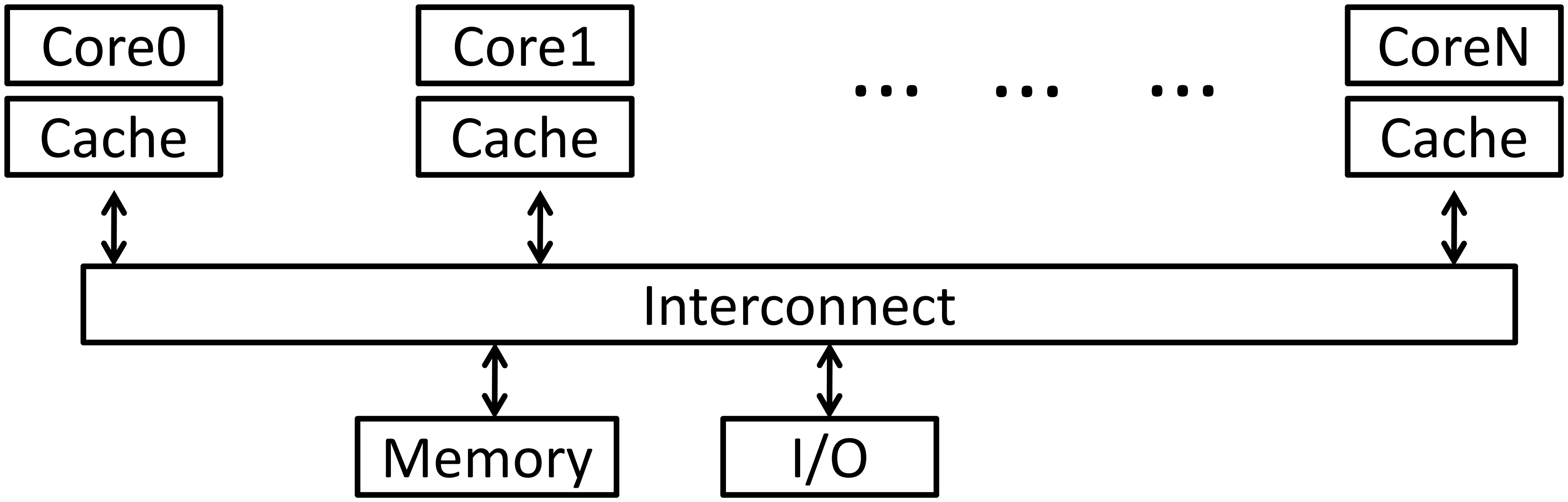
Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {
        x = x + 1;
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
        x = x + 1;
}
```

What will the value of x be after both loops finish?



Coherence Defined

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
  - Aka write serialization

Cache Coherence Protocols

Operations performed by caches in multiprocessors to ensure coherence

- Migration of data to local caches
  - Reduces bandwidth for shared memory
- Replication of read-shared data
  - Reduces contention for access

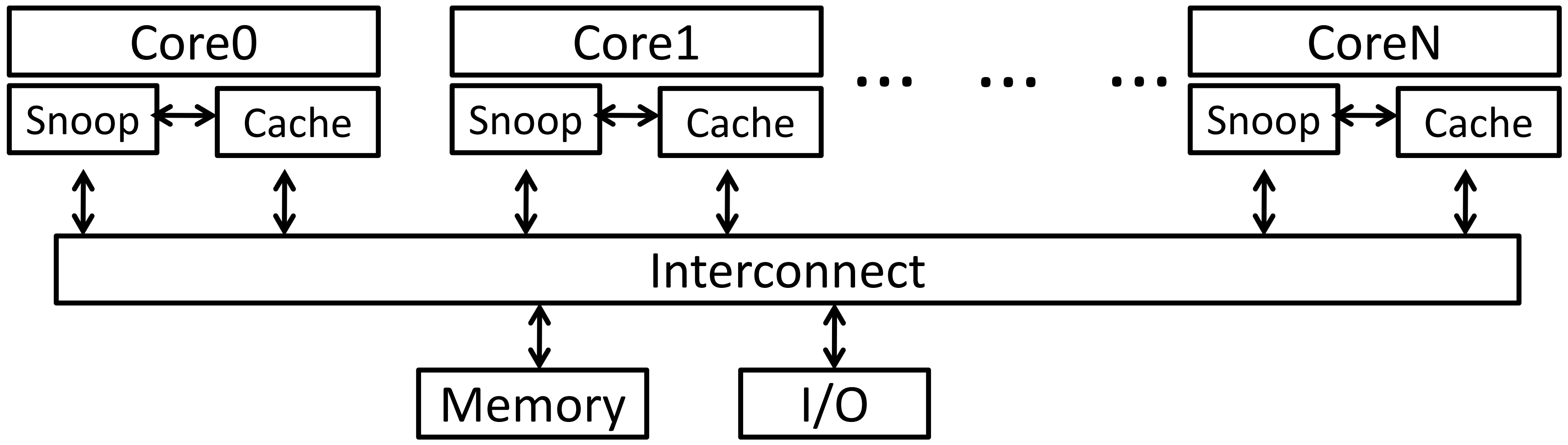
Snooping protocols

- Each cache monitors bus reads/writes

Snooping

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



Invalidating Snooping Protocols

Cache gets **exclusive access** to a block when it is to be written

- Broadcasts an invalidate message on the bus
- Subsequent read in another cache misses
  - Owning cache supplies updated value

Time Step	CPU activity	Bus activity	CPU A's cache	CPU B's cache	Memory
0					0
1	CPU A reads X	Cache miss for X	0		0
2	CPU B reads X	Cache miss for X	0	0	0
3	CPU A writes 1 to X	Invalidate for X	1		0
4	CPU B read X	Cache miss for X	1	1	1

Writing

Write-back policies for bandwidth

Write-invalidate coherence policy

- First invalidate all other copies of data
- Then write it in cache line
- Anybody else can read it

Permits one writer, multiple readers

In reality: many coherence protocols

- Snooping doesn't scale
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory

Takeaway

Informally, Cache Coherency requires that reads return most recently written value.

With multiprocessors maintaining cache coherency can be difficult and requires cache coherency protocols like Snooping cache coherency protocols.

Next Goal: Synchronization

Is cache coherency sufficient?

i.e. Is cache coherency (***what*** values are read) sufficient to maintain consistency (***when*** a written value will be returned to a read). Both coherency and consistency are required to maintain consistency in shared memory programs.

Is Cache Coherency Sufficient?

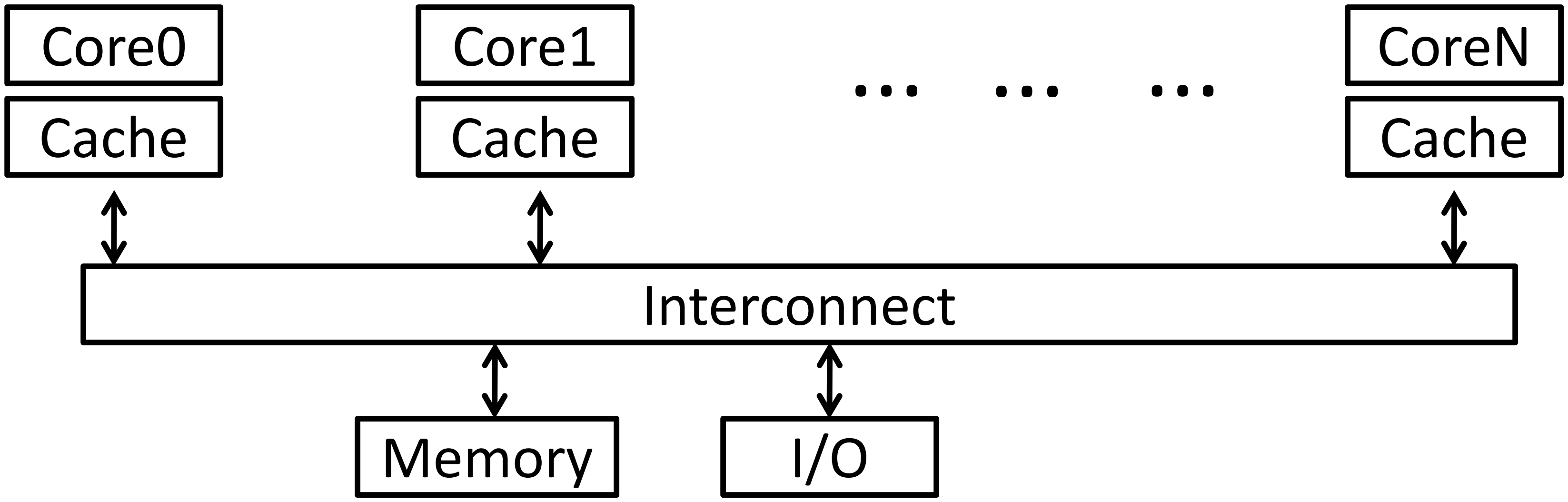
Thread A (on Core0)

```
for(int i = 0, i < 5; i++) {
        LW $t0, addr(x)
        ADDIU $t0, $t0, 1
        SW $t0, addr(x)
}
```

Thread B (on Core1)

```
for(int j = 0; j < 5; j++) {
        LW $t0, addr(x)
        ADDIU $t0, $t0, 1
        SW $t0, addr(x)
}
```

Very expensive and difficult to maintain consistency



Synchronization

Two processors sharing an area of memory

- P1 writes, then P2 reads
- Data race if P1 and P2 don't ***synchronize***
  - Result depends of order of accesses

Hardware support required

- Atomic read/write memory operation
- No other access to the location allowed between the read and write

Could be a single instruction

- E.g., atomic swap of register $\leftrightarrow$ memory (e.g. ATS, BTS; x86)
- Or an atomic pair of instructions (e.g. LL and SC; MIPS)

Synchronization in MIPS

Load linked: `LL rt, offset(rs)`

Store conditional: `SC rt, offset(rs)`

- Succeeds if location not changed since the `LL`
  - Returns 1 in rt
- Fails if location is changed
  - Returns 0 in rt

Example: atomic swap (to test/set lock variable)

```
try: MOVE $t0,$s4  ;copy exchange value
     LL  $t1,0($s1);load linked
     SC  $t0,0($s1);store conditional
     BEQZ $t0,try  ;branch store fails
     MOVE $s4,$t1  ;put load value in $s4
```

Any time a processor intervenes and modifies the value in memory between the LL and SC instruction, the SC returns 0 in $t0, causing the code to try again.

Mutex from LL and SC

Linked load / Store Conditional

m = 0; // m=0 means lock is free; otherwise, if m=1, then lock locked

```
mutex_lock(int *m) {
   while(test_and_test(m)){}
}


int test_and_set(int *m) {
     old = *m;   LL    Atomic
     *m = 1;     SC
     return old;
}
```

Mutex from LL and SC

Linked load / Store Conditional

```
m =0;
mutex_lock(int *m) {
   while(test_and_test(m)){}
}

int test_and_set(int *m) {
try: →  LI $t0, 1
        LL $t1, 0($a0)
        SC $t0, 0($a0)  ← BEQZ $t0, try
        MOVE $v0, $t1
}
```

Mutex from LL and SC

Linked load / Store Conditional

```
mutex_lock(int *m) {
   while(test_and_test(m)){}
}

int test_and_set(int *m) {
 try:
      LI $t0, 1
      LL $t1, 0($a0)
      SC $t0, 0($a0)
      BEQZ $t0, try
      MOVE $v0, $t1
}
```

Mutex from LL and SC

Linked load / Store Conditional

```
mutex_lock(int *m) {
   test_and_set:
             LI $t0, 1
             LL $t1, 0($a0)
             BNEZ $t1, test_and_set
             SC $t0, 0($a0)
             BEQZ $t0, test_and_set
}

mutex_unlock(int *m) {
      *m = 0;
}
```

Mutex from LL and SC

Linked load / Store Conditional

```
mutex_lock(int *m) {
   test_and_set:
           LI $t0, 1
           LL $t1, 0($a0)
           BNEZ $t1, test_and_set
           SC $t0, 0($a0)
           BEQZ $t0, test_and_set
}

mutex_unlock(int *m) {
     SW $zero, 0($a0)
}
```

This is called a
Spin lock
Aka spin waiting

Alternative Atomic Instructions

Other atomic hardware primitives

- test and set (x86)
- atomic increment (x86)
- bus lock prefix (x86)
- compare and exchange (x86, ARM deprecated)
- linked load / store conditional

(MIPS, ARM, PowerPC, DEC Alpha, …)

very expensive

Now we can write parallel and correct programs

Thread A

```
for(int i = 0, i < 5; i++) {

    mutex_lock(m);

        x = x + 1;

    mutex_unlock(m);

}
```

Thread B

```
for(int j = 0; j < 5; j++) {

    mutex_lock(m);

        x = x + 1;

    mutex_unlock(m);

}
```

Takeaway

Informally, Cache Coherency requires that reads return most recently written value.

With multiprocessors maintaining cache coherency can be difficult and requires cache coherency protocols like Snooping cache coherency protocols.

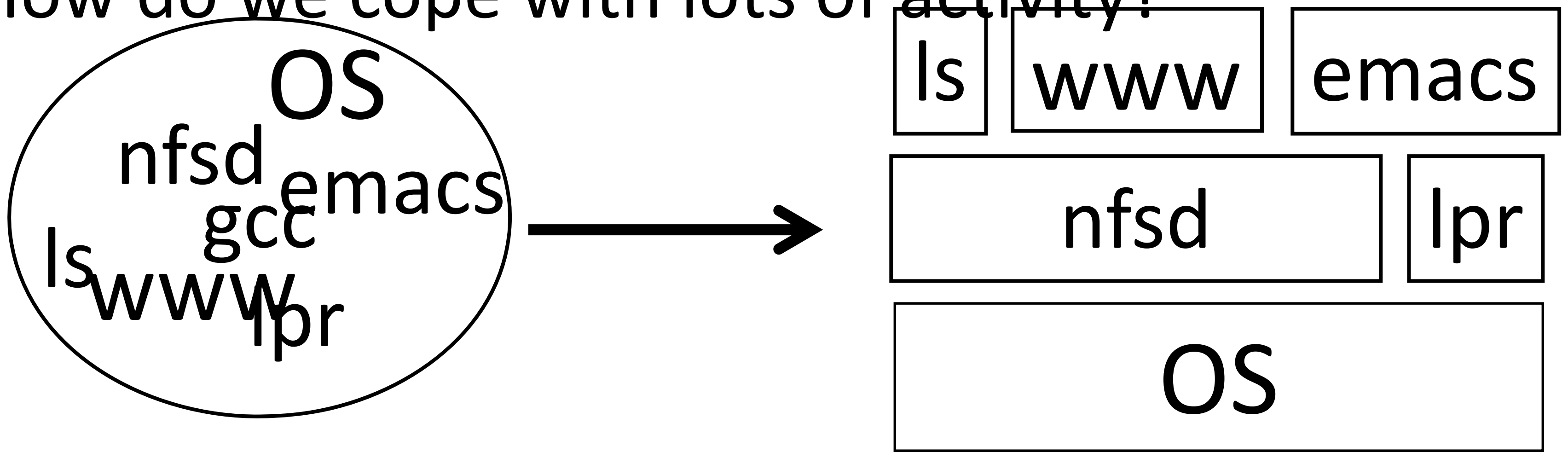
Cache coherency controls ***what*** values are read, but may be insufficient or very expensive to maintain consistency (***when*** a written value will be returned to a read). We need synchronization primitives to more efficiently implement parallel and correct programs.

Next Goal

How do we write parallel programs?

Processes

How do we cope with lots of activity?



Simplicity? Separation into processes

Reliability? Isolation

Speed? Program-level parallelism

Process and Program

Process

OS abstraction of a running computation

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

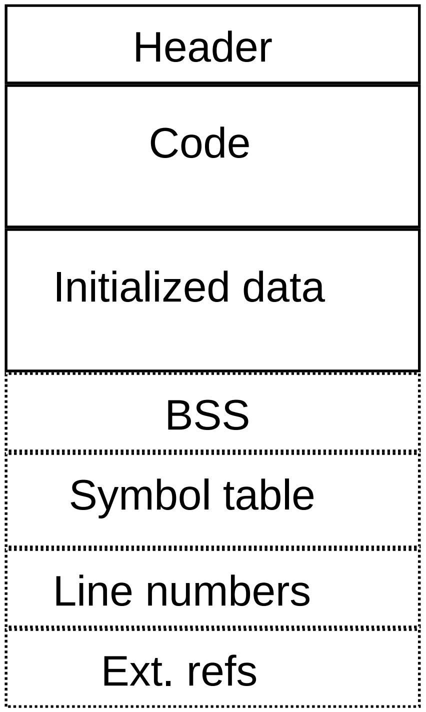
From process perspective

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

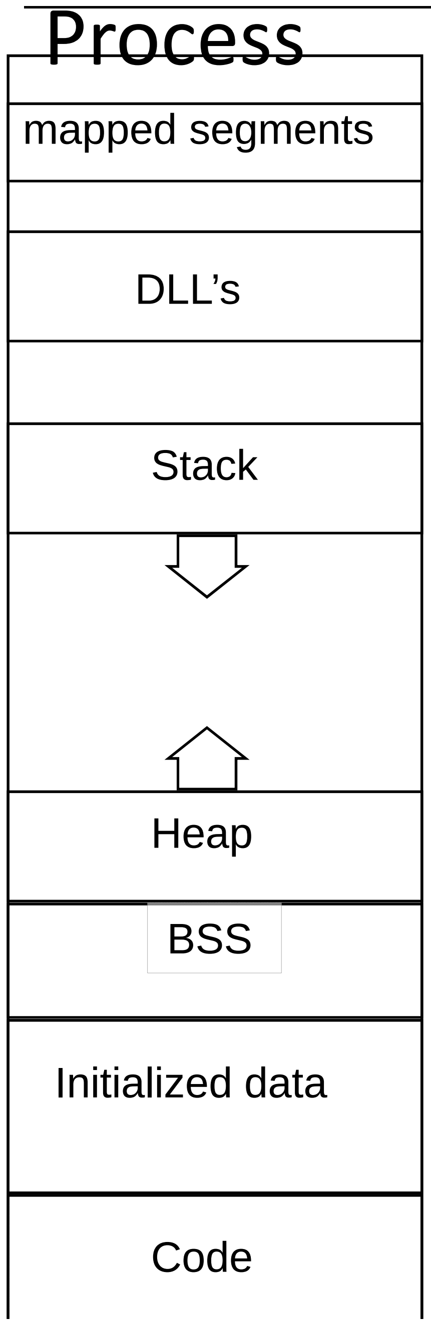
Program

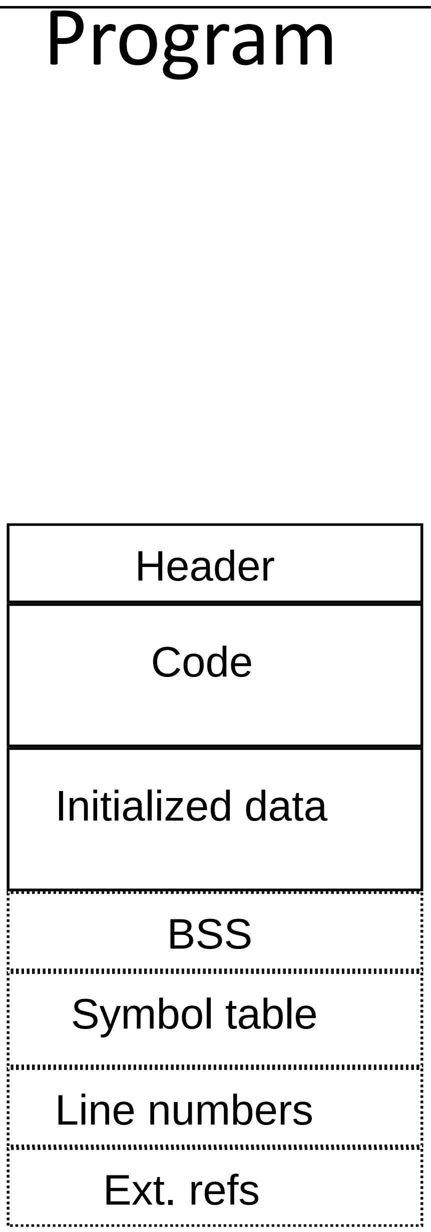
“Blueprint” for a process

- Passive entity (bits on disk)
- Code + static data



Process and Program





Role of the OS

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?

How to create a process?

Q: How to create a process?

pstree example

```
$ pstree | view -
init-+-NetworkManager-+-dhclient
     |-apache2
     |-chrome-+-chrome
     |        `-chrome
     |-chrome---chrome
     |-clementine
     |-clock-applet
     |-cron
     |-cupsd
     |-firefox---run-mozilla.sh---firefox-bin-+-plugin-cont
     |-gnome-screensaver
     |-grep
     |-in.tftpd
     |-ntpd
     `-sshd---sshd---sshd---bash-+-gcc---gcc---cc1
                                 |-pstree
                                 |-vim
                                 `-view
```

Processes Under UNIX

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!

Example

```
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 “Hey”
I’m process 23511
Hey from 23512
Hey from 23511, child is 23512
```

Inter-process Communication

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(), …

Inter-process Communication

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

Processes are heavyweight

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

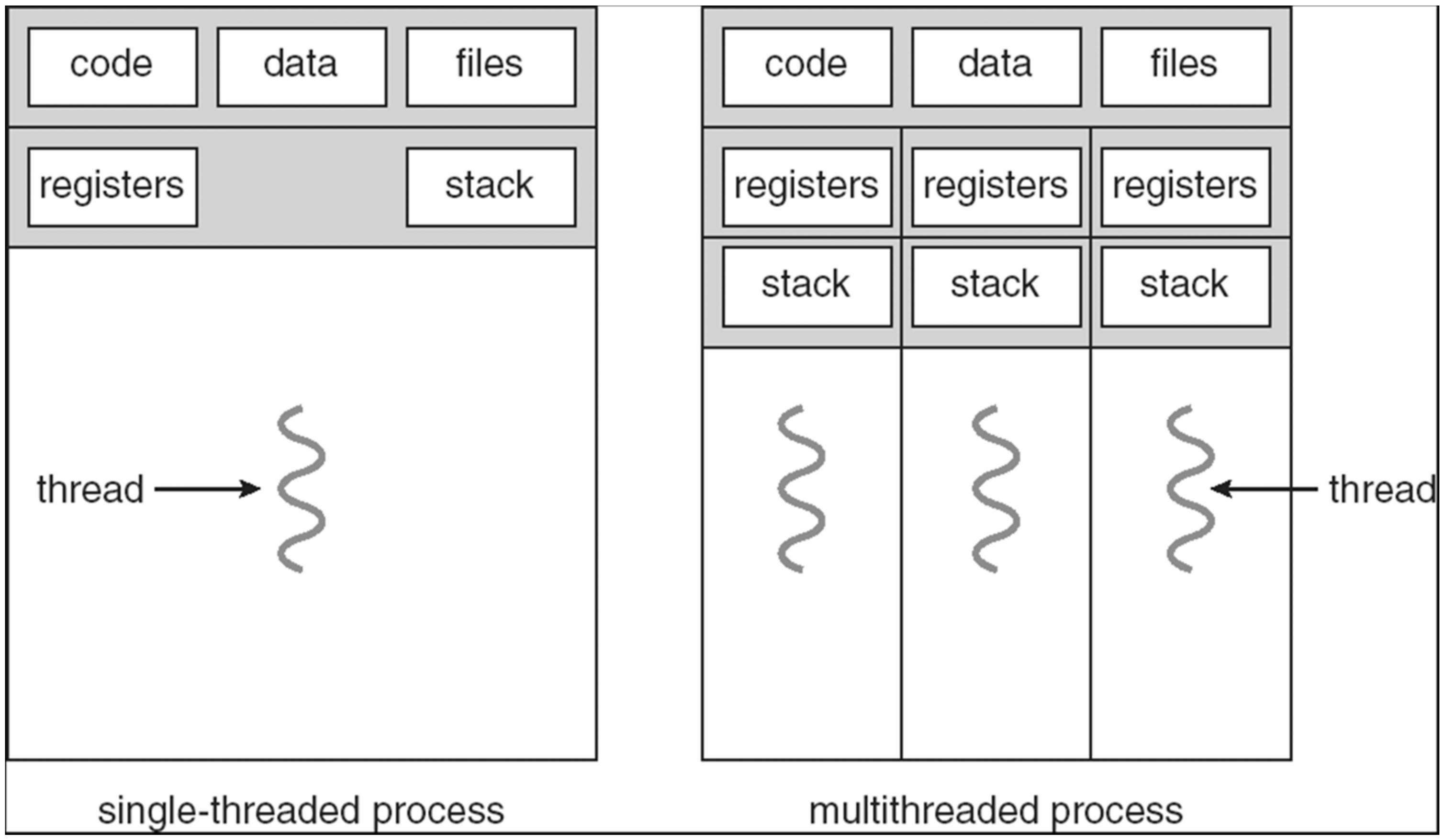
OS abstraction of a single thread of control

- The unit of scheduling
- Lives in one single process

From thread perspective

- one virtual CPU core on a virtual multi-core machine

Multithreaded Processes



Threads

```
#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);
}
```

Threads

```
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
```

Example Multi-Threaded Program

Example: Apache web server

```
void main() {
setup();
     while (c = accept_connection()) {

     req = read_request(c);
          hits[req]++;
     send_response(c, req);


     }
     cleanup();
}
```

Race Conditions

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!

Race conditions

Race Condition

Timing-dependent error when accessing shared state

- Depends on scheduling happenstance
  … e.g. 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

Critical sections

What if we can designate parts of the execution as critical sections

- Rule: only one thread can be “inside”

```
Thread 52                 Thread 205



read hits                 read hits
addi                      addi
write hits                write hits
```

Mutexes

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)
}
```

Takeaway

Informally, Cache Coherency requires that reads return most recently written value.

With multiprocessors maintaining cache coherency can be difficult and requires cache coherency protocols like Snooping cache coherency protocols.

Cache coherency controls ***what*** values are read, but may be insufficient or very expensive to maintain consistency (***when*** a written value will be returned to a read). We need synchronization primitives to more efficiently implement parallel and correct programs.

Processes and Threads are the abstraction that we use to write parallel programs? Fork and Joint and Interprocesses communication (IPC) can be used to coordinate processes. Threads are used to coordinate use of shared memory within a process.

Next time

Next time. Higher level synchronization primitives (other abstractions to implement a critical section beyond mutexes)?

Administrivia

Project3 *due next week*, Monday, April 22nd

- Design Doc due *yesterday*, Monday, April 15th
- **Games night Friday, April 26th, 5-7pm.**
- **Location: B17 Upson**

Homework4 is *due tomorrow*, Wednesday, April 17th

- Work alone
- Question1 on Virtual Memory is pre-lab question for Lab4

Prelim3 is *next week*, Thursday, April 25th

- Time and Location: 7:30pm in Phillips 101 and Upson B17
- Old prelims are online in CMS

Administrivia

Next three weeks

- Week 12 (Apr 15): Project3 design doc due and HW4 due
- Week 13 (Apr 22): Project3 due and Prelim3
- Week 14 (Apr 29): Project4 handout

Final Project for class

- Week 15 (May 6): Project4 design doc due
- Week 16 (May 13): Project4 due