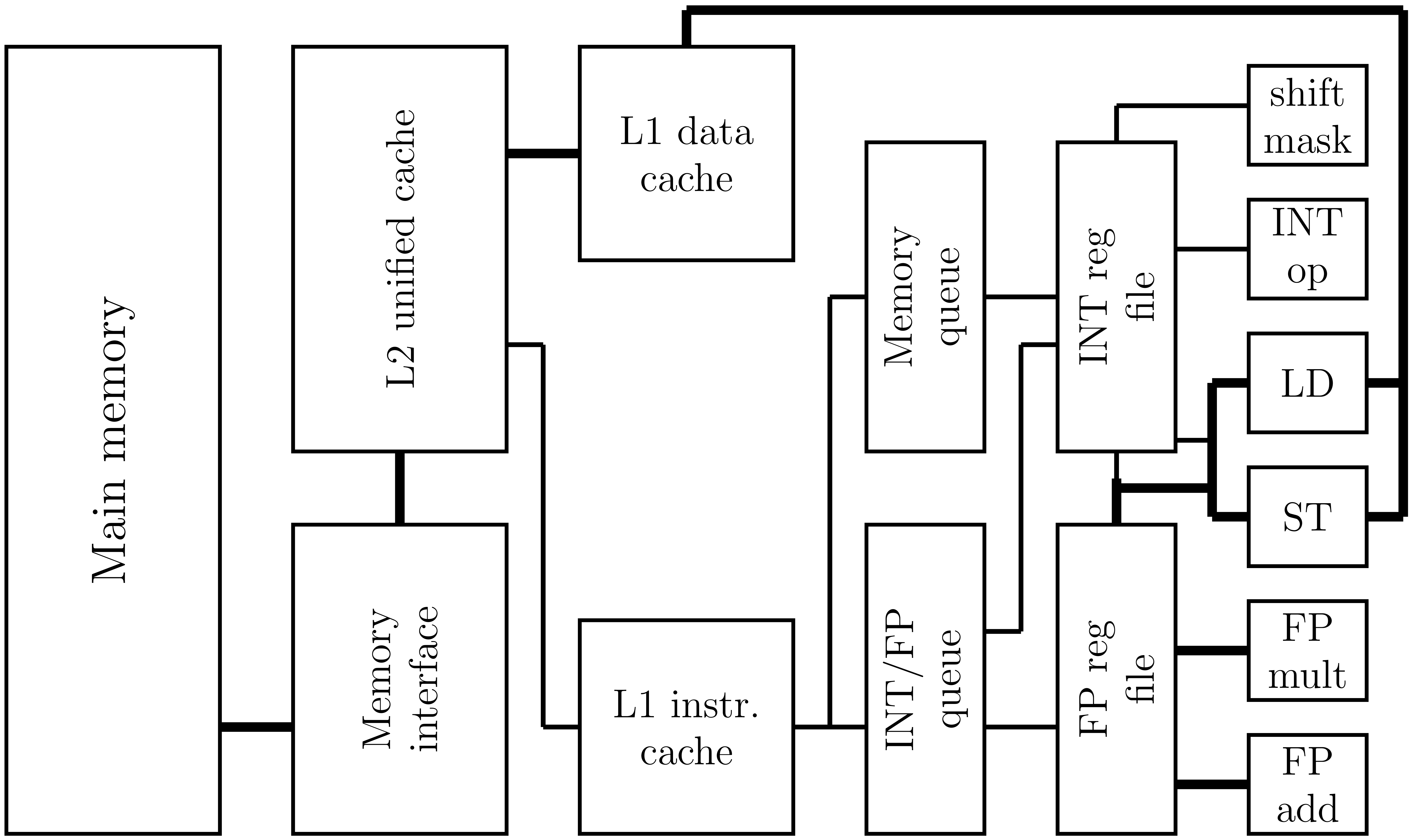
Introduction to Parallel Processing

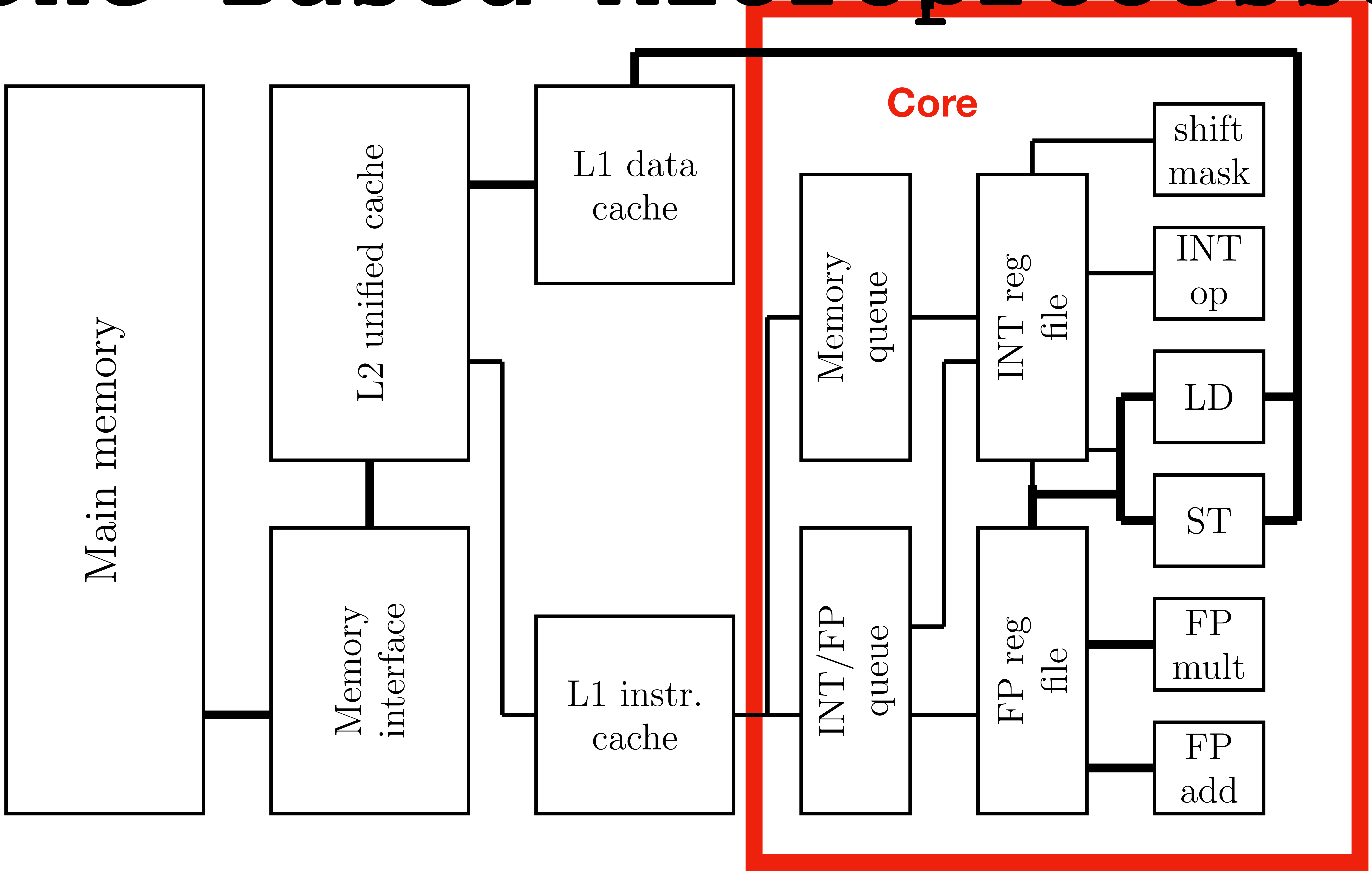
Lecture 24 : Shared Memory Parallelism

Professor Amanda Bienz

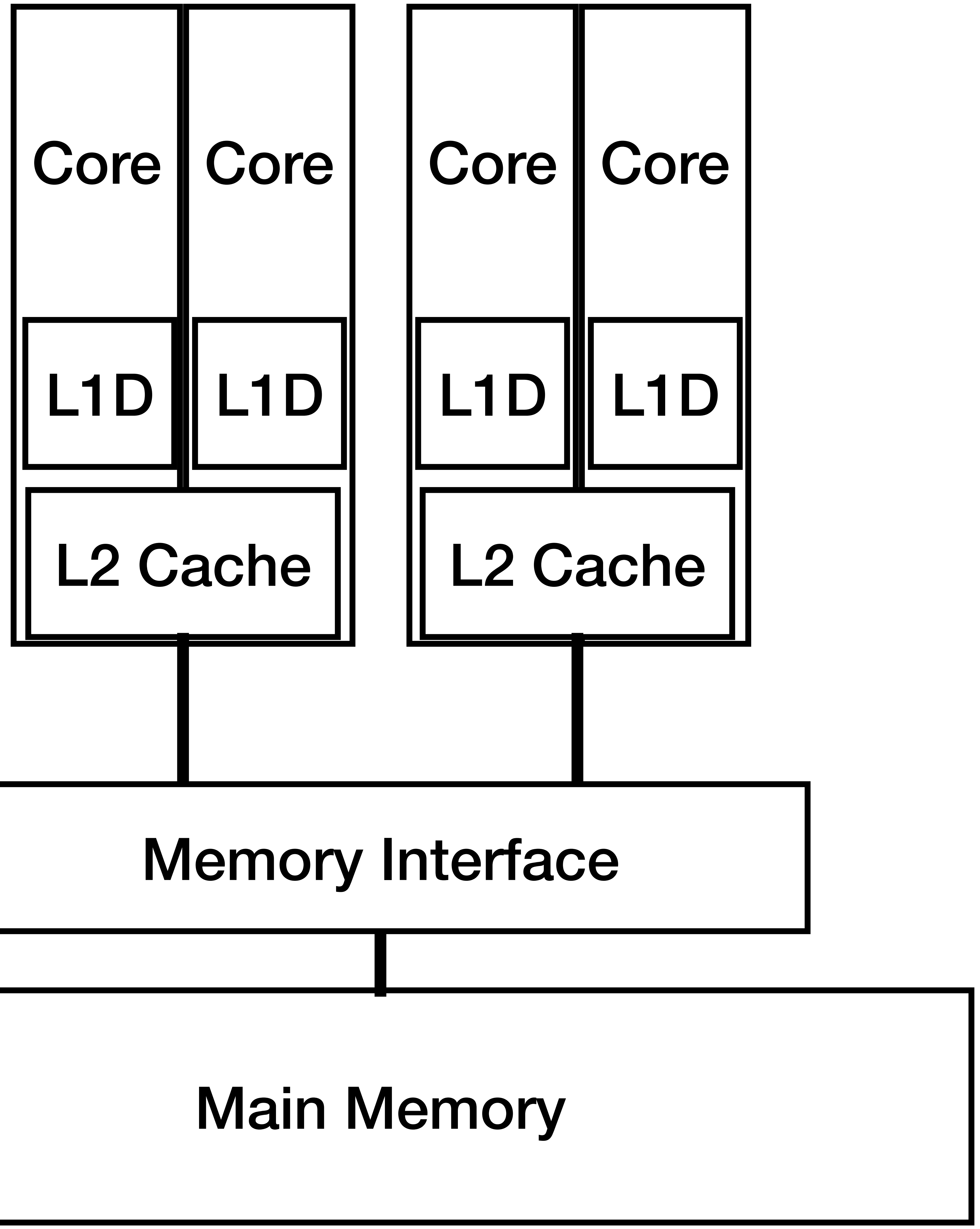
Cache-Based Microprocessor



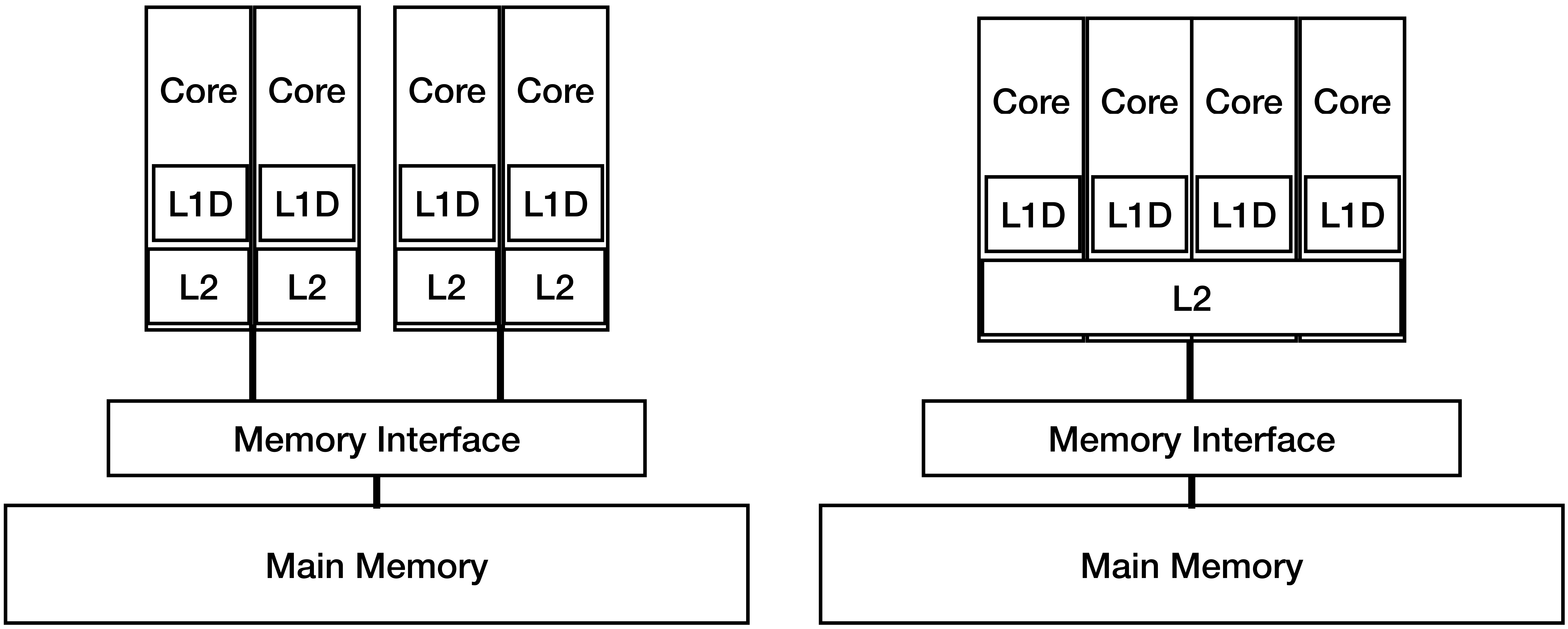
Cache-Based Microprocessor



Shared Memory Processor



Shared Memory Processor



Threads

- A ***thread*** is a single stream of control in the flow of a program
  - An independent sequence of instructions

```
For I = 0 to n:
    For j = 0 to n:
        C[i][j] = dot_poduct(get_row(a, i), get_col(b, j))
```

- n^2 different threads that can be executed independently

Threads

- A ***thread*** is a single stream of control in the flow of a program
  - An independent sequence of instructions

```
For I = 0 to n:
    For j = 0 to n:
        C[i][j] = get_thread(dot_poduct(get_row(A, i), get_col(B, j)))
```

- n^2 different threads that can be executed independently
- Underlying system schedules the threads across the processes

Threads

```
For I = 0 to n:
    For j = 0 to n:
        C[i][j] = get_thread(dot_poduct(get_row(A, i), get_col(B, j)))
```

- Each thread must have access to matrices A, B, C
- Use shared main memory to accomplish this
- All of main memory is globally accessible by each thread
- All function calls within a thread are visible only to the thread
- Values accessed by threads are stored locally

Why Threads?

- Software Potability : threaded applications can be developed in serial (on single core machines)
    - Can run on parallel machines without any changes
    - Not architecture dependent
- Latency Hiding : Memory access latency is a big bottleneck, both in serial and parallel codes. When multiple threads can execute on a single process, this latency can be hidden (while one process is accessing memory, another is performing operations)

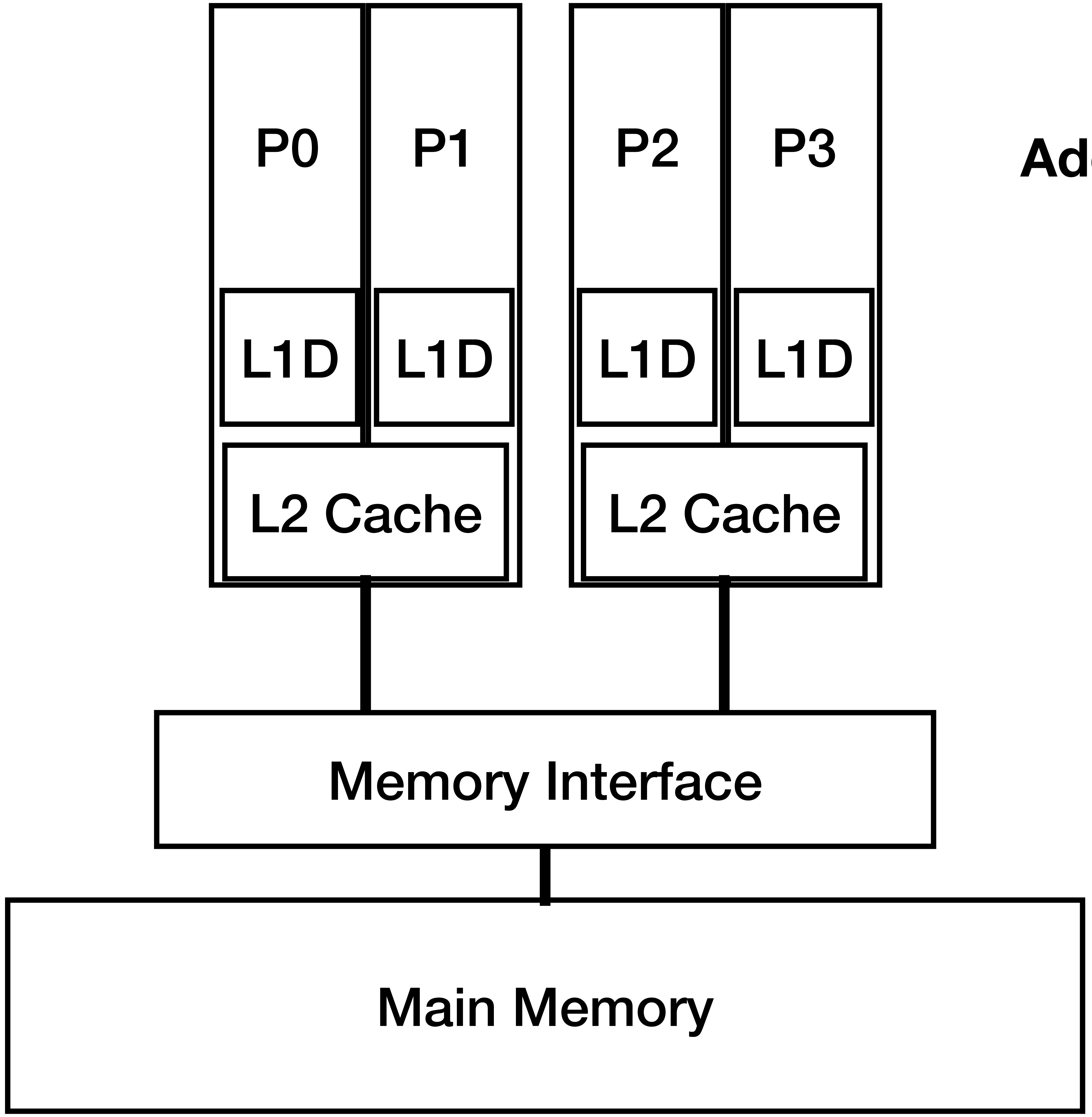
Why Threads?

- Scheduling / Load Balancing : parallel applications require programmer to split up data evenly so each process has same amount of work. Sometimes this is easy, but very difficult in unstructured or dynamic codes

- Ease of programming : Easier than MPI

- Widespread Use

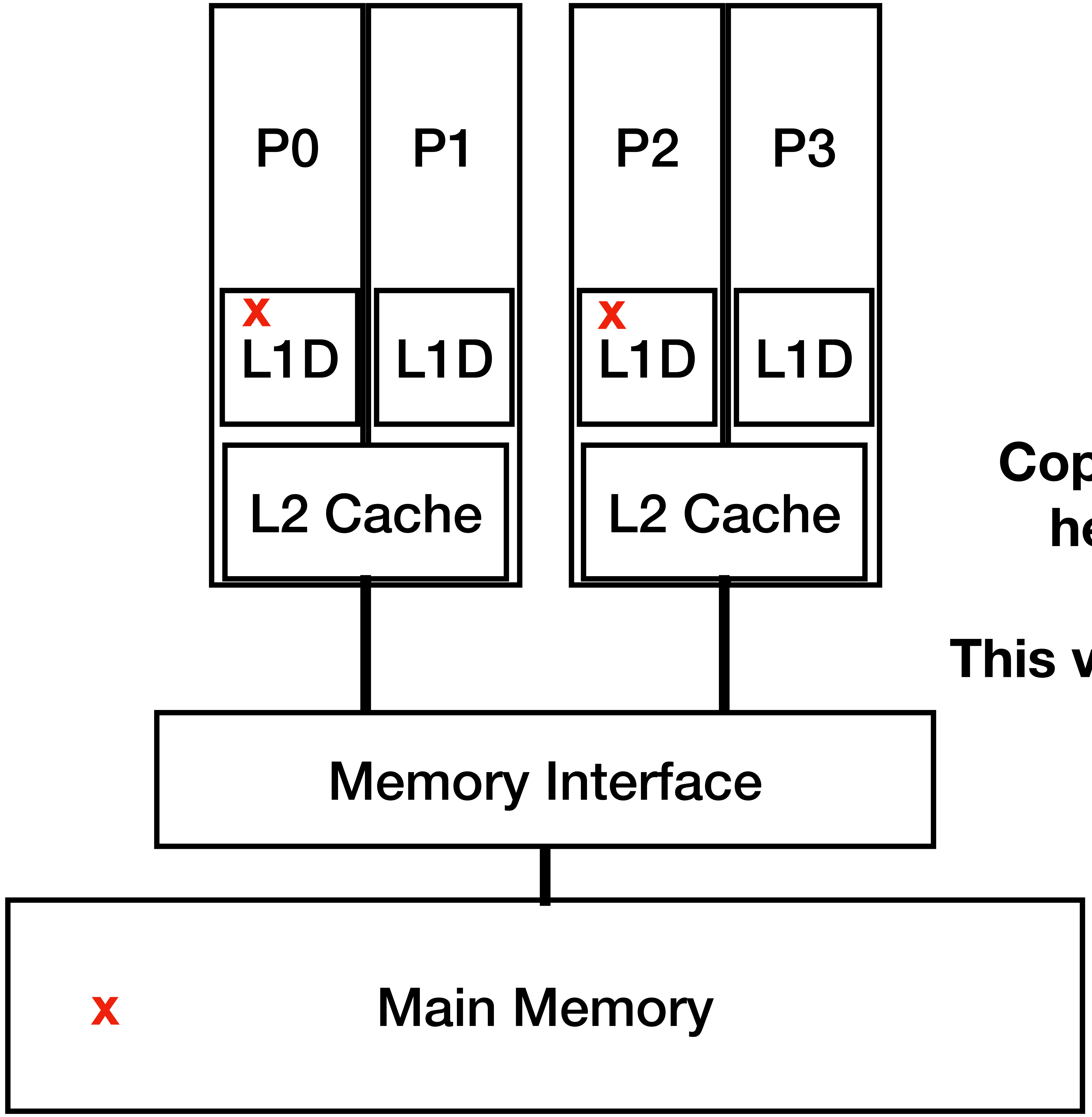
Cache Coherence



Additional hardware required to keep multiple copies of data consistent with each other

When there are multiple copies of data how to ensure different processes can operate on data in manner that follows semantics

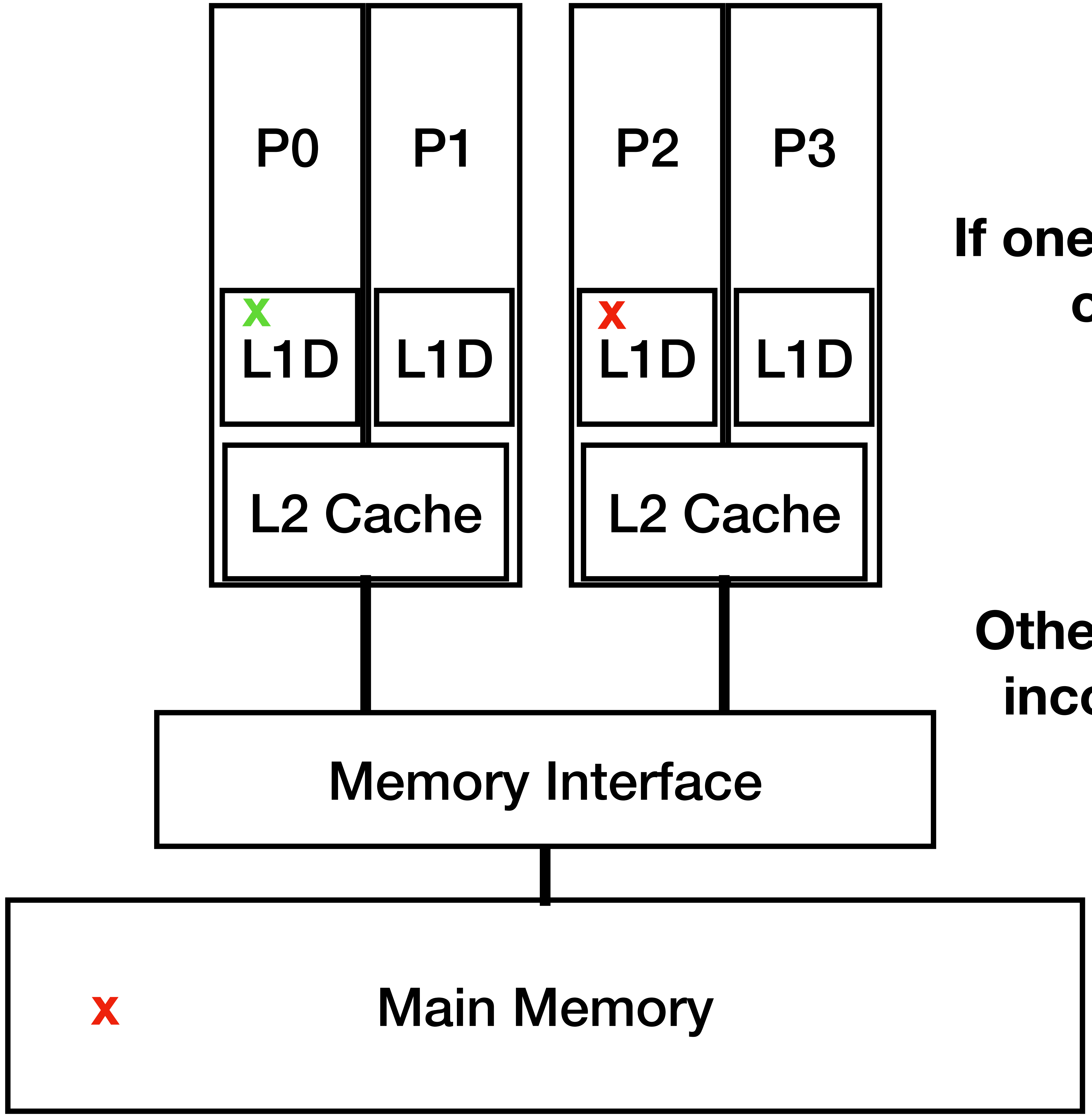
Cache Coherence



Copies of x in shared memory, held by P0, and held by P1

This variable is considered *shared*

Cache Coherence

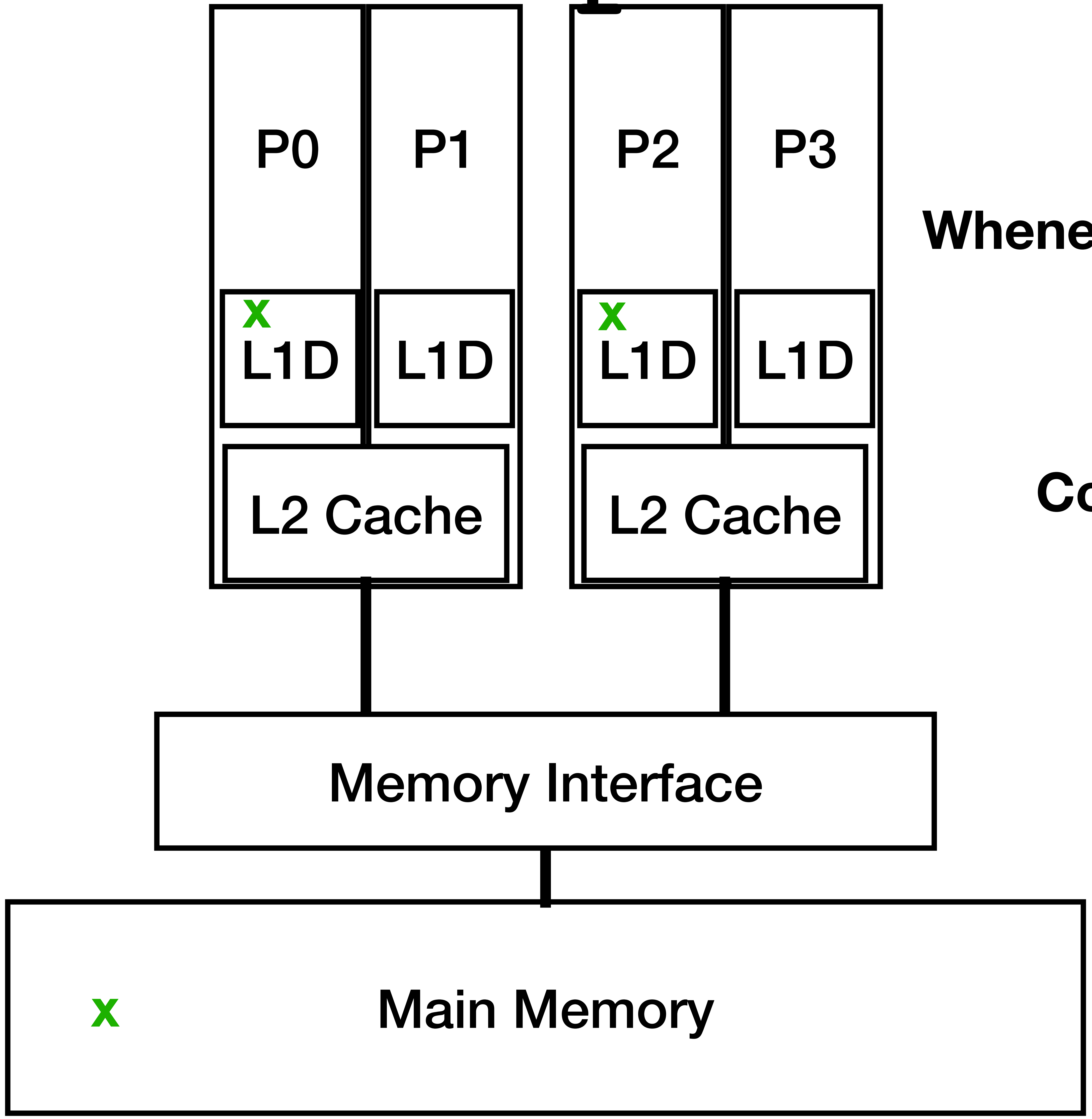


If one process changes it's copy other values are either:

1. **Invalidated**
2. **Updated**

Otherwise, P2 could work with incorrect version of the data

Update Protocol

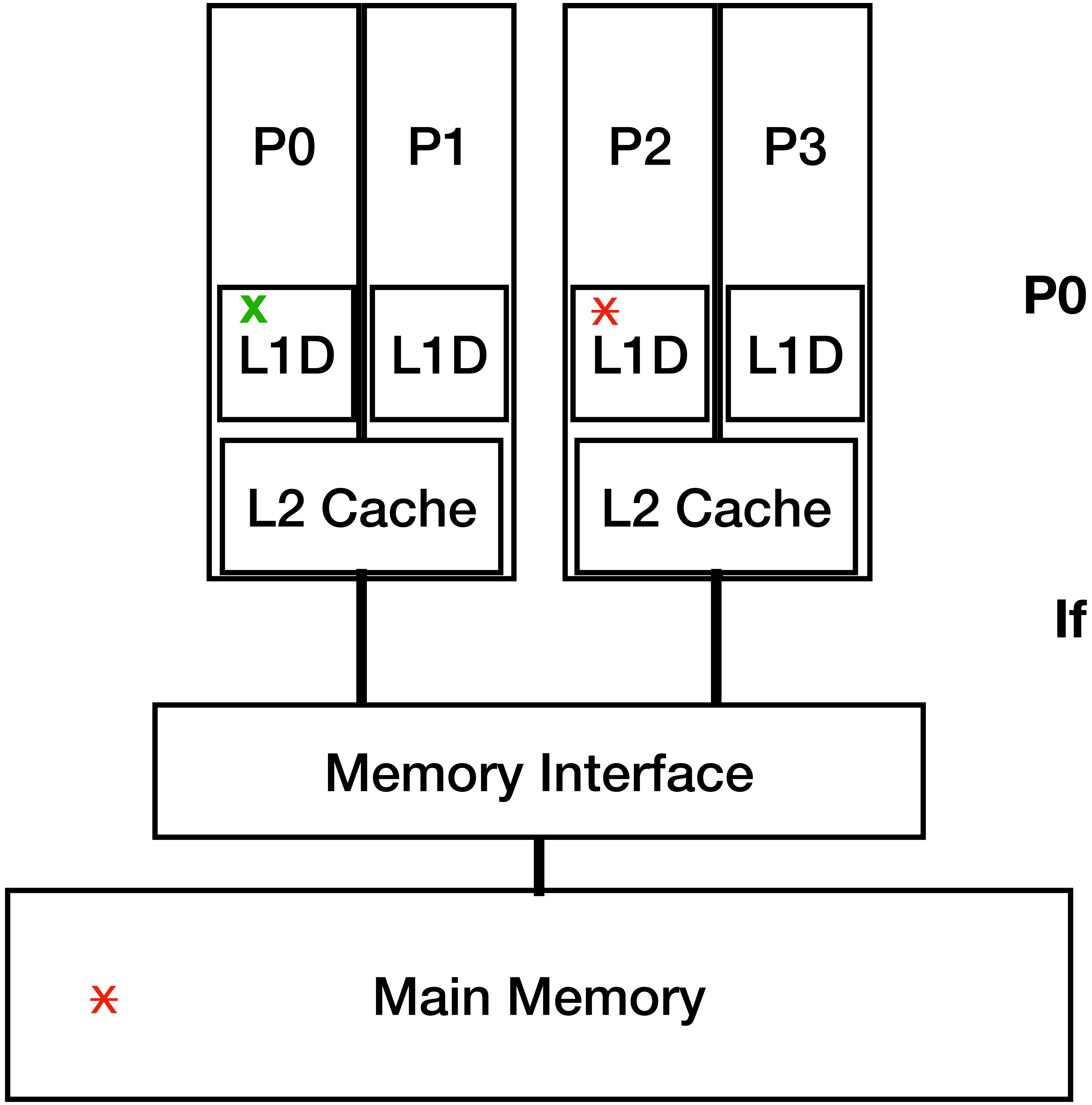


Whenever data is written, all copies in system are updated

Excess overhead if P2 never uses x again

Communication : full cache line is written to main memory and sent to P2

Invalidate Protocol



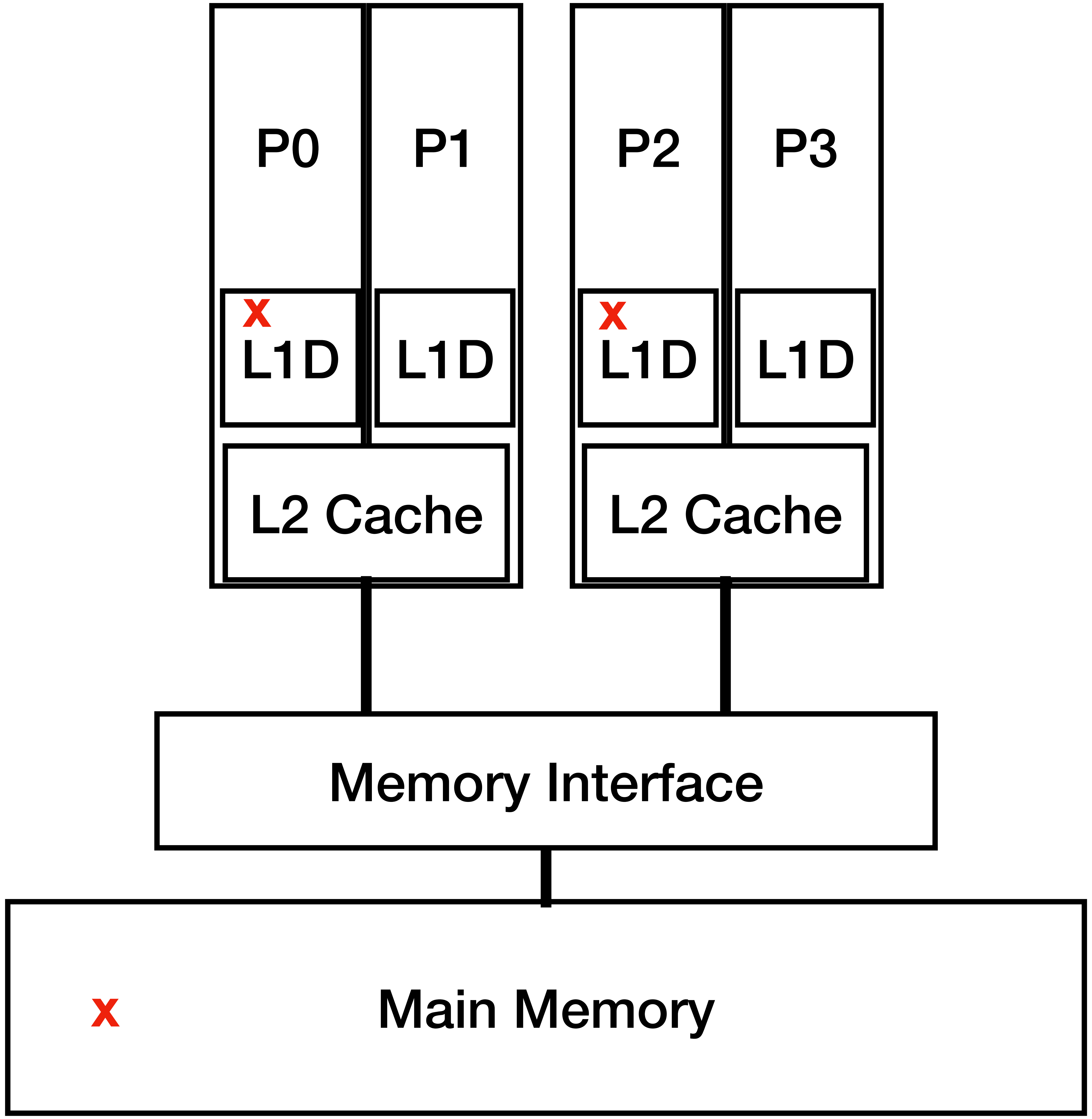
P0 marks x as dirty, invalidating values stored on P2 and in main memory

Invalidates cache line on first update

If P0 updates x again, or other values on same cache line, P2 doesn't need to know

Typically the process that is used today

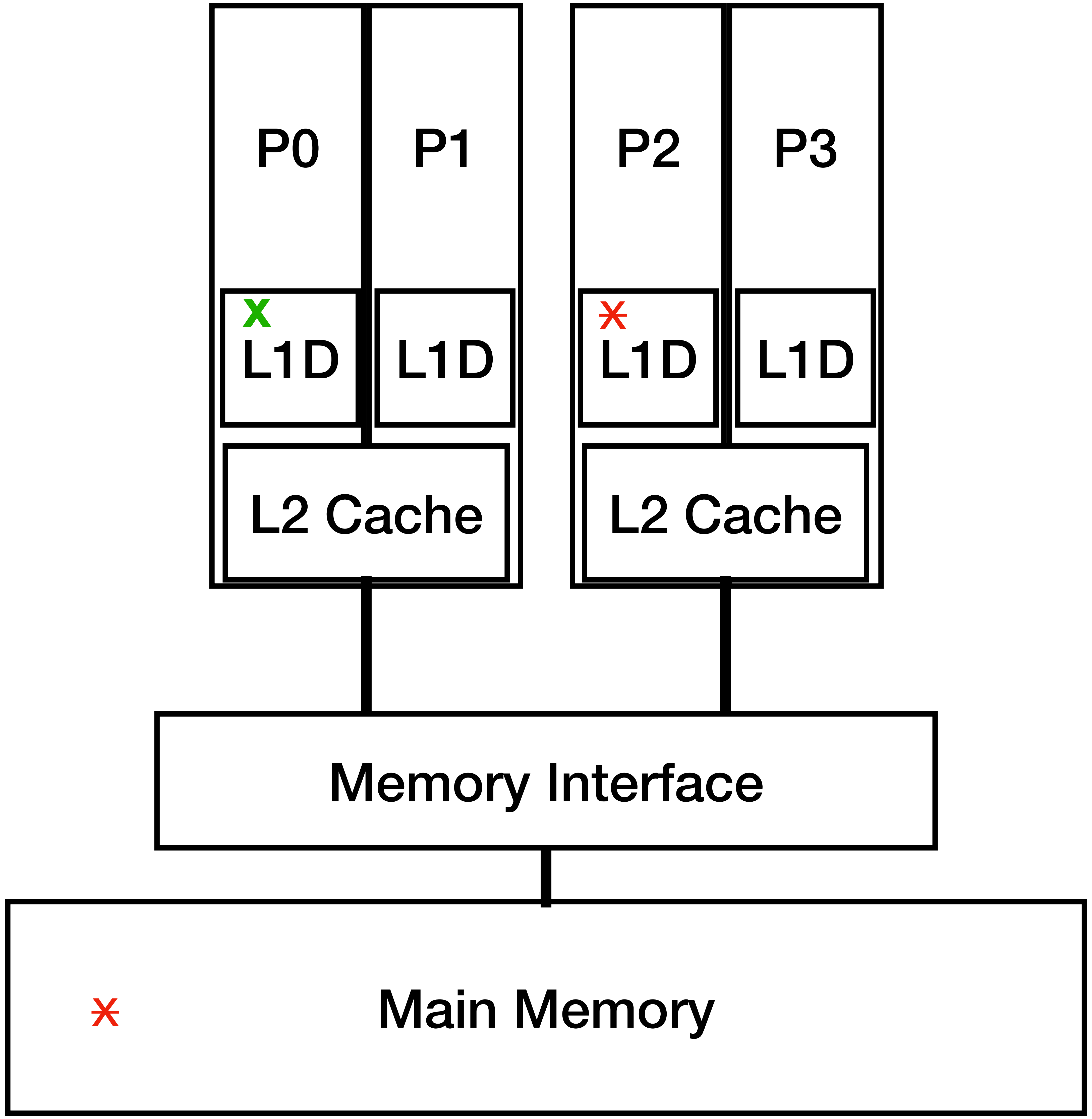
Cache Coherence



Processes P0 and P2 hold copies of x

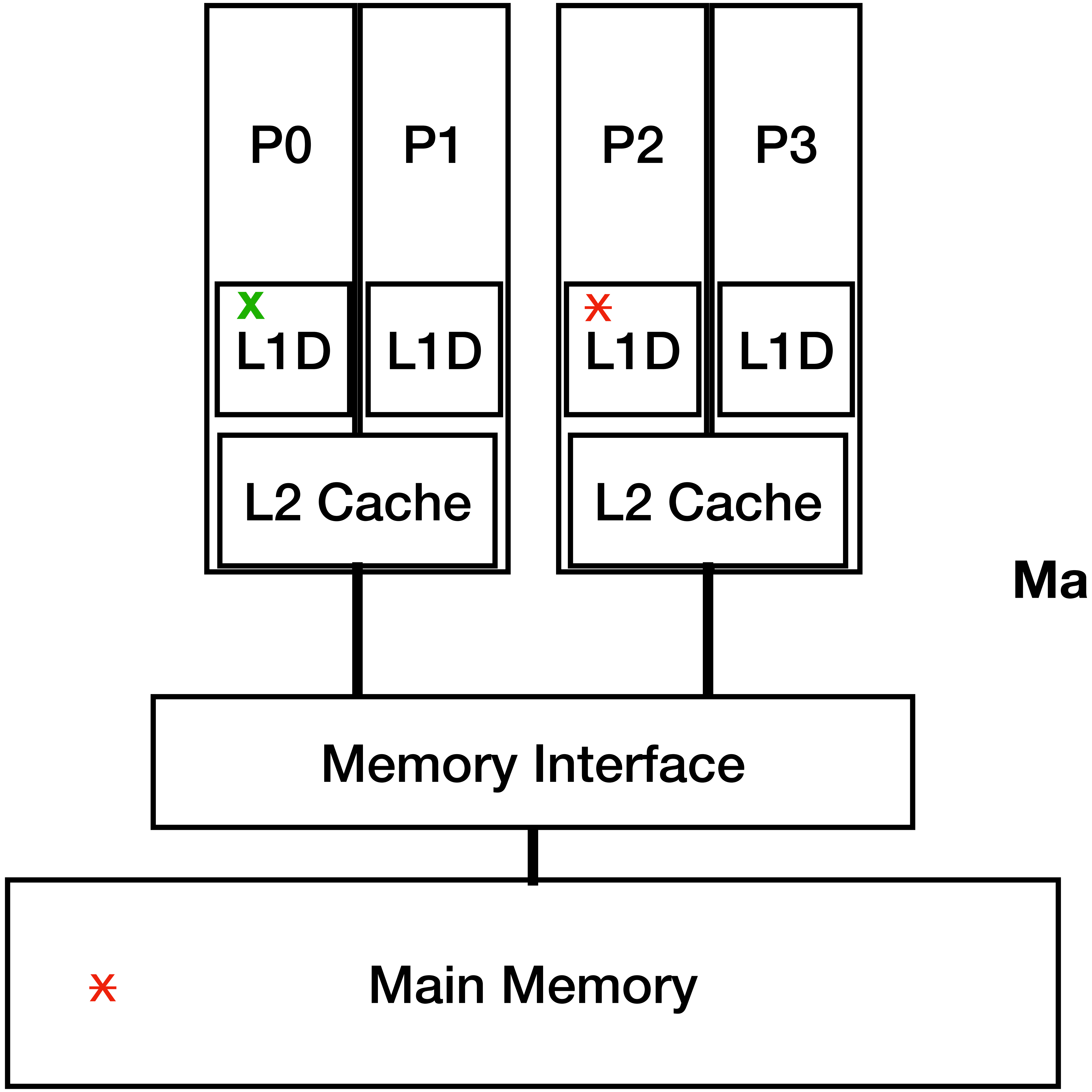
Here, x is *shared*

Cache Coherence



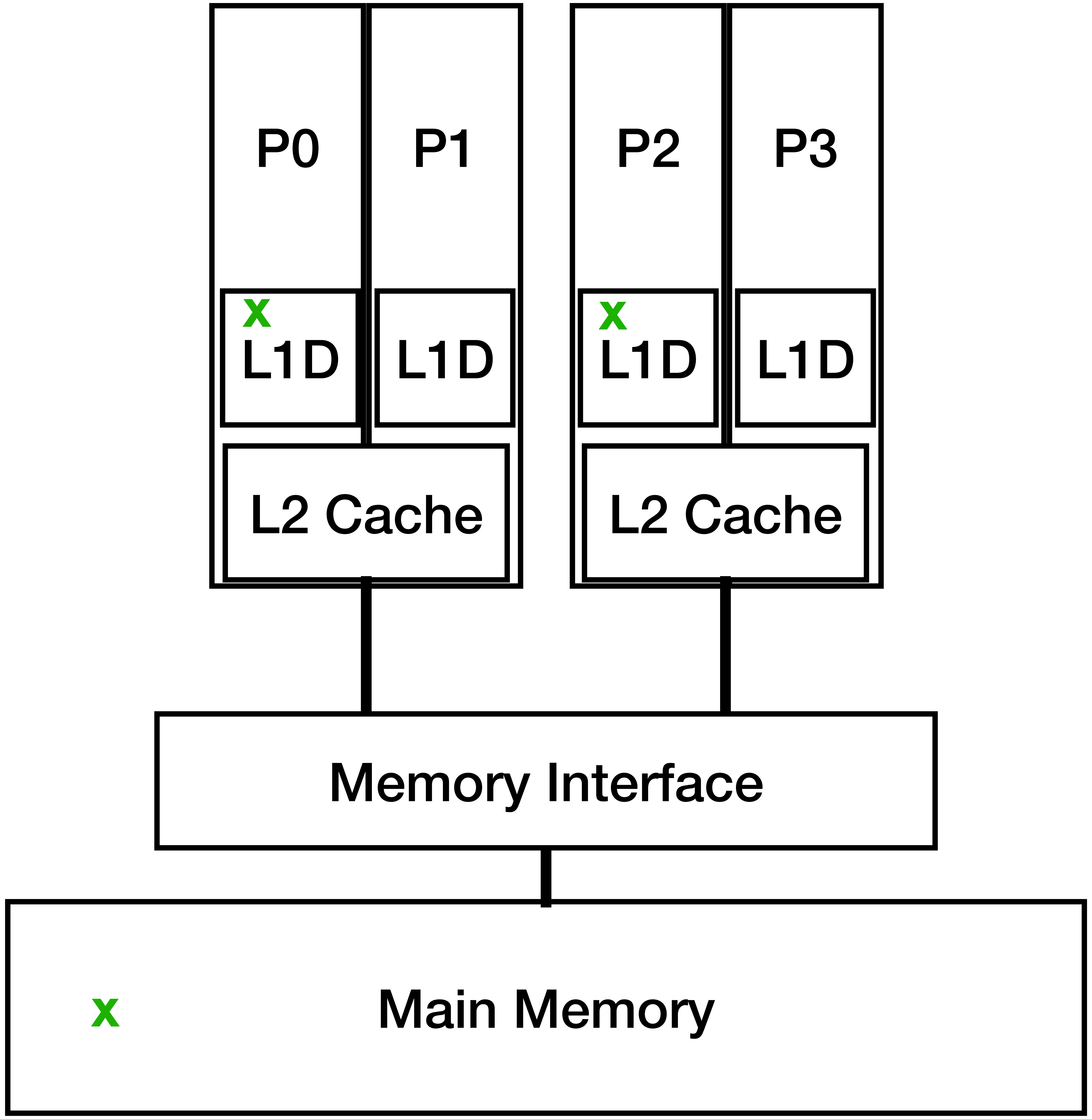
Processes P0 holds a *dirty* variable
P2 holds an *invalid* variable

Cache Coherence



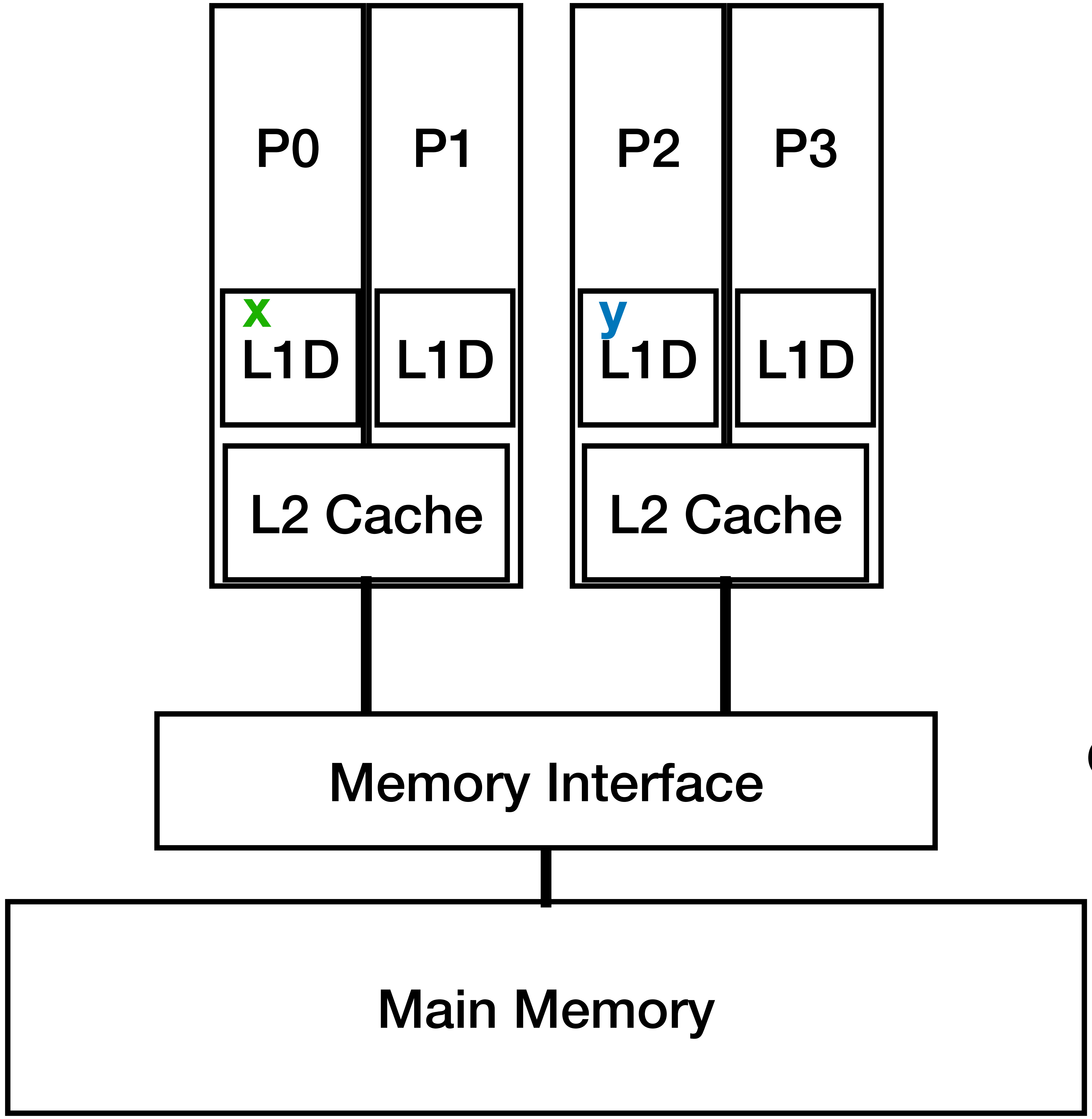
If P2 loads x:
Attempts to fetch from main memory
Main memory has variable marked as dirty by P0
P0 services the request

Cache Coherence



Now, back to *shared* state

False Sharing



P0 stores to x
P2 stores to y

What if x and y are on the same cache line?

System only invalidates / updates by cache line
Can't detect the updates aren't actually the same

Key Takeaways

- If one process is writing to a shared variable, want all other processes to avoid accessing that variable if possible
- If one process is writing to a shared variable, want all other processes to avoid accessing **nearby** variables if possible

Race Conditions

- Operating system keeps caches up to date, but you still need to be careful not to write broken code
- Threads : mostly share the same address space
  - Heap : dynamically allocated global memory
  - Data section : statically allocated global memory
- A thread's local variables may be either **shared** or **private**
- **If multiple threads update one shared variable at the same time, you can have a race condition**

Helpful Tips

- Multiple threads must not concurrently write to a shared variable
- **Solution 1** : Get rid of shared variable
  - Hello world : each thread could have its own thread_id
- **Solution 2** : Break code into independent sections