CS3350B Computer Organization
Chapter 5: Parallel Architectures

Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Thursday March 21, 2019

Outline

Needing Multicore Architectures

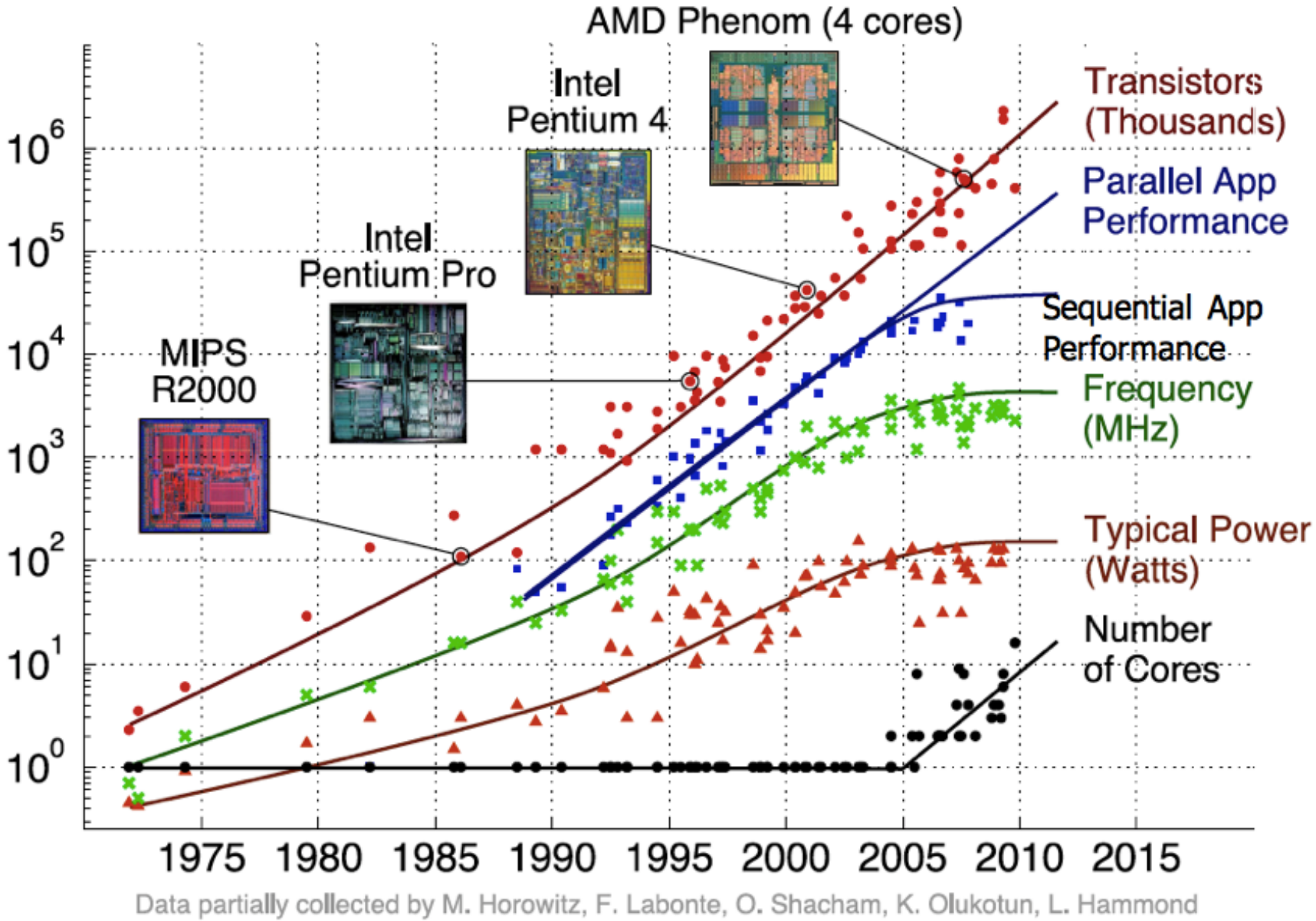
Recall:

- Processor-Memory gap.
- Power Wall.
- Moore's law failing?
- Great Ideas in Computer Architecture: Performance via Parallelism.

New reasons:

- ILP has hit a peak within current power/thermal constraints.
- Can leverage vector operations and SIMD processors but this limits parallelism to a single instruction type.

Needing Multicore: Performance Plot



SIMD Processors

- **SIMD processors** are sometimes called **vector processors**.
- However, not all SIMD processors are vector processors.
- They execute a Single Instruction on Multiple Data elements.
- In a vector processor the data elements must be *adjacent*.
- SSE (Streaming SIMD Extension) extends x86 for vectorized instrs.
  - ↳ Modern CPUs: intel, AMD.
- Ex: The loop can be unrolled and executed using a 128-bit (4x32-bit int) vectorized instruction.

```
int i = 0;
for (i; i < n; i++) {
    A[i] += 10;
}
```

```
int i = 0;
for (i; i < n; i += 4) {
    A[i] += 10;
    A[i+1] += 10;
    A[i+2] += 10;
    A[i+3] += 10;
}
```

Flynn's Taxonomy

Flynn's Taxonomy is a characterization of an architecture's parallelism.

	Single Instr. Stream	*Multiple Instr. Streams*
Single Data Stream	SISD	MISD
Multiple Data Streams	SIMD	MIMD

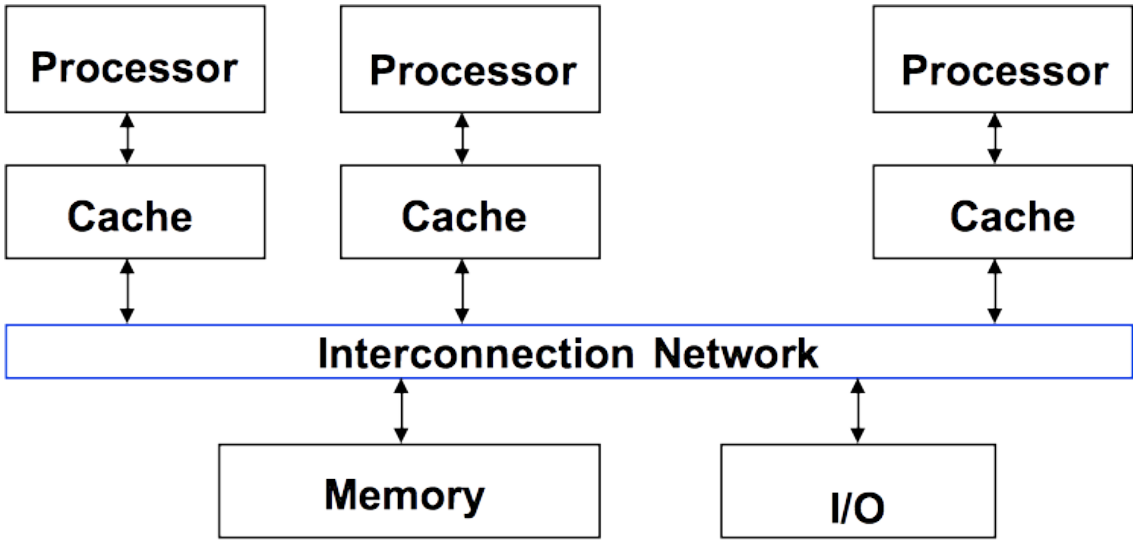
- **SISD** – the normal architecture; basic *von Neumann architecture*.
- **SIMD** – single instruction applied to multiple data elements. Sometimes called a vector processor.
- **MISD** – obscure and not usually used. A data pipeline where each stage performs a different operation.
- **MIMD** – a multiprocessor; multiple processors fetch different instructions and operate on different data.

Outline

Multiprocessors

Multiprocessors belong to the MIMD type.

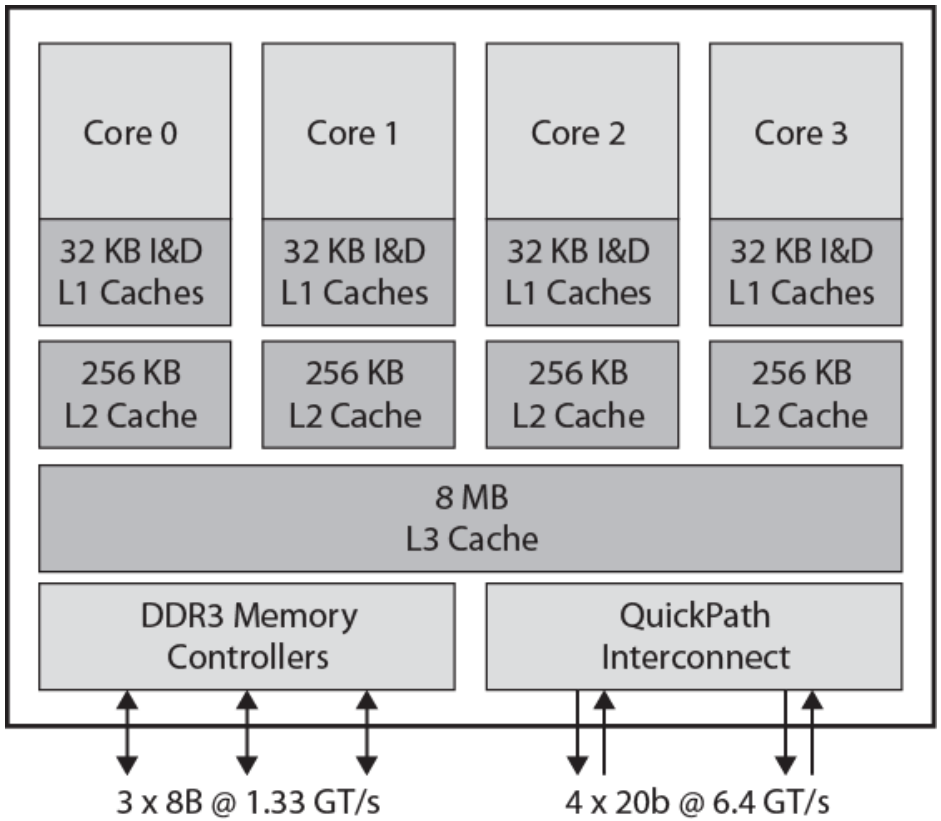
- Multiple, independent processors executing different instructions/programs.
- A computer with literally multiple processors.



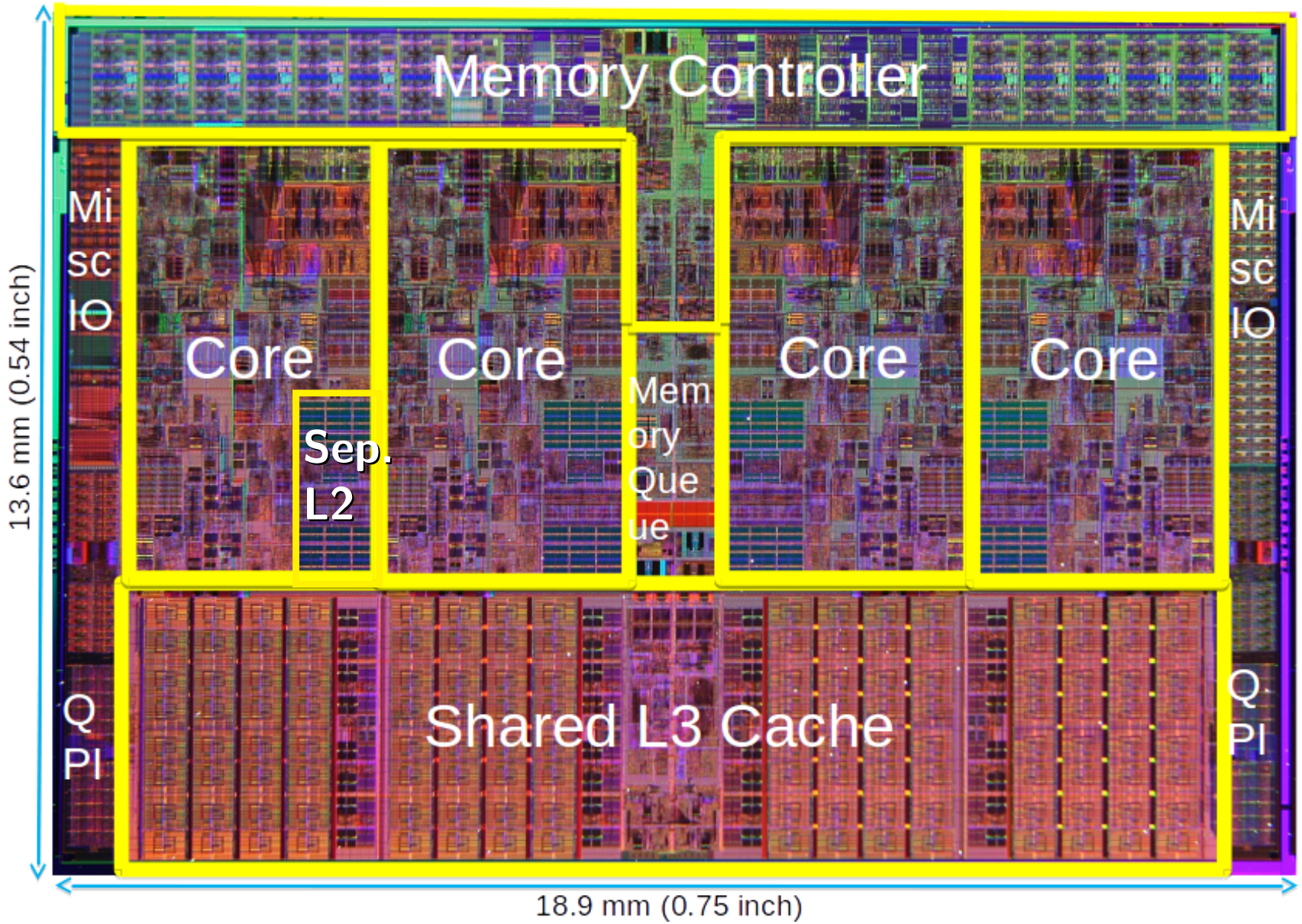
Multi-core Processors

Multi-core processors are still MIMD type but differ from multi-processors.

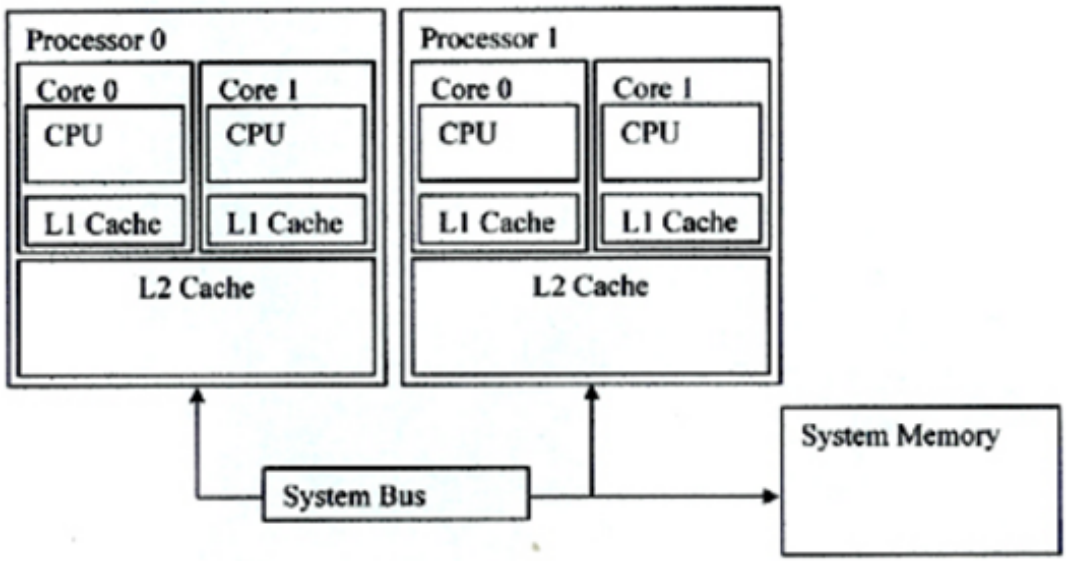
- Also called chip-level multiprocessor; the processor itself has multiple processors (datapaths).
- Contrast with superscalar (which is only SISD).
- Each core can operate a completely different instruction stream.

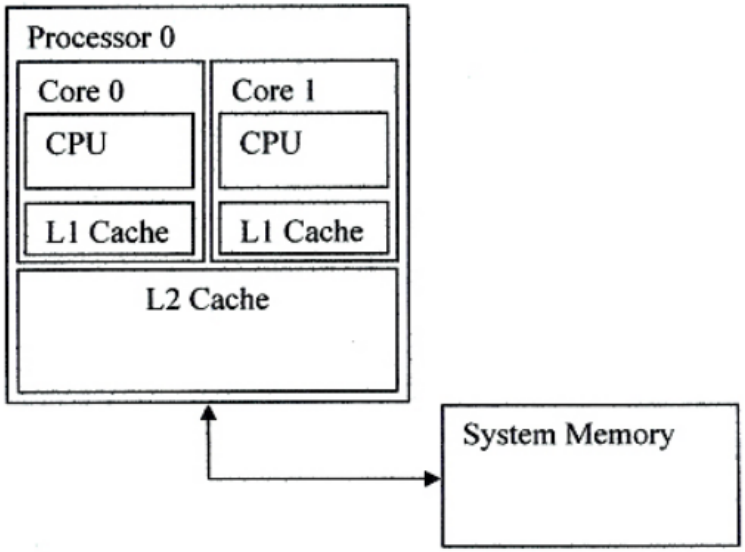


Modern Multi-Core Circuitry



Multiprocessor vs Multi-core



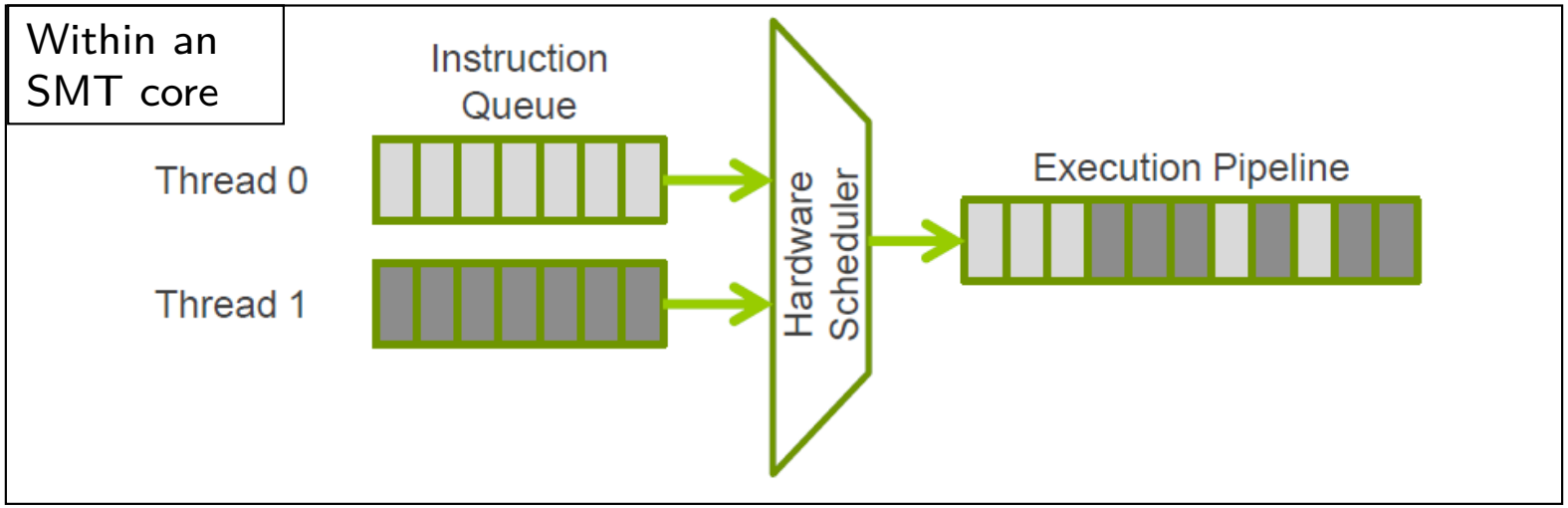


- Each processor in a multiprocessor system can also be multi-core.
- To operating system each core seen as an independent processor.
- **But** notice that cores within a processor usually share some cache. This gives better performance to processes which need to share information (e.g. multiple threads within a single process).

Simultaneous Multi-threading

Simultaneous Multi-threading (SMT) is hardware-level support within a single core to handle multiple threads at once.

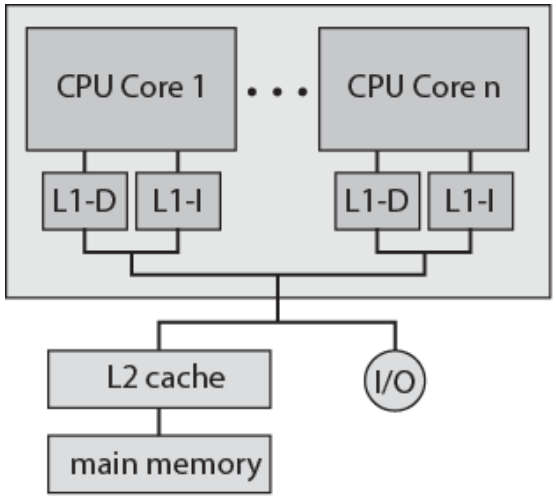
- One SMT core presents itself to the OS as multiple processors, one for each possible thread.
- Called hyperthreading by intel.
- Not necessarily completely duplicate hardware as in cores, but some redundancy in datapath.
- Sort of like superscalar for ILP but with different instruction streams.



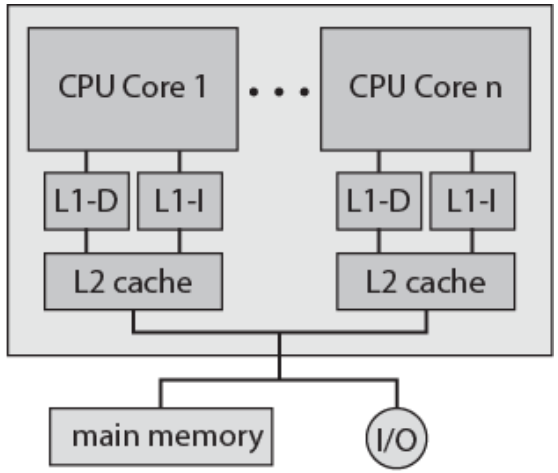
Pros and Cons of Multi (core) Processors

- Possible throughput increases are stellar.
- Allows multi-tasking in OS.
- Parallelism tackles power wall, moore's law, performance bottlenecks.
- However, a single program cannot always make use of multiple cores or processors.
- Must program explicitly for parallelism.
- Parallel programming is hard.
  - ↳ Must explicitly program for **thread-level parallelism** (TLP)
  - ↳ Some tools try to help: compiler vectorization, Open-MP, Cilk.
- Still only one **global** memory.

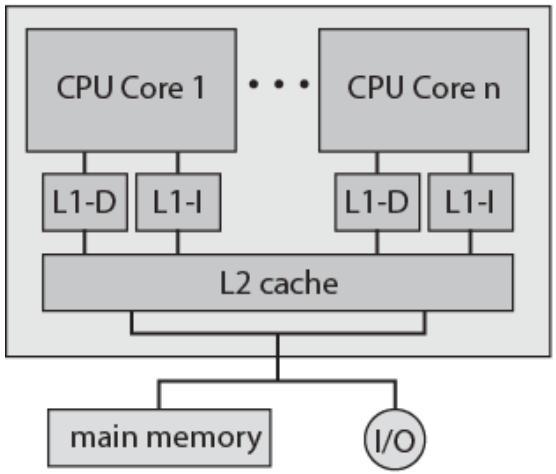
Multi-core Configurations



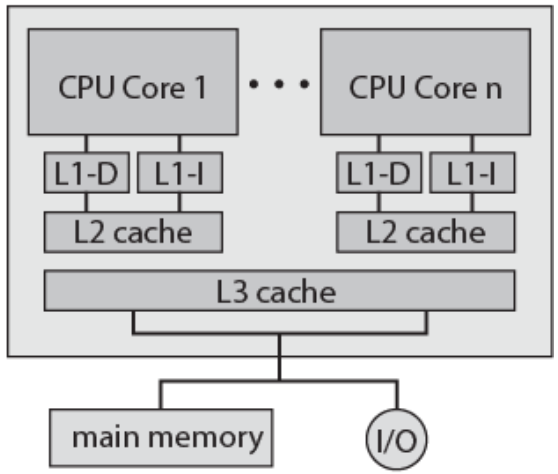
(a) Dedicated L1 cache



(b) Dedicated L2 cache



(c) Shared L2 cache



(d) Shared L3 cache

Outline

The Problem of Cache Coherence

- If each processor (core) has its own (L1) cache then it also has its own copy of memory.
- If two cores both have a copy, how does one core get the write updates of another core?
- This is the problem of **cache coherence**.
- Writing back to lower levels of cache which are shared by multiple processors (cores) is not sufficient. Must also propagate change upwards?
- This is particularly a problem if this lowest level is very slow memory. What can we do?

Note: cache coherency is usually performed on a per cache block basis, not per memory word/address.

Cache Coherency and Consistency

Coherency: If processor P1 reads a value of a particularly memory address after processor P2 writes to it (and no other writes have occurred to that address) then P1 must read the value P2 wrote.

(Sequential) **Consistency**: all writes to a memory address must be performed in some sequential order.

- Coherency is about *what* value is read while consistency is *when* the value is read.

Cache coherency requires two parts in a solution:

- Write propagation: writes to a cache block must be propagated to other copies of that same cache block in other caches.
- Transaction serialization: reads and writes to a particular address must be seen by all processors in the same order.

Snooping Policies

Whenever a processor reads from or writes to a lower level of cache, these transactions are broadcast on some shared global bus.

A **snooping policy** (a.k.a bus sniffing) has all cores monitor this bus for all requests and responses and act accordingly.

- **Write invalidate** protocol: when a write is broadcast, all other cores which share a copy of the address being written to **invalidate** their copies.
- **Write update** protocol: when a write is broadcast, all other cores which share a copy of that address being written to **update** their local copy with the value which was broadcast.
- In either case, serialization is handled by mutually exclusive access to global bus. Core stalls if bus is busy.

Invalidate vs Update

Consider multiple writes to same memory address without any reads.
Consider writes to adjacent memory words within a single cache block.

- Both of these cases common due to temporal/spatial locality.
- Both of these cases require multiple update signals and new values to be sent across the bus.
- However, only one invalidate signal would be needed in either case.

Bus bandwidth is a precious resource.

- Invalidate protocols preferred due to less bandwidth required.
- Invalidate protocols used in modern CPUs over update protocols.

The MESI Protocol (1/2)

MESI: an invalidate snooping protocol which adds several optimizations to further reduce bus bandwidth.

- Recall that a normal cache has a "valid" state and, if using a write-back policy, a "dirty" state.
- MESI adds "Exclusive" and "Shared" states. M $\Rightarrow$ dirty, I $\Rightarrow$ invalid.
- If cache block is "Exclusive" avoid broadcasting a write.

MESI can be described by looking at what happens on read misses, read hits, write misses, and write hits.

The MESI Protocol (2/2)

Each cache block in each processor's cache has one of 4 states:

Modified: The cache block is exclusively owned by the processor and is dirty (i.e. it differs from main memory).

Exclusive: The cache block is exclusively owned by the processor and is clean (i.e. the same value as main memory).

Shared: The cache block is shared between multiple processors and is clean.

Invalid: The cache block was loaded into cache but its value is no longer valid.

.

MESI Read Hit

- The cache block's state is one of MES to be considered a hit.
- Nothing to do; just return the value and do not change states.
- If state is M then the cache block is dirty but that's fine for this particular core.

MESI Read Miss (1/2)

Case 1: No copies anywhere.

- The requesting core waits for response from lower level memory.
- The value is stored in cache with state E.

Case 2: One cache has an E copy.

- The snooping cache hears read request, puts copy of value on bus.
- The access to lower level memory is abandoned.
- The requesting core reads value from bus.
- Both cores set cache block's state to S.

MESI Read Miss (2/2)

Case 3: Several caches have an S copy.

- One snooping cache hears request, puts copy on bus.
- The access to lower level memory is abandoned.
- The requesting core reads value from bus and sets state to S.
- All copies are still in state S.

Case 4: One cache has an M copy.

- The snooping cache hears read request, puts copy on bus.
- The access to lower level memory is abandoned.
- The requesting core reads value from bus and sets state to S.
- The snooping core sets state to S **and** writes updated value back to main memory.

MESI Write Hit

The cache block's state is one of MES to be considered a hit.

If in M state:

- Cache block is exclusively owned and already dirty.
- Update cache value, no state change.

If in E state:

- Cache block is exclusively owned and clean.
- Update cache value, set state to M.

If in S state:

- Cache block is shared but clean.
- Requesting core broadcasts invalidate on the bus.
- Snooping cores with S copy change to I state.
- Requesting core updates cache value and sets state to M.

MESI Write Miss

This one is tricky. Need to *read* first and then *write.*

Case 1: No other copies.

- Read from main memory, write new value in cache, set state to M.

Case 2: Other copy is in E state / Other copies are in S state.

- The requesting cache requests Read With Intent To Modify.
- The snooping cache(s) hear RWITM request, puts copy on bus, sets own state to I.
- Requester reads value from bus, updates value, sets state to M.

Case 3: Other copy in M state

- The snooping cache hears RWITM, puts copy on bus, writes back to main memory, sets its own state to I.
- Requester reads value from bus, updates value, sets state to M.

MESI: An Example

Consider a system with 3 processors where all processors are referencing the same cache block.

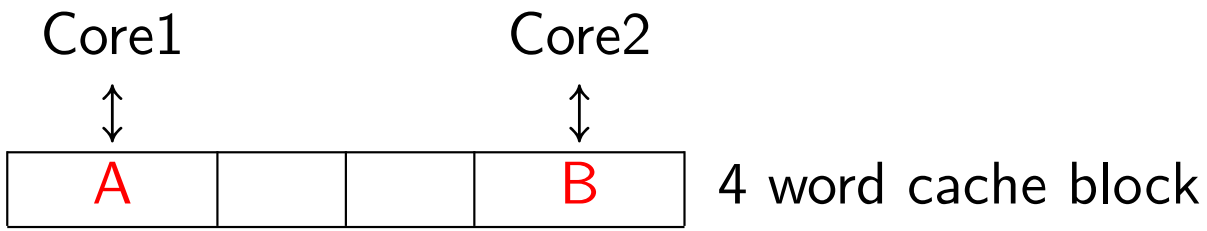
Using the MESI protocol fill the table to describe the cache block's state in each processor's cache.

Request	P1	P2	P3	Data Supplier
Initially	-	-	-	-
P1 Read	E	-	-	Main Mem.
P1 Write	M	-	-	-
P3 Read	S	-	S	P1 Cache
P3 Write	I	-	M	-
P1 Read	S	-	S	P3 Cache
P3 Read	S	-	S	-
P2 Read	S	S	S	P1 or P3 Cache

Outline

Block Size and Cache Coherency

- Recall: Memory always moved to and from a cache as a full cache block.
- In MESI, a write invalidates *an entire cache block*.
- Cache blocks larger than 1 word can cause **false sharing**.
  - ↳ Two cores are writing to different memory addresses that just happen to be in the same cache block.
  - ↳ False sharing disproportionately increases cache misses in write-invalidate schemes.
  - ↳ Can cause **thrashing** of invalidate signals.
  - ↳ Not a problem if cores share a cache (but, in modern processors, each core has its own L1 cache).



True Sharing vs False Sharing

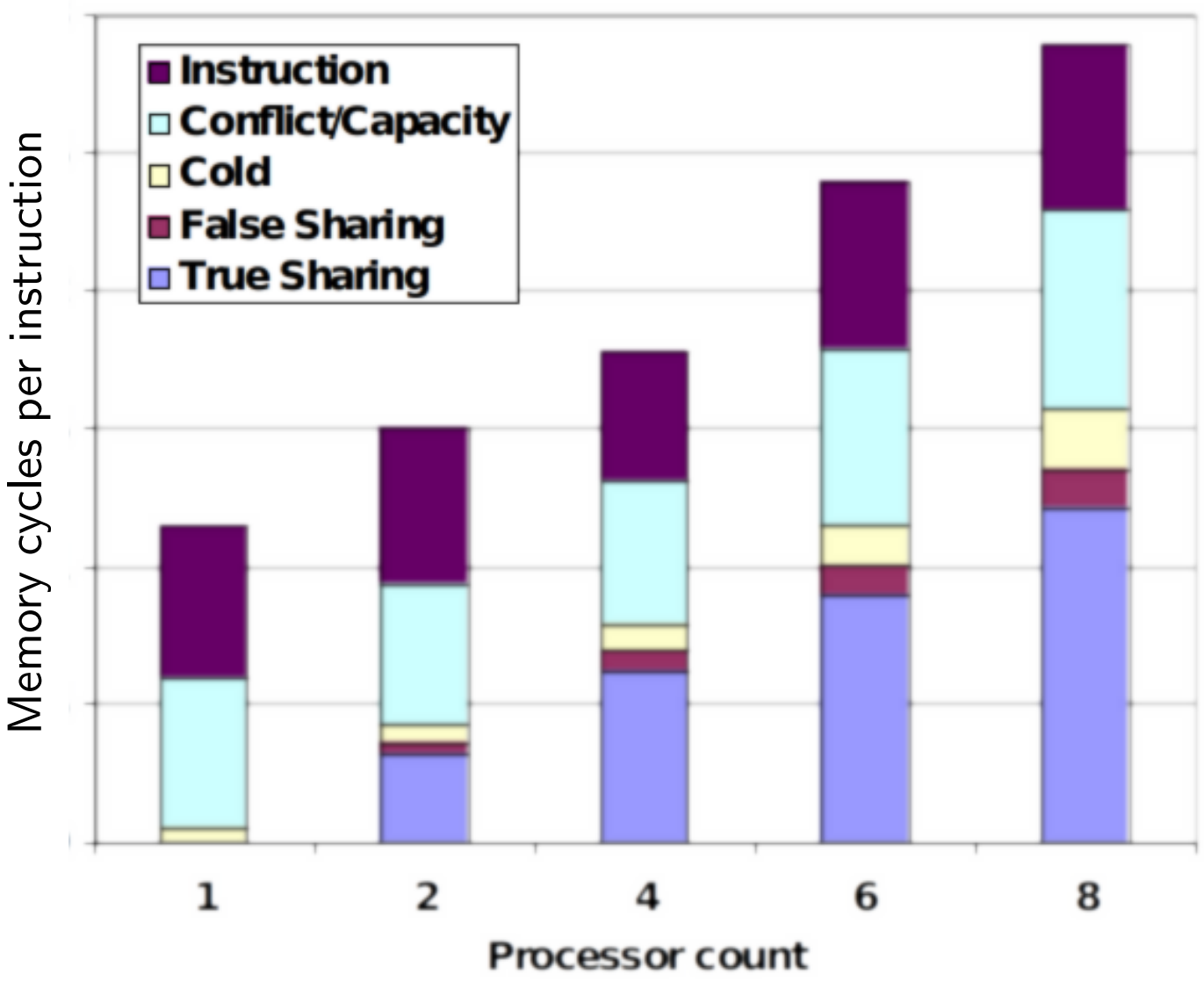
True sharing caused by communication of data between threads.

- Data is truly invalided and must be read again.
- Would still occur if cache block size was 1 word.

False sharing caused by multiple processors writing to different memory addresses within a single cache block.

- The cache block is shared but no word within the block is actually shared.
- Miss would **not** occur if cache block size was 1 word.
- Invalidation on each write causes a cache miss for the other processor.
- Goes back and forth in this way $\Rightarrow$ thrashing.

Multiprocessor Cache Performance



False Sharing Example

Let x1 and x2 belong in the same cache block. Processor 1 and Processor 2 each want to access either x1 or x2. Let us assume both processors begin with the cache block loaded into cache.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		Hit; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		Hit; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

Combating False Sharing

Change the algorithm

- Change the "stride" of the loop to access data in different cache line.
- Re-evaluate: do these threads *really* need to be accessing memory which is this close together?
- Have different threads access different, far away parts of memory.

Change the data structure

- Add "padding" to data structures so that data is better aligned in memory (with respect to memory word or cache line).

Outline

What is Multithreading?

Multithreading is **not** multi-core or multiprocessor.

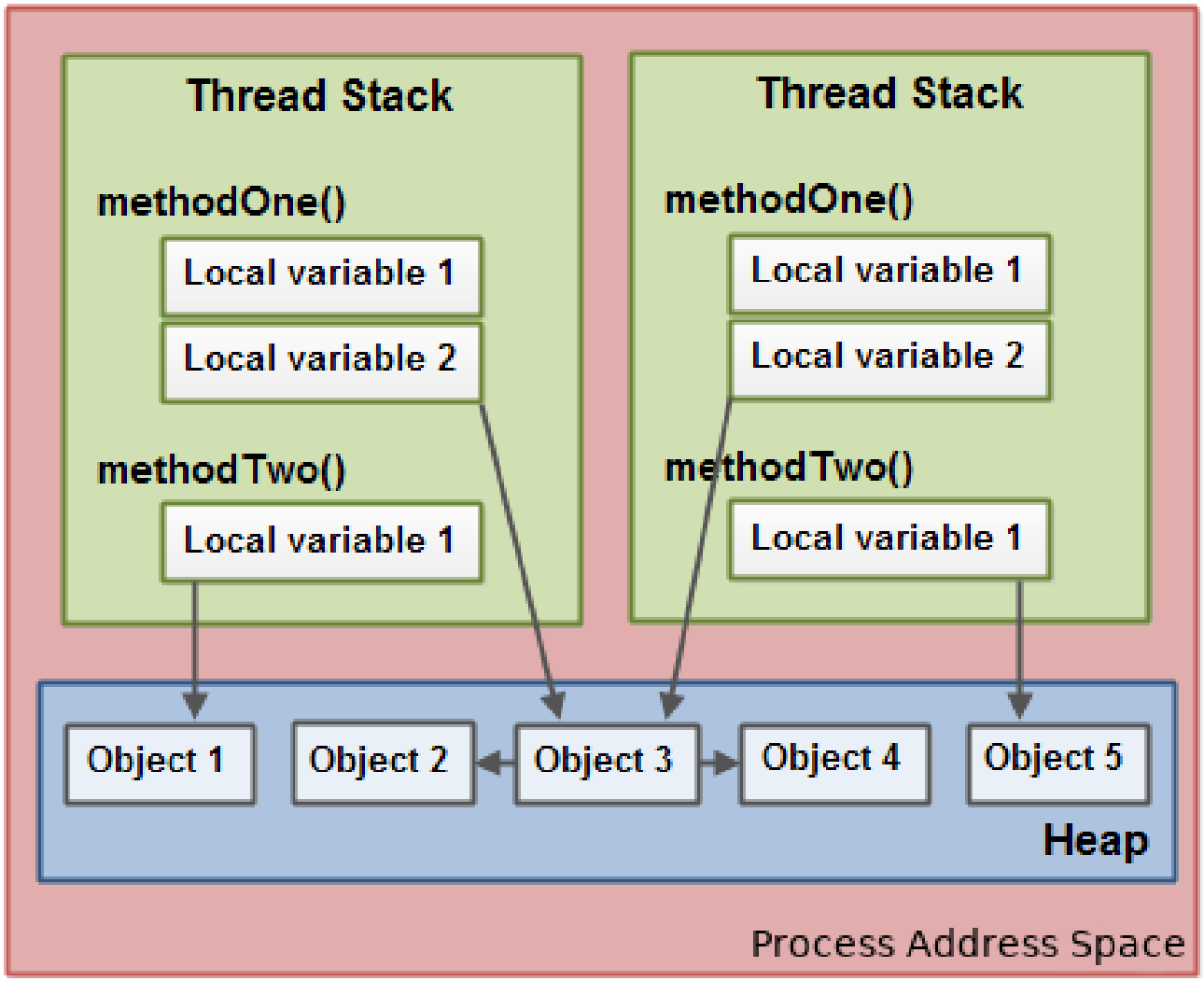
↳ But together they give us **performance**.

Multithreading is a programming model which allows for multiple, concurrent **threads** of execution, each with their own **context**.

- Thread: the smallest processing unit that can be scheduled by the OS.
- Context: A thread's own and unique local variables, PC, register values, stack.
- But, threads within the same process share an address space (heap).

Process: An instance of a program being executed. Usually handled by the operating system and requires a lot of overhead to instantiate and set-up properly. A process can contain many threads.

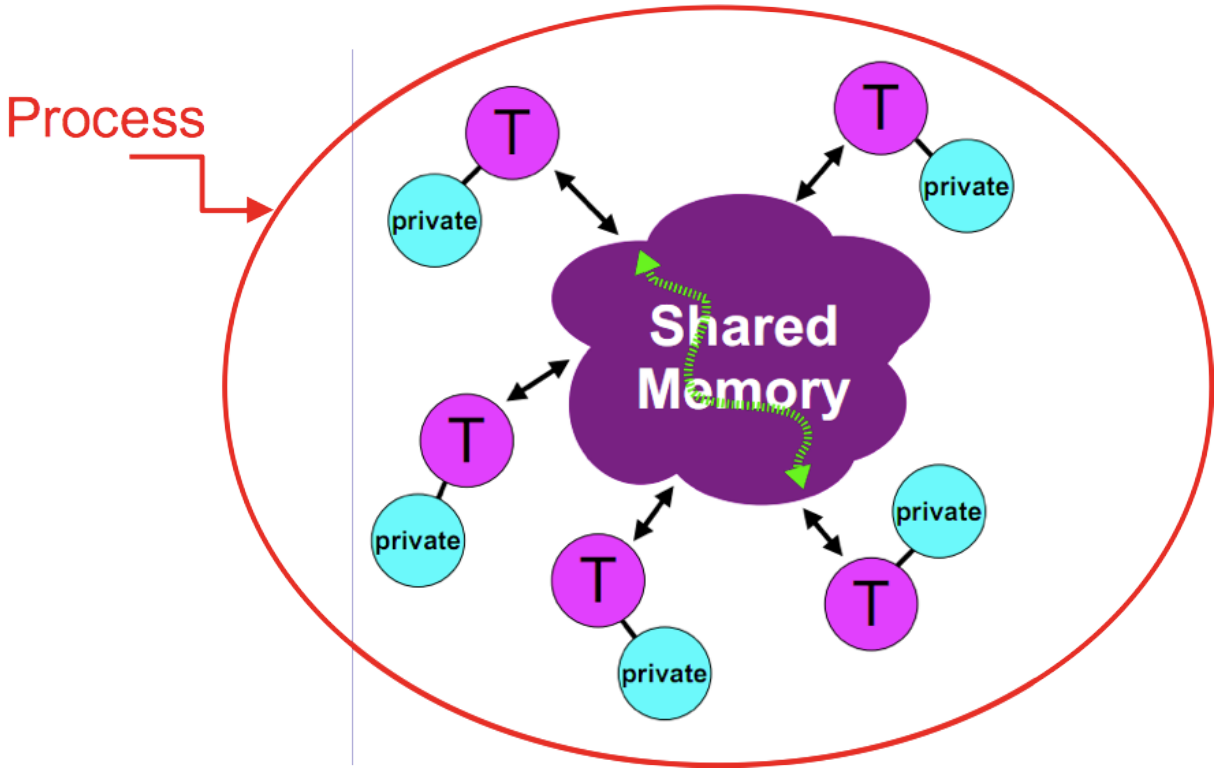
Multithreading Memory Model (1/2)



Multihreading Memory Model (2/2)

It is also possible to separate the heap into "thread-local" or private memory and shared memory.

Different programming languages allow this construct.



Multithreading and Multi-core

Using multiple threads does not require multiple cores (processors).

- A single processor can handle multiple threads via **time-division multiplexing**: the sharing of time on the datapath between threads.
  - ↳ This requires **context switching**—updating the state of the processor's register file, PC, stack pointer, etc. to match the thread's context.
- With multiple cores (processors), threads can run simultaneously.
  - ↳ Each core/processor has its own registers, etc. ⇒ no context switching.
  - ↳ If the number of threads exceeds the number of processors (cores) then multiple threads must run on one processor (core) via context switching.
- **Thread scheduling** is *hard*. Generally, the OS and hardware handle this. We'll ignore this.
  - ↳ Preemptive scheduling: threads are interrupted and context switches are forced.
  - ↳ Non-Preemptive scheduling: a.k.a cooperative scheduling, threads yield themselves to allow others to run. Threads are not interrupted.

Context Switching

- To switch contexts, the context/state begin switched from must first be saved in some way.
- Generally, this occurs by storing all the values of the registers, PC, etc. in some special data structure (e.g. a process control block) and storing that somewhere in memory.
  - ↳ Usually in the operating system's memory address space.
- Context switching is expensive, particularly if threads are not from the same processes.
  - ↳ Of course, an operating system can switch between multiple processes.

Data Races

- Via MESI, we know cache coherency is a problem.
- If two threads both attempt to write to same memory location at the same time, one must be first.
- Recall: transaction serialization.
- But, what order do the writes occur?
- **Non-Determinism**

```
void setAddress(int* addr, int val) {
    *addr = val;
}

int main(int argc, char** argv) {
    int* p = new int[1];
    *p = 0;

    std::thread t1(setAddress, p, 1);
    std::thread t2(setAddress, p, 2);
    t1.join();
    t2.join();
    std::cerr << "p: " << *p << "\n";
    return 0;
}
```

Data Races In Action

```
alex@niflheim:~/foo$ g++ NonDetEx.cpp -lpthread
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 1
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
alex@niflheim:~/foo$ ./a.out
p: 2
```

More Data Races

- What happens if multiple threads reading and writing?
  - ↳ Need synchronization between threads.
- What are the possible values of p that could be printed?

```
void incrAddr(int* address) {
    int val = *address;
    val++;
    *address = val;
}

int main(int argc, char** argv) {
    int* p = new int[1];
    *p = 0;

    std::thread t1(incrAddr, p);
    std::thread t2(incrAddr, p);
    t1.join();
    t2.join();
    std::cerr << "p: " << *p << "\n";
    return 0;
}
```

More Data Races

- What happens if multiple threads reading and writing?
  - ↳ Need synchronization between threads.
- What are the possible values of p that could be printed?
- 1 *or* 2 ⇒ **non-determinism**.
- Context switch could occur between reading from address and writing back updated result.

```
void incrAddr(int* address) {
    int val = *address;
    val++;
    *address = val;
}

int main(int argc, char** argv) {
    int* p = new int[1];
    *p = 0;

    std::thread t1(incrAddr, p);
    std::thread t2(incrAddr, p);
    t1.join();
    t2.join();
    std::cerr << "p: " << *p << "\n";
    return 0;
}
```

Fixing Data Races

- To fix data races we need thread **synchronization**.
  - ↳ Only one thread can execute some **critical section** at one time.
  - ↳ This is called **mutual exclusion**.
- We generally use **locks** whose "ownership" allows a thread to access a critical section.
- If a thread tries to "lock" (a.k.a take/own/capture) a lock that is already locked by another thread, it waits for the lock to be unlocked and then tries to lock it again.

```
std::mutex mutex;
void incrAddr(int* address) {
    mutex.lock(); //wait here until successful lock
    int val = *address;
    val++;
    *address = val;
    mutex.unlock();
}
```

Implementing a Lock

Semaphore: a counter used to control access to a critical section.

- Locks usually use a specialized **binary semaphore**: its value is 0 or 1.

The simplest lock is a **spinlock**.

- It waits by "spinning " until the lock can be acquired.
- A bad, non-working, but simple example:

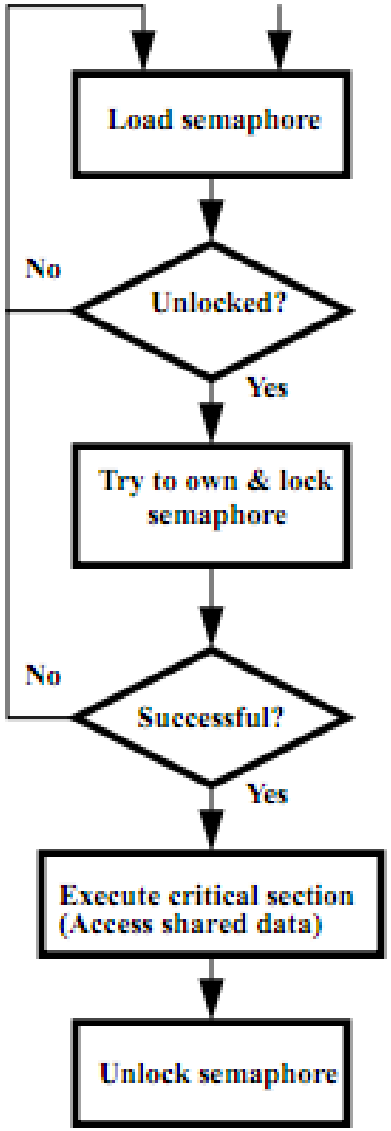
```
void spinlock::lock() {
    int spins = 0;
    while(this.semaphore == 1) {
        ++spins;
    }
    this.semaphore = 1;
}
```

- If multiple threads spinning on same spinlock, which one is first to set the semaphore to 1? (i.e. which one owns the lock?) Which one goes back to spinning?

Better Lock Implementation: "Test and Set"

- "Test and set" is an **atomic** operation that sets a variable's value, returning its old value.
- Atomic: an operation (many instructions) which is viewed as happening instantly across all threads. Cannot be interrupted by a context switch.
- `this.setSemaphore()` is atomic.

```
void spinlock::lock() {
    int spins = 0;
    while(this.semaphore == 1) {
        ++spins;
    }
    int oldVal = this.setSemaphore(1);
    if (oldVal == 1) {
        //another thread got there first
        this.lock();
    }
}
```



Automatic Multithreading

- Some tools exist to automatically parallelize code.
  - ↳ Cilk (GCC 5-7), OpenMP (GCC 4.2+)
- These tools are good but can't replace a fine-tuned implementation of a human.
- *But* writing efficient parallel code is even harder than getting it working in the first place.
  - ↳ Automatic tuning of parallel code ⇒ one of my current research topics.

```
void parallelAdd(int* a, int n) {
    //cilk_for: each loop iteration done in parallel
    cilk_for(int i = 0; i < n; ++i) {
        a[i] += 10;
    }
}
```

Summary

- Serial execution has hit many walls: memory, power, ILP.
- Parallelism needed for performance.
- One processor can give us **vectorized** instructions (SIMD).
- Multi-core and multiprocessor (MIMD) gives us a way to effectively use multithreading.
- Cache coherency a problem with multiple cores, but can be fixed with snooping policies (e.g. MESI).
- False sharing impacts cache performance.
- Multithreading ⇒ Parallelism on Multiprocessors ⇒ Performance.

Want more? CS4402 - Distributed and Parallel Systems