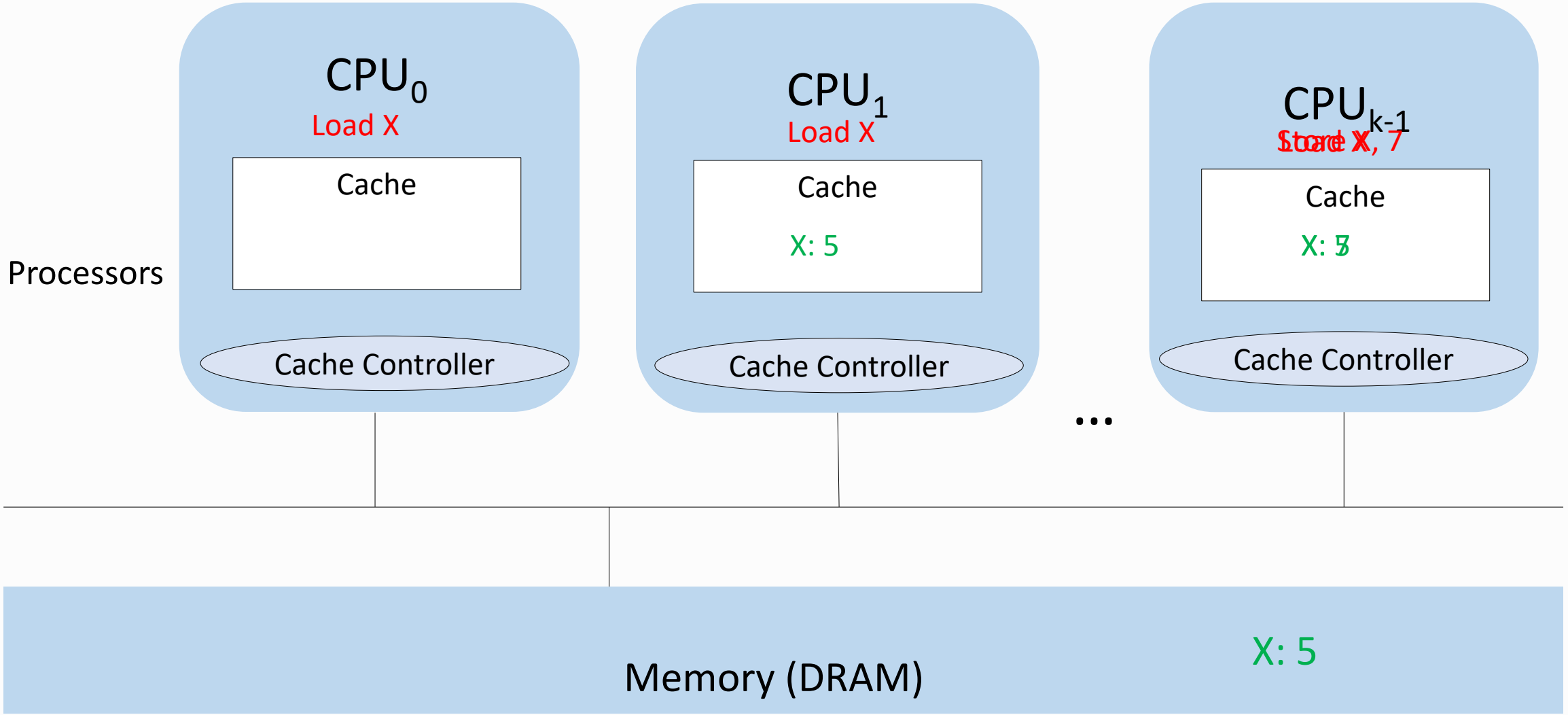
Shared Memory Multiprocessors

Multiple Processors Accessing a Common Memory

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Shared Memory Multiprocessor and Caches



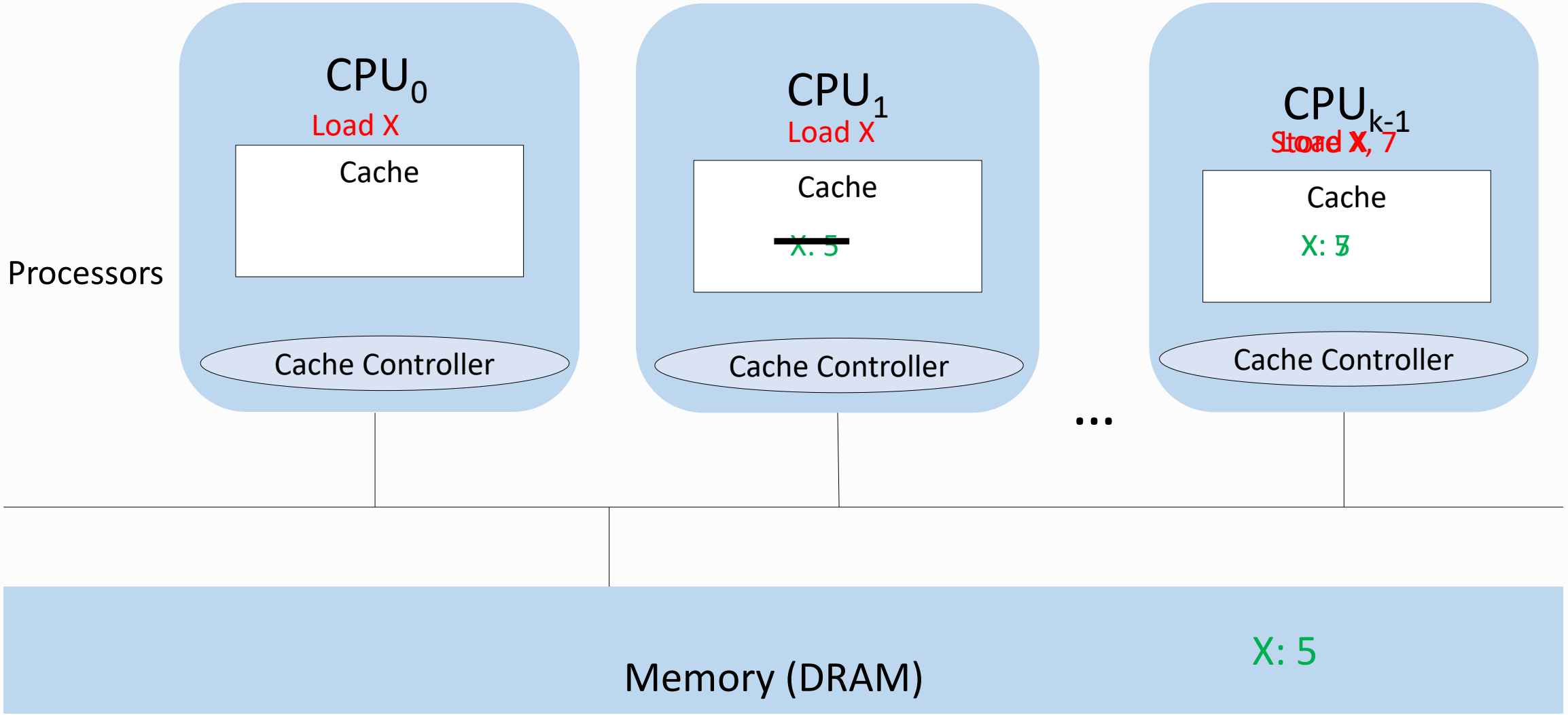
Cache Coherence, Snoopy Caches, and MSI Protocol

- The cache controllers watch every transaction on the bus
  - Even if its not their transaction ... hence "*snoopy*"
- Each cache line is in one of three states:
  - M: Modified - my copy is "dirty" (i.e., up to date) ... don't let others use the stale copy from the DRAM
  - S: Shared - I have a copy, and I know others have a copy, and none of them is M
  - I: Invalid - I have a copy, but another processor's cache has a later version. Don't let my processor use my copy

The Illinois Protocol (a.k.a. MESI Protocol)

- Prof. Janak Patel et. al
- Each cache line is in one of four states:
  - M: Modified
  - E: Exclusive: I have a copy, no other caches have a copy and it is clean
    - i.e. same as the copy in DRAM
  - S: Shared: I have a copy; I know others have a copy, and none of them is M
  - I: Invalid
- Each cache controller monitors transactions on the shared bus
  - Snooping

Shared Memory Multiprocessor and Caches



Real Cache Coherence Hardware

- Real cache coherence hardware is more complex than that
- Instead of a single shared bus, a complex interconnect
- L1, L2, L3 Caches
- Shared caches:
  - L1 typically private to a core (or two)
  - SMT: symmetric multi-threading
    - So, a core is typically multiple (2 or 4) hardware threads (hyperthreads)
  - L3 is typically shared

Performance Take-Aways (Lessons)

- Cache traffic “serving” latest versions of cache lines can be a bottleneck
  - Bus traffic
- Data written by one core and read by another core(s) causes cache traffic
  - This is based on the cache line, not a variable
  - I.e., if you “load X”, not just the 4 bytes of X, but the surrounding 32 (say) bytes of a cache-line have to be moved from a neighboring core’s cache to yours (or from the DRAM)

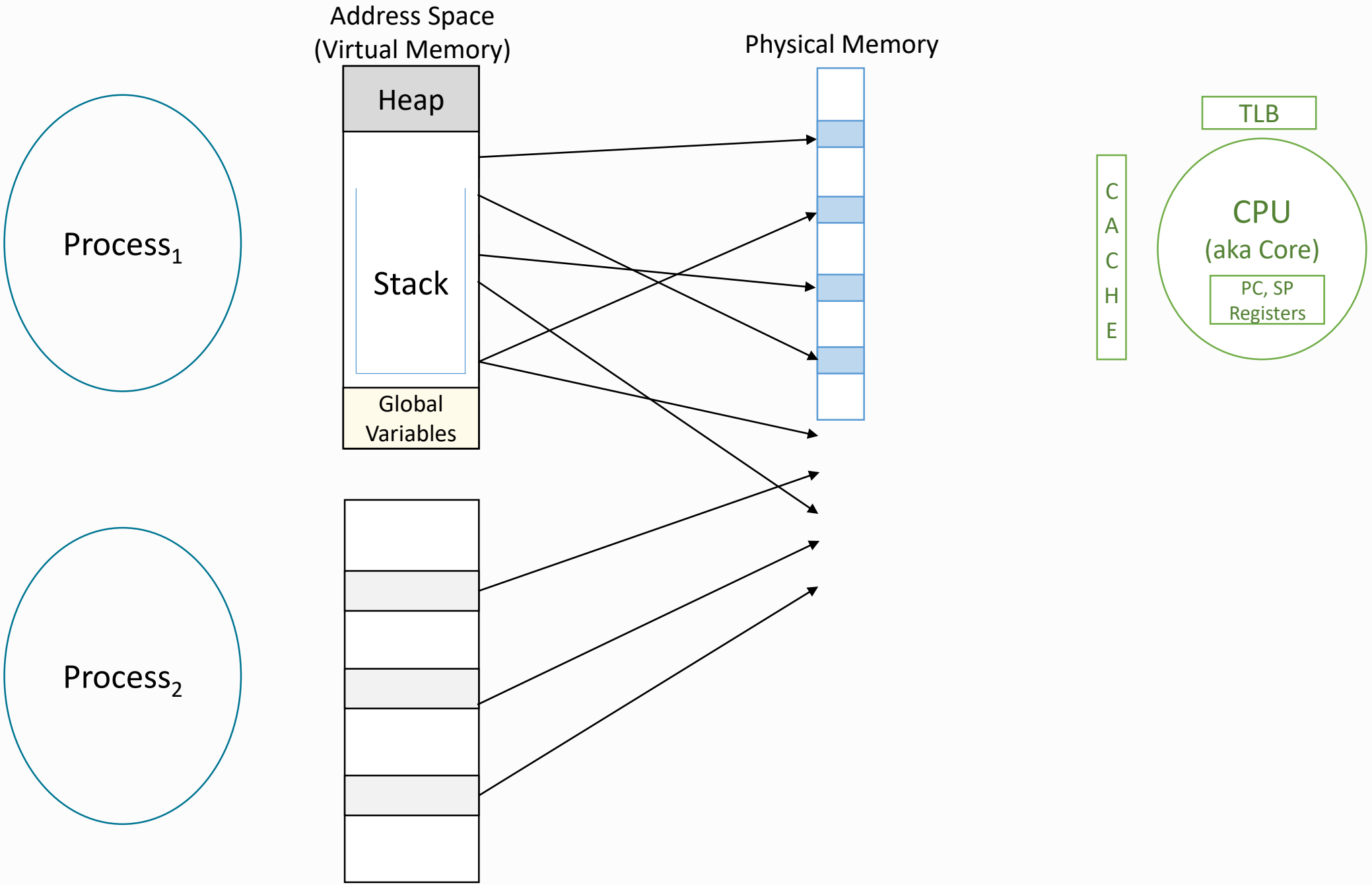
Shared Address Space Programming: Basics

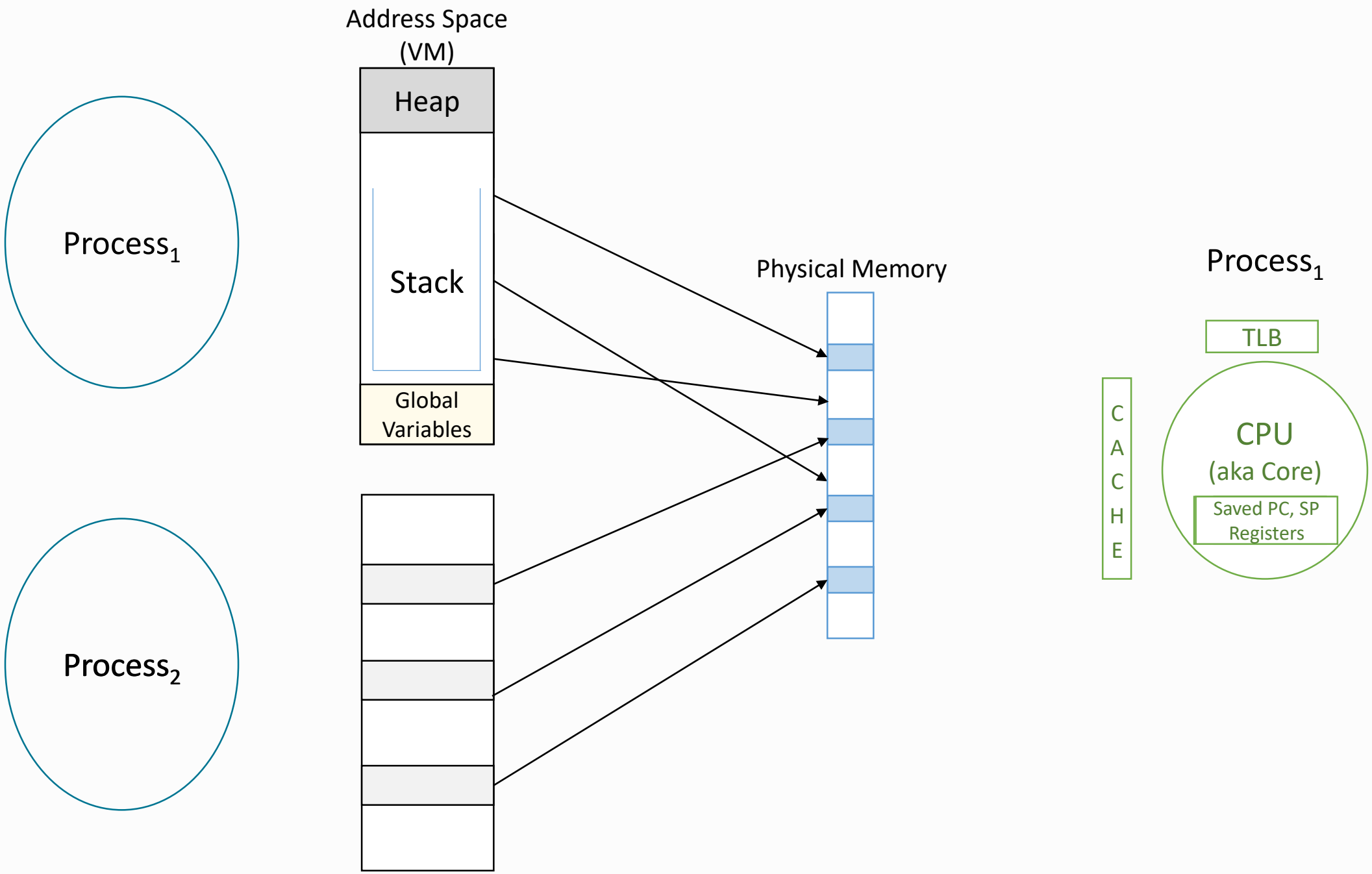
Processes, Threads, Address Space, and the Operating System's View

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Processes and Address Spaces

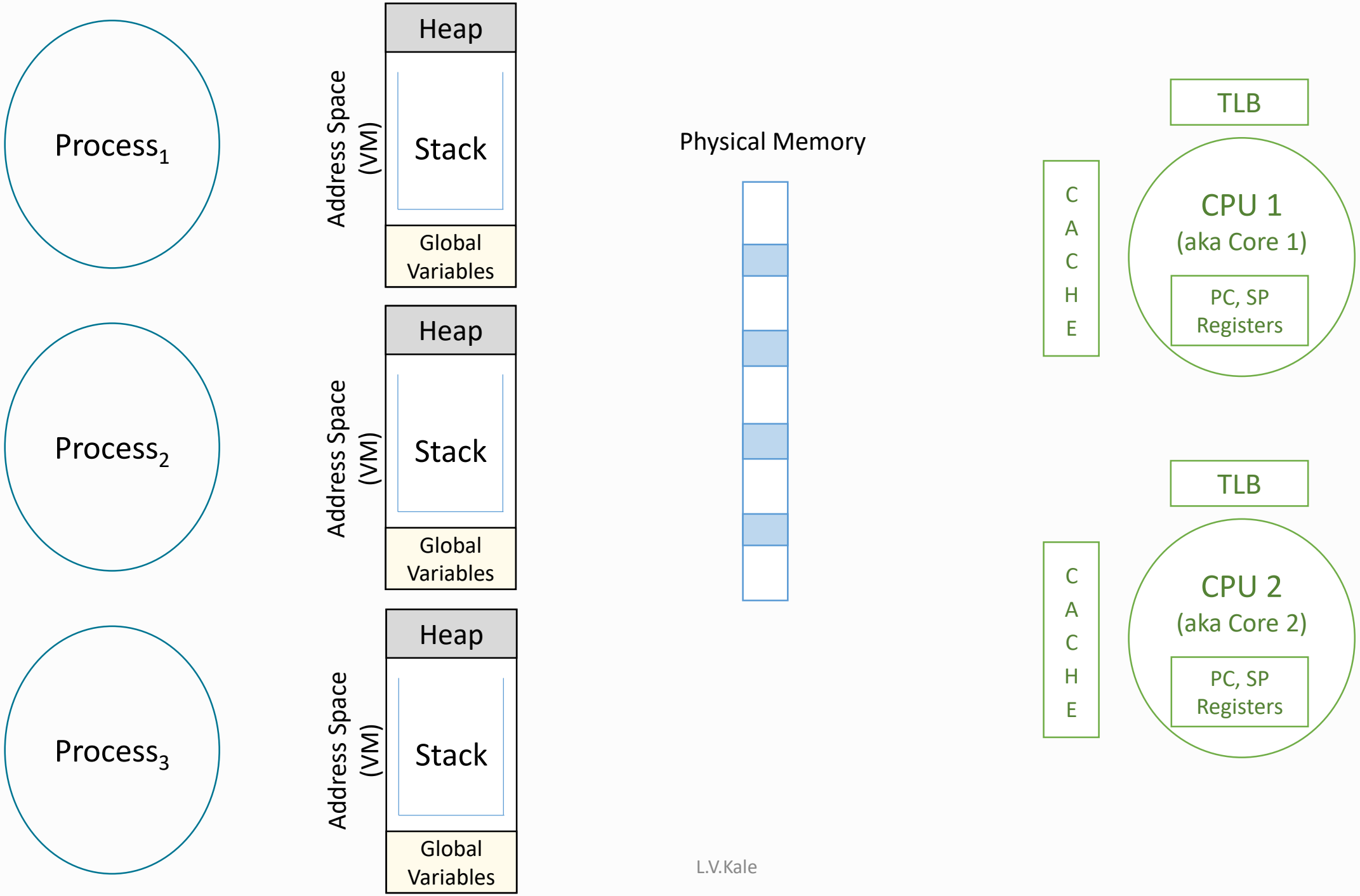
- To understand programming of multicore processors, and its performance issues, it is useful to understand:
  - Processes
  - Address spaces and their relation to real physical memory
  - Division of address space: global variables, stack, and heap
  - Operating system (OS) scheduler
  - Threads and their relationship to the above
- Basic challenges of shared address space programming
  - And how (in what sequence) we want to address them in this course


Address Space
(Virtual Memory)
Physical Memory
Heap
Stack
Global
Variables
Process$_1$
Process$_2$
TLB
CACHE
CPU
(aka Core)
PC, SP
Registers


Address Space
(VM)
Heap
Stack
Global
Variables
Process1
Process2
Physical Memory
Process1
TLB
CACHE
CPU
(aka Core)
Saved PC, SP
Registers

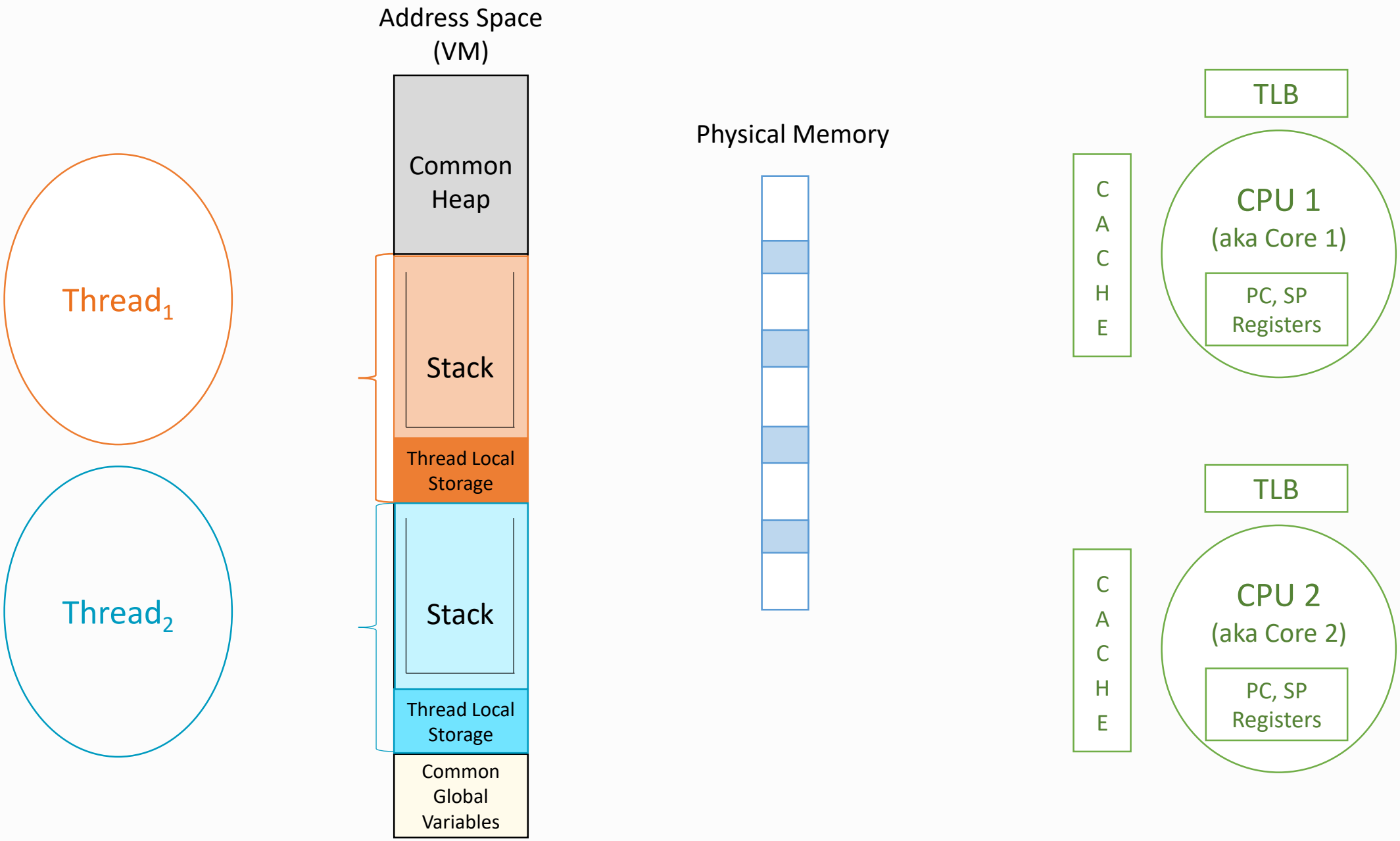
Processes and Time Sharing

- A process' view of memory is its address space. Its "logical" or "virtual" memory
- Pages from virtual memory are assigned to (live in) pages of physical memory
- OS runs a process for a time-quanta (a few milliseconds) and then interrupts it, so it can run another process, if any
  - As well as short OS services, daemons
  - This is called a context switch
- Registers, including PC (program counter) and SP (stack pointer), as well as data registers, are saved in memory when a process is context switched out
  - When there are multiple processes
- Note that when the same process is scheduled again:
  - The cache may not contain data you had brought in earlier
  - From your process's point of view, the cache is said to be "polluted"
  - Of course, your data is safe in memory (eviction handles that)


Process1
Address Space (VM)
Heap
Stack
Global Variables
Process2
Address Space (VM)
Heap
Stack
Global Variables
Process3
Address Space (VM)
Heap
Stack
Global Variables
Physical Memory
TLB
CACHE
CPU 1
(aka Core 1)
PC, SP
Registers
TLB
CACHE
CPU 2
(aka Core 2)
PC, SP
Registers

OS Visible Threads

- To write parallel programs on a multicore processor (aka Node)
  - You can't quite have two processes running on two cores
    - Because each will run on its own address space. So no coordination and no "shared memory programming"
- Instead, let us a have something smaller than a process
  - LWP (light-weight process), pthread (POSIX thread), Windows threads, fibers
    - We will talk about pthreads, but the same idea applies to Windows, etc.
  - Each process "contains" several pthreads
  - They all share the same address space
  - Pthreads are visible to operating systems (OS) for the purpose of scheduling
  - I.e., a pthread is like a process, from the point of view of scheduling, but subordinate to (or part of) a process in that all pthreads belonging to a process have the same shared address space


Address Space
(VM)
Common
Heap
Thread$_1$
Stack
Thread Local
Storage
Thread$_2$
Stack
Thread Local
Storage
Common
Global
Variables
Physical Memory
TLB
C
A
C
H
E
CPU 1
(aka Core 1)
PC, SP
Registers
TLB
C
A
C
H
E
CPU 2
(aka Core 2)
PC, SP
Registers

OS Visible Threads

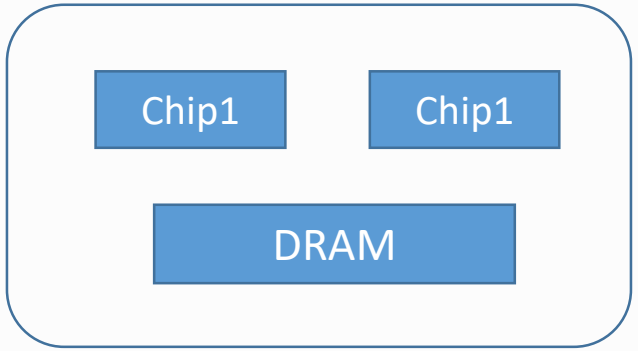
- When you are writing a tightly coupled, performance-oriented parallel program, you do want to keep each pthread running on one fixed core
  - To the extent possible
  - So there is no cache pollution
  - You can try to keep the “working set” of each thread within the caches
    - Recall: working set is the subset of memory that the program is accessing at the current time
  - Corollary: over-subscription (more pthreads than cores) is bad
- BTW: there are other situations, such as some web-server programs, where this affinity is a non-issue
  - Time scales maybe shorter than OS quanta
  - Disk I/O times may dominate
  - Over-subscription may be ok, if many threads are waiting for their I/O
    - From disk or network

Simultaneous Multi-threading (SMT) / Hyperthreading

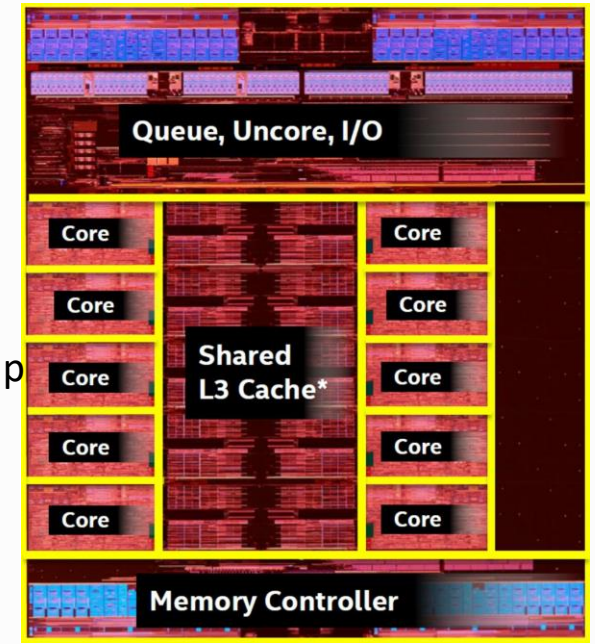
- Most machines today support two-way SMT
- Although the word “thread” is used, this has nothing (directly) to do with the OS-visible threads we saw (pthreads, etc.)
- A two-way SMT means each core can be thought of as two cores
  - It has two sets of registers, including PC, SP (so it is executing its own instruction stream)
  - But shares floating point unit/s, L1 caches, etc.
  - When one hardware thread is waiting (e.g., for a load or a floating point operation), the other thread runs
  - So, you will get close to a two-fold speedup only when each thread is running code that has a lot of such “waiting” (latencies)

A Little More on Hardware Hierarchy

- "System" or "host" or "node" or "box," may contain multiple processors
  - *Node* is a term that comes from clusters and supercomputers, which include many such nodes connected by some network (Infiniband, ethernet, or proprietary network)
- Processor or socket: a single chip, may contain multiple cores
- Core: may contain two-way or four-way hardware threads
- Memory:
  - L1 cache private to a core (or two)
  - L2 may be private or shared
  - L3 shared (across all processors in a node)
  - Memory controller, typically one per chip



Intel Broadwell chip



Source: PCWorld / https://goo.gl/fcyXAP

OpenMP: Motivation and Introduction

Can We Execute Loop Iterations in Parallel

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Basic Idea of OpenMP

- Let us provide simple, easy to use abstraction/s that cover common ways of exploiting parallelism
  - And that can make your sequential program parallel
- If you start with a sequential program, where will you find opportunities for parallelism?
  - A program runs billions of instructions a second on a single core
    - But the code is not that long ☺
  - It generates those instructions from:
    - Recursion
    - Loops!

Let Us Parallelize Loops!

- Loops are iterative behavior
  - E.g., for different values ***i*** between **0** and **N-1** do the <u>*same*</u> work (loop body)
  - Hmmm … Make different cores (pthreads) do different iterations
    - Well … We don't have that many cores. So let each core do a section of iterations

Non-loop code
Loop
Loop
Non-loop code
Loop

- Obviously, even though the text of the program may have more lines in non-loop portions, it is clear (right?) that most of the execution time is spent in loops
  - At least for most programs

Can We Run All Loops in Parallel?

- Well ... Let us look at a few examples:
- The question we ask is:
  - Can I ask different cores to do distinct iterations in parallel?
  - Of course you can
  - But: It must also get the same result as the sequential code!

```
for (i = 0; i < N; i++)
  A[i] = i * i * 0.23;
```

Loop1

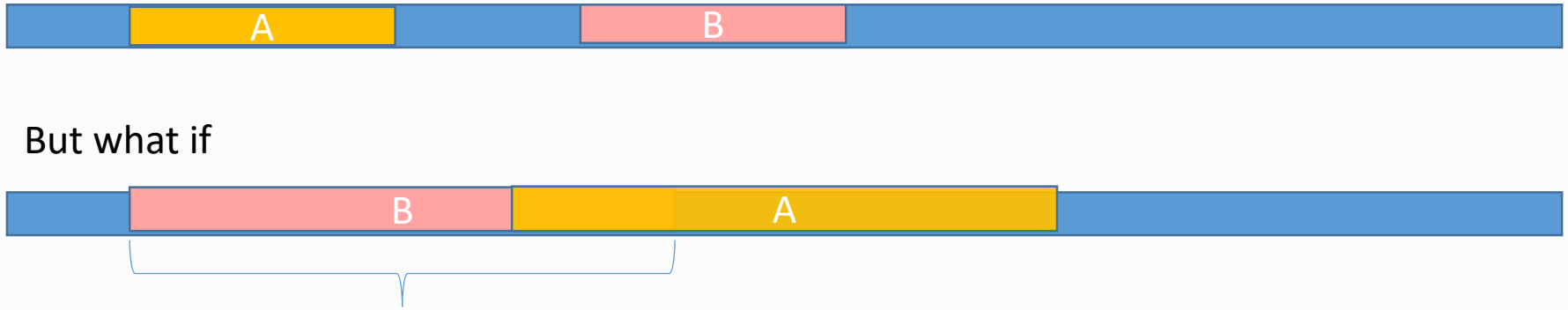
```
for (i = 0; i < N; i++)
  A[i] = x * B[i];
```

Loop2

Loop2 … Reconsidered

```
for (i = 0; i < N; i++)
  A[i] = x * B[i];
```

- Are you sure you get the same result?
  - Recall, your address space is a linear sequence of addresses



End of B overlaps with beginning of A

This can happen in C, C++

It is called “aliasing” … The same location is part of A as well as part of B

The result won’t be the same as the sequential execution! So, not parallel unless we know there is no aliasing

More Examples ... Assume No Aliasing

```
for (i = 1; i < N; i++)
  A[i] += A[i-1];
```

No ... Imagine thread 5 is working on i=50 iteration, and thread 3 on i=49 ... 49th iteration may happen after 50th, but 50th iteration needs value calculated by 49th!

```
for (i = 0; i < N; i++)
  Y[i] += a * X[i];
```

Can be done in parallel

```
for (i = 0; i < N; i++)
  Y[i] += X[ A[i] ];
```

Can be done in parallel

Loops May or May Not Be Parallelizable

- Can the compiler find out and decide?
  - Sometimes, but not always
- So, we need programmer's help
- OpenMP:
  - Let the programmer tell the compiler a particular loop is ok to execute in parallel
  - Let the compiler do the rest of the mechanics of parallel execution
- As we will see
  - OpenMP supports additional ways of parallelizing code beyond parallel loops
  - But we will focus on parallel loops first

Another example : sum over an array

```
sum = 0;
for (i = 0; i < N; i++)
  sum += A[i];
```

Hmm ... Needs more thought

Things Can Get Tricky

- Suppose you write a function to withdraw money from a bank account
- What happens when two threads execute this at about the same time?

```
withdraw(acct, w) {
  if (acct->balance > w)
    acct->balance -= w;
  else  error…
}
```

They both might conclude there is adequate balance, enter the *then* statement and withdraw more than they should! (E.g., balance is 100, w for each process is 80)

What is worse: even when the balance is adequate, it may leave a wrong result!

Things Can Get Really Tricky

- Assume that a global variable *b* has value 100 before two threads execute the following code:

Thread0:

b = b – 30;

Thread1:

b = b – 50;

Note: No if statements

The threads may execute any order, but you expect the value of ***b*** to be 20 afterward

Load R1,b

Sub R1, R1, 30

Store R1, b

Load R1,b

Sub R1, R1, 50

Store R1, b

But each statement is actually 3 machine instructions!

The instructions may interleave and make the final value be any of 50, 70, or 20!

T1

```
Load R1, b
Sub R1, R1, 30
Store R1, b
```

T2

```
Load R1, b
Sub R1, R1, 50
Store R1, b
```

```
T1: Load R1, b
T1: Sub R1, R1, 30
T1: Store R1, b
T2: Load R1, b
T2: Sub R1, R1, 50
T2: Store R1, b
```

b = 20

```
T1: Load R1, b
T2: Load R1, b
T1: Sub R1, R1, 30
T1: Store R1, b
T2: Sub R1, R1, 50
T2: Store R1, b
```

b = 50

```
T1: Load R1, b
T1: Sub R1, R1, 30
T2: Load R1, b
T2: Sub R1, R1, 50
T2: Store R1, b
T1: Store R1, b
```

b = 70

Array Sum

- So, the array-sum example is somewhat problematic
  - Although, since addition is commutative-associative, it feels like there should be a way to do it in parallel
  - We will return to this example later

OpenMP: History and Parallel Loops

Development of OpenMP, Pragmas, and Loops

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

History of OpenMP

- Research on automatic parallelization via compilers started at Illinois with David Kuck as a pioneering researcher
  - Years of deep research have yielded significant successes
  - However, in practice, it has been inadequate
- Shared memory multiprocessors became commercially available in the early 80s
- David Kuck initiated a group called Parallel Computing Forum to develop extensions to Fortran to support explicit, user-specified, parallel loops and other constructs
- Vendors/compilers provided their own separate directives: similar, but not identical
- Eventually, this led to a committee developing OpenMP standard, in 1997
- This standardized 15 years of shared-address-space programming practice

OpenMP philosophy

- Pragma in C/C++:
  - Hints/commands/info that the programmer provides to the compiler
  - OpenMP pragma's always begin with the word "omp"
  - `#pragma omp <text-of-the-openmp-pragma>`
- Ignoring pragma, you get a correct sequential program
  - Pragmas only provide hints/suggestions to the compiler
  - Exceptions: calls into OpenMP runtime (prefixed by omp_)
- User decides what to execute in parallel; compiler automates the rest
  - Trusting programmer completely, but occasionally providing feedback

OpenMP examples

```
..
#include <omp.h>
..

#pragma omp parallel for
for (i = 0; i < N; i++)
    A[i] = i * i * 0.23;
```

omp.h must be included

#pragma omp parallel for
is a special case of an OpenMP *directive*

Later, we will learn general forms of it

Program must be compiled with OpenMP options

The options vary from compiler to compiler:
gcc, llvm/clang: **-fopenmp**
Intel compilers: **-openmp**
Etc.

OpenMP example

```
#pragma omp parallel for
for (i = 1; i < N; i++)
    A[i] = A[i-1] + A[i+1];
```

This is not a proper parallel loop, because a value written in one iteration (say i=X) is used in the next iteration i=X+1

But if you tell OpenMP to do it, it will happily execute in parallel. The results won't be the same as sequential execution

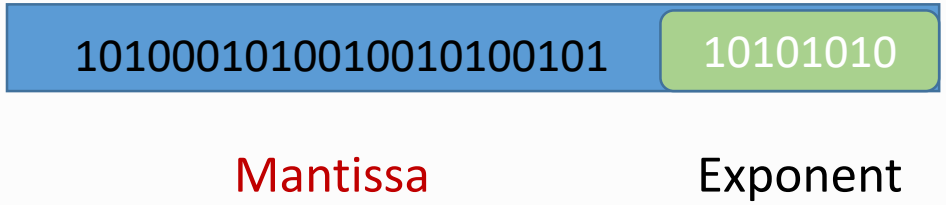
A good compiler may give you a warning, but it doesn't have to

How to decide if a loop is parallel

- Basic rule: if the result of executing it in parallel will *always* be the same as sequential execution
  - Same as: small floating point differences arising out of (say) adding numbers in different order are ok

A little note on floating point numbers

- What problem does the floating-point representation solve?
  - Representing numbers with high dynamic range
  - Tiny numbers like 0.00000000000001234 or
  - Large numbers like 98345600000000
  - But you have only 32 bits (single precision “float”)



$$\text{Value} = 1.\text{mantissa} \times 2^{(\text{exponent}-127)}$$

Floating point example

- When you add two numbers, the system shifts (as it must) the mantissa to equalize exponent, then adds the mantissas together
- This shift causes some bits to be dropped off
- Example: in Oct code ... (Oct: 3 bits written as 1 digit between 0 … 7)

$$
\begin{aligned}
& 1.04753625 * 2^6 \\
+\ & 1.60000000 * 2^6 \\
=\ & 2.64753625 * 2^6
\end{aligned}
$$

$$
\begin{aligned}
& 1.04753625 * 2^3 \\
+\ & 1.60000000 * 2^6 \\
=\ & 0.10475362 * 2^6 \\
+\ & 1.60000000 * 2^6 \\
=\ & 1.70475362 * 2^6
\end{aligned}
$$

Floating point issues

- So, adding numbers in different order may give you (slightly, hopefully) results
  - In decimal, assume you have only 4 digits available
  - Consider 5751 + 21.43 + 16.40 + 2.543
  - I.e., $5.751 * 10^3 + 2.143 * 10 + 1.640 * 10 + 2.543 * 1$
  - (5751 + 21.43) + (16.40 + 2.543) = (5772) + (18.94) = 5790
  - 5751 + ((21.43 + 16.40) + 2.543) = 5751 + (40.37) = 5791

Floating point issues

- So, adding numbers in different order may give you (slightly, hopefully) results
  - In decimal, assume you have only 4 digits available
  - Consider 5751 + 21.43 + 16.40 + 2.083
  - I.e., $5.751 * 10^3 + 2.143 * 10 + 1.640 * 10 + 2.543 * 1$
  - (5751 + 21.43) + (16.40 + 2.083) = (5772) + (18.48) = 5790
  - 5751 + ((21.43 + 16.40) + 2.083) = 5751 + (40.37) = 5791

Floating point arithmetic is not associative

- Consequences:
  - Comparing for "equality" is not useful
  - Instead of "if (x==y)" you may use "if abs(x-y) < THRESHOLD"
- From the parallelization point of view:
  - You must decide if reordering of floating point operations is ok for your purpose
    - x + (y+z) , or (x+y) + z or even (x+z) + y
  - In most cases it is
  - If so, some loops are parallelizable
- Basic rule: if the result of executing it in parallel will *always* be the same as sequential execution
  - The past few slides explained "same as" part

How to decide if a loop is parallel

- Basic rule: if the result of executing it in parallel will *always* be the same as sequential execution
- **Why *always*?**
  - You have to be careful that some possible execution with some number of threads doesn't produce the wrong result
  - You definitely don't want a program that runs correctly 99.9% of the time
  - Because it will fail once in a while and it'd be hard to debug

How to decide if a loop is parallel

- Basic rule: if the result of executing it in parallel will *always* be the same as sequential execution
- **How do we deal with "always"?**
- It is useful to imagine you are running with an infinite number of threads and that they can slow down, pause, or speed up as much as they want
- Next, we will look at a few rules of thumb that allow us to quickly decide, in case of some special patterns, if the loop is parallel
- Later, we will learn OpenMP constructs that allow parallelization of loops that don't look to be parallel by the above rule
- We will also learn how to refactor (change) your code

Simple rules

- If
  - **(A)** The loop only reads some arrays/and writes other arrays,
    - I.e., there is no array (or scalar variable) that's both written and read by the loop
  - **And**
  - **(B)** Each iteration writes to a different portion of array
- Then the loop can be safely parallelized

```
#pragma omp parallel for
for (i = 1; i < N; i++)
    B[i] = A[i-1] + A[i+1];
```

```
#pragma omp parallel for
for (i = 1; i < N; i++)
    B[i] = x * A[i-1];
```

The converse of this rule is not necessarily true (i.e., if the rule does not apply, it may still be possible to execute the loop in parallel)

Simple rules: when the loop is not parallel

- If one iteration writes to a variable and the next iteration reads it
- If one iteration writes to a location and some subsequent iteration reads it
- If a later iteration writes to a variable and an earlier iteration reads it

```
#pragma omp parallel for
for (i = 1; i < N; i++)
  B[i-1] = x * A[i] * B[i];
```

There are tricks where you can ***restructure*** the code so it can be run in parallel ... more on that later

Using simple rules, are these parallelizable?

```
for (i = 1; i < N; i++)
  B[i] += A[i];
```

- Yes (A and B)

```
for (i = 1; i < N; i++)
  B[0] += A[i-1];
```

- No (B)

```
for (i = 1; i < N; i++)
  A[N-i] += A[i-1];
```

- No (write and later reads)

Assume N is 1000. iteration i=5, (e.g.) is writing to A[995] and reading from A[4]. Iteration i=996 is writing to A[4]. So, iteration 5 may use either stale or new value in A[4], depending on execution order

```
for (i = 0; i < N; i++)
  A[i] += A[(i-N/2)%N];
```

- No (write and earlier reads)