CSc 422
Introduction to Parallel and Distributed Computing

- Instructor: David Lowenthal
- TAs:
  - Mark Nguyen
  - Krishna Prashanth Thummanapelly

Parallelizing Programs

- Goal: speed up programs using multiple processors/cores

When is speedup important?

- Applications can finish sooner
  - Search engines
  - High-res graphics
  - Weather prediction
  - Nuclear reactions
  - Bioinformatics
  - AI

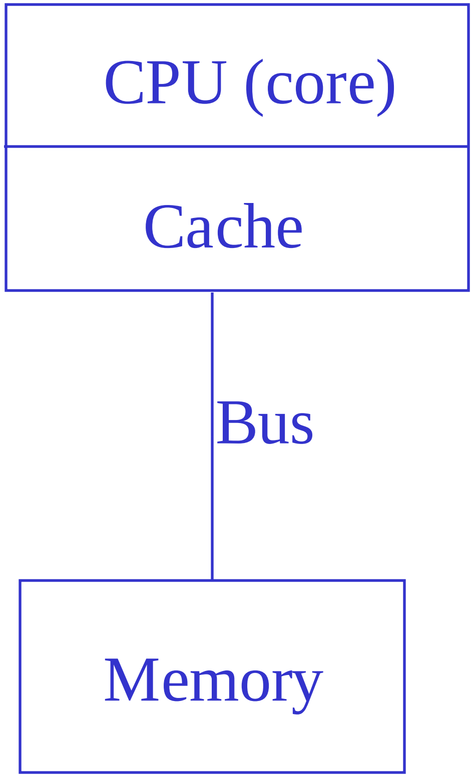
Types of parallel machines

- Special purpose
  - GPU, FPGA
- General purpose (our focus in this course)
  - Shared-memory multiprocessor (“multicore”)
  - Distributed-memory multicomputer

SIMD vs. MIMD

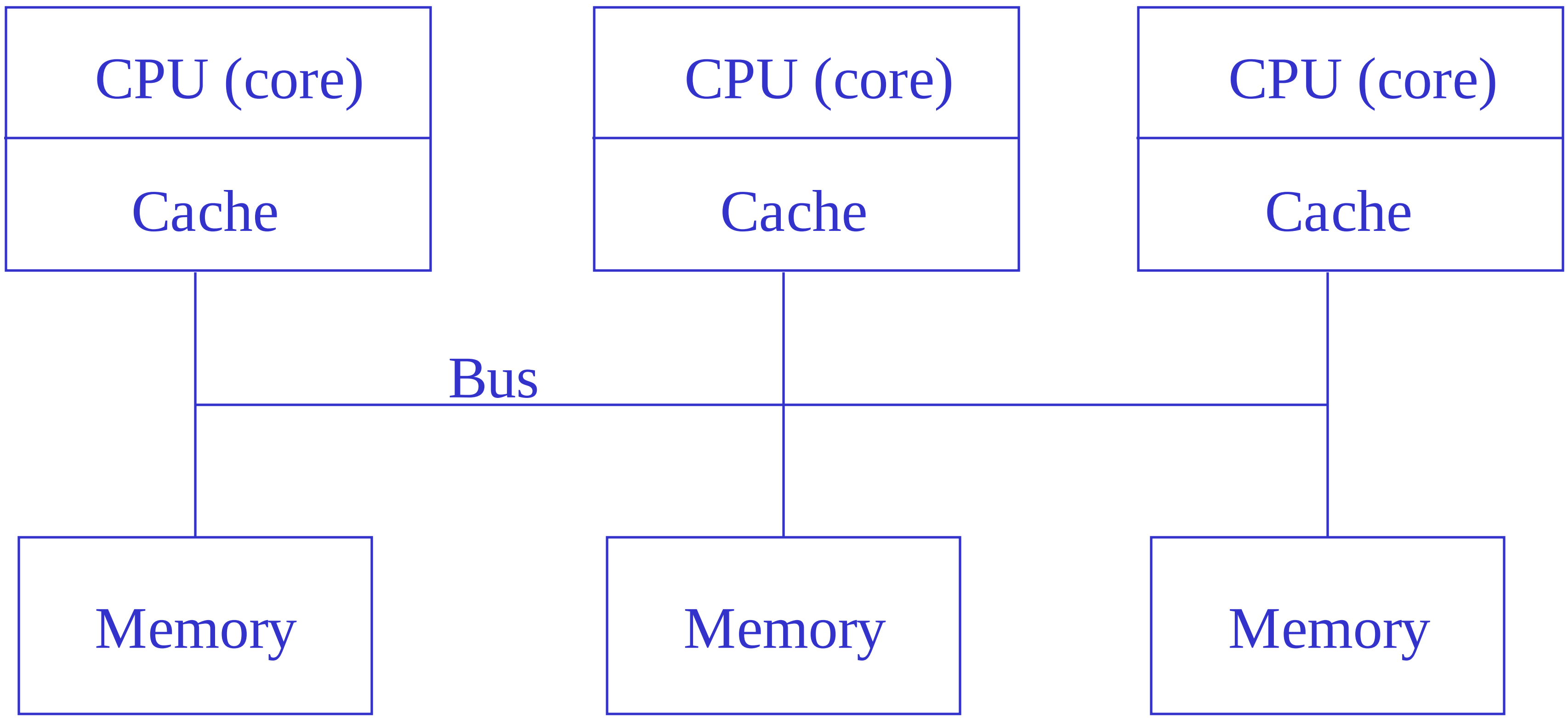
- SIMD: single instruction, multiple data
  - GPU is in this category
- MIMD: multiple instruction, multiple data
  - Multicore and multicomputer in this category

Review: Sequential Computer



What is the simplest way to extend this to a parallel computer?

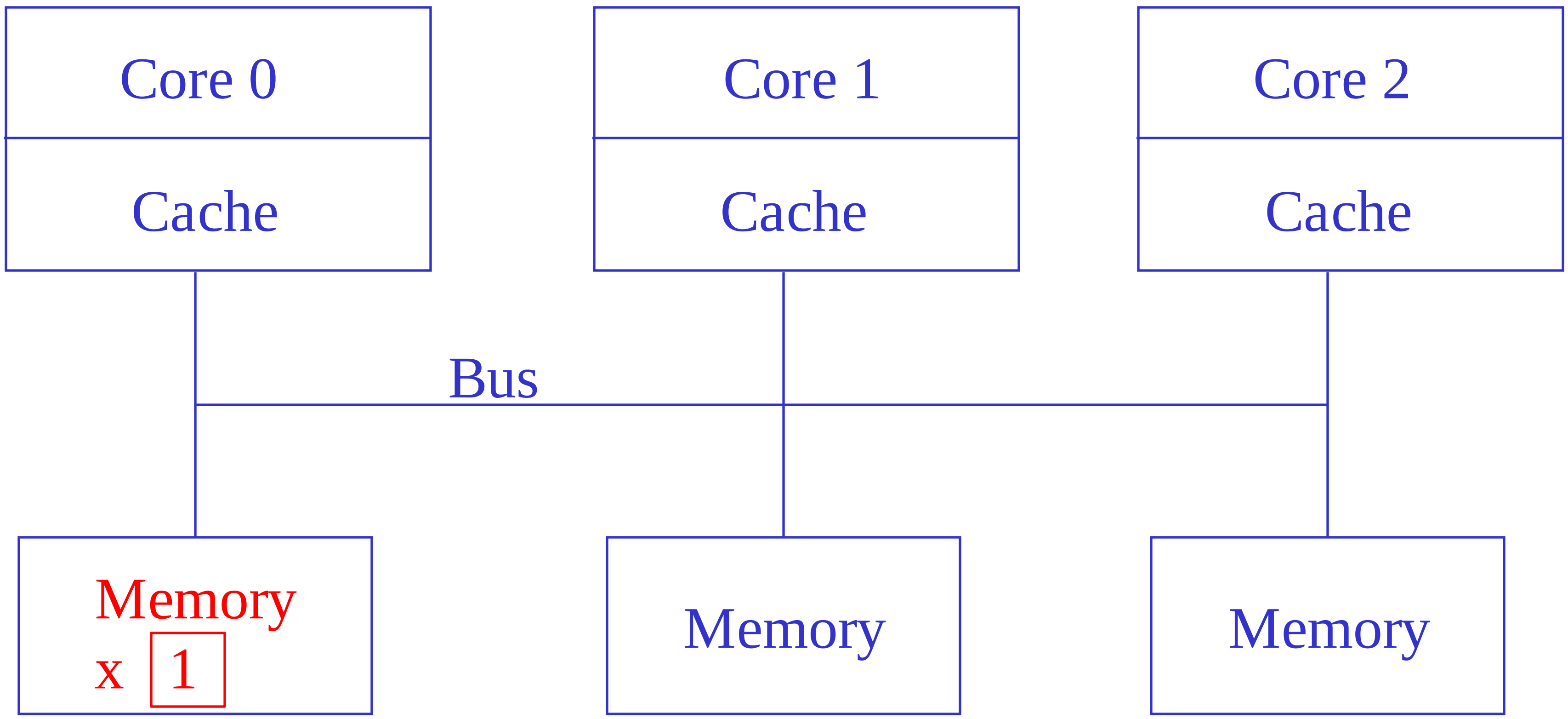
Shared-Memory Multiprocessor ("Multicore")



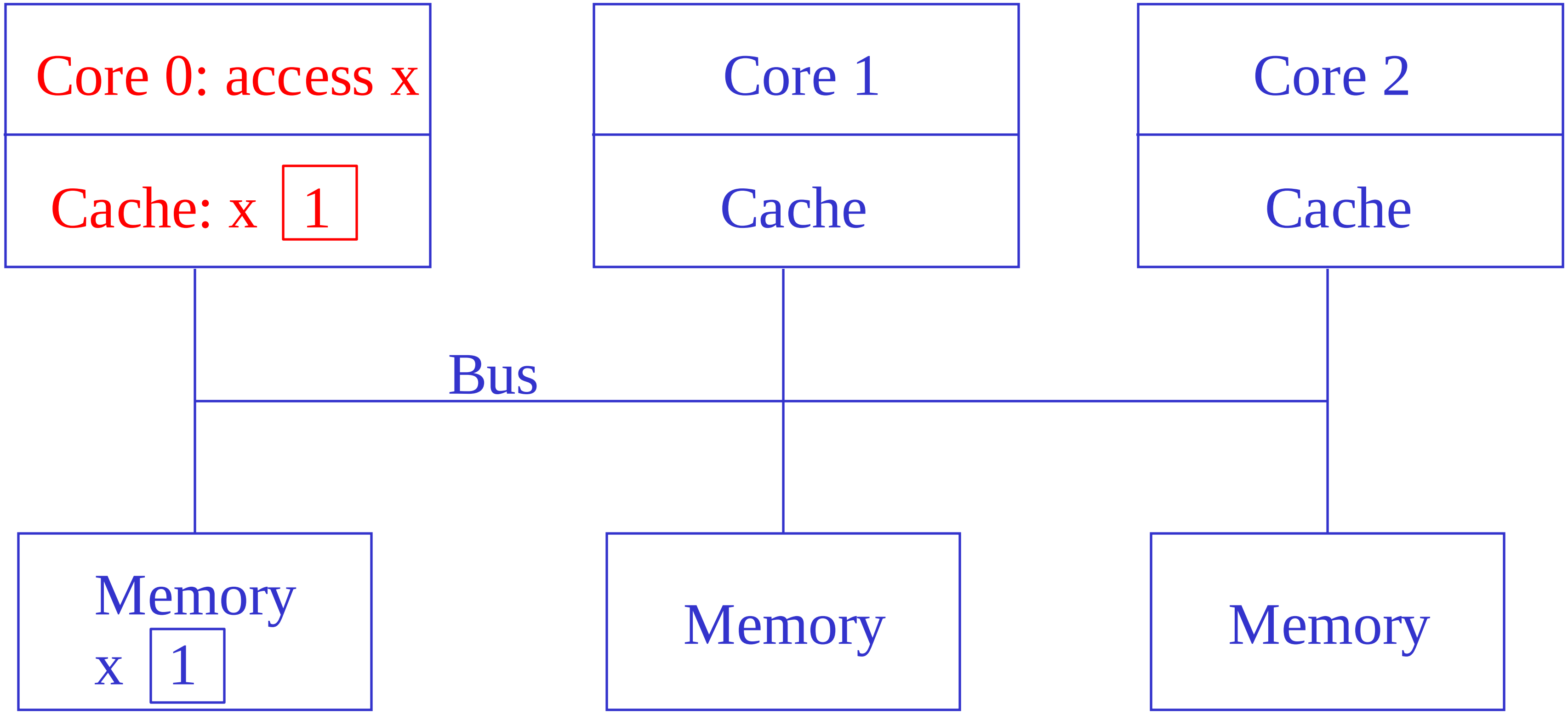
Memory is shared; Cache coherence is an issue

MIMD machine; each core executes independent instruction stream

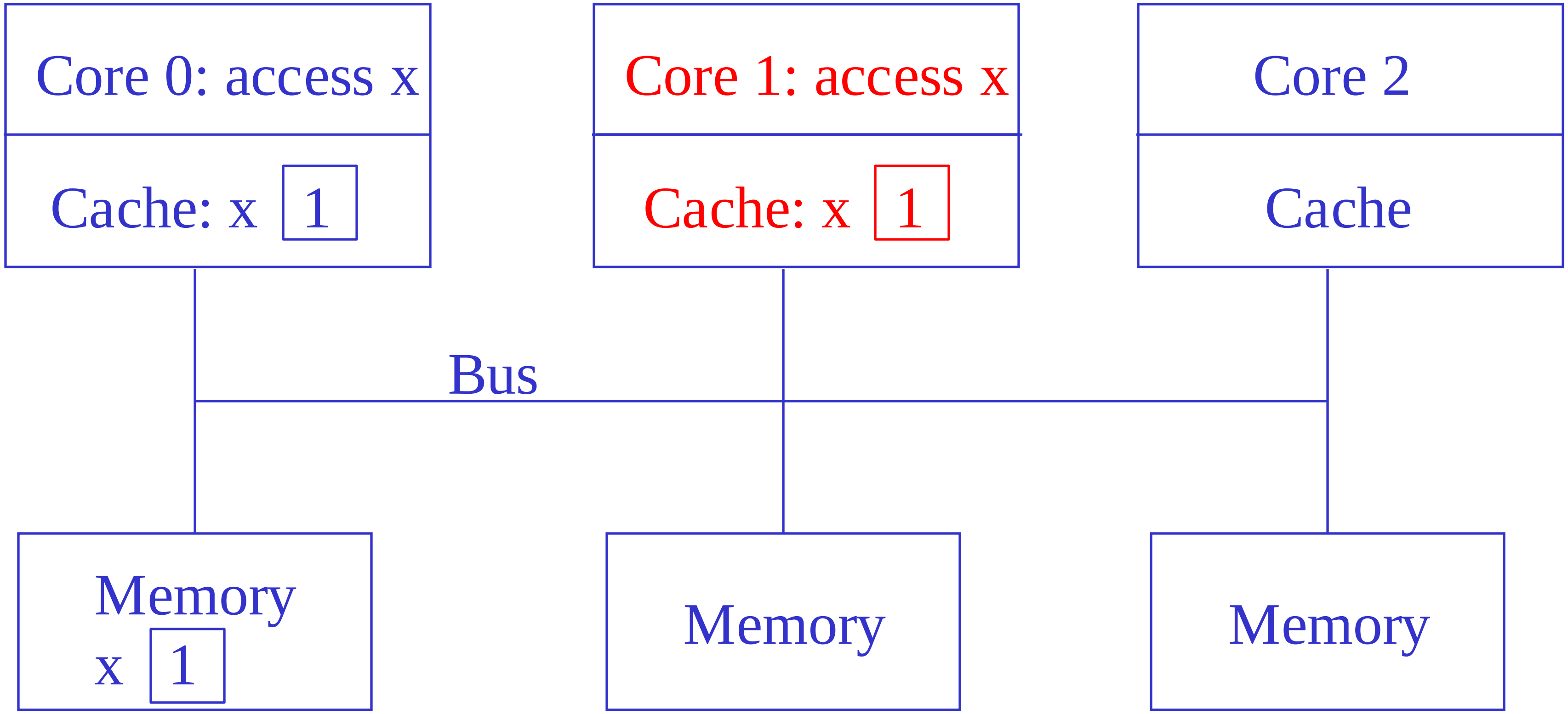
Cache Coherence Example
Initial State



Cache Coherence Example
First core accesses a variable

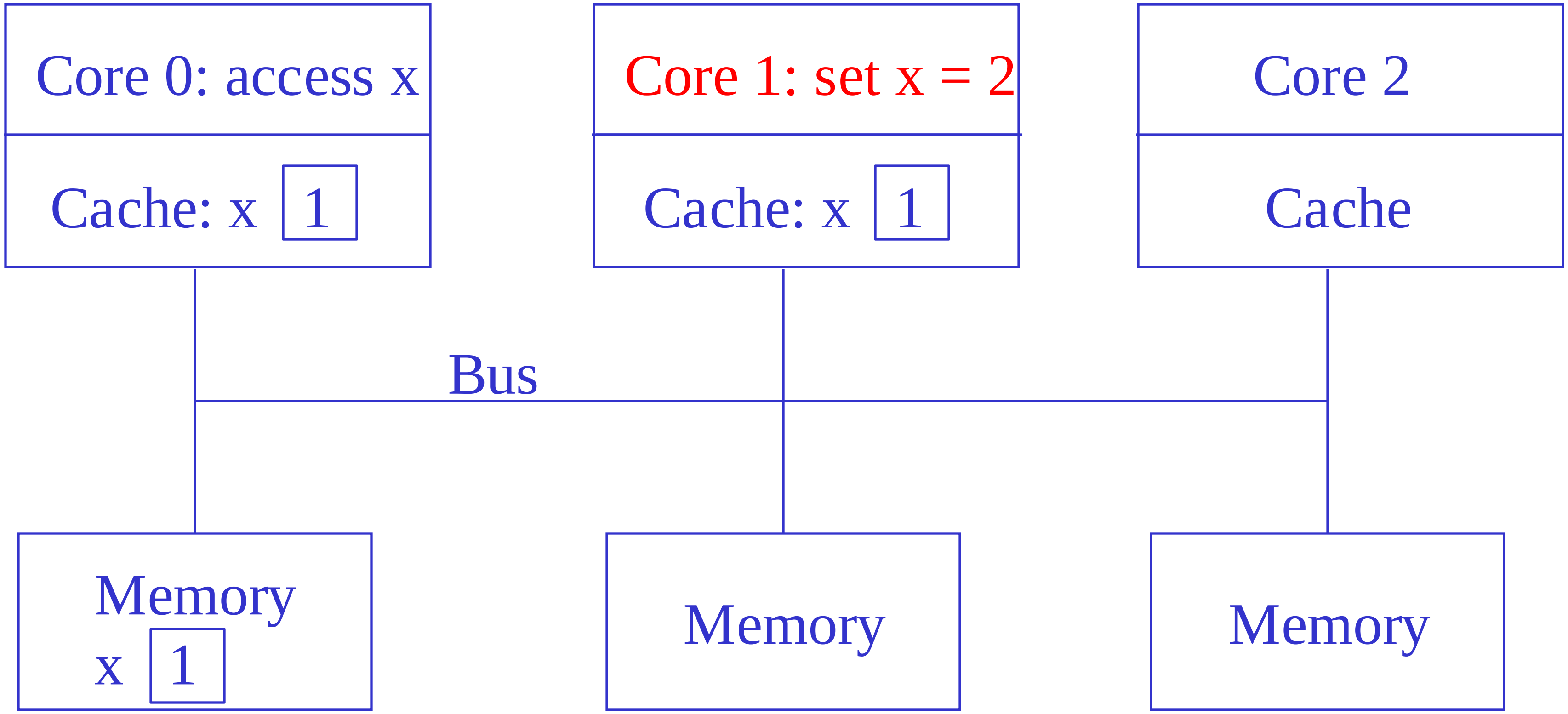


Cache Coherence Example
Second core accesses same variable



No issues: cores 0 and 1 can both read x’s value out of their cache

Cache Coherence Example
Either core writes to a variable

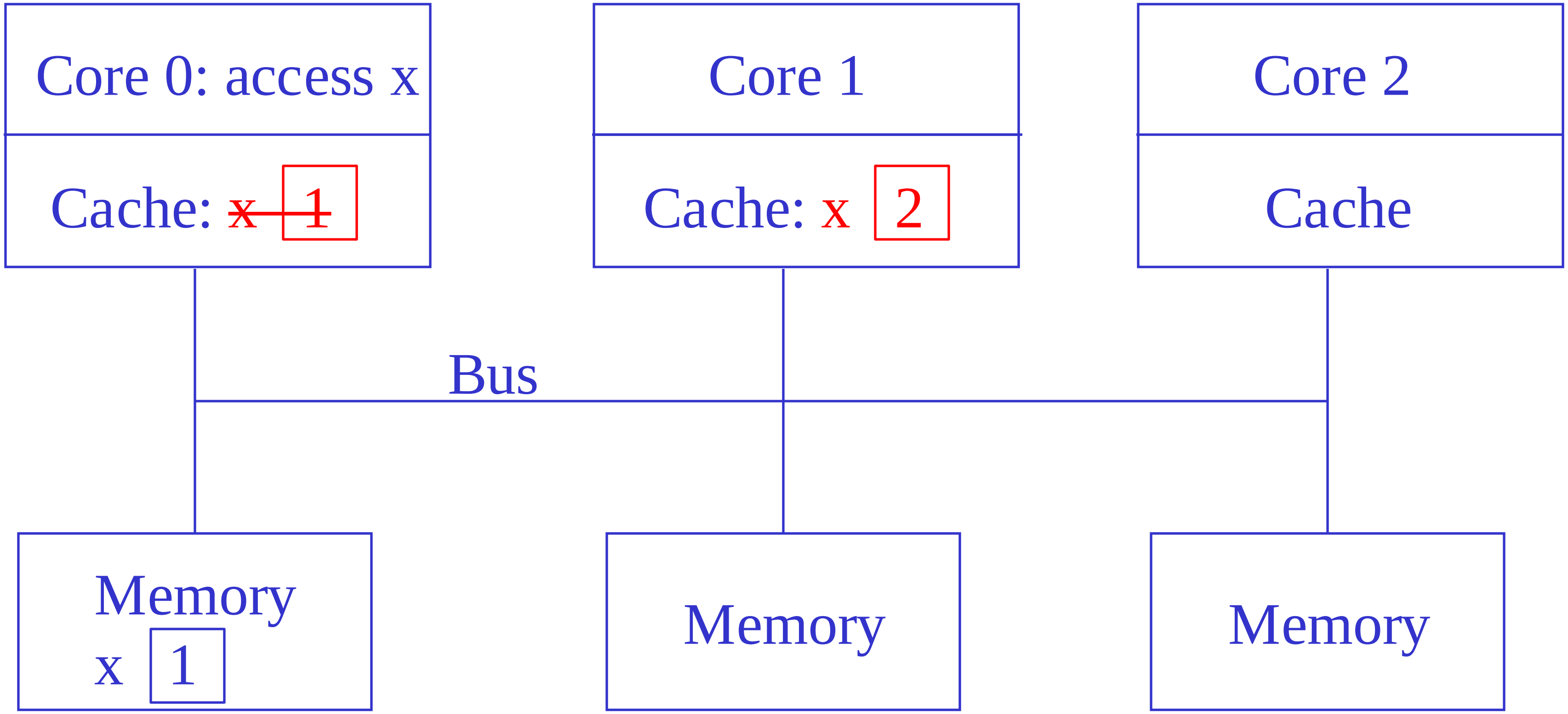


Now what happens?

Cache Coherence

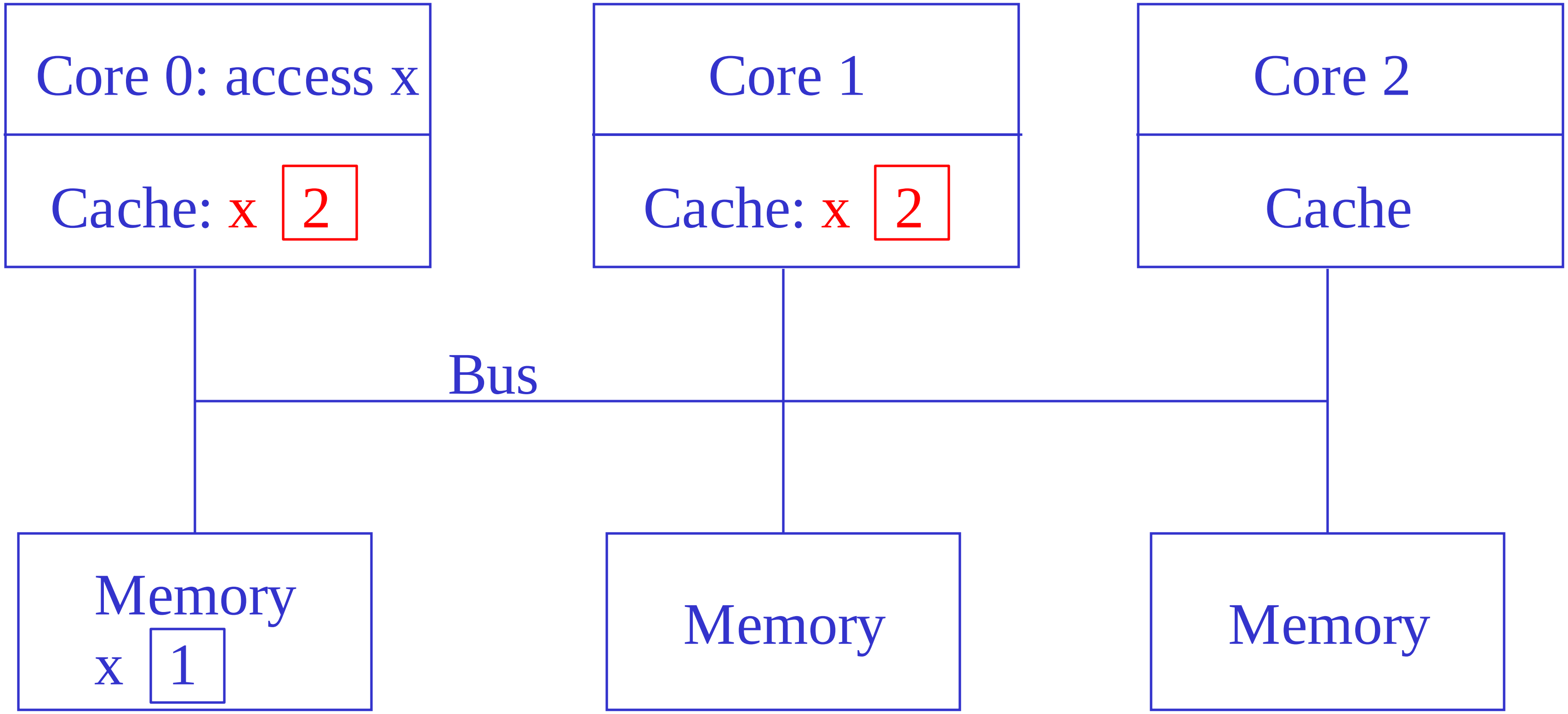
- Cached copies must remain consistent
  - Two ways to do so
    - Invalidate all but one cached copy
    - Update all cached copies
- Additionally, the memory copy can be:
  - Updated on every write (write-through)
  - Updated when cached copy is evicted (write-back)

Cache Coherence Example
Invalidate + Write Back



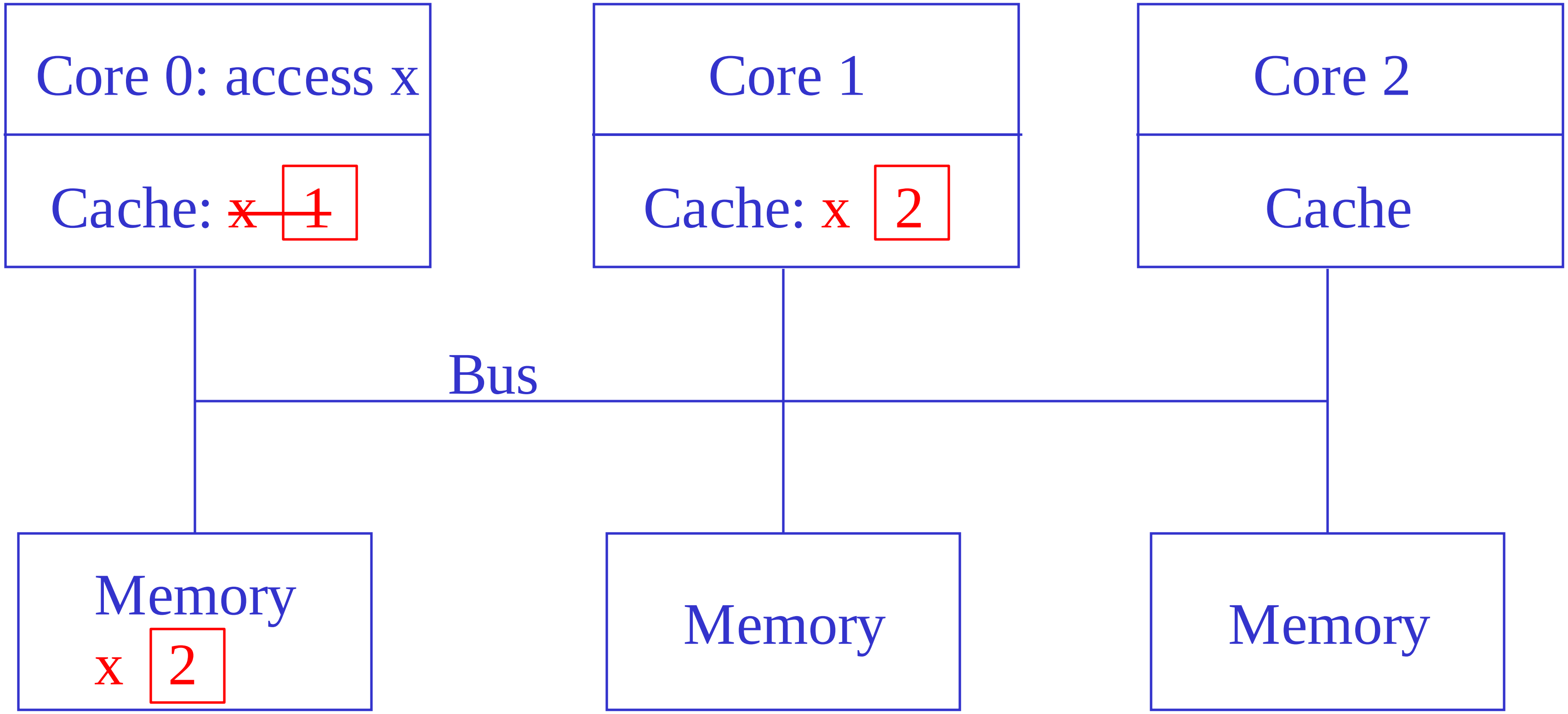
Cache Controller invalidates all cached copies except the writer's

Cache Coherence Example
Update + Write Back



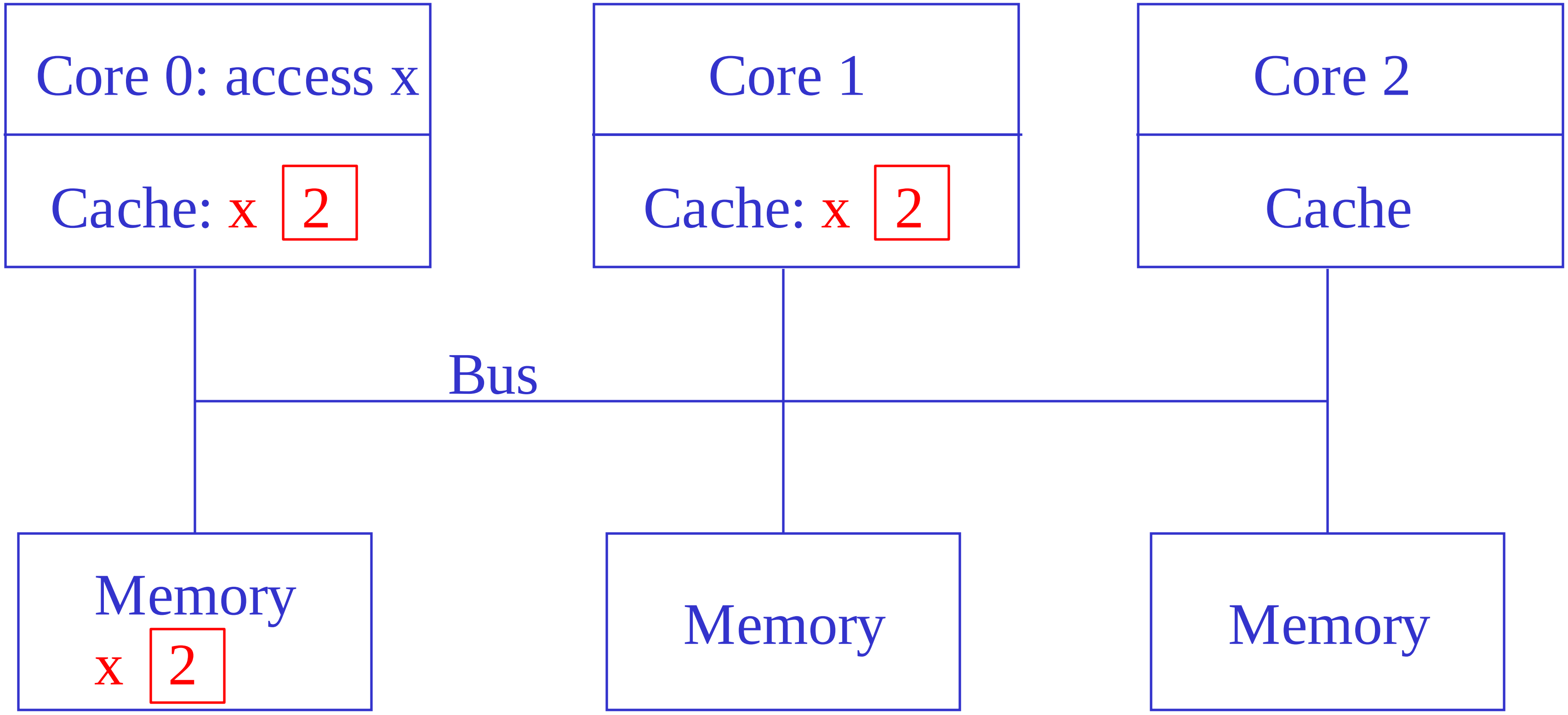
Cache Controller ensures all cached copies are updated

Cache Coherence Example
Invalidate + Write Through



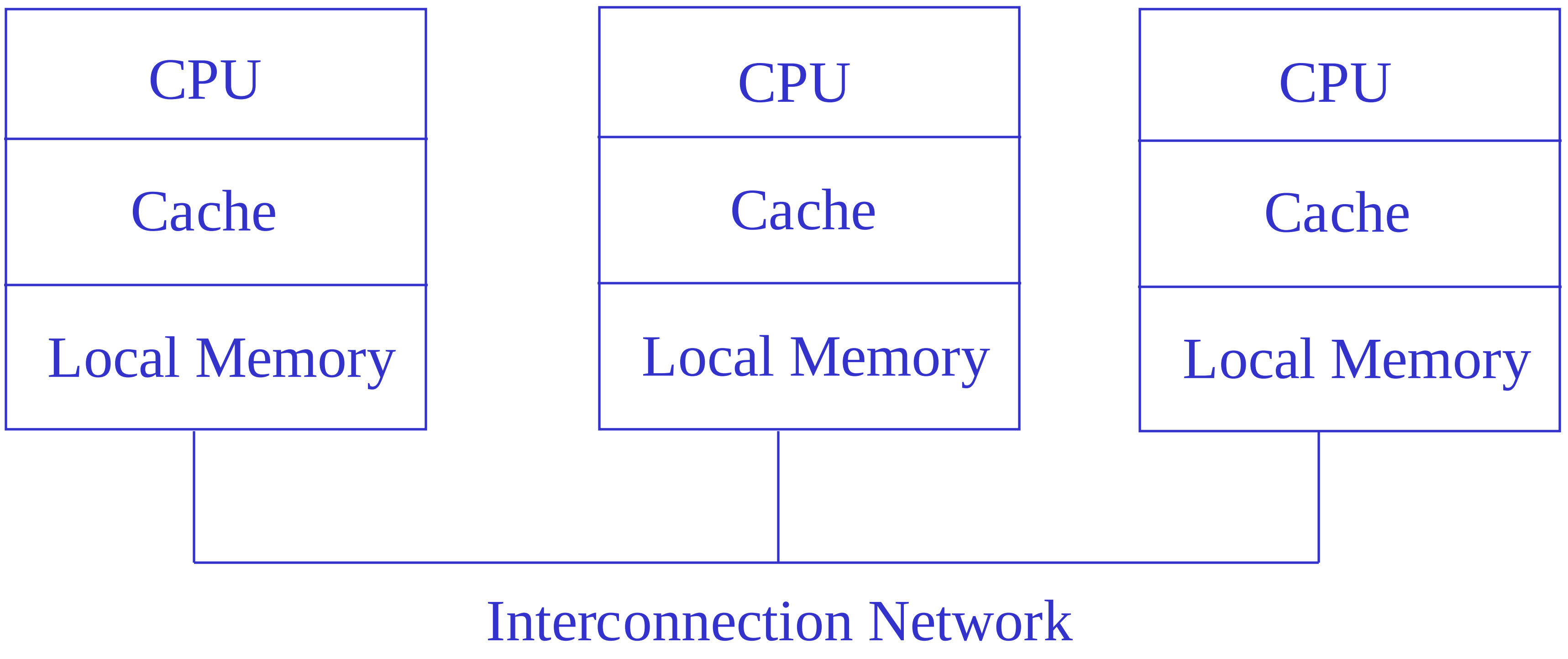
A write updates the writer's cached copy and the memory copy
All other cached copies are invalidated

Cache Coherence Example
Update + Write Through



All values are updated

Distributed Memory Multicomputer



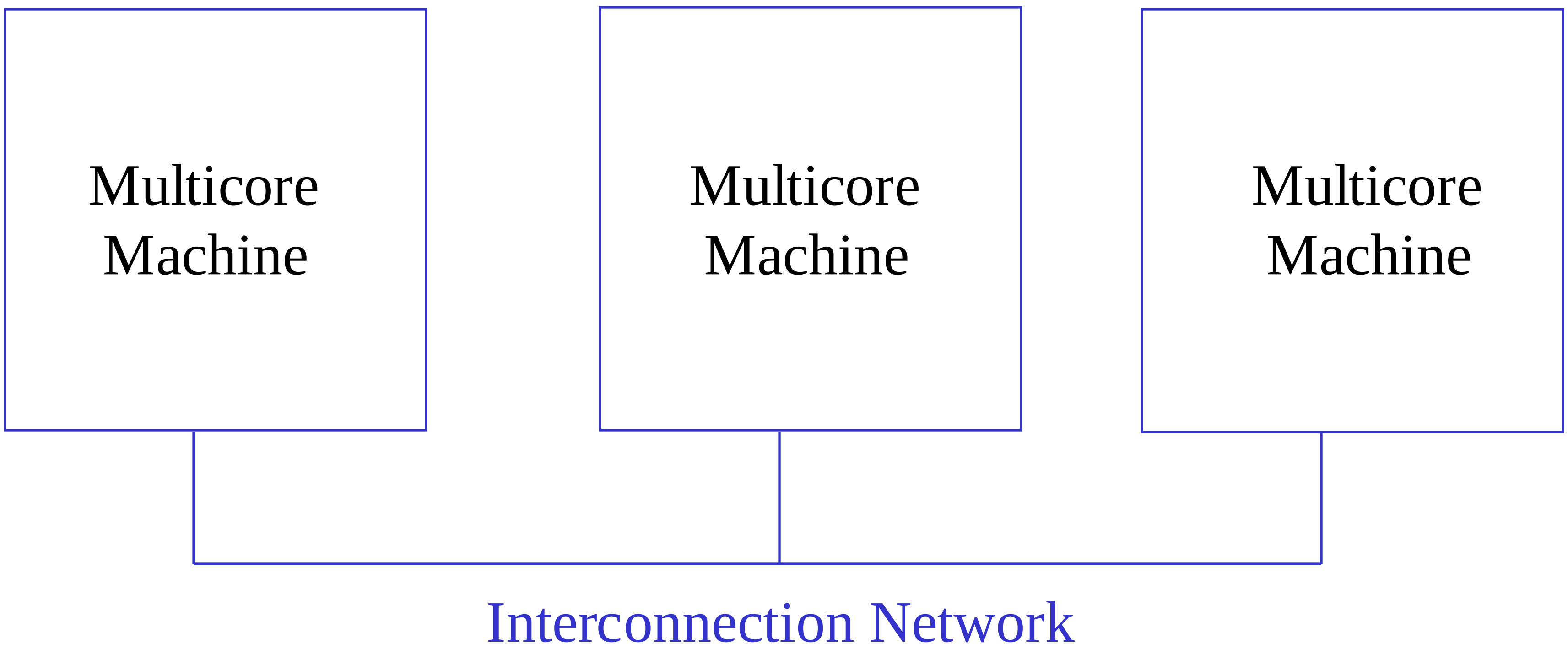
Memory is **not** shared

Also a MIMD machine

Multicomputer Details

- Each machine (“node”) is a full computer
  - Cache and memory are separate
  - CPUs cannot access each other’s memory directly
    - Only can do so through messages over the interconnect

All Machines today are Multicore (this is still a multicomputer)



Hybrid approach
Memory is **not** shared between machines

Real-World Supercomputer Example

- El Capitan (Lawrence Livermore National Lab)
  - 1.7 Exaflops
  - 11,136 nodes and 11M+ total CPU+GPU cores
  - 4 GPUs and one CPU (with 24 cores) per node
    - Made by HPE/Cray
  - 512 GB memory/node
  - Just under 30 MW of power

If you are interested:

https://top500.org/lists/top500/2024/11/

Key Advantage/Disadvantage: Shared-Memory Multiprocessors

- Advantage:
  - Can write sequential program, profile it, and then parallelize the expensive part(s)
    - No other modification necessary
- Disadvantage:
  - Does not scale to large core counts
    - Bus saturation, hardware complexity

Key Advantage/Disadvantage: Distributed-Memory Multicomputers

- Advantage:
  - Can scale to large numbers of nodes
- Disadvantage:
  - Harder to program
    - Must modify *entire* program even if only a small part needs to be parallelized

(Sequential) Matrix Multiplication

```
double A[n][n], B[n][n], C[n][n]
for i = 0 to n-1
  for j = 0 to n-1
    double sum = 0.0
    for k = 0 to n-1
      sum += A[i][k] * B[k][j]
    C[i][j] = sum
```

Question: how can this program be parallelized?

Steps to parallelization

- First: find parallelism
  - Concerned about what can *legally* execute in parallel
  - At this stage, expose as *much* parallelism as possible
  - Partitioning can be based on data structures or function

Other steps are architecture dependent

Finding Parallelism in Matrix Multiplication

- Can we parallelize the inner loop?

(Sequential) Matrix Multiplication

```
double A[n][n], B[n][n], C[n][n]
for i = 0 to n-1
  for j = 0 to n-1
    double sum = 0.0
    for k = 0 to n-1
      sum += A[i][k] * B[k][j]
    C[i][j] = sum
```

Finding Parallelism in Matrix Multiplication

- Can we parallelize the inner loop?
  - No, because *sum* would be written concurrently

Finding Parallelism in Matrix Multiplication

- Can we parallelize the inner loop?
  - No, because *sum* would be written concurrently
- Can we parallelize the outer loops?

(Sequential) Matrix Multiplication

```
double A[n][n], B[n][n], C[n][n]
for i = 0 to n-1
  for j = 0 to n-1
    double sum = 0.0
    for k = 0 to n-1
      sum += A[i][k] * B[k][j]
    C[i][j] = sum
```

Finding Parallelism in Matrix Multiplication

- Can we parallelize the inner loop?
  - No, because *sum* would be written concurrently
- Can we parallelize the outer loops?
  - Yes, because (1) the write sets are disjoint for all iterations,
    - Write set for iteration (i,j) is $sum_{i,j}$, *C[i][j]*
  - And (2) the read set for iteration (i,j) does not contain $\text{sum}_{x,y}$ or C[x][y], unless x==i and y==j
    - Read set for process (i,j) is $sum_{i,j}$, *A[i][k=0:n-1], B[k=0:n-1][j]*
  - Note: we have the option to parallelize just one of these loops

Finding Parallelism in Loops in General

- Assume a loop of the form:

for i := 0 to n-1

Body

- The read set of iteration *i* of the loop is the set of all variables that are read on iteration *i*
- The write set of iteration *i* of the loop is the set of all variables that are written on iteration *i*

Finding Parallelism in Loops in General

- Assume a loop of the form:

  for i := 0 to n-1

  Body

- Let the read set for iteration *i* of the loop be R_i and the write set be W_i
- The iterations of the loop are independent if **all** of the following are true:
  - $R_i \cap W_j == \emptyset$, for all $i \neq j$
  - $W_i \cap R_j == \emptyset$, for all $i \neq j$
  - $W_i \cap W_j == \emptyset$, for all $i \neq j$

Terminology

- *co* statement: creates concurrency

  co i := 0 to n-1

  Body

  oc

- Semantics: *n* instances of body are created and executed concurrently until the *oc*
  - All instances must complete before single thread proceeds after the *oc*
- Implementation: create *n* threads, join them at the *oc*
- Can also be written *co* b_1 // b_2 // … // b_n *oc*

Notation

- I will use the following three versions of *co* interchangeably; they mean the same thing

co i := 0 to n-1

    Body_i

oc

co i := 0 to n-1 {

    Body_i

}

co Body_0 // Body_1 // … // Body_{n-1} *oc*

Terminology

- *Process* statement: also creates concurrency

  process i := 0 to n-1 {

  Body

  }
- Semantics: *n* instances of body are created and executed in parallel until the end of the *process*
- Implementation: create *n* processes
- No synchronization at end

Need to understand what processes/threads are!

Processes

- History: OS had to coordinate many activities
  – Example: deal with multiple users (each running multiple programs), incoming network data, I/O interrupts
- Solution: Define a model that makes complexity easier to manage
  – Process (thread) model

What's a process?

- Informally: program in execution
- Process encapsulates a physical processor
  - everything needed to run a program
    - code ("text")
    - registers (PC, SP, general purpose)
    - stack
    - data (global variables or dynamically allocated)
    - files
- NOTE: a process is sequential

Examples of Processes

- Shell: creates a process to execute command

  lectura:> ls foo

  (shell creates process that executes “ls”)

  lectura:> cat foo & grep bar & wc

  (shell creates three processes, one per command)

- OS: creates a process to manage printer
  - process executes code such as:

    wait for data to come into system buffer

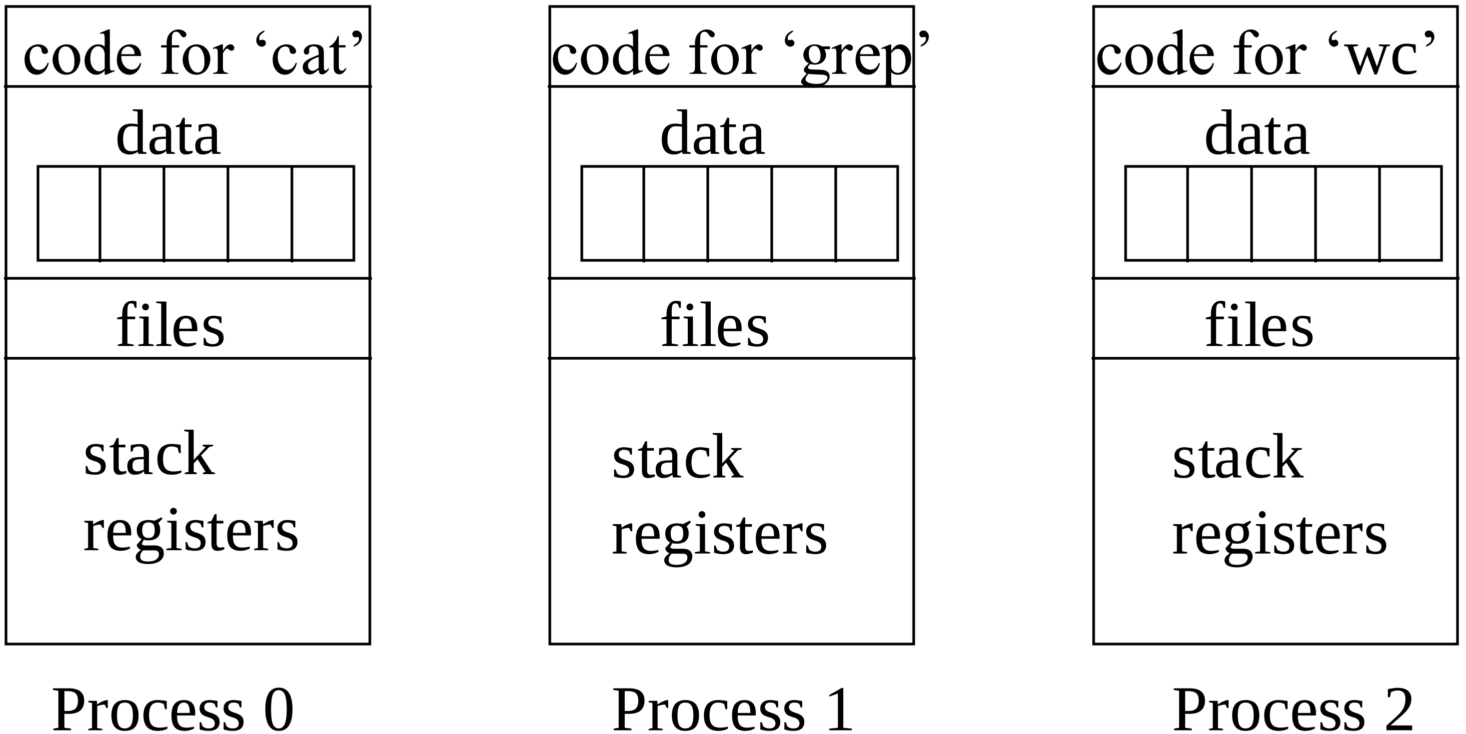
    move data to printer buffer

Creating a Process

- Must somehow specify code, data, files, stack, registers
- Ex: UNIX
  - Use the fork( ) system call to create a process
  - Makes an **exact** duplicate of the current process
    - (returns 0 to indicate child process)
  - Typically, exec( ) is run on the child

We will not be doing this (systems programming)

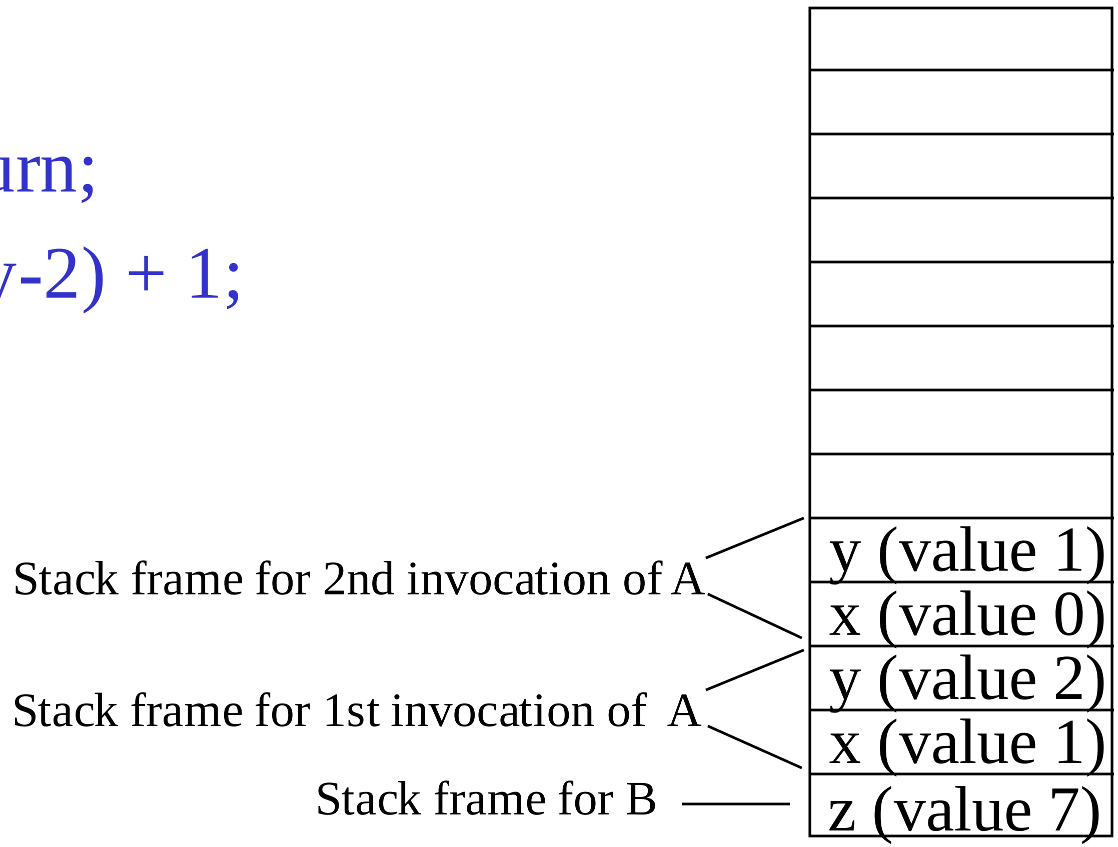
Example of Three Processes



OS switches between the three processes ("multiprogramming")

Review: Run-time Stack

```
A(int x) {
    int y = x+1;
    if (x == 0) return;
    else return A(y-2) + 1;
}
B( ) {
    int z = 7;
    A(1);
}
```



Decomposing a Process

- Process: everything needed to run a program
- Consists of:
  - Thread(s)
  - Address space

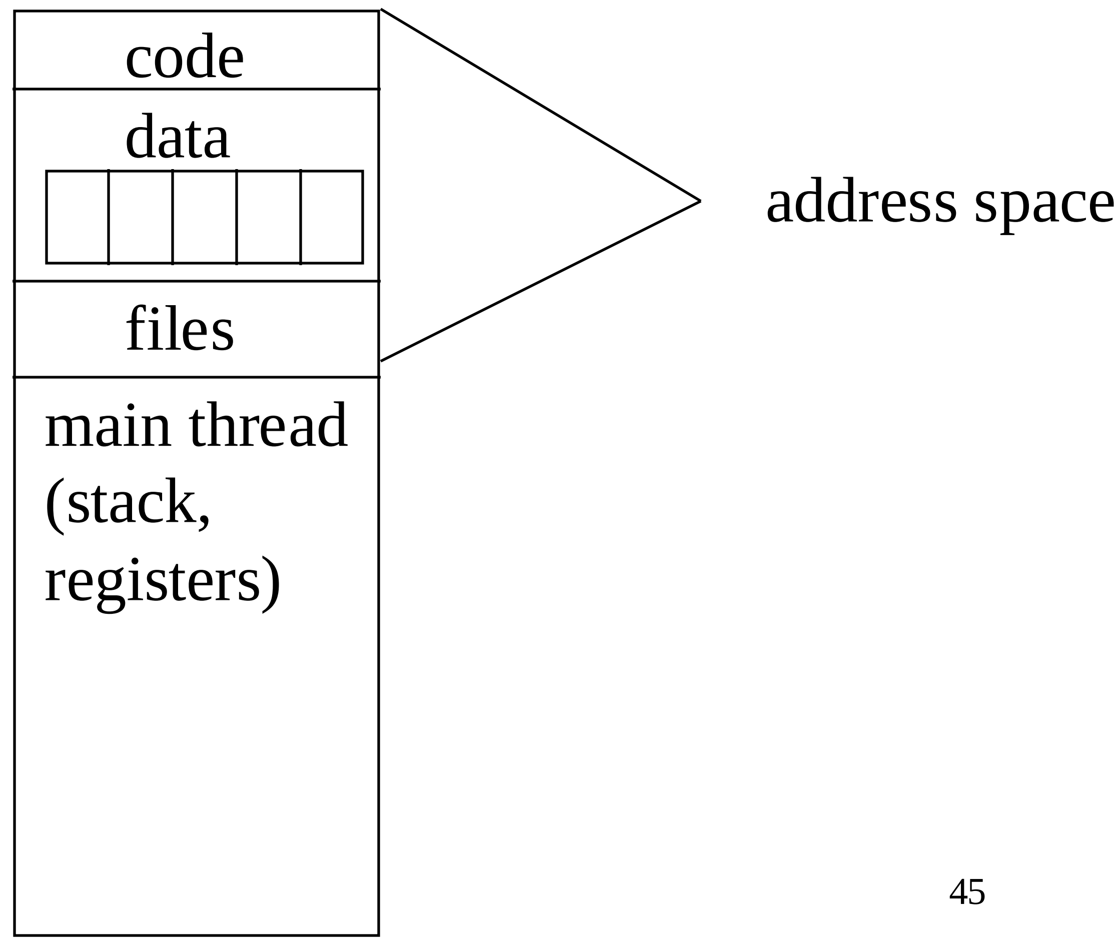
Thread

- Sequential stream of execution
- More concretely:
  - program counter (PC)
  - register set
  - stack
- Sometimes called lightweight process

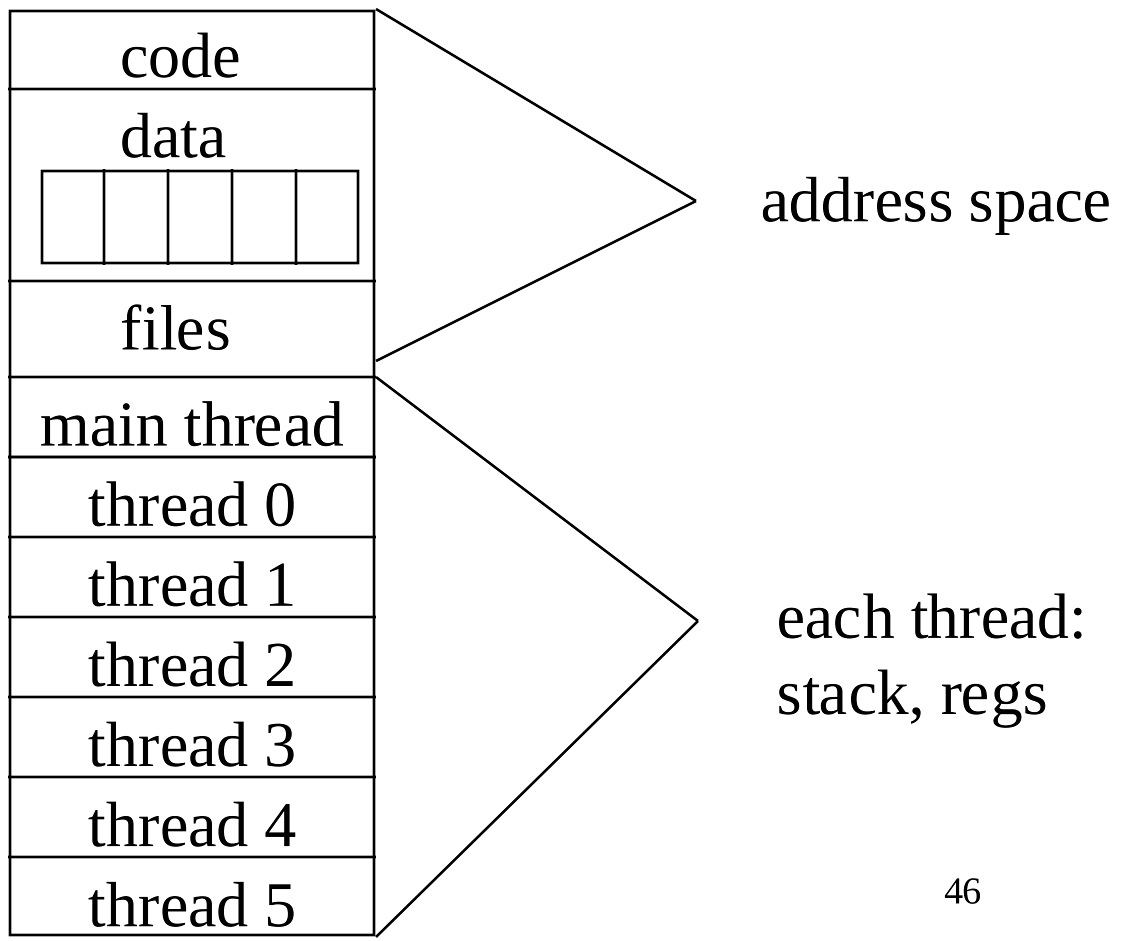
Address Space

- Consists of:
  - code
  - contents of main memory (data)
  - open files
- Address space can have > 1 thread
  - threads share memory, files
  - threads have separate stacks, register set

One Thread, One Address Space



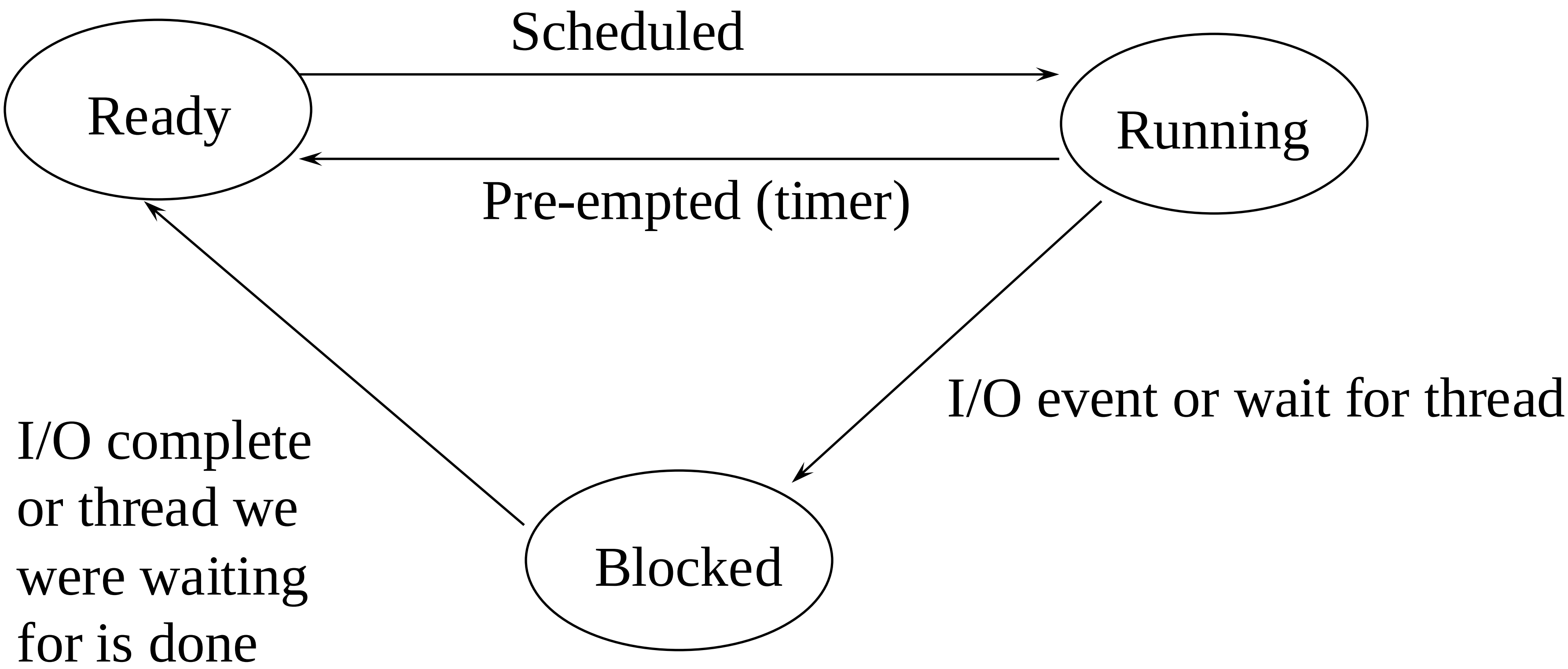
Many Threads, One Address Space



Thread States

- Ready
  - eligible to run, but another thread is running
- Running
  - using CPU
- Blocked
  - waiting for something to happen

Thread State Graph



Scheduler

- Decides which thread to run
  - (from ready list only)
- Chooses from some algorithm
- From point of view of CSc 422, the scheduler is something we cannot control
  - We have no idea which ready thread will be run
  - Our programs must not depend on a particular ready thread running before or after another ready thread

Context Switching

- Switching between 2 threads
  - change PC to current instruction of new thread
    - **might need to restart old thread in the future**
  - must save exact state of first thread
- What must be saved?
  - registers (including PC and SP)
  - what about stack itself?

Procedure Call Picture (time goes down)

```
Procedure A( ) {                    Procedure B( ) {




   call B( )
                                        return
                                    }


   call B( )



}
```

Procedure Call Picture (time goes down)

Procedure A( ) {

call B( )

call B( )

}

Procedure B( ) {

return

}

Procedure Call Picture (time goes down)

Procedure A( ) {

call B( )

call B( )

}

Procedure B( ) {

return

}

Procedure Call Picture (time goes down)

Procedure A( ) {

call B( )

call B( )

}

Procedure B( ) {

return

}

Procedure Call Picture (time goes down)

Procedure A( ) {

call B( )

call B( )

}

Procedure B( ) {

return

}

Context Switching Picture (time goes down)

Thread A	Thread B
Switch to B	
	Switch to A
Switch to B	
	Switch to A

Context Switching Picture (time goes down)

Thread A

Thread B

Switch to B

Switch to A

Switch to B

Switch to A

Context Switching Picture (time goes down)

Thread A

Thread B

Switch to B

Switch to A

Switch to B

Switch to A

Recall: Procedure Call Picture (time goes down)
(So far this looks the same as context switching)

Procedure A( ) {

call B( )

call B( )

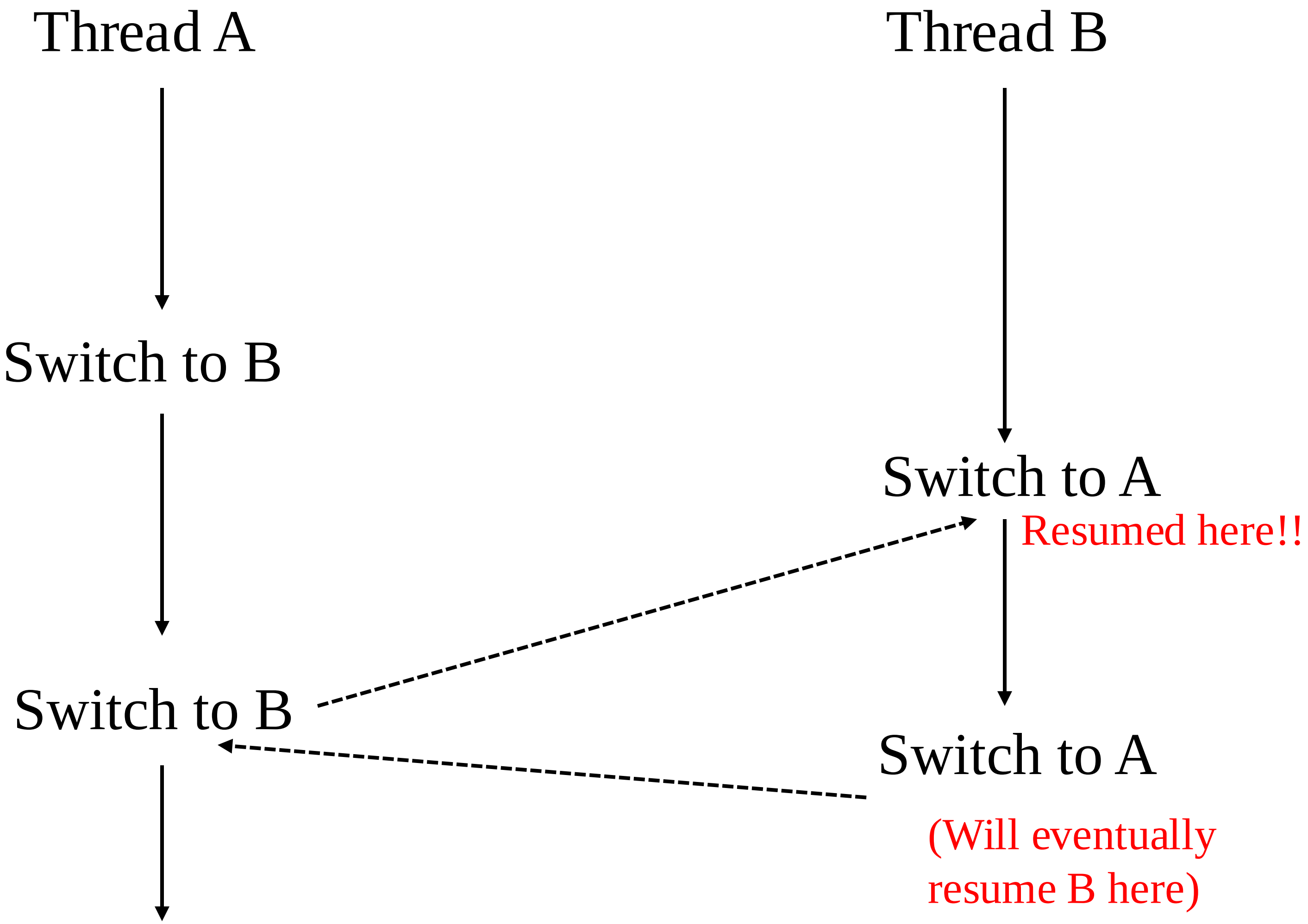
}

Procedure B( ) {

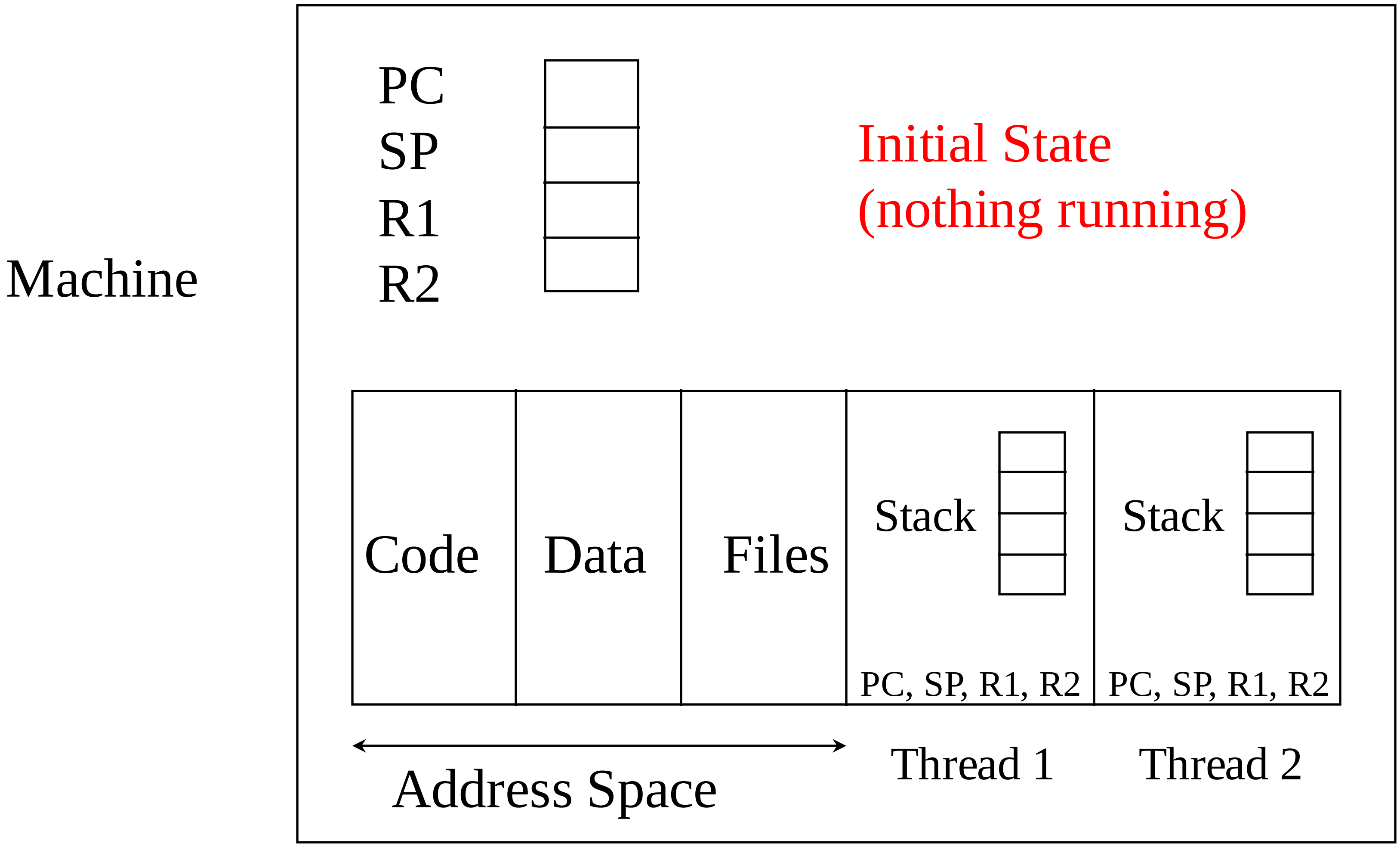
return

}

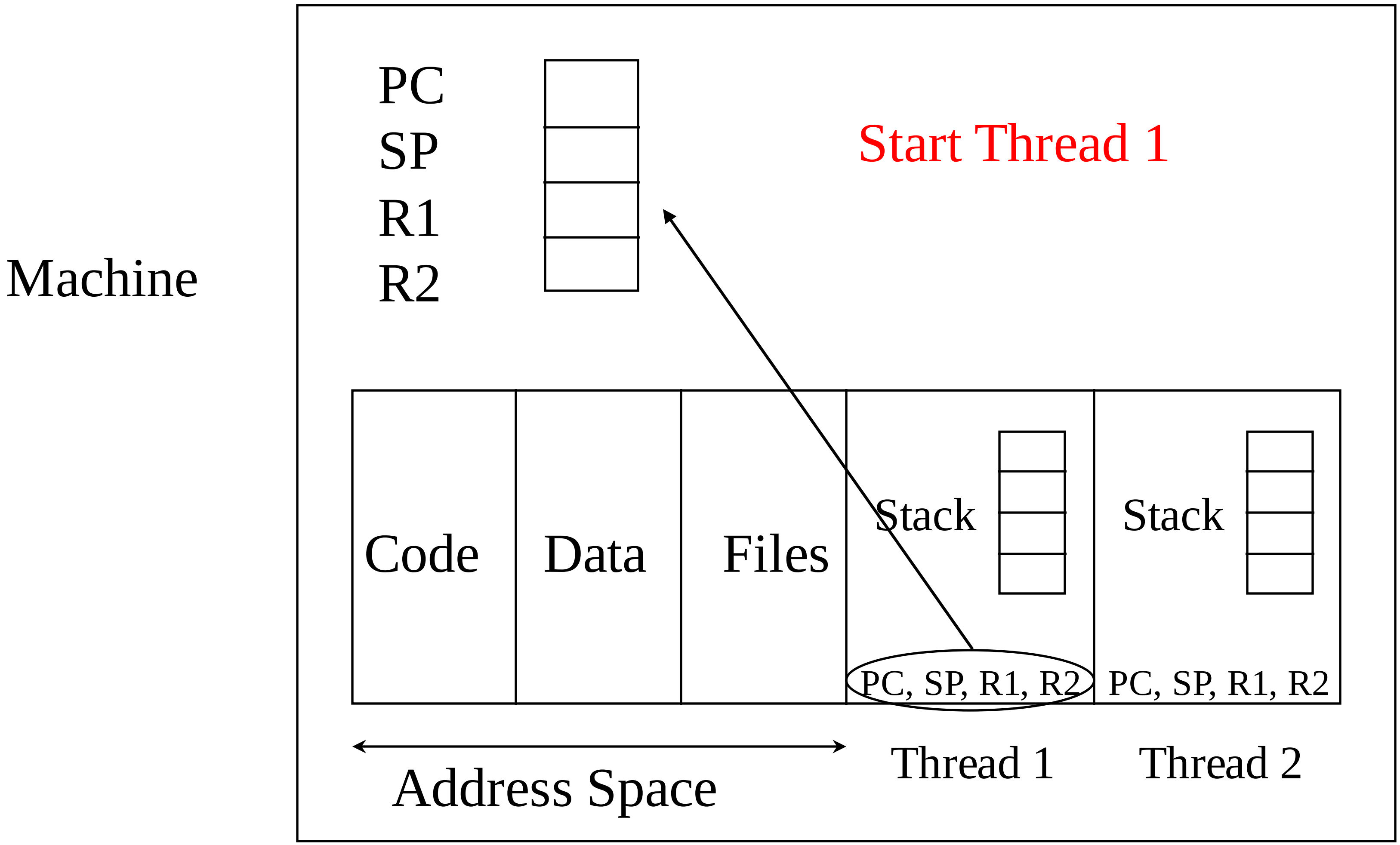
Context Switching Picture (time goes down)



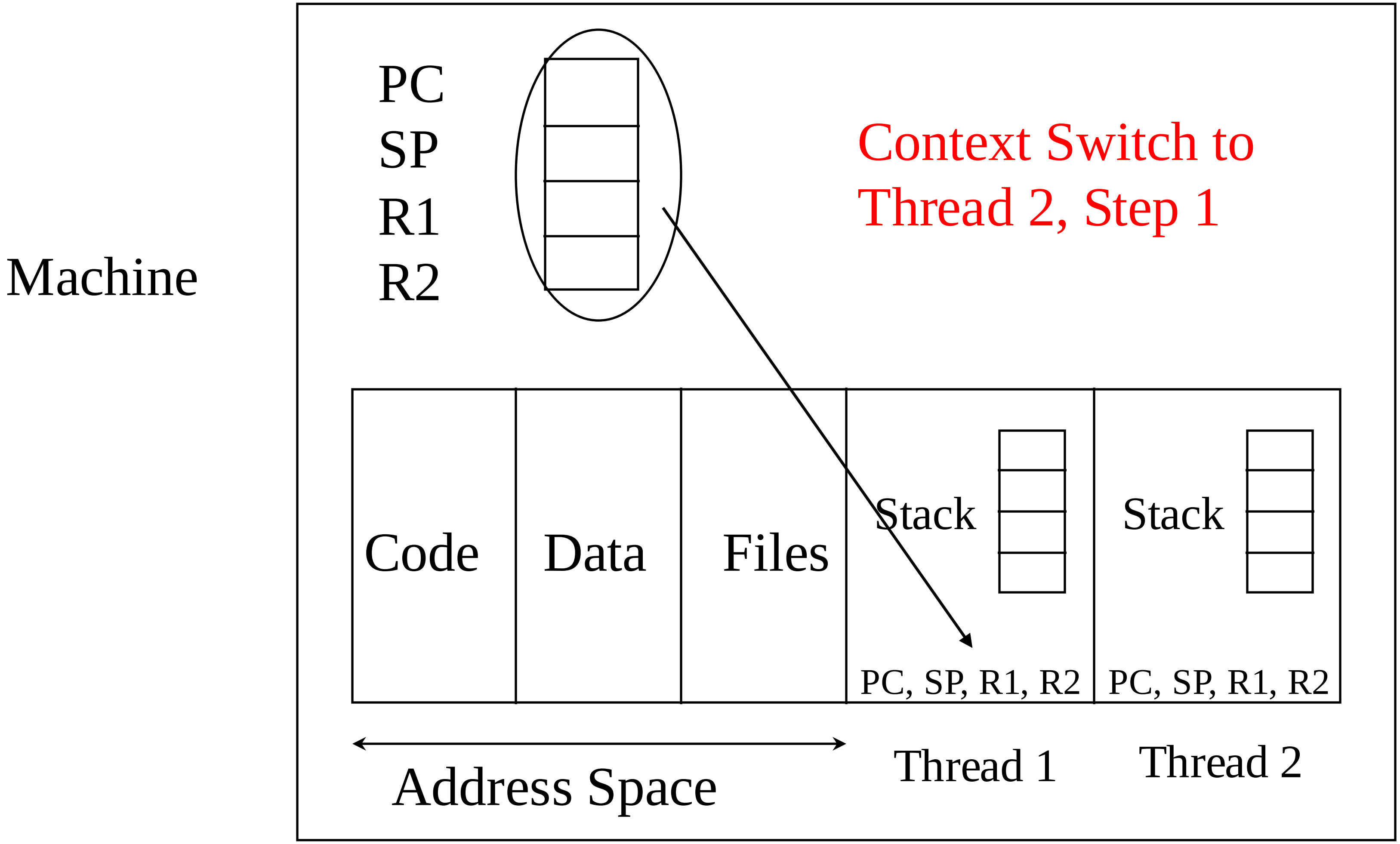
Multiple Threads, One Machine (Single Core)



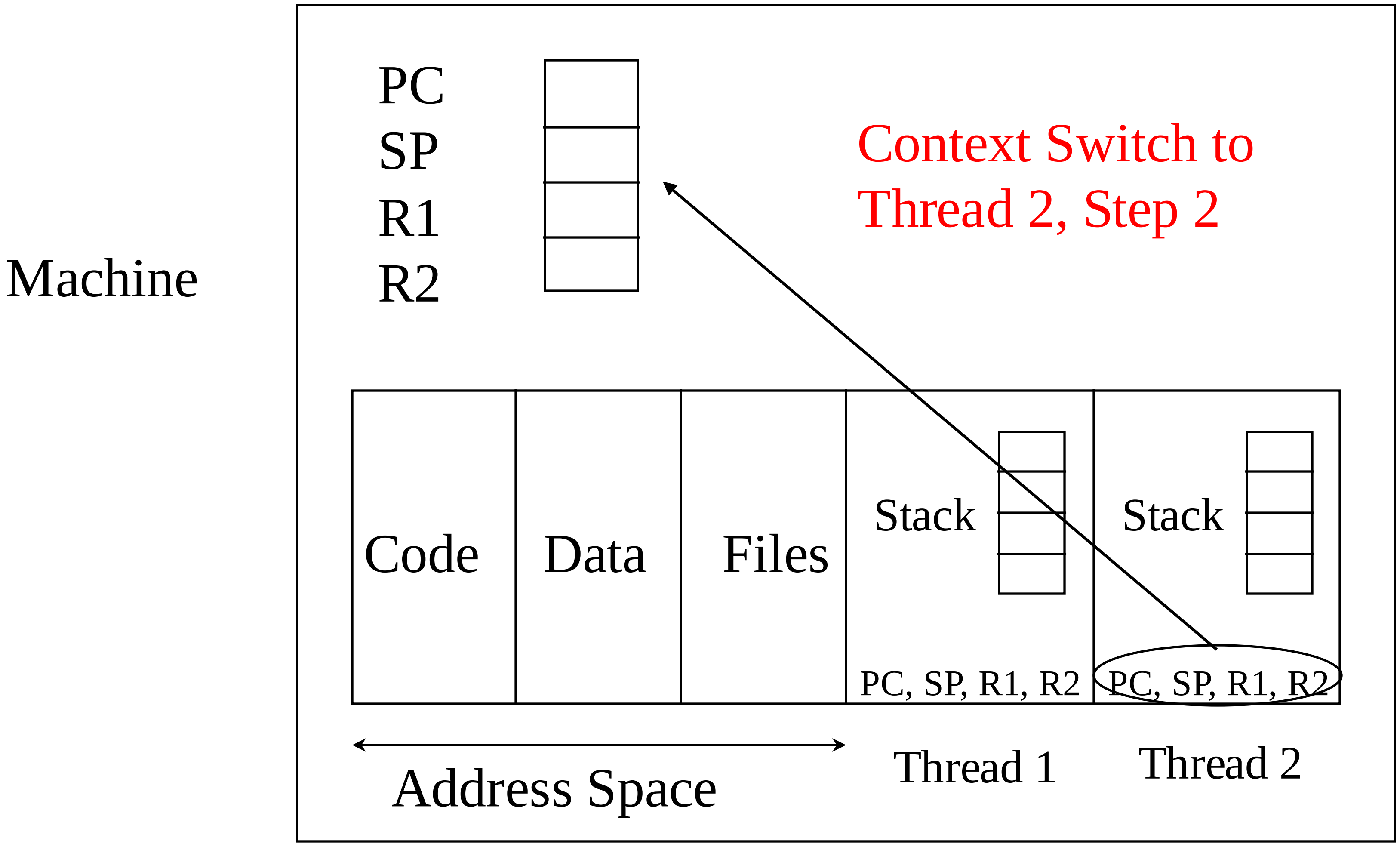
Multiple Threads, One Machine (Single Core)



Multiple Threads, One Machine (Single Core)



Multiple Threads, One Machine (Single Core)



Why Save General Purpose Registers?

(Suppose x == y == 0 initially)

- code for Thread 1

  foo( )

  x := x+1

  x := x*2

  Assembly code:

  R1 := R1 + 1 /* !! */

  R1 := R1 * 2

  Suppose context switch occurs after line “!!”

- code for Thread 2

  bar( )

  y := y+2

  y := y-3

  Assembly code:

  R1 := R1 + 2

  R1 := R1 - 3

Why Save General Purpose Registers?

(Suppose x == y == 0 initially)

- code for Thread 1

  foo( )

  x := x+1

  x := x*2

- code for Thread 2

  bar( )

  y := y+2

  y := y-3

Register allocation is
outside of our control

Assembly code:

R1 := R1 + 1 /* !! */

R1 := R1 * 2

Assembly code:

R1 := R1 + 2

R1 := R1 - 3

Suppose context switch
occurs after line “!!”

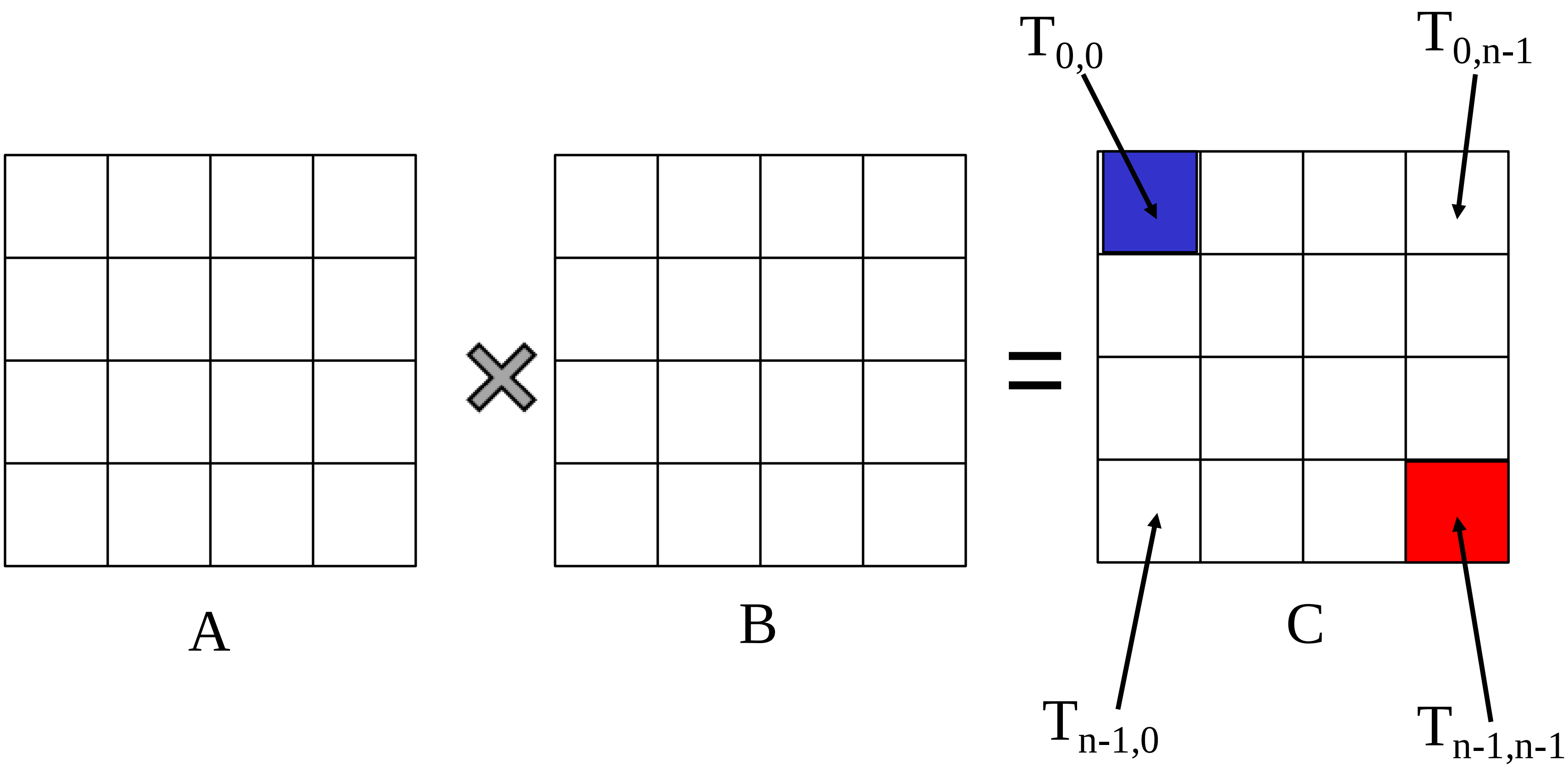
Example: Basic Threads (Code available on website)

Matrix Multiplication, n^2 threads

```
double A[n][n], B[n][n], C[n][n]
co i = 0 to n-1  {
  co j = 0 to n-1  {
    double sum = 0.0
    for k = 0 to n-1
      sum += A[i][k] * B[k][j]
    C[i][j] = sum
  }
}
```

We already argued the two outer "for" loops were parallelizable

Picture of Matmult, n^2 threads



Steps to parallelization

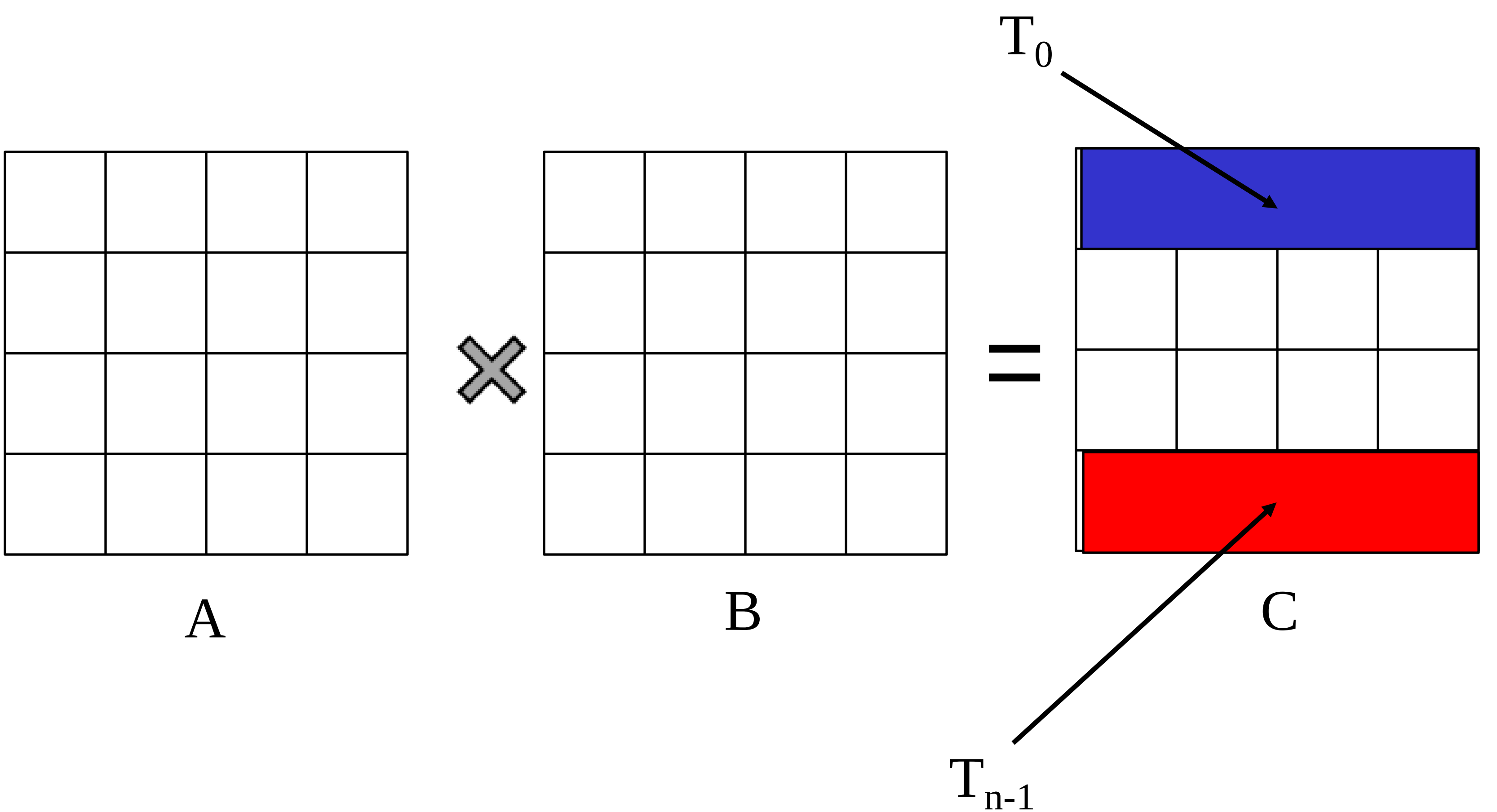
- Second: control the *granularity* (amount of work done per thread)
  - Must trade off advantages/disadvantages of fine granularity
    - Advantages: better load balancing, better scalability
    - Disadvantages: process/thread overhead and more dependencies between threads
  - Combine small threads into larger ones to coarsen granularity
    - Try to keep the load balanced

Matrix Multiplication, n threads

```
double A[n][n], B[n][n], C[n][n]
co i = 0 to n-1  {
  for j = 0 to n-1  {
    double sum = 0.0
    for k = 0 to n-1
      sum += A[i][k] * B[k][j]
    C[i][j] = sum
  }
}
```

This is plenty of parallelization
if the number of cores is <= n

Picture of Matmult, n threads

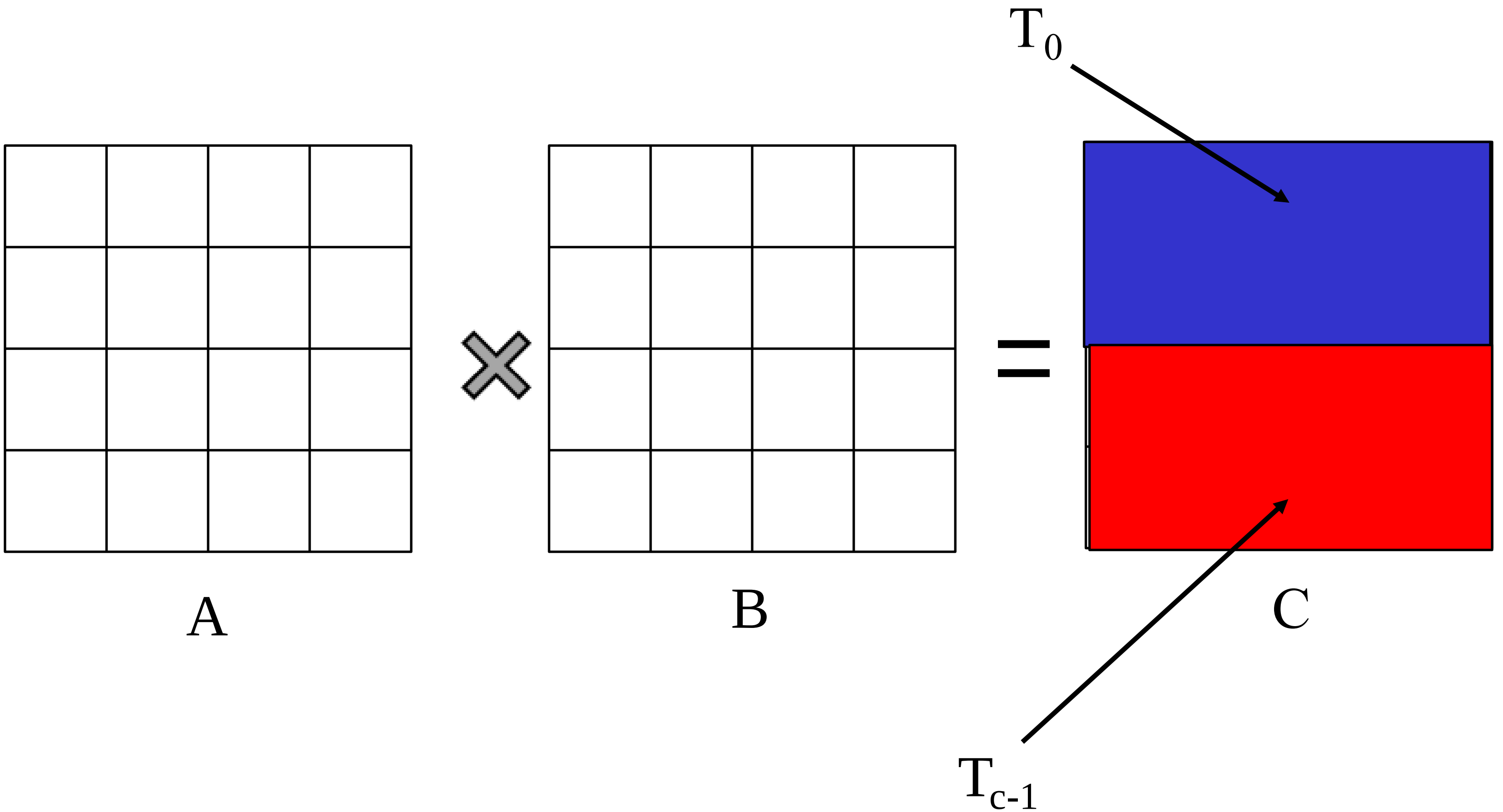


Matrix Multiplication, c threads

```
double A[n][n], B[n][n], C[n][n]
co i = 0 to c-1  {
  startrow = i * n / c; endrow = (i+1) * n / c - 1
  for r = startrow to endrow
    for j = 0 to n-1  {
      double sum = 0.0
      for k = 0 to n-1
        sum += A[r][k] * B[k][j]
      C[r][j] = sum
    }
}
```

Assuming c is the number of available cores, this works well…but why?

Picture of Matmult, c threads In this example, c == 2



Note the last thread is subscripted by c, not n

Example: Matrix Multiplication Using Threads (Code available on website)

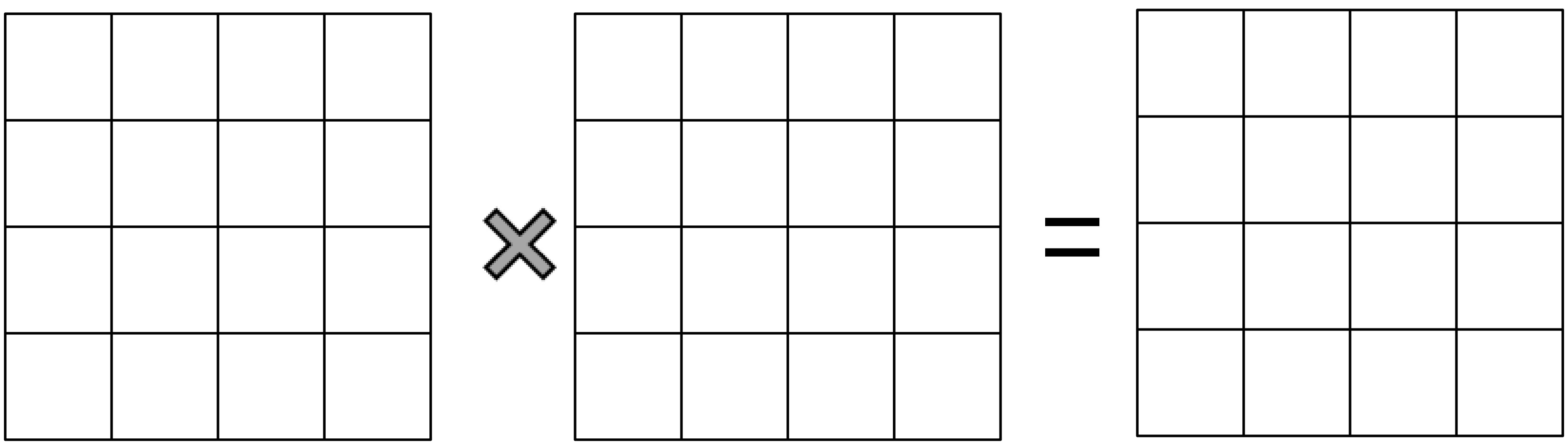
Steps to parallelization

- Third: distribute computation and data
  - Assign which processor does which computation
    - **The co statement does *not* do this**
  - If memory is distributed, decide which processor stores which data (why needed?)
    - One can also choose to replicate data
  - Goals: minimize communication and balance the computational workload
    - Often conflicting

Steps to parallelization

- Fourth: synchronize and/or communicate
  - If shared-memory machine, synchronize
    - Both mutual exclusion and sequence control
      - Locks, semaphores, condition variables, barriers, reductions (topics that will consume several weeks)
  - If distributed-memory machine, communicate
    - Message passing
    - Typically, communication involves implicit synchronization

Parallel Matrix Multiplication---Distributed-Memory Version

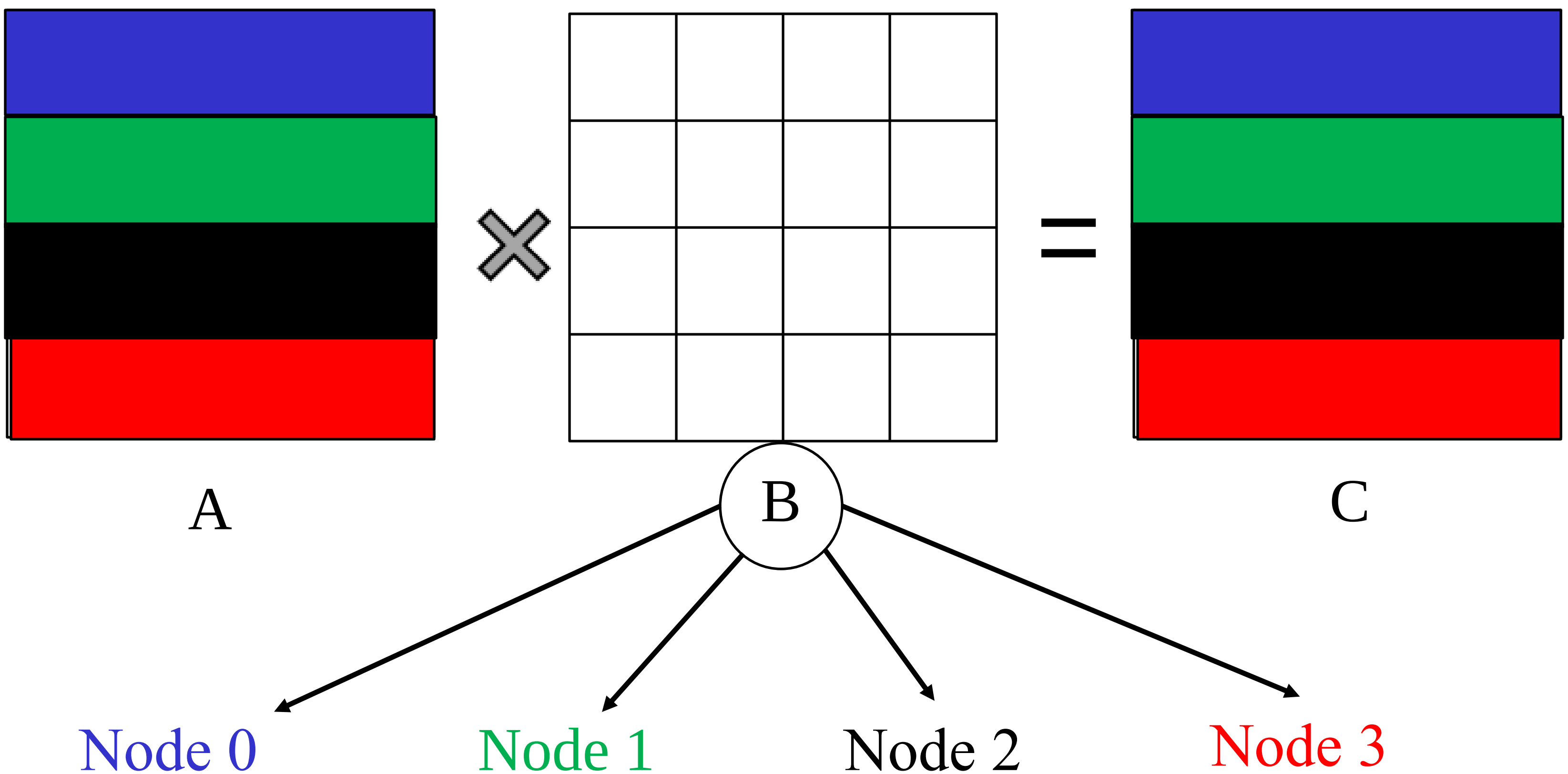


A B C

Node 0 Node 1 Node 2 Node 3

How do we distribute A, B, and C?

Parallel Matrix Multiplication---Distributed-Memory Version



Array B must be replicated (distributed in its entirety to all nodes)

Parallel Matrix Multiplication---Distributed-Memory Version

```
process worker [i = 0 to p-1] {
 double A[n][n], B[n][n], C[n][n]  // wasting space!
 startrow = i * n / p; endrow = (i+1) * n / p – 1
 if (i == 0)  {
    for j = 1 to p-1 {
       sr = j * n / p; er = (j+1) * n/p – 1
       send A[sr:er][0:n-1], B[0:n-1][0:n-1] to process j
    }
 else
    receive A[startrow:endrow][0:n-1], B[0:n-1][0:n-1] from 0
```

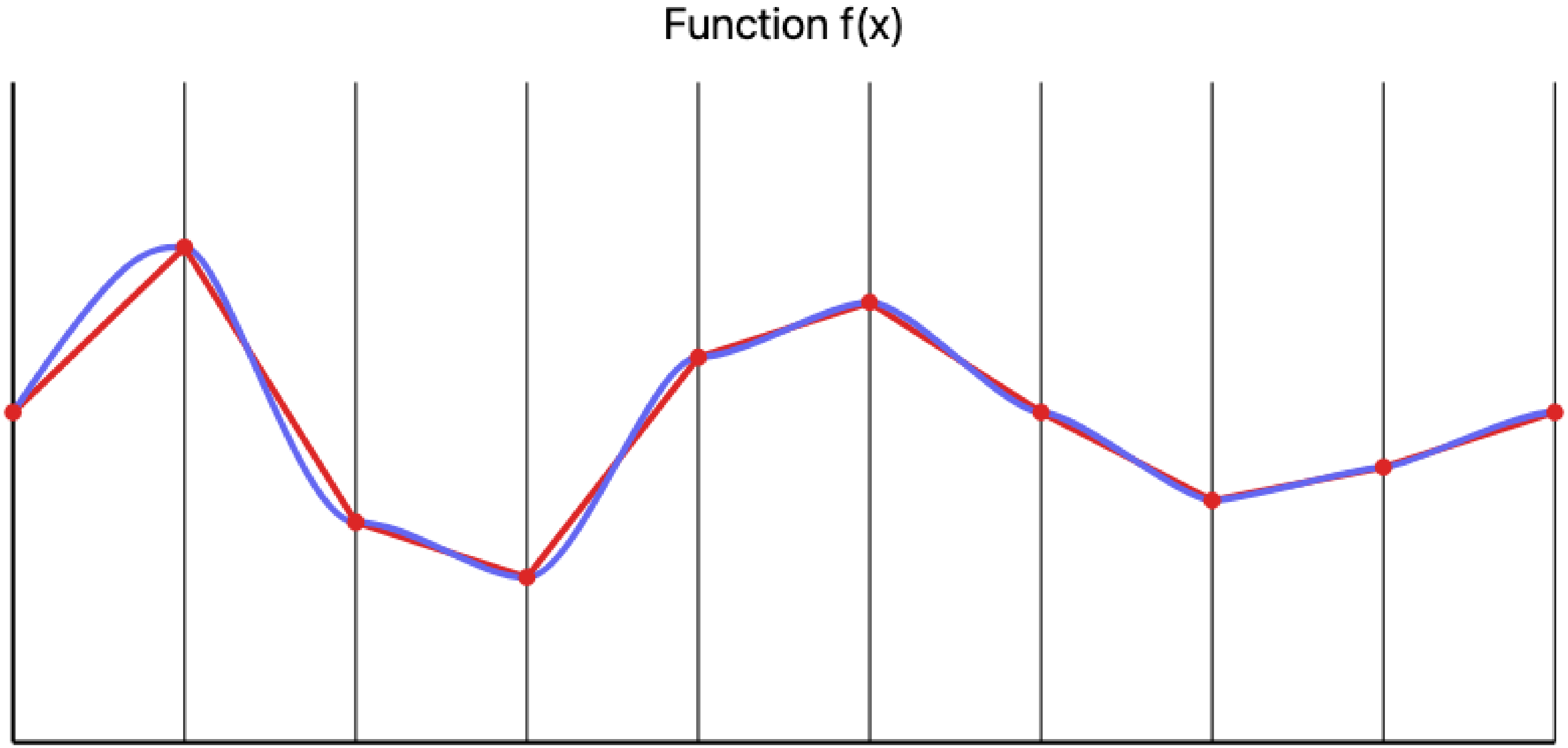
Parallel Matrix Multiplication---Distributed-Memory Version

```
for i = startrow to endrow
  for j = 0 to n-1 {
    double sum = 0.0
    for k = 0 to n-1
      sum += A[i][k] * B[k][j]
    C[i][j] = sum
  }
  // here, need to send my piece back to administrator
  // how do we do this?
} // end of process statement
```

Steps to parallelization (summary so far)

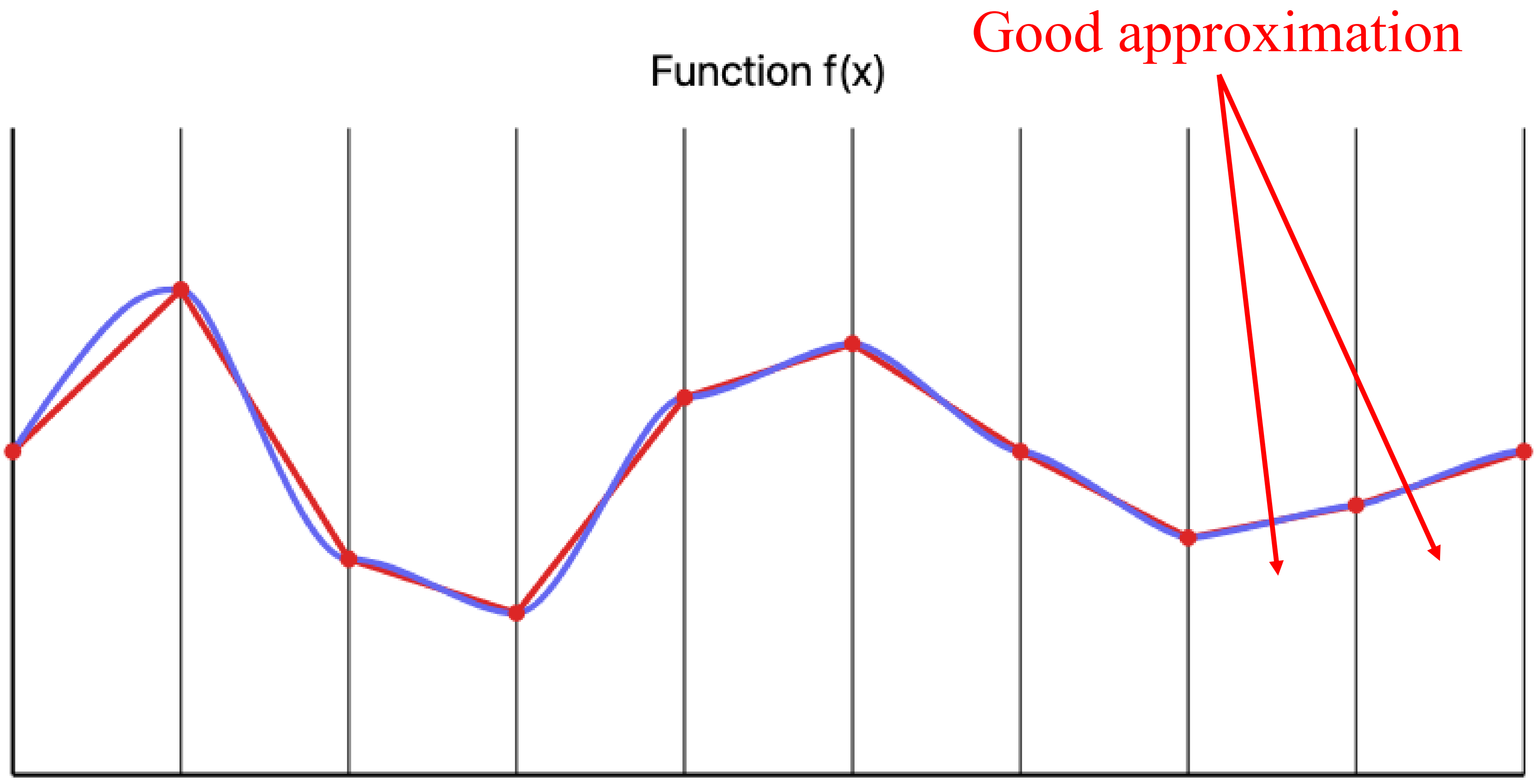
- First: find parallelism
- Second: control (potentially coarsen) granularity
- Third: distribute computation and data
- Fourth: synchronize and/or communicate

Numerical Integration



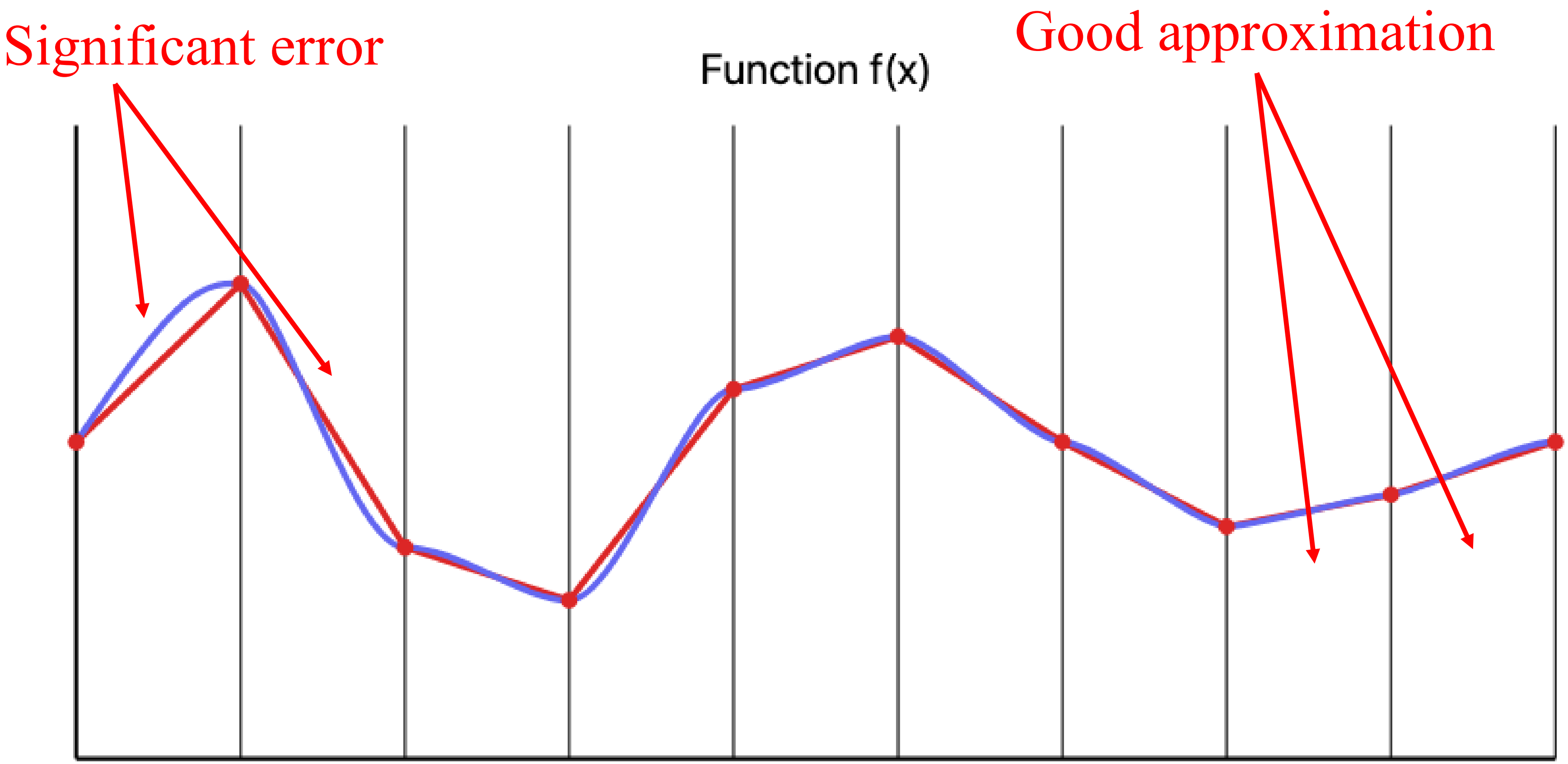
Area under the curve ≅ the sum of the areas of the black/red trapezoids

Numerical Integration



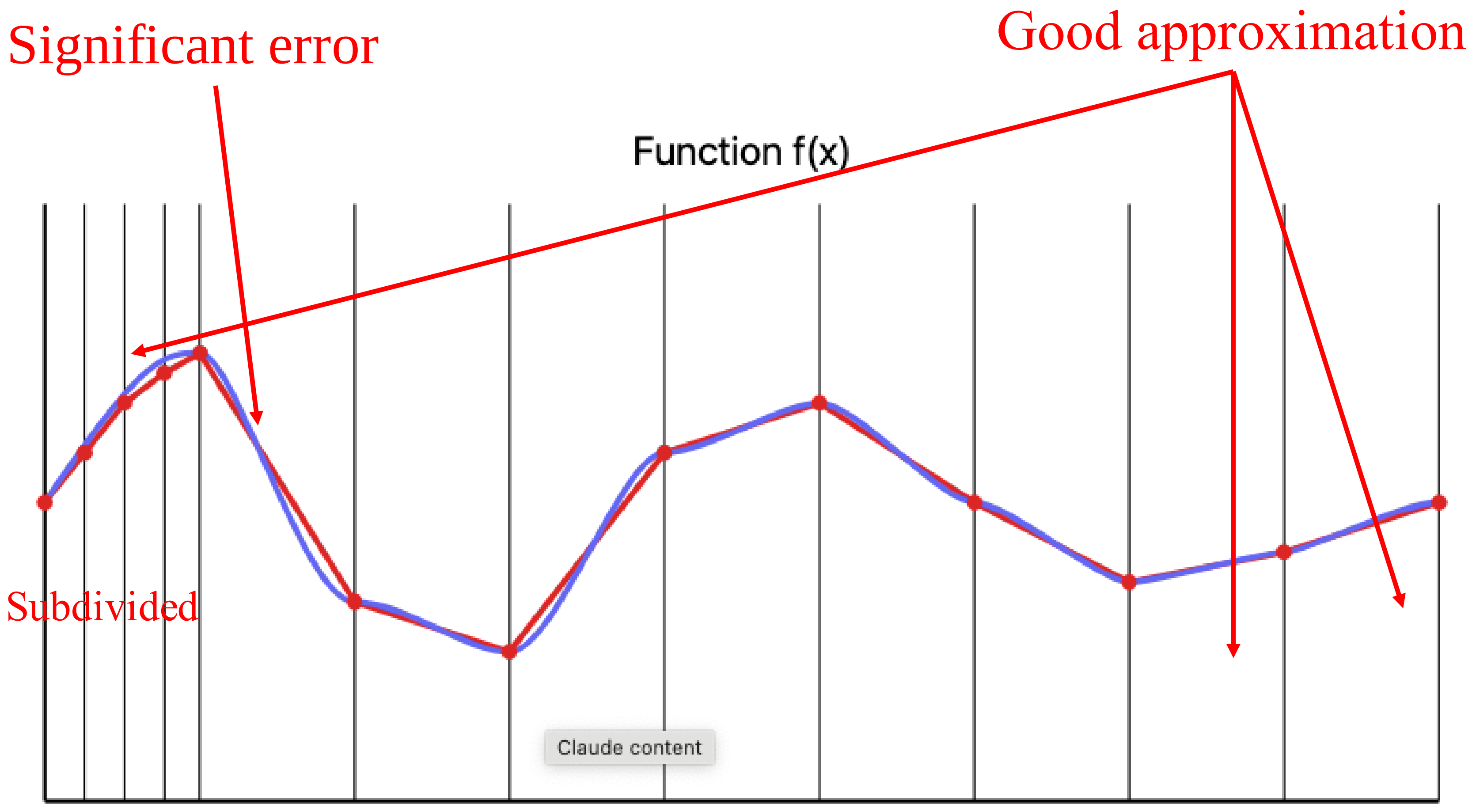
Area under the curve ≅ the sum of the areas of the black/red trapezoids

Numerical Integration



Area under the curve ≅ the sum of the areas of the black/red trapezoids

Numerical Integration



Area under the curve ≅ the sum of the areas of the black/red trapezoids

Adaptive Quadrature: Sequential (Recursive) Program

```
double f( ) {…}  // some arbitrary function
double area(a, b) {
  double c
  c := (a+b)/2
  compute area of each half and area of whole
  if (close)
    return area of whole
  else
    return area(a,c) + area(c,b)
}
```

Adaptive Quadrature: Parallel (Recursive) Program

```
double f( ) {…}  // some arbitrary function
double area(a, b) {
 double c, leftArea, rightArea
 c := (a+b)/2
 compute area of each half and area of whole
 if (close)
   return area of whole
 else {
   co leftArea = area(a,c)  // rightArea = area(c,b) oc
   return leftArea + rightArea
 }
}
```

Challenge with Adaptive Quadrature

- For efficiency, must control granularity (step 2)
  - Without such control, granularity will likely be too fine
  - Can stop thread creation after "enough" threads created
    - Hard in general, as do not want cores idle either
  - Thread implementation can perform work stealing
    - Idle cores take a thread and execute that thread, but care must be taken to avoid synchronization problems and/or efficiency problems

Steps to parallelization

- Fifth: assign processors to tasks (only if using task and data parallelism)
  - Must also know dependencies between tasks
  - Task parallelism is typically used if limits of data parallelism are reached

This slide is for completeness; we will not study this in CSc 422

Steps to parallelization

- Sixth: parallelism-specific optimizations
  - Examples: message aggregation, overlapping communication with computation
    - Most of these refer to message-passing programs (targeting distributed-memory multicomputers)

Steps to parallelization

- Seventh: acceleration
  - Find parts of code that can run on GPU/FPGA/Cell/etc., and optimize those parts
  - Difficult and time consuming
    - But may be quite worth it

This slide is also for completeness; we will (probably) not study this in CSc 422

Pipelines

- Example:
  - (abstract) lec:> a | b | c | …
  - (concrete) lec:> ps | grep dkl
- Producer/Consumer paradigm
  - In example above, the thread executing “ps” is the producer, and the thread executing “grep” is the consumer
  - Implemented by a bounded buffer (will study this in a couple of weeks)

Sequential Grep

```
void grep (file f, pattern pat) {
  string line
  while ((line = read(f)) != EOF) {
    found = search (line, pat)
    if (found)
       print line
  }
}
```

Assume we have two cores

Apply our Steps

- Find parallelism
  - Can read next line while searching current line
- Coarsen granularity: put off for now
- Distribute computation (we are assuming shared memory)
  - One thread reads, another thread searches
- Synchronize
  - co/while vs. while/co
- Optimizations: not relevant for this program

Concurrent Grep, First Attempt

```
string line[2]; int next = 0
void readNext( ) { return ((line[next] = read (f)) != EOF)) }
void grep (file f, pattern pat) {
  int retval = readNext( ); next = 1
  while (retval != 0) {
    co
        found = search (line[1-next], pat);
        if (found) print line
    //
        retval = readNext( )
    oc
    next = 1 - next
  }
}
```

Notes on Concurrent Grep, First Attempt

- Style:
  - “co inside while”
- Problem:
  - Thread creation and synchronization on each iteration of while loop
    - Overhead leads to slowdown, not speedup

Concurrent Grep, Better Version

- Style:
  - "while inside co"
  - Co is invoked once
    - One arm of co is the search, the other is the read
    - Turns into producer/consumer paradigm, so similar to pcBusyWait.c example already online (and textbook has details)