AMATH 483/583
High Performance Scientific Computing

Lecture 10:
Processes, Threads, Concurrency, Parallelism

Xu Tony Liu, PhD

Paul G. Allen School of Computer Science & Engineering

University of Washington

Seattle, WA

Announcements

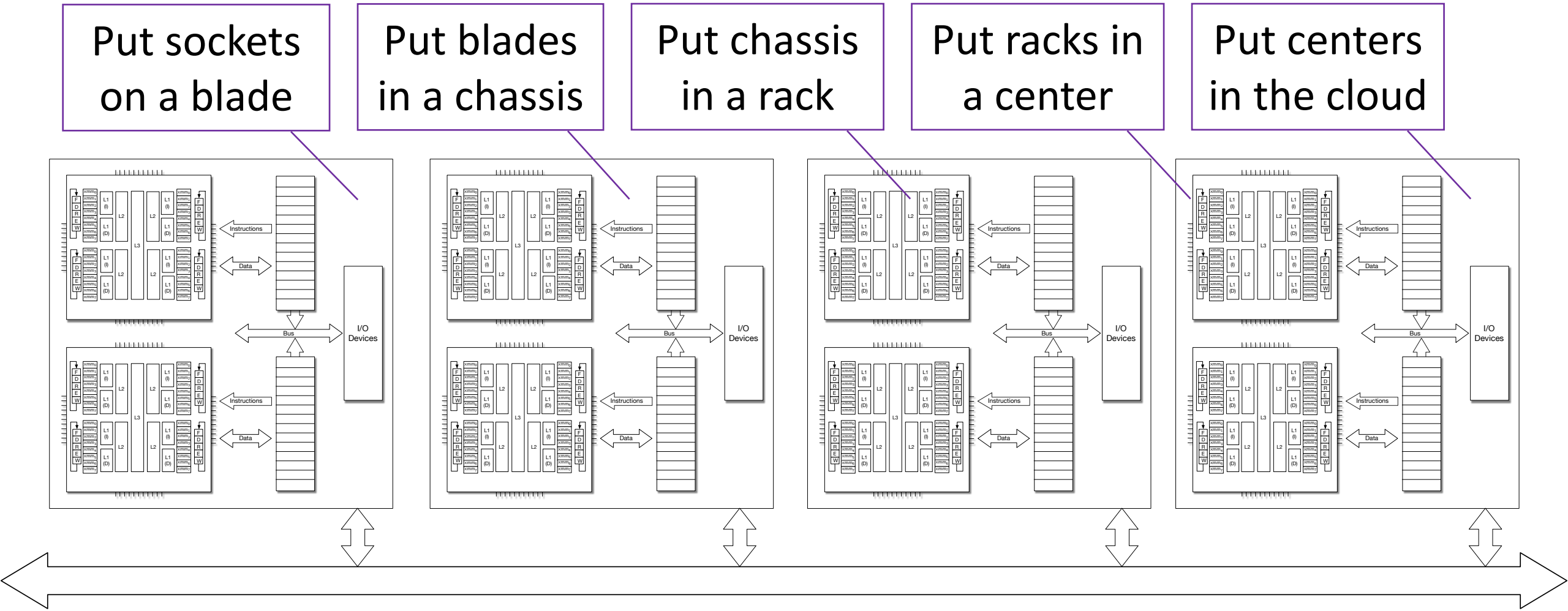
- Mid Term out this noon 04/28/2022 due 11:59AM 05/05/2022
- **The exam may not be discussed with anyone except course instructors**
- You may contact the instructors via *private* messages on Piazza to clarify questions

Overview

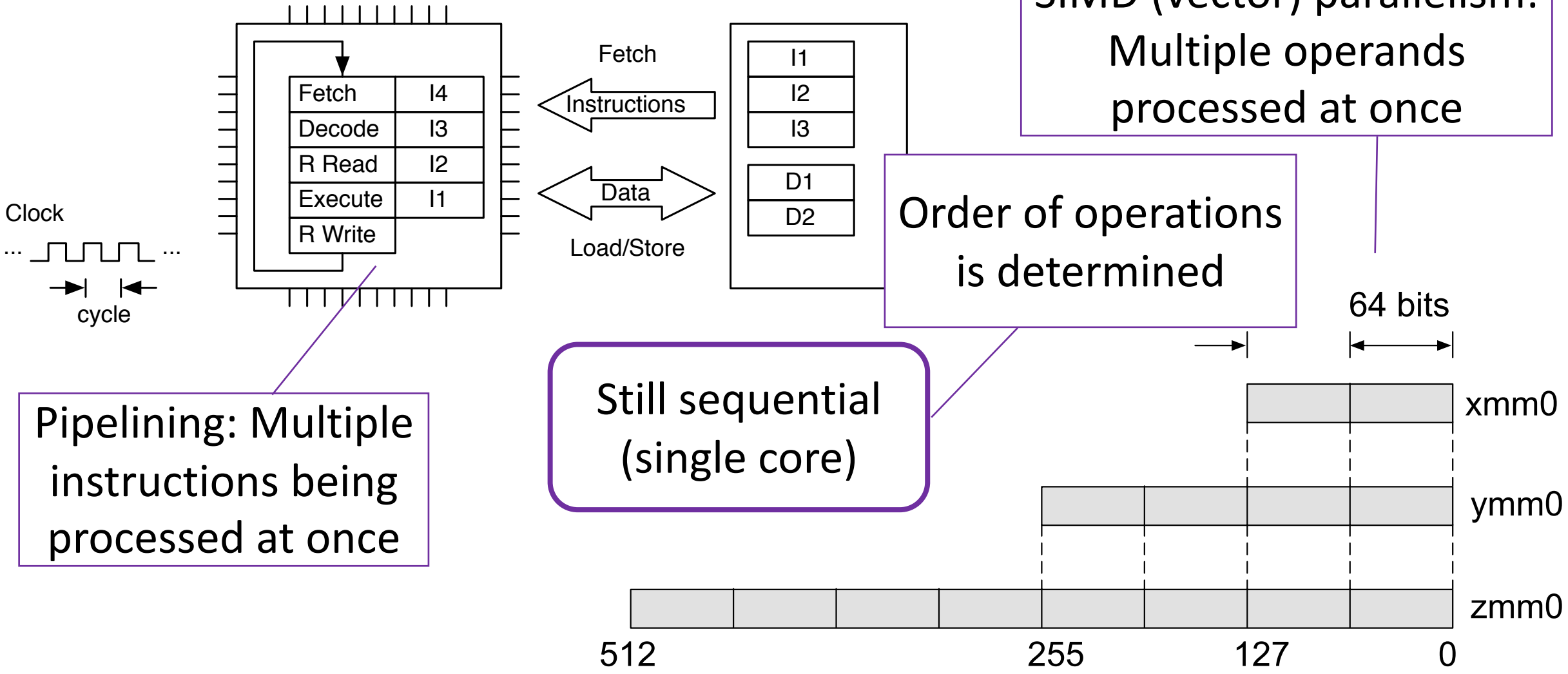
- Review
  - SISD, SIMD
- Multiple cores (MIMD)
- Concurrency
- Processes
- Parallelism
- Threads
  - Multithreading

Supercomputers (HPC)

Schematically



Parallelism and HPC so far



SIMD (vector) parallelism: Multiple operands processed at once

Order of operations is determined

Pipelining: Multiple instructions being processed at once

Still sequential (single core)

General Performance Principles

- Work harder
  - Faster core
- Work smarter
  - Branch predictions, etc.
  - Better compilation
  - Better algorithm
  - Better implementation
- More workers
  - More cores

Dennard scaling (ended 2005)

Higher optimization level, e.g., -O2, -O3, -Ofast, etc.

e.g., Strassen's algorithm

We did this

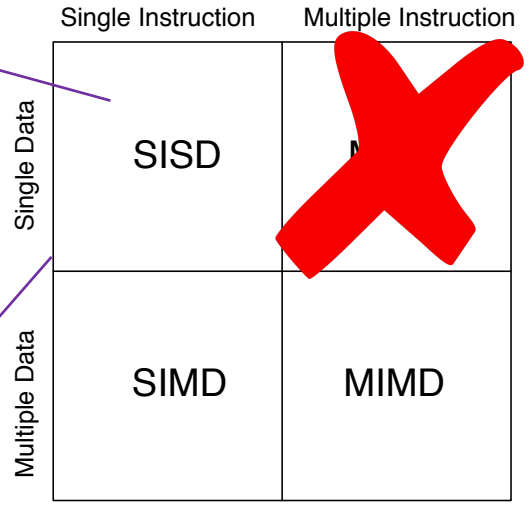
Parallel Computing

Flynn's Taxonomy (Aside)

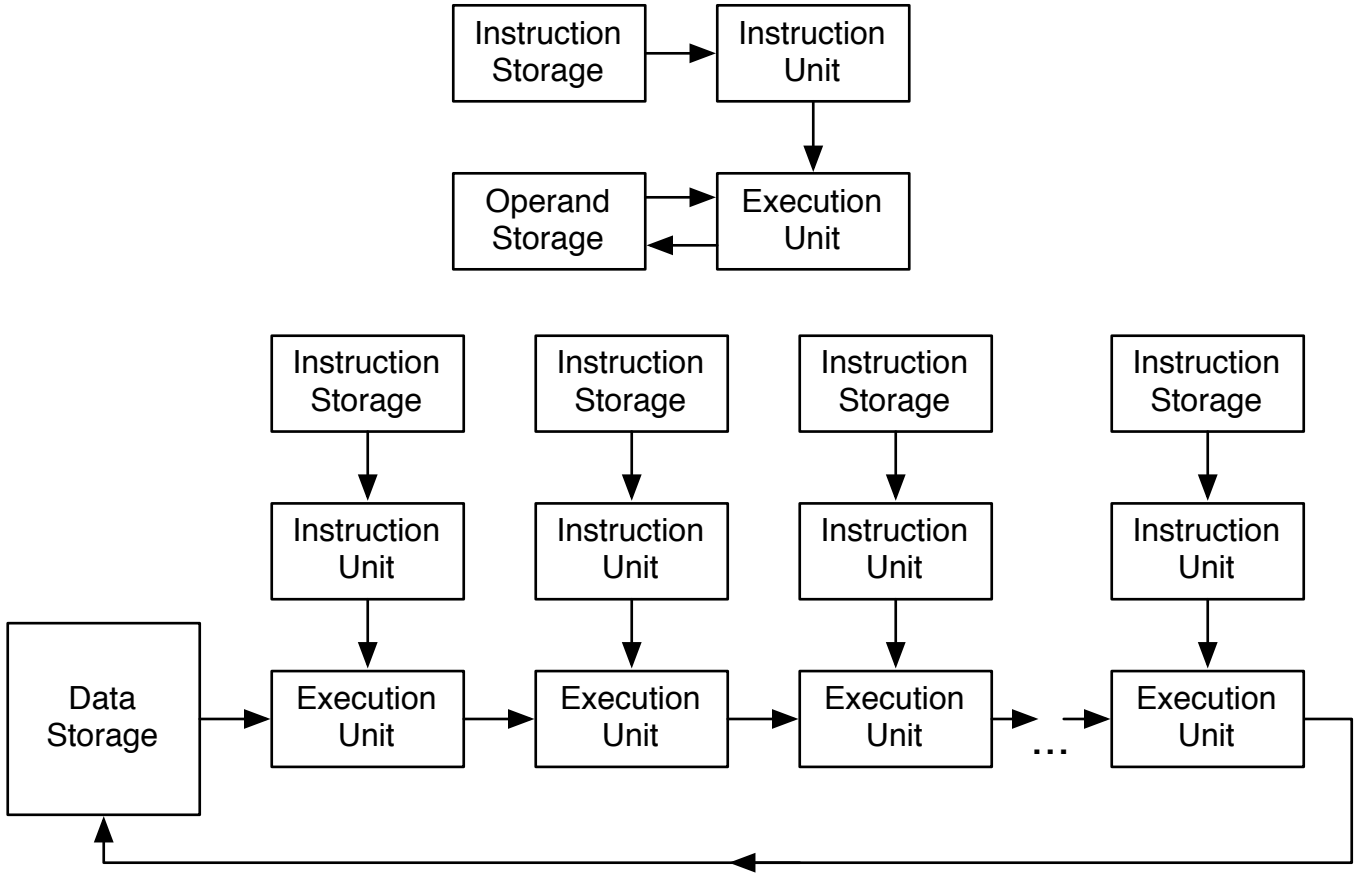
Anyone in HPC must know Flynn's taxonomy

- **Classic** classification of parallel architectures (Michael Flynn, 1966)

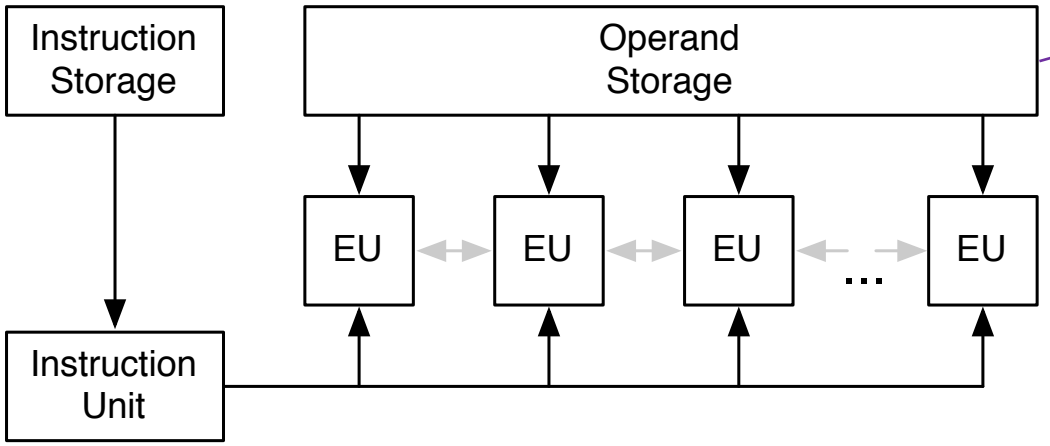
Plain old sequential



Based on multiplicity of instruction streams, data storage

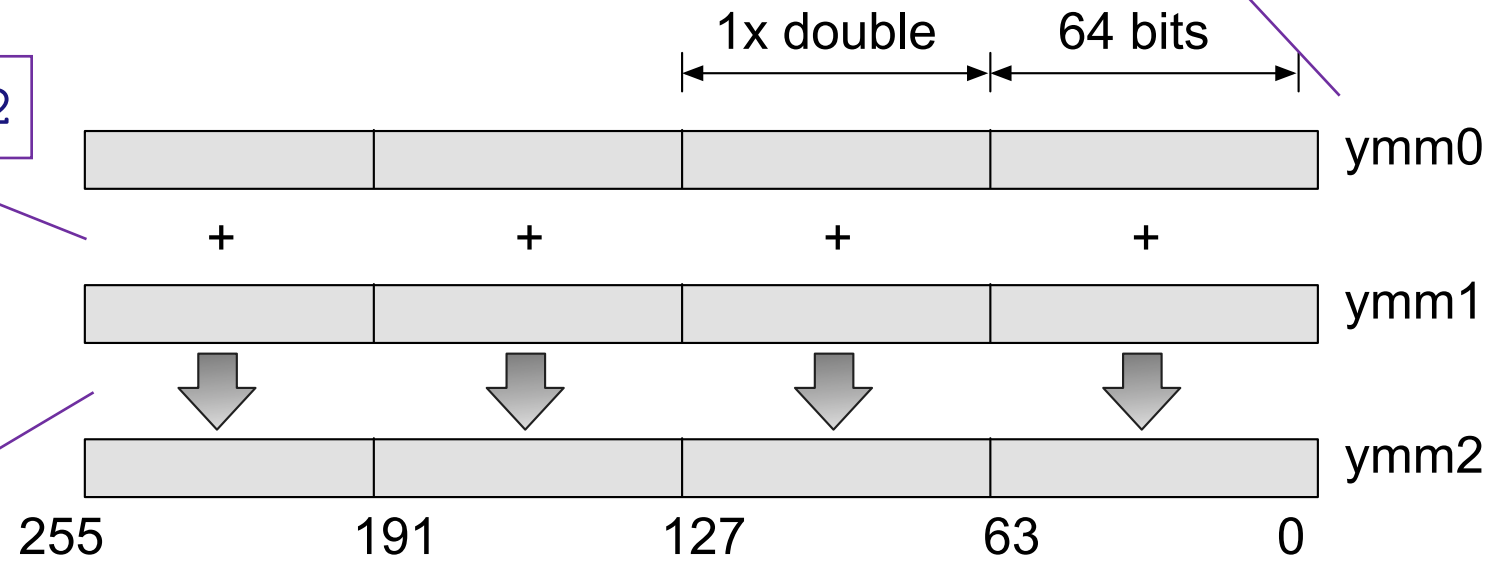


SIMD in SSE/AVX



Flynn's original conceptual model

ymm are 256 bit registers




```
vfadd231pd  %ymm0, %ymm1, %ymm2
```

One machine instruction

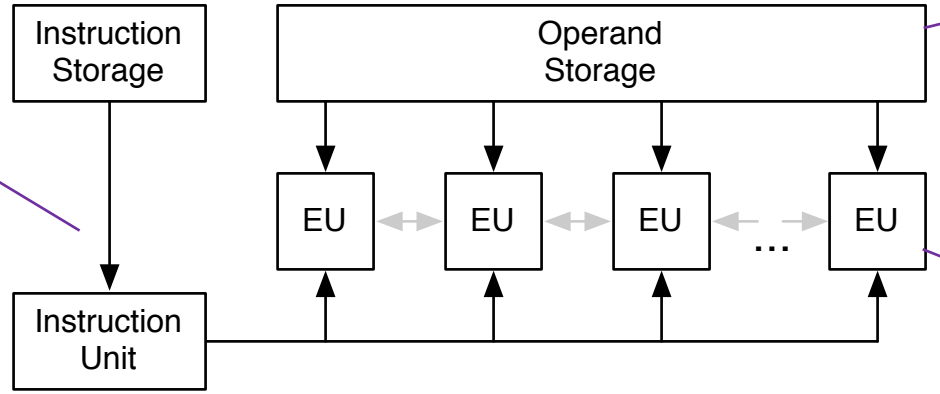
Adds all four doubles ***simultaneously***

SIMD and MIMD

- Two principal parallel computing paradigms (multiple op

Single instruction at a time

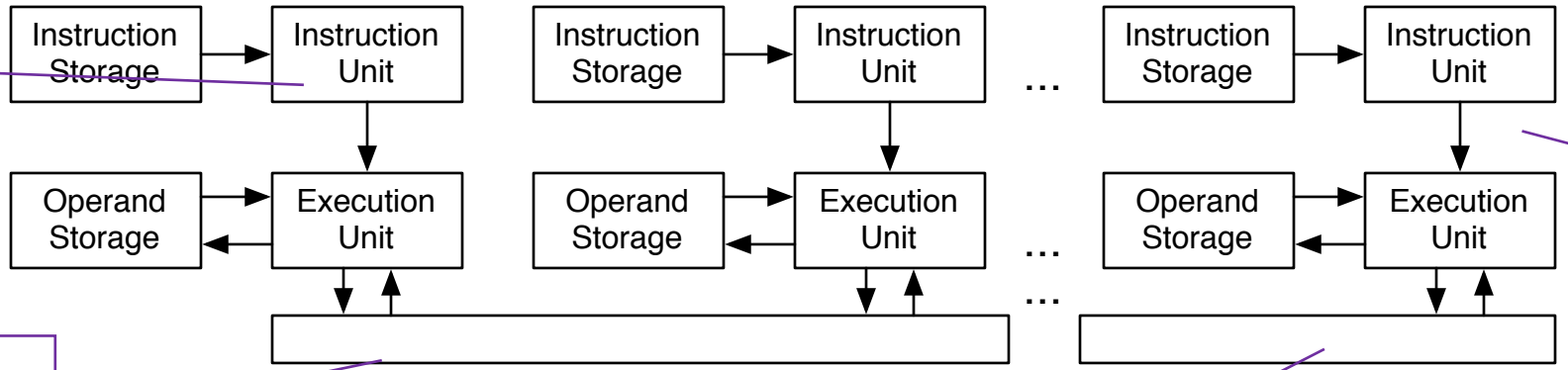
But each have their own data



Multiple instructions

All execution units execute in (c)lock step

EUs run independently (w own instrs)



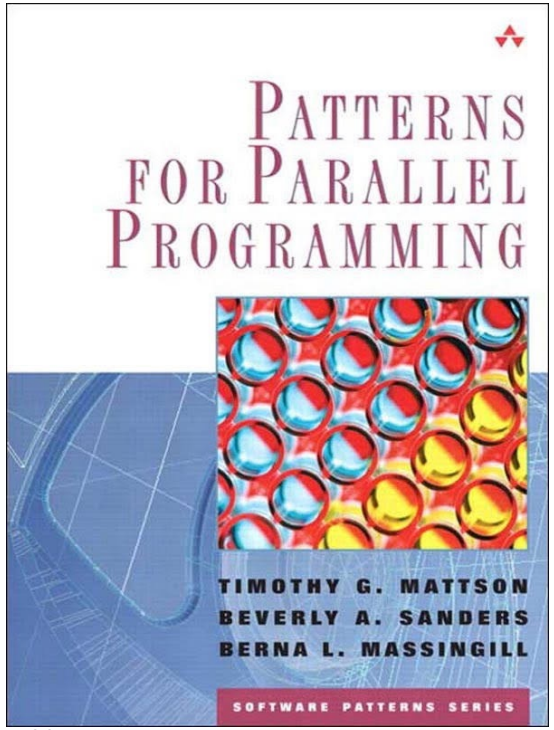
Coming up next

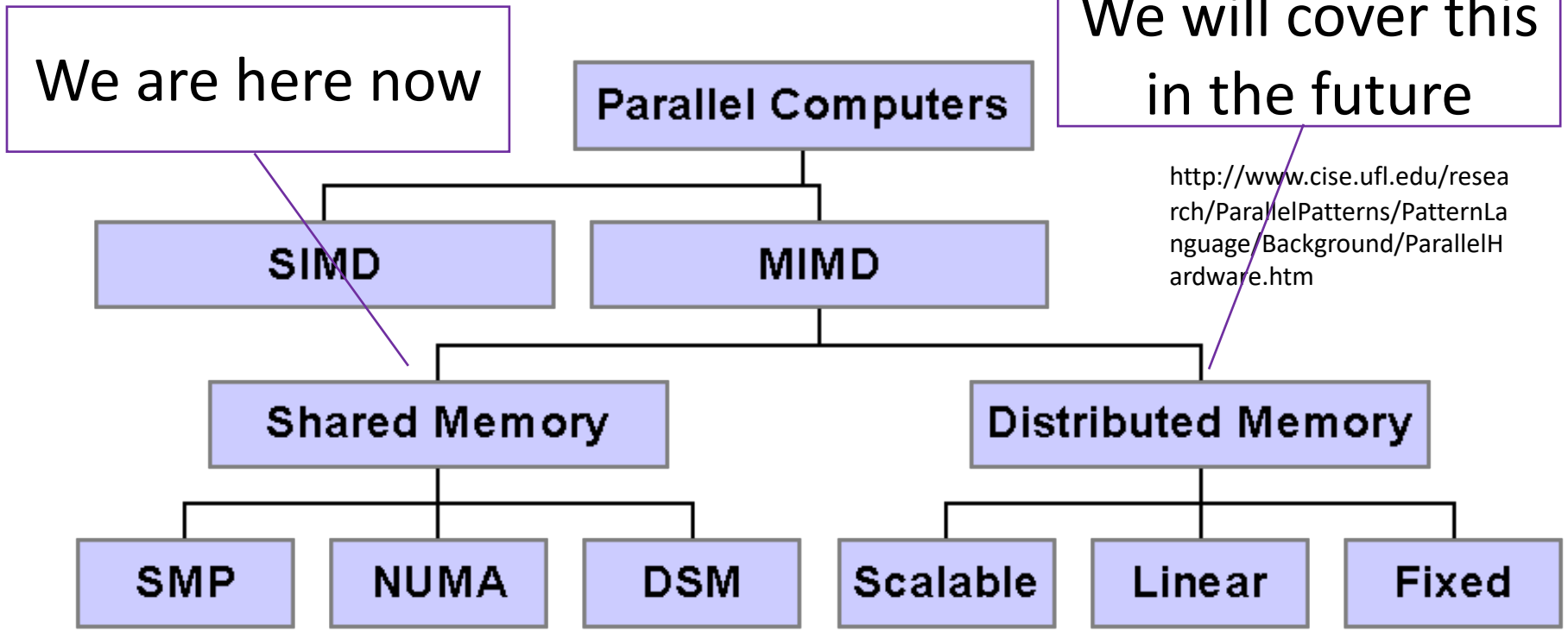
Shared Memory

Not Shared

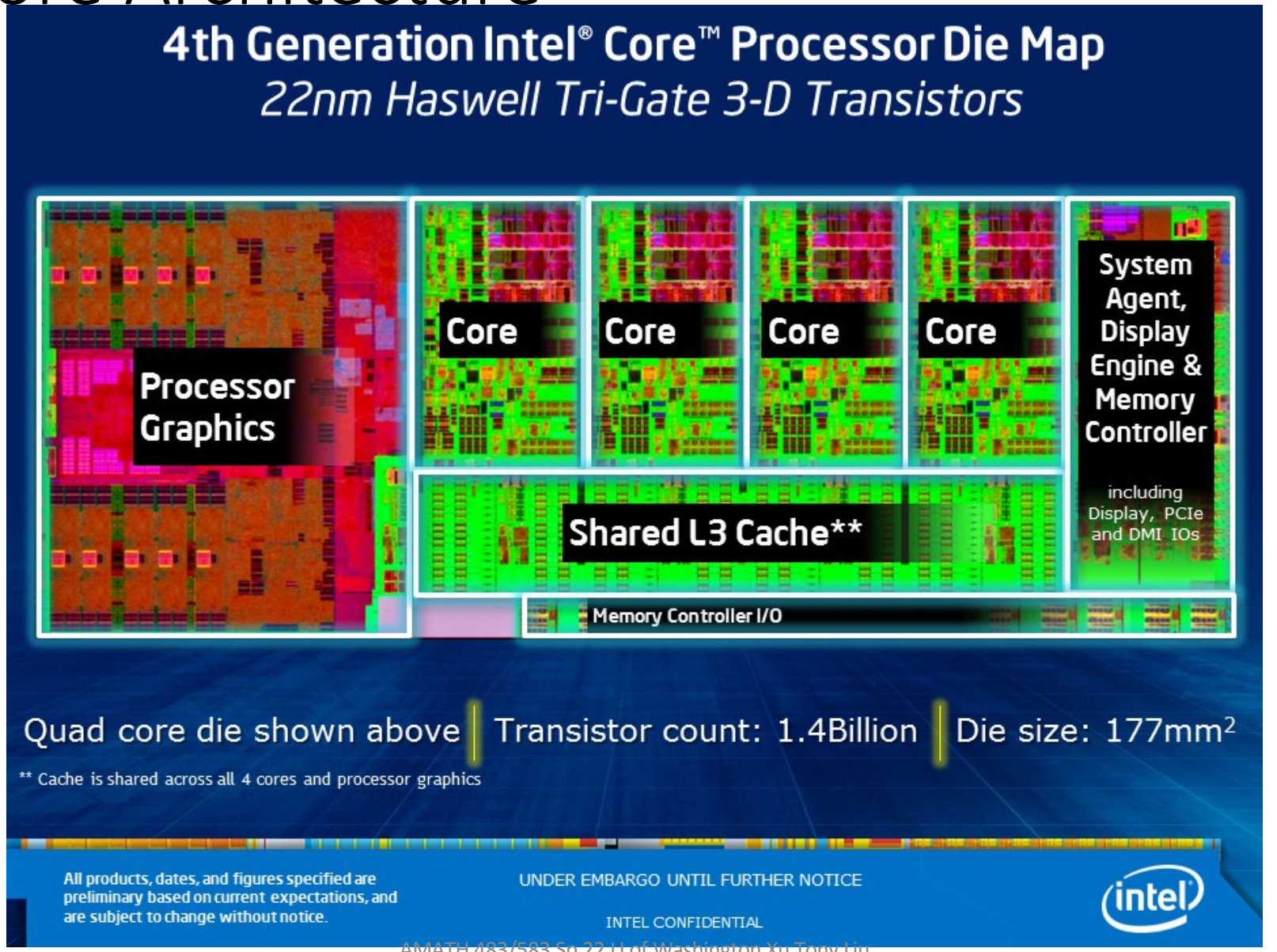
A More Refined (Programmer-Oriented) Taxonomy

- Three major modes: SIMD, Shared Memory, Distributed Memory
- Different programming approaches are generally associated with different modes of parallelism (threads for shared, MPI for distributed)
- A modern supercomputer will have all three major modes present





Multicore Architecture



Multicore for HPC

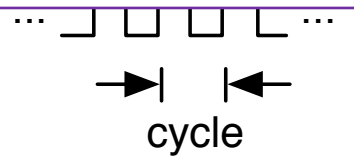
- How do multicore chips operate (how does the hardware work)?
- How do they get high performance?
- How does the software exploit the hardware (how do we write our software to exploit the hardware)?
- What are the abstractions that we need to use to reason about multicore systems?
- What are the programming abstractions and mechanisms?
- Terminology: Program, process, thread
- More terminology: Parallel, concurrent, asynchronous

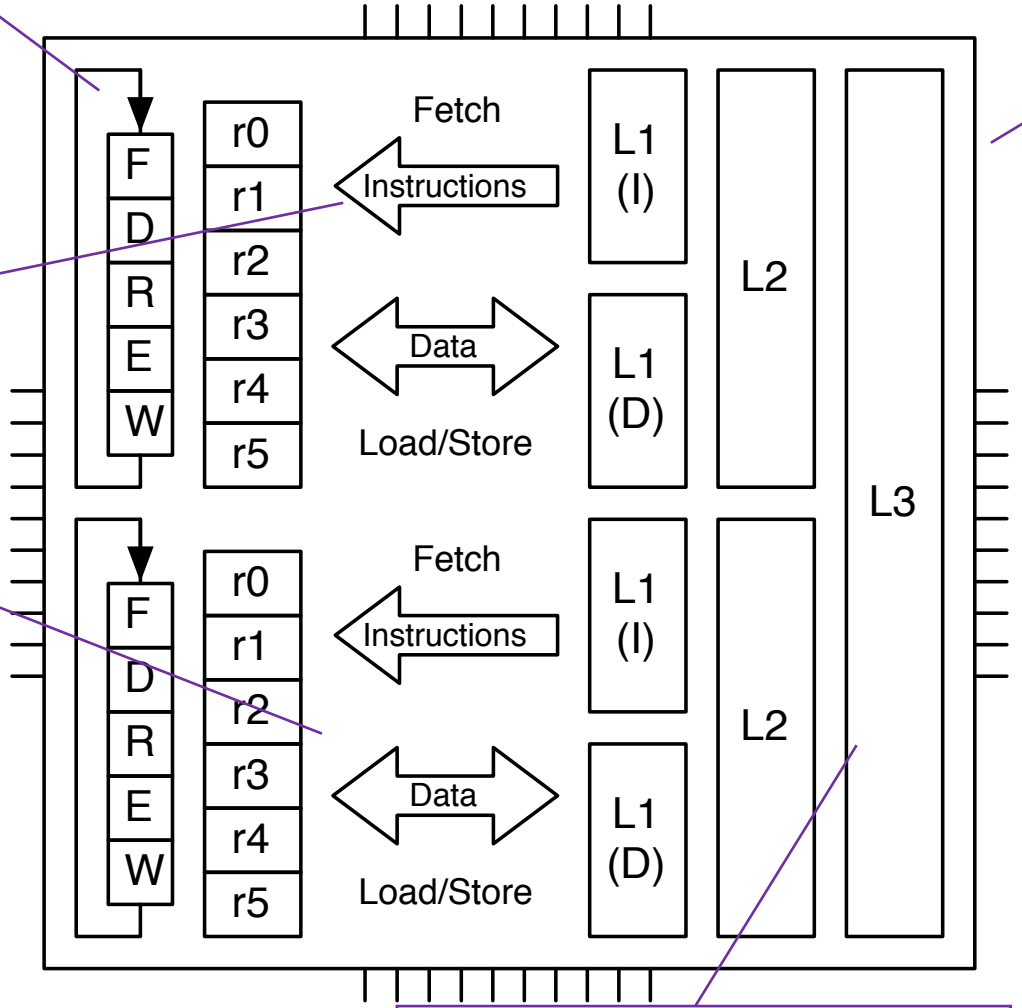
Multicore Architecture

Core is a FDREW + regs

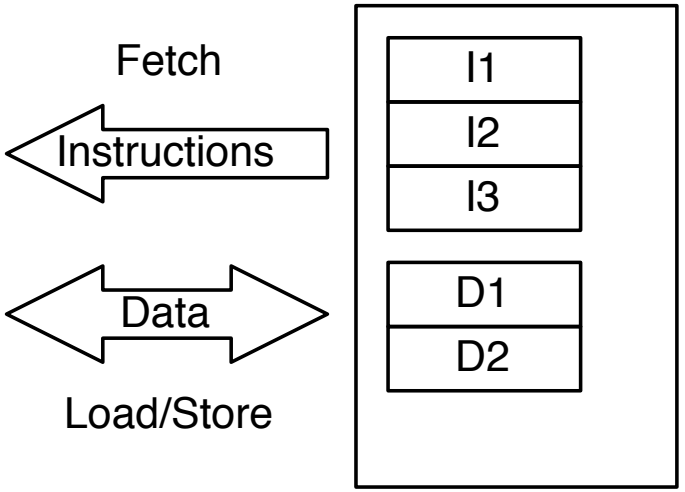
Each runs its own sequence of instructions

Each can access its own data





Any CPU in the last 4-5 years



But memory might be shared

Each has memory hierarchy

Sequential Example

- You are the TA for AMATH 483 and must grade 22 exams
- The exam has 8 questions on it
- It takes 3 minutes to grade one question

- How long will it take you to grade all the exams?

Parallelization Example

- You are the TA for AMATH 483 and must grade 22 exams
- The exam has 8 questions on it
- It takes 3 minutes to grade one question
- You ask 21 friends who agree to help you

- How long will it take the 22 of you to grade all the exams?

- Describe your approach
- List your assumptions

Parallelization Example

- You are the TA for AMATH 483 and must grade 1012 exams (1012 = 46 * 22)
- The exam has 8 questions on it
- It takes 3 minutes to grade one question
- You ask 21 friends who agree to help you

- How long will it take the 22 of you to grade all the exams?

- Describe your approach
- Describe another approach
- List your assumptions

Another Parallelization Example

- You are the TA for AMATH 483 and must grade 8 exams
- The exam has 22 questions on it
- It takes 3 minutes to grade one question
- You ask 21 friends who agree to help you

- How long will it take the 22 of you to grade all the exams?

- Describe your approach

Parallelization Example

- You are the TA for AMATH 483 and must grade 368 exams (368 = 46 * 8)
- The exam has 22 questions on it
- It takes 3 minutes to grade one question
- You ask 21 friends who agree to help you

- How long will it take the 22 of you to grade all the exams?

- What if you had 368 friends? 368*22?

Compare And Contrast – Concurrency vs Parallelism

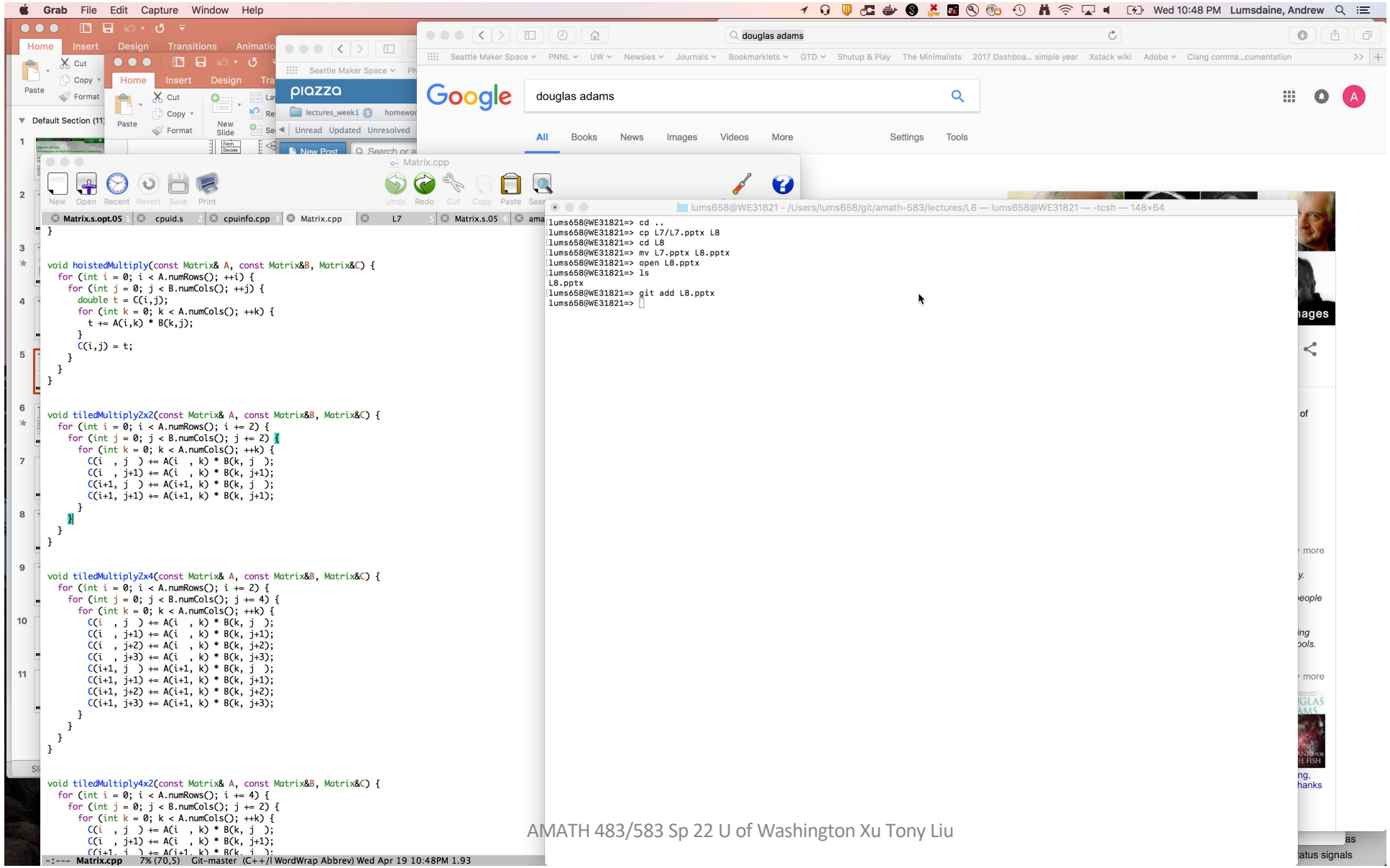
- Time for everyone grades one exam: 8*3 minutes
- Time for everyone grades one question: 3 minutes

- How (why) did you use the approaches you did?

- Concurrency
  - When two or more tasks can start, run, and complete in overlapping time periods
- Parallelism
  - When tasks **literally** run at the same time

We will come back on this

How Do We Run Many Programs at the Same Time?



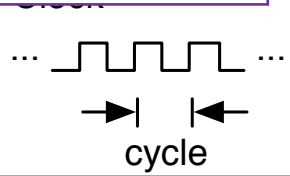
Running a Program

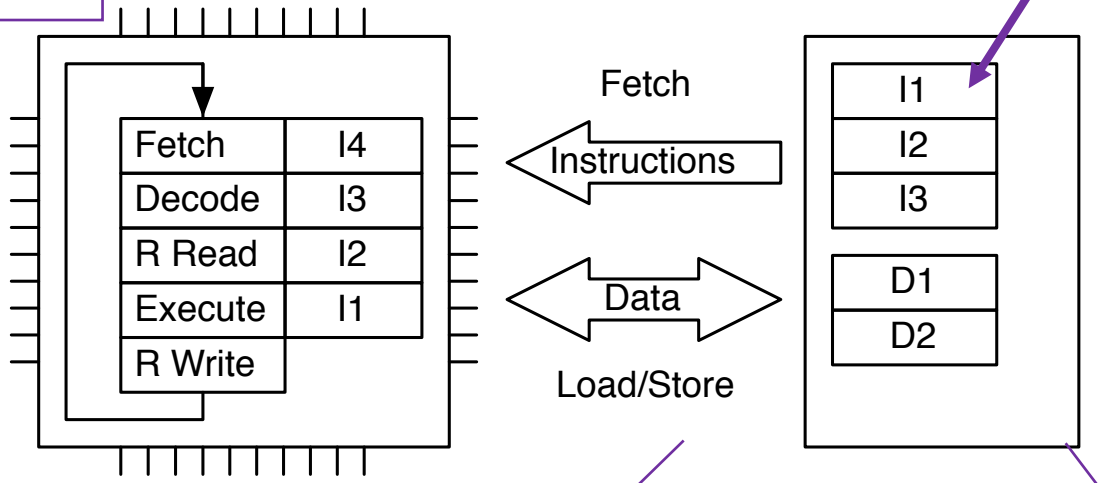
When a CPU is executing bytes from one program

Bytes from program stored in memory

```
        .globl  __Z15hoistedMultiplyRK6MatrixS1_RS_
        .p2align        4, 0x90
__Z15hoistedMultiplyRK6MatrixS1_RS_:    ## @_Z15hoistedMultiplyRK6MatrixS1_RS_
        .cfi_startproc
## BB#0:
        pushq   %rbp
Ltmp16:
        .cfi_def_cfa_offset 16
Ltmp17:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp18:
        .cfi_def_cfa_register %rbp
        pushq   %r15
        pushq   %r14
        pushq   %r13
        pushq   %r12
        pushq   %rbx
Ltmp19:
        .cfi_offset %rbx, -56
Ltmp20:
        .cfi_offset %r12, -48
Ltmp21:
        .cfi_offset %r13, -40
Ltmp22:
        .cfi_offset %r14, -32
Ltmp23:
        .cfi_offset %r15, -24
        movq    (%rdi), %rax
        movq    %rax, -120(%rbp)        ## 8-byte Spill
        testq   %rax, %rax
        je      LBB2_9
## BB#1:
        movq    8(%rsi), %rcx
        testq   %rcx, %rcx
        je      LBB2_9
## BB#2:
        movq    16(%rsi), %r12
        movq    8(%rdx), %rax
        movq    %rax, -104(%rbp)        ## 8-byte Spill
        movq    16(%rdx), %rdx
        movq    8(%rdi), %rax
        movq    16(%rdi), %r13
        leaq    -1(%rcx), %rsi
        movq    %rsi, -88(%rbp)         ## 8-byte Spill
        movl    %ecx, %esi
```

It isn't executing bytes from another



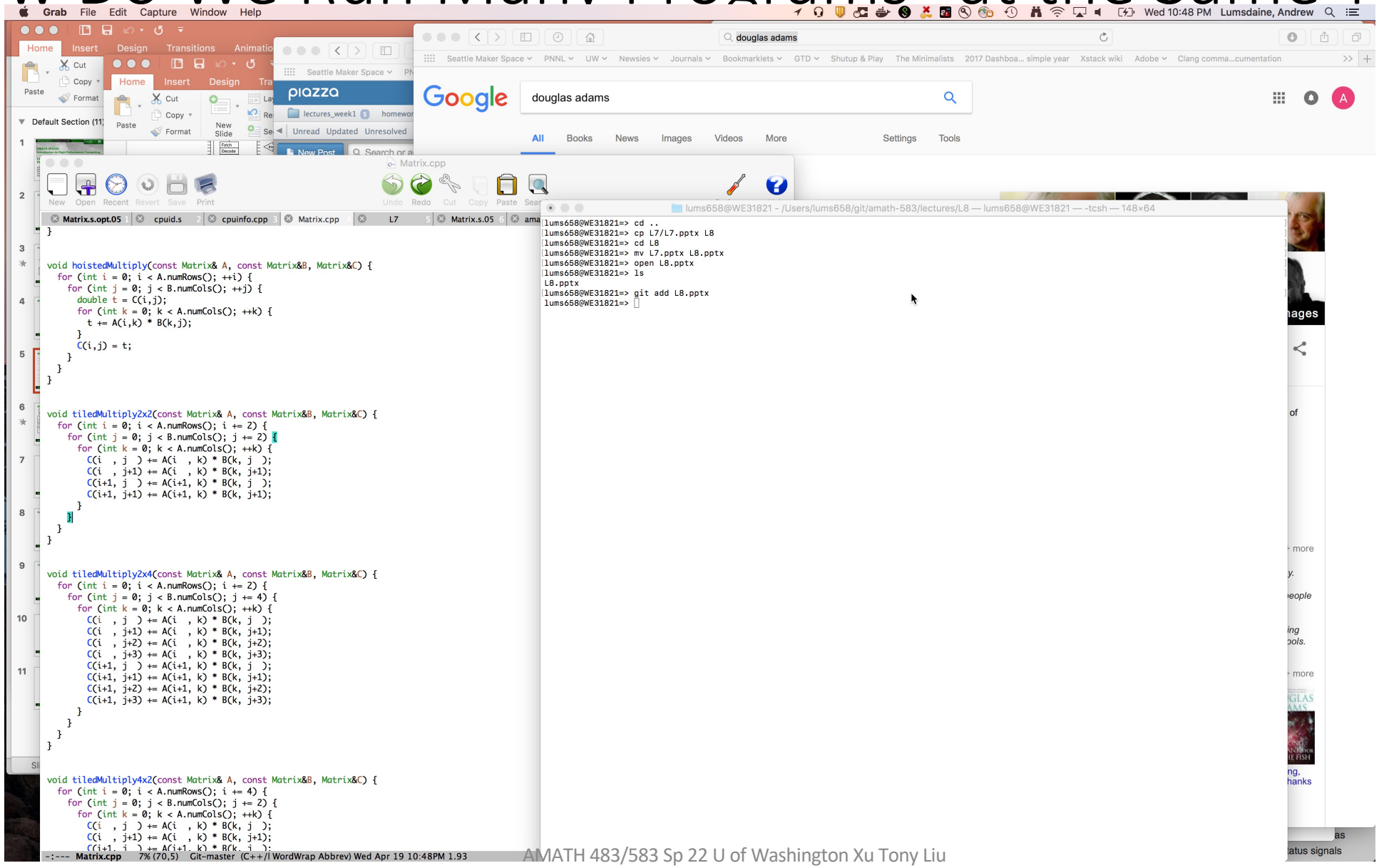


Including from the OS (just another program)

How does another program run?

How did the bytes get here?

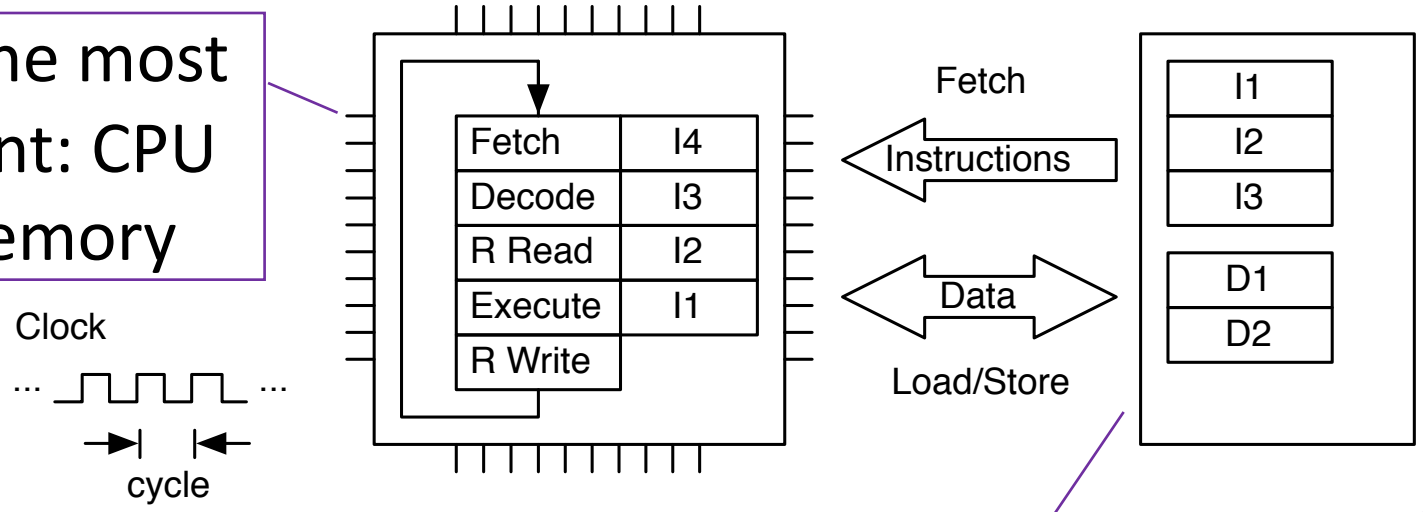
How Do We Run Many Programs "at the Same Time"?



A Word About Operating Systems

- An operating system is ***a (system) program*** that provides a standard interface between the resources of a computer and the users of the computer

Two of the most important: CPU and memory

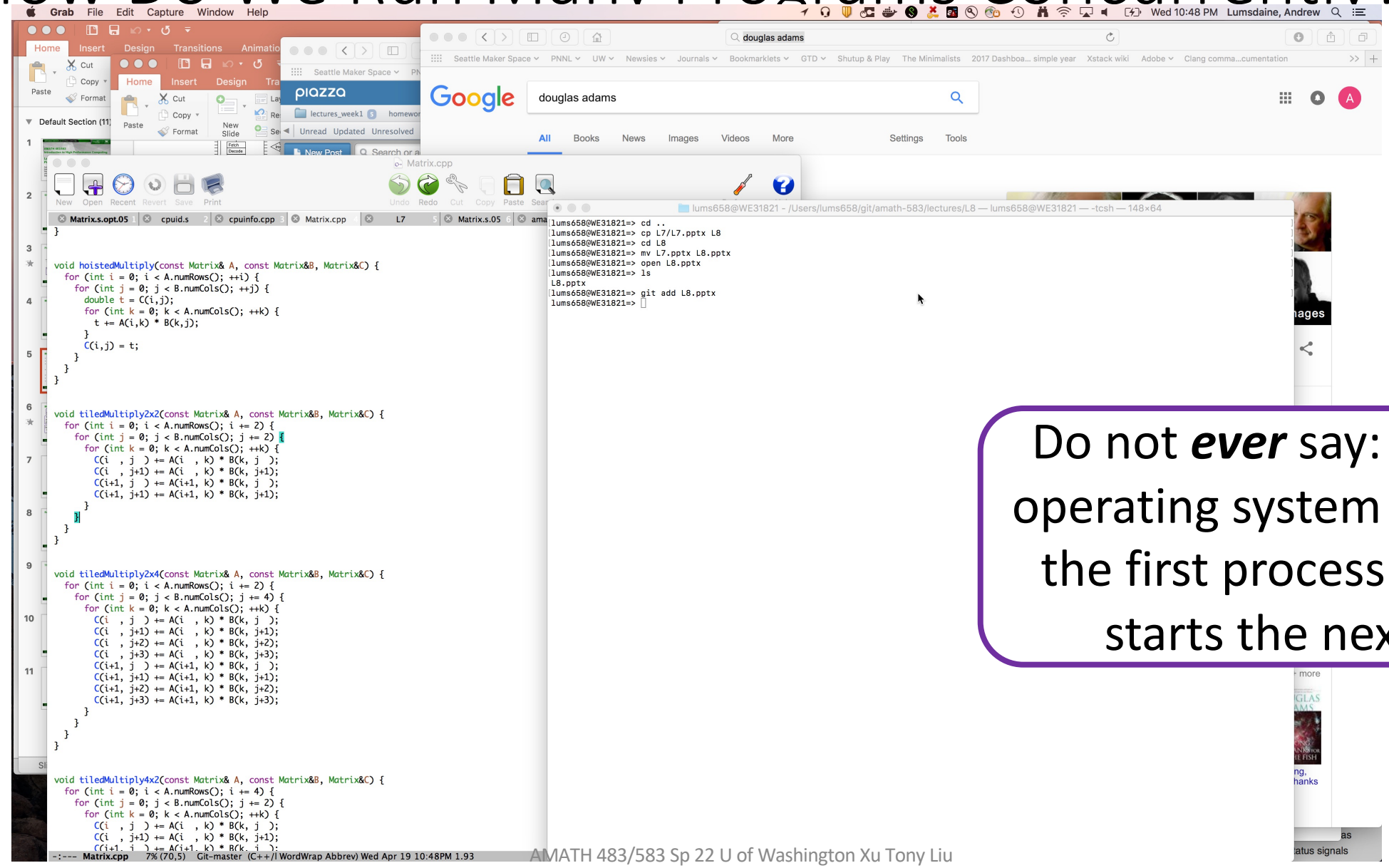


Also, file system, I/O, network, etc

Two of the most important: CPU and memory

For HPC, these are the most important

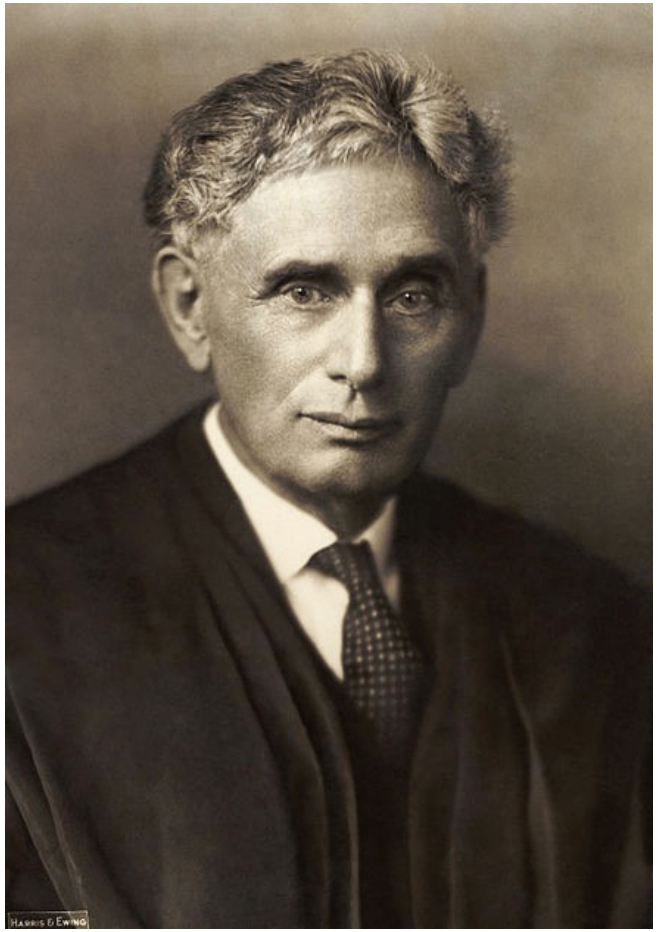
How Do We Run Many Programs Concurrently?



Do not ***ever*** say: ”the operating system stops the first process and starts the next

Processes

- A process is an abstraction for a collection of resources to represent a (running) program
  - CPU
  - Memory
  - Address space



The Operating System Can Run When…

- The process whose instructions are being executed by the CPU (the running process) requests a service from the OS (makes a ***system call***)
- In response to a hardware interrupt
- It does not spontaneously run
- It is not somehow running in the background
- Again, when the CPU is executing instructions for one program, it is not executing instructions for another program
- The only way anything happens on the computer is if the CPU executes instructions that make it happen

Process Abstraction

Stored in Process Control Block (PCB)

Set of information about process resources

Sufficient to be able to start a process after stopped

Also for accounting / administrative purposes

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

What does program counter represent?

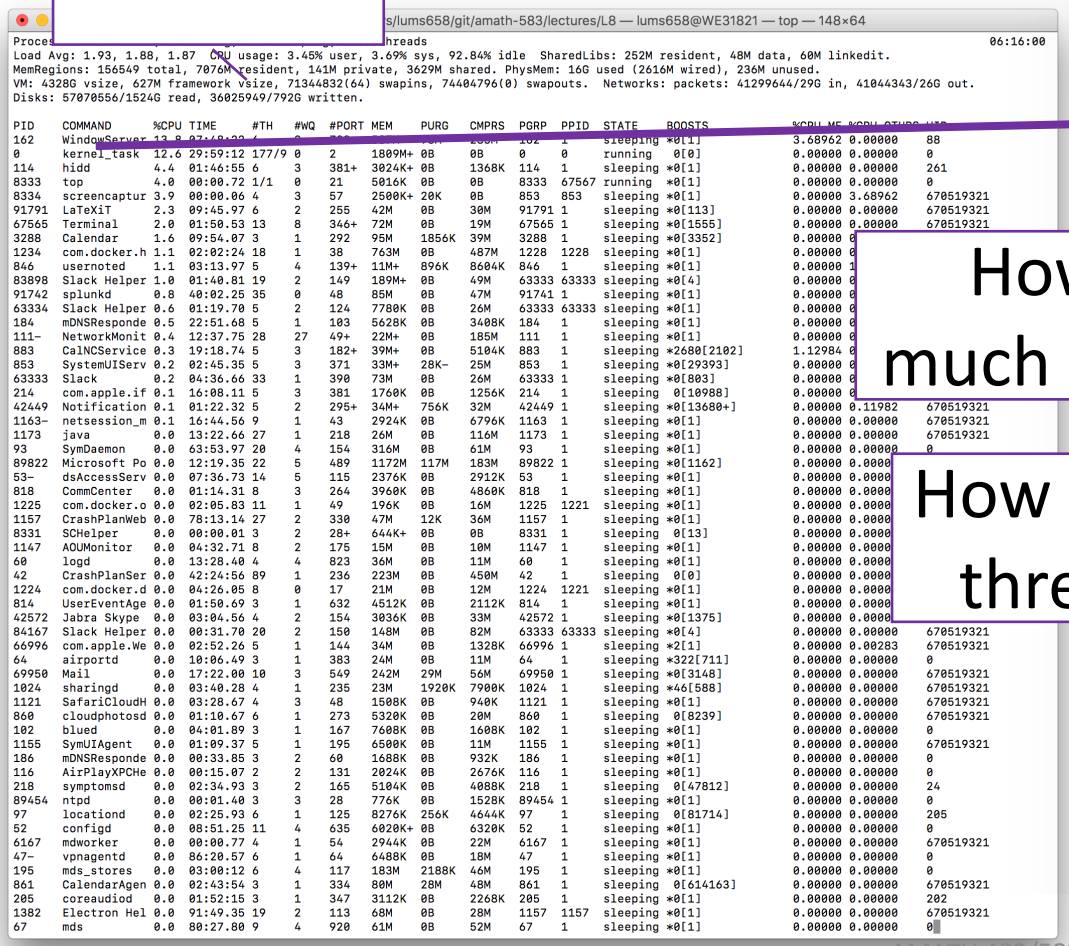
The Process Concept

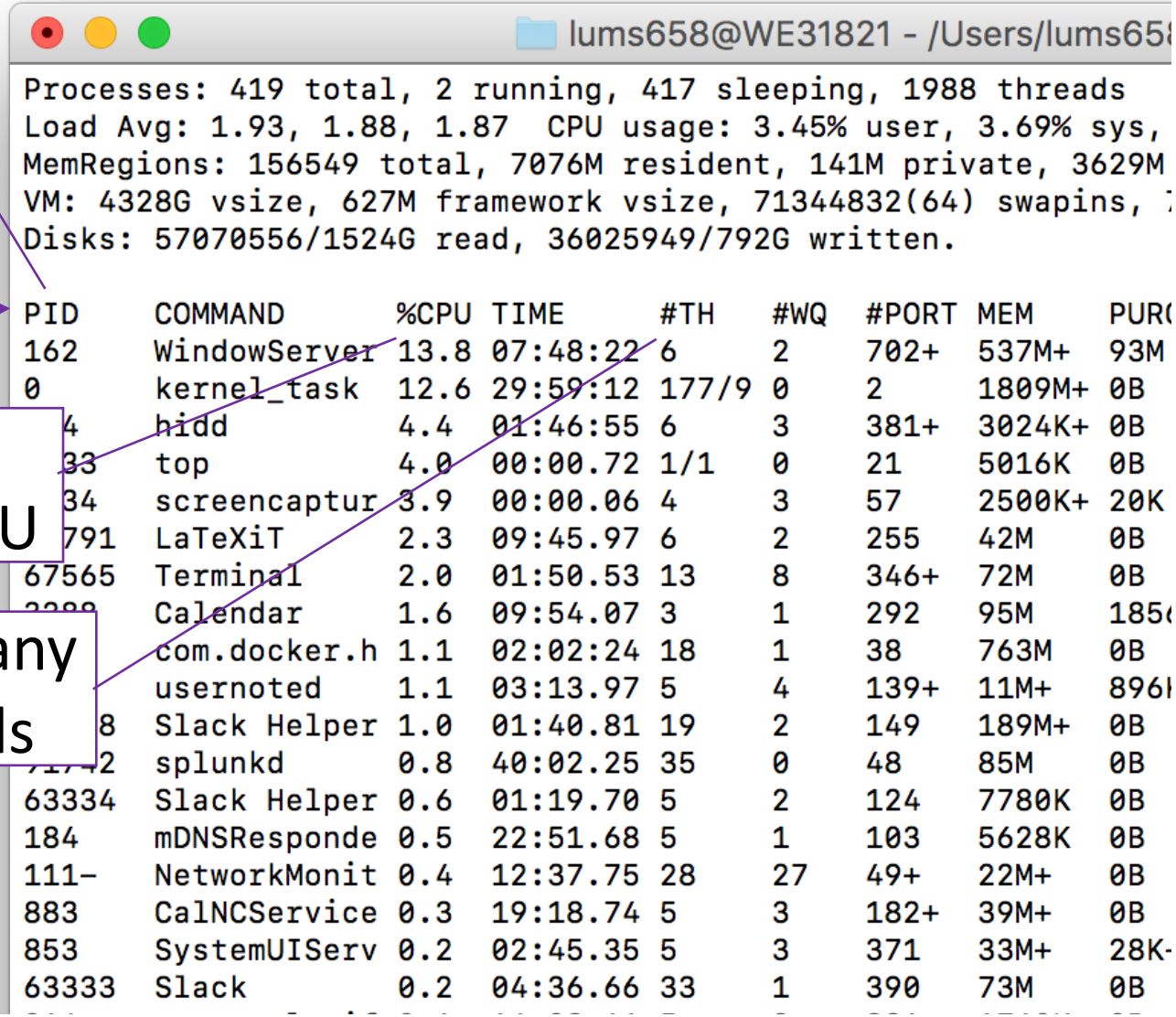
$ top -u

Process ID

How much CPU

How many threads





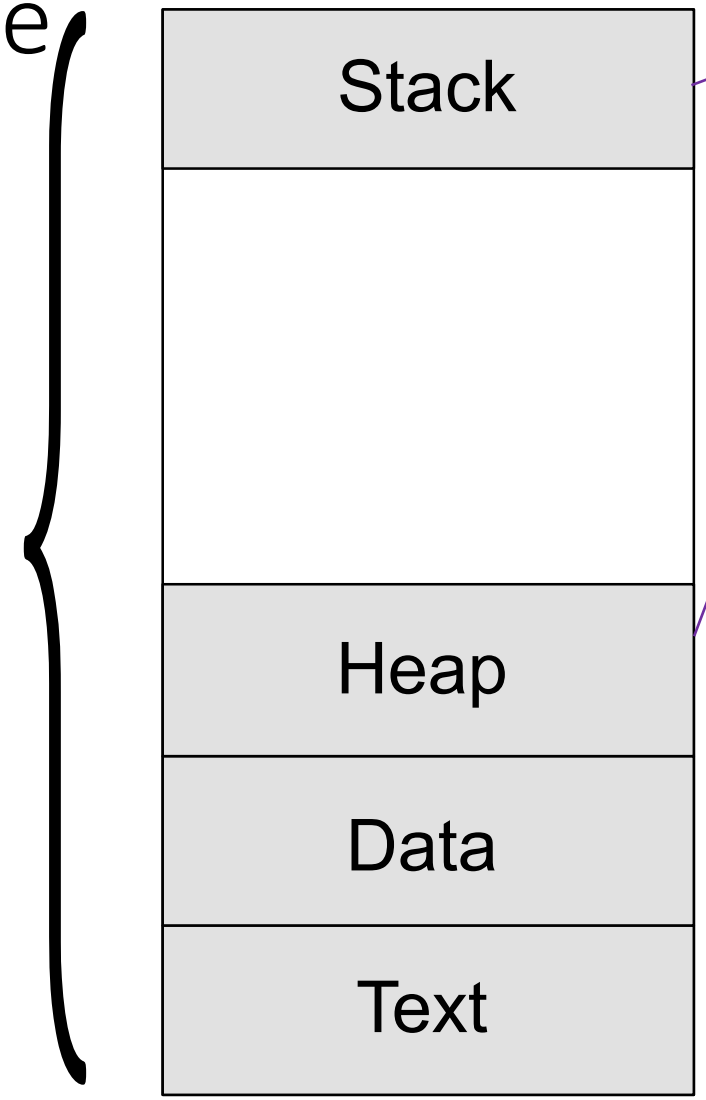
Process address space

Memory resources for each process

All 32/48/64 bits

How can each process use all the address space?

Address Space



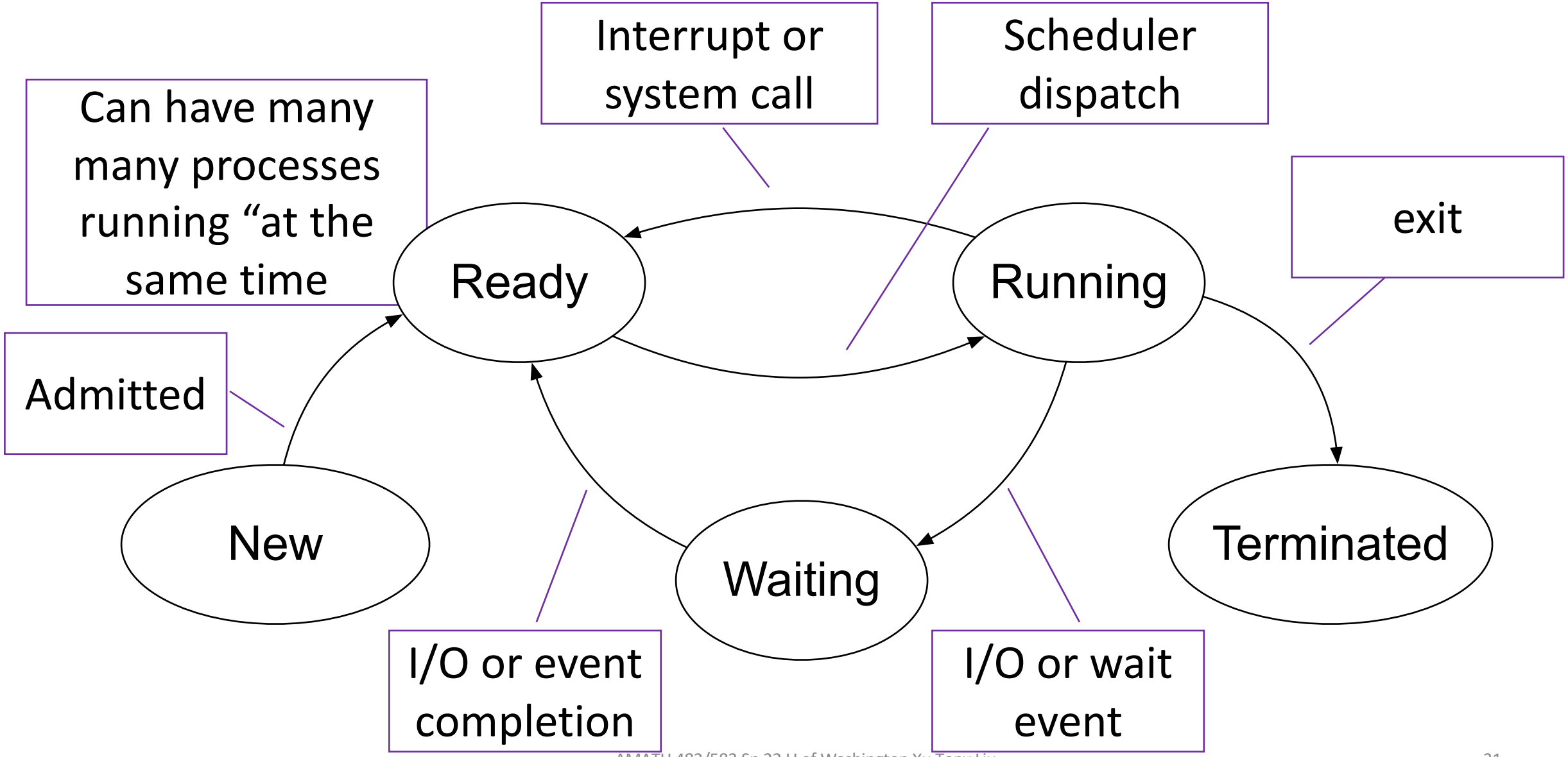
Created and managed at run time

Created and managed at run time

Compiled / Linked

Stored Program

Process Lifetime



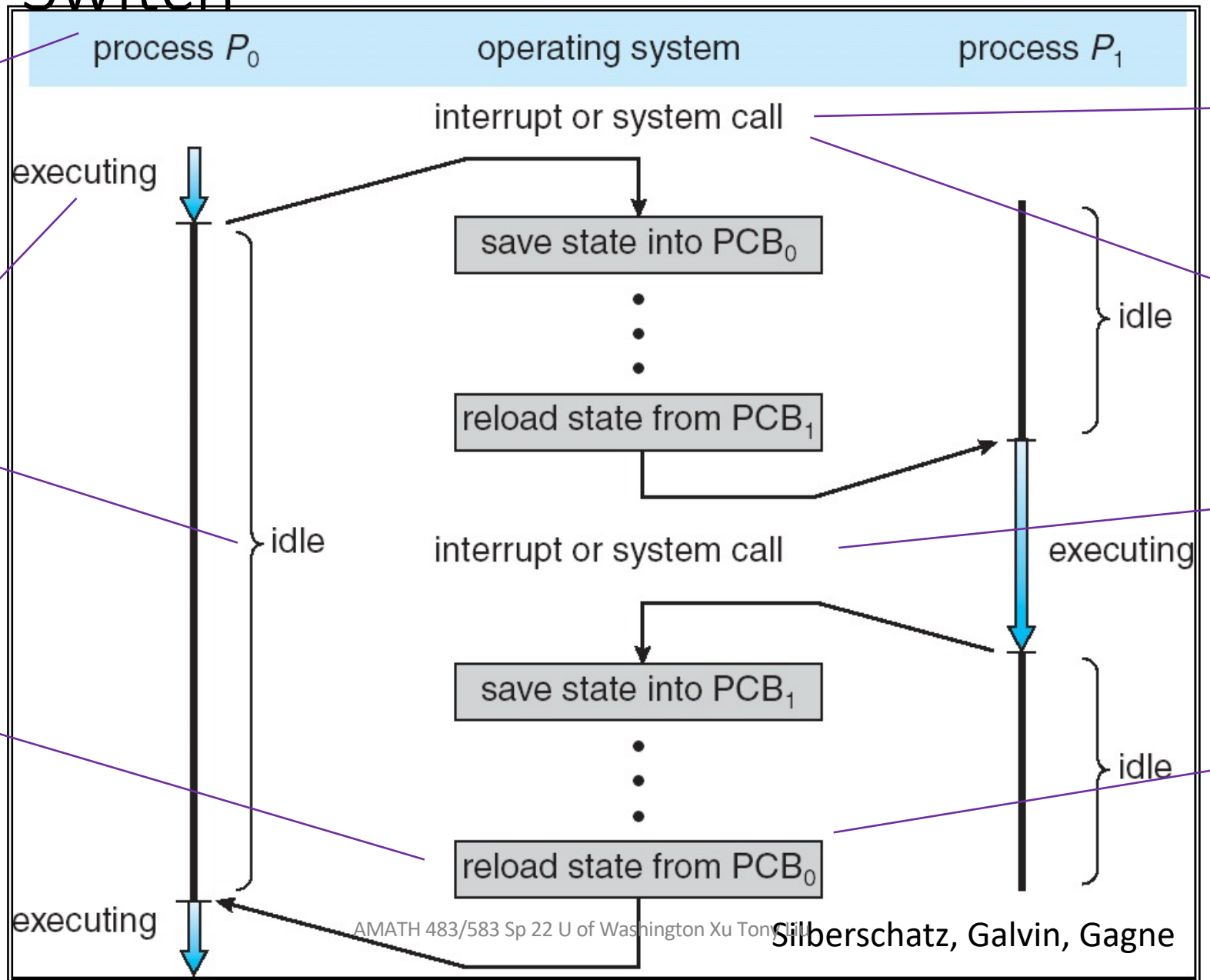
Context Switch

P0 and P1 are running processes

What does this mean?

And this?

PCB = Process Control Block



External to OS

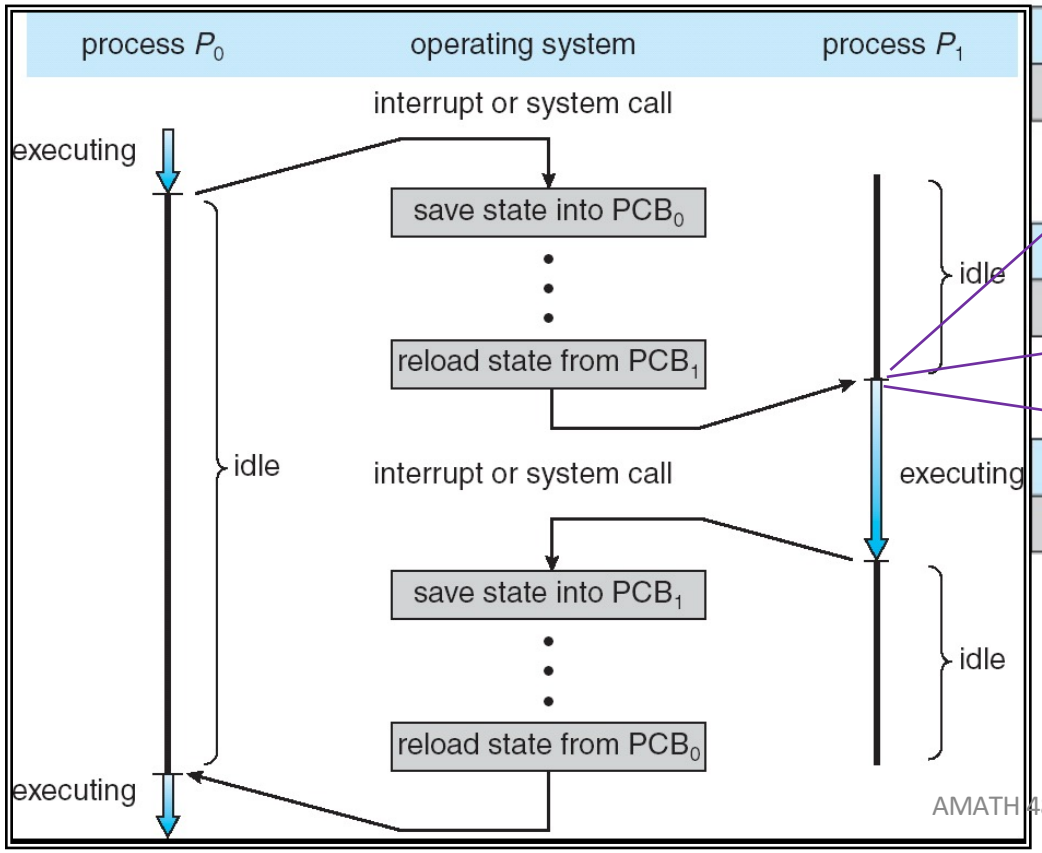
OS does not do this

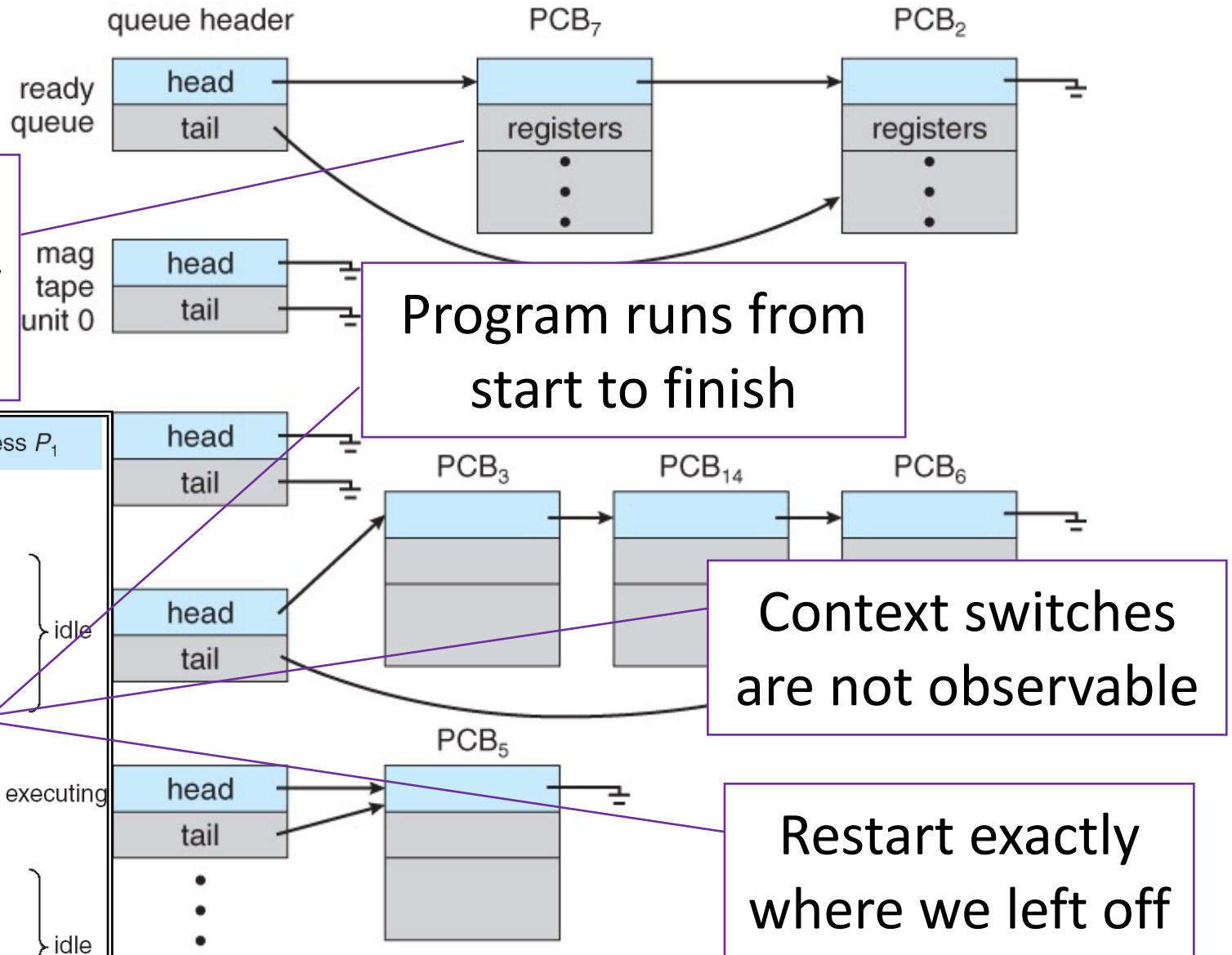
External to OS

Expensive!

Process Queues

A process control block (PCB) has all information necessary to manage a process





Program runs from start to finish

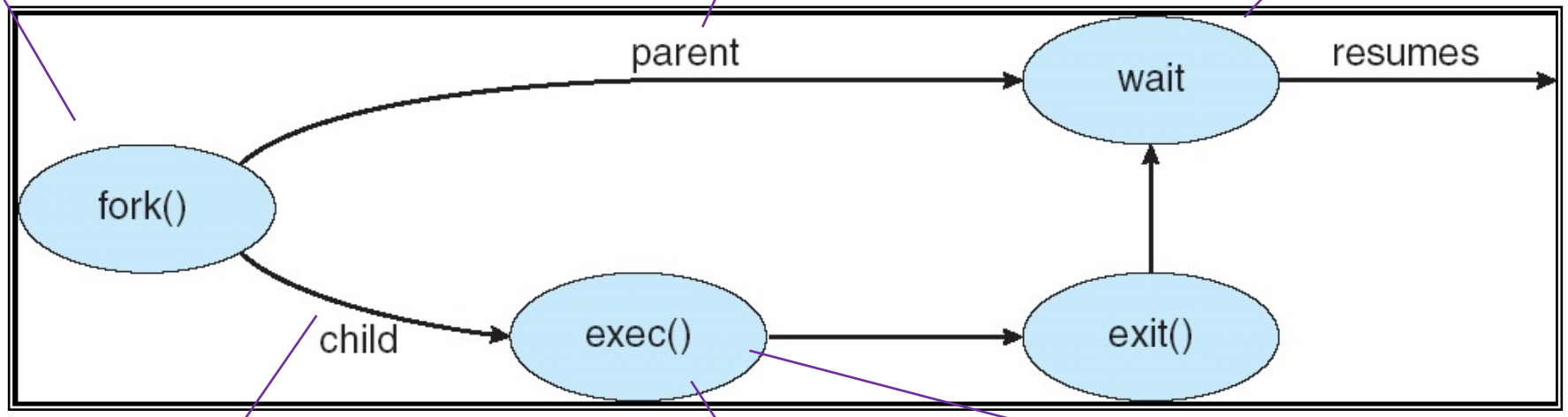
Context switches are not observable

Restart exactly where we left off

Process invokes fork()

The other process (the ”parent) keeps executing

Can wait for other process to complete



The OS makes a copy of the original process and makes it runnable

One of the processes (the “child”) runs exec()

Which pulls in new program bits to run

You see this fork/exec/wait almost all the time with one particular program you run (which?)

Example: process creation in UNIX

One process calls fork()

```
#include <unistd.h>

int main () {

  fork();

  return 0;
}
```

Each process "thinks" it called fork() and returned

Two processes return from fork()

```
#include <unistd.h>

int main () {

  fork();

  return 0;
}
```

Two processes return from fork()

```
#include <unistd.h>

int main () {

  fork();

  return 0;
}
```

fork() make an exact copy

Example

```
int main() {
{
  int pids[20];

  for (int i = 0; i < 20; ++i) {
    pids[i] = fork();
  }

return 0;
}
```

fork() returns a PID identifier

Loop 20 times

Call fork() 20 times

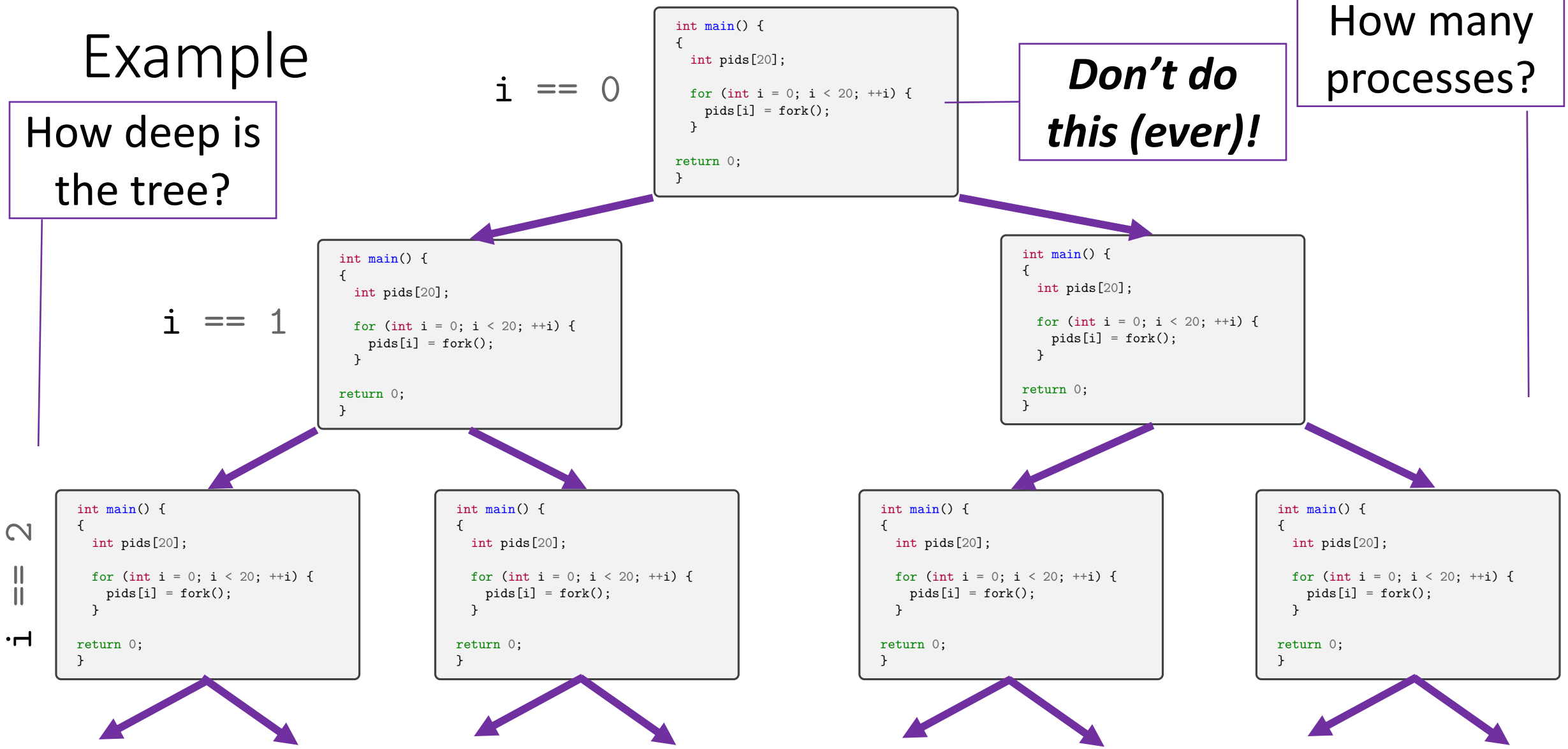
How many processes get created?

Example

How deep is the tree?

How many processes?

Don't do this (ever)!



man fork()

```
#include <unistd.h>
pid_t fork();
```

The child process has a unique id

Upon successful completion, fork() returns a value of 0 to the child process and the returns the process ID of the child process to the parent process

lums658@WE31821 - /Users/lums658/git/amath-583/lectures/L8 — lums658@WE31821 — less ‹ man fork — 135×52

```
FORK(2)                   BSD System Calls Manual                  FORK(2)

NAME
     fork -- create a new process

SYNOPSIS
     #include <unistd.h>

     pid_t
     fork(void);

DESCRIPTION
     fork() causes creation of a new process.  The new process (child process) is an exact copy of the calling process (parent
     process) except for the following:

           o   The child process has a unique process ID.

           o   The child process has a different parent process ID (i.e., the process ID of the parent process).

           o   The child process has its own copy of the parent's descriptors.  These descriptors reference the same underlying
               objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an
               lseek(2) on a descriptor in the child process can affect a subsequent read or write by the parent.  This descriptor
               copying is also used by the shell to establish standard input and output for newly created processes as well as to set
               up pipes.

           o   The child processes resource utilizations are set to 0; see setrlimit(2).

RN VALUES
     Upon successful completion, fork() returns a value of 0 to the child process and returns the process ID of the child process to
     the parent process.  Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global
     variable errno is set to indicate the error.

RS
     fork() will fail and no child process will be created if:

     [EAGAIN]           The system-imposed limit on the total number of processes under execution would be exceeded.  This limit is
                        configuration-dependent.

     [EAGAIN]           The system-imposed limit MAXUPRC (<sys/param.h>) on the total number of processes under execution by a single
                        user would be exceeded.

     [ENOMEM]           There is insufficient swap space for the new process.

CY SYNOPSIS
     #include <sys/types.h>
     #include <unistd.h>

     The include file <sys/types.h> is necessary.

SEE ALSO
:
```

Example Revisited

```
int main() {
{
  pid_t pids[20];

  for (int i = 0; i < 20; ++i) {
    pids[i] = fork();
    if (pids[i] == 0)
      break;
  }

return 0;
}
```

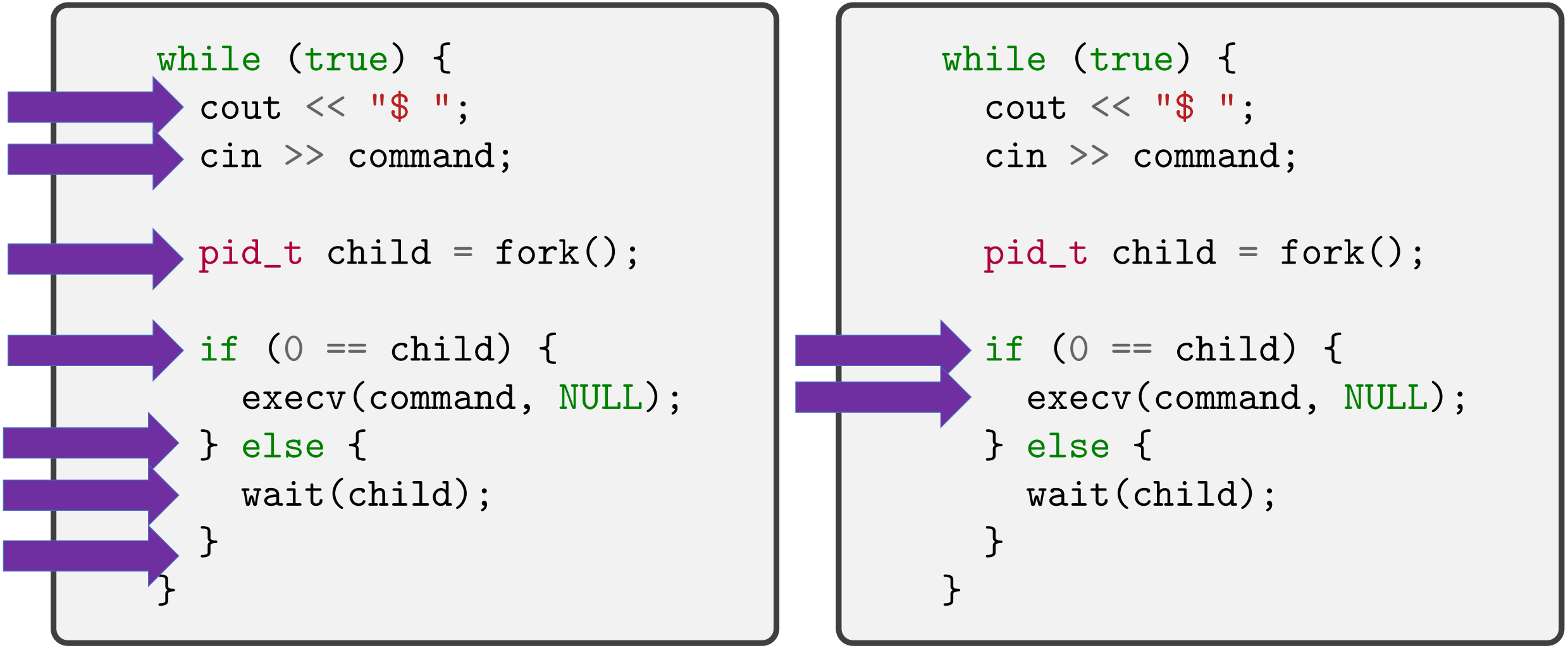
Get return value of fork()

How many processes now?

If zero, the process is a child

If no, the process is the parent, keep going

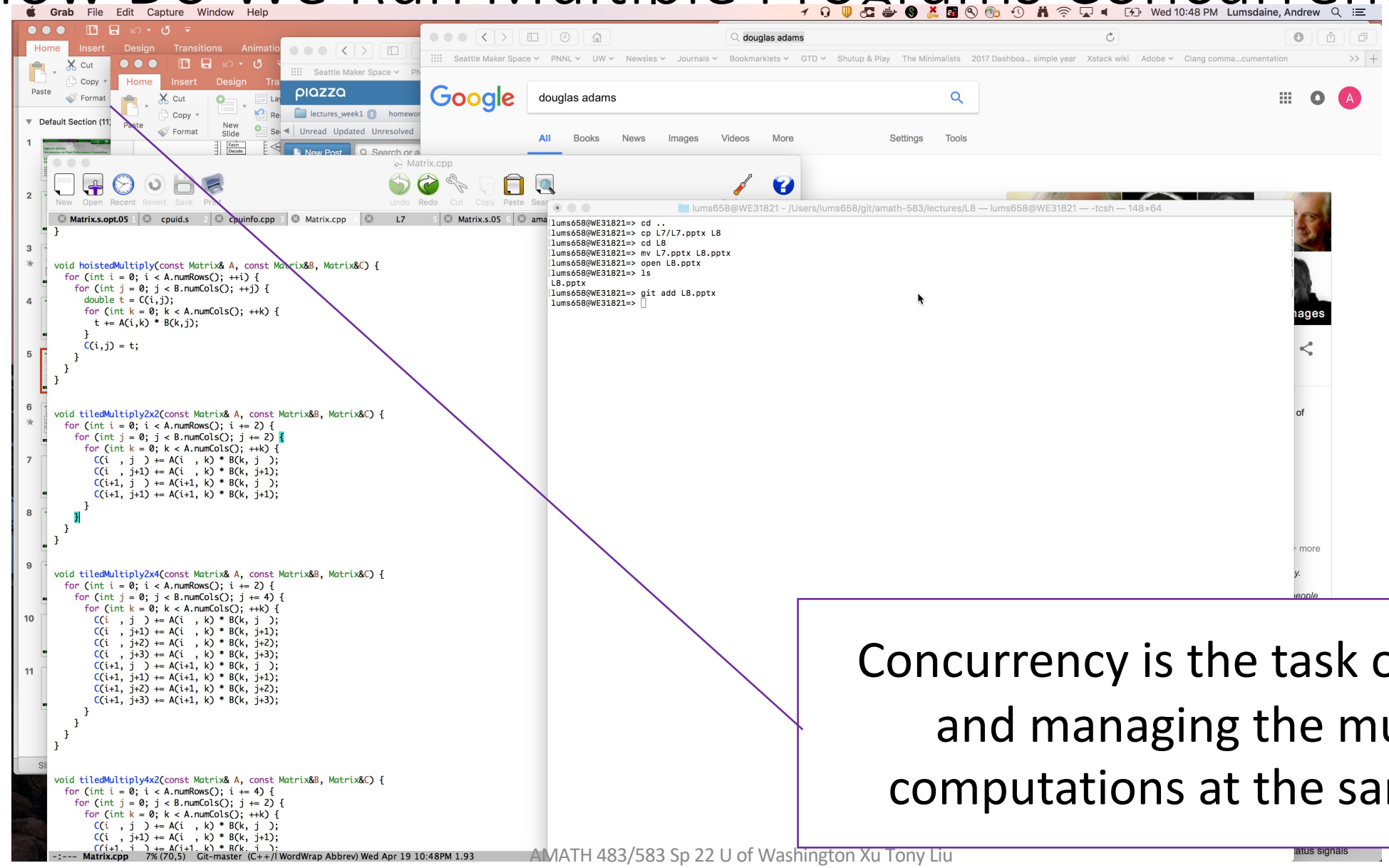
Process creation in UNIX (fork / exec pattern)



```
while (true) {
  cout << "$ ";
  cin >> command;

  pid_t child = fork();

  if (0 == child) {
    execv(command, NULL);
  } else {
    wait(child);
  }
}
```

```
while (true) {
  cout << "$ ";
  cin >> command;

  pid_t child = fork();

  if (0 == child) {
    execv(command, NULL);
  } else {
    wait(child);
  }
}
```

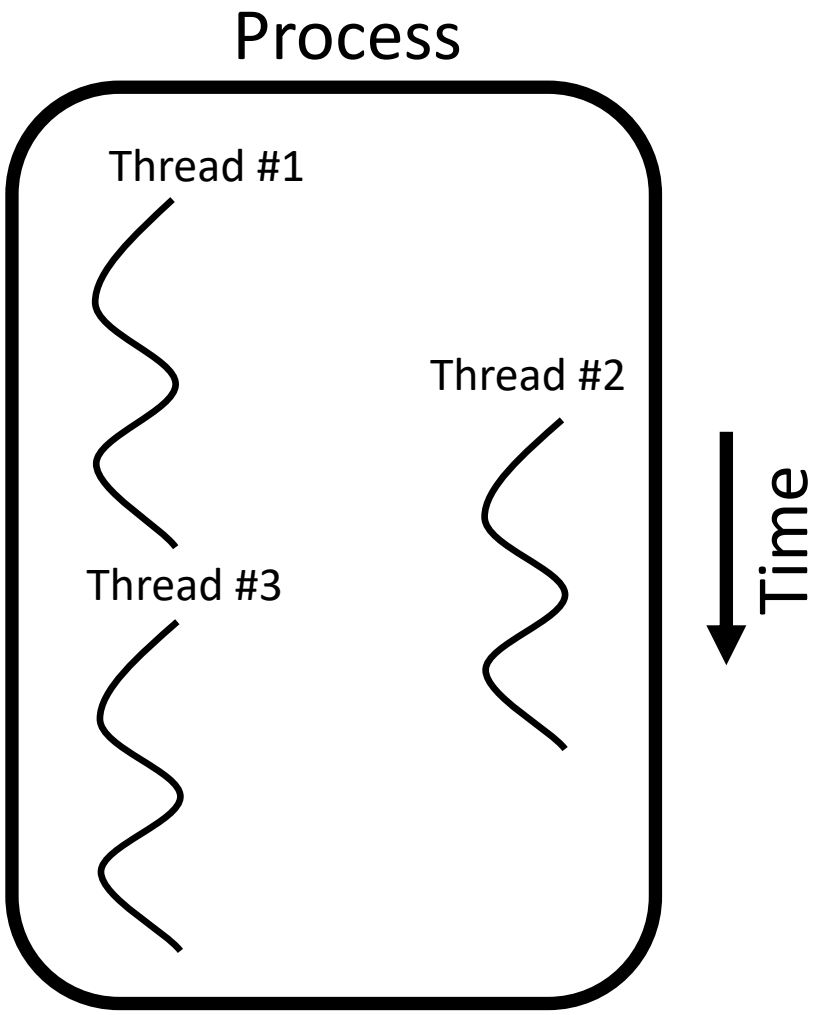
How Do We Run Multiple Programs Concurrently?



Concurrency is the task of running and managing the multiple computations at the same time

Processes and Threads

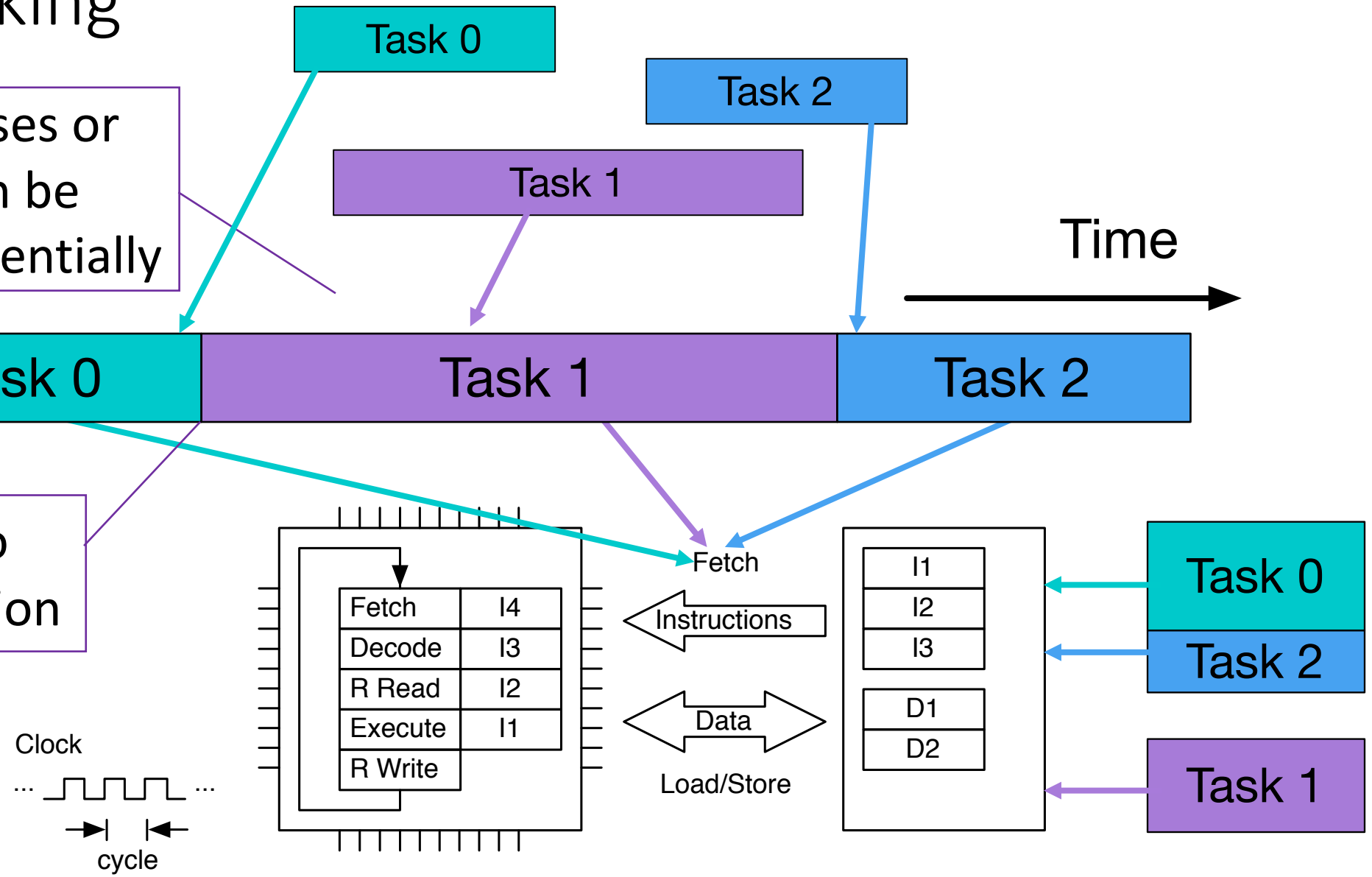
- A process is an abstraction for a collection of resources to represent a (running) program
  - CPU
  - Memory
  - Address space
- A thread is an abstraction of execution (using the resources within a process)
  - Can share an address space



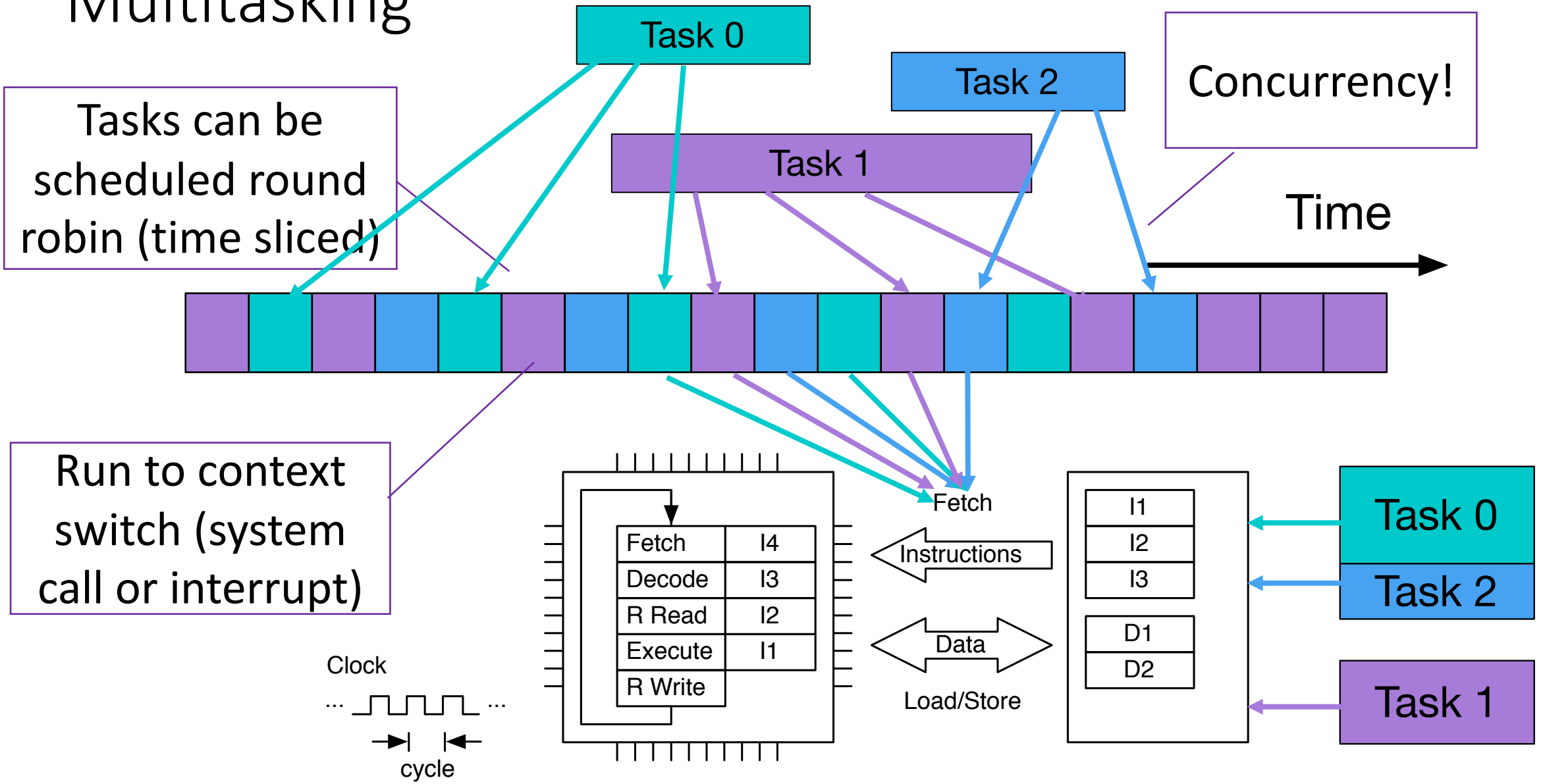
Multitasking

Tasks (processes or threads) can be scheduled sequentially

Run to completion



Multitasking

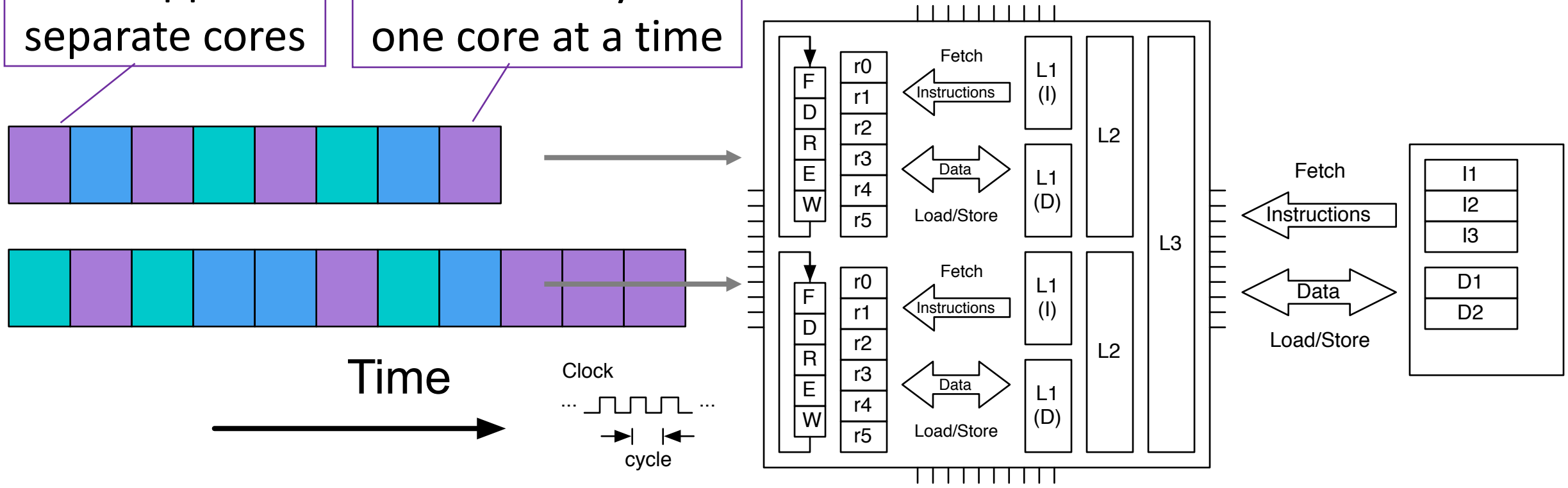


Multitasking on Multicore

Concurrency!

Time sliced and mapped to separate cores

A single threaded task can only use one core at a time



Multitasking on Multicore

Parallelism!

What about L1, L2?

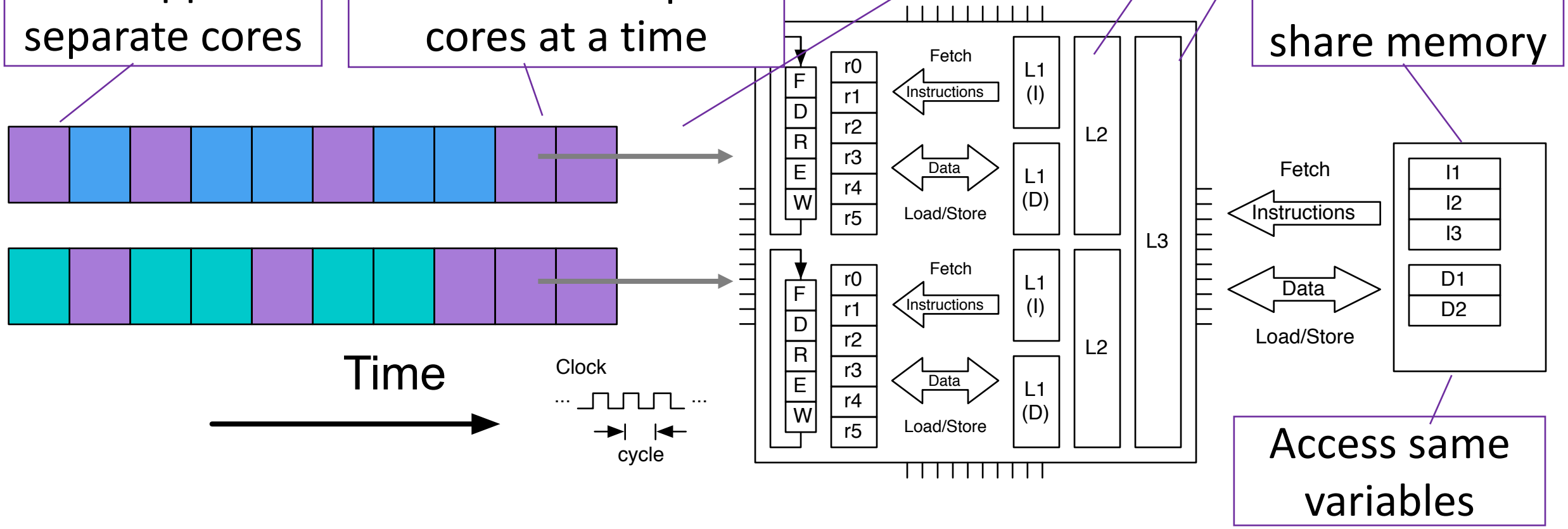
Time sliced and mapped to separate cores

A multithreaded task can use multiple cores at a time

Shorter run time!

And L3 cache

Threads can share memory



Access same variables

Cache Coherence

Hardware managed

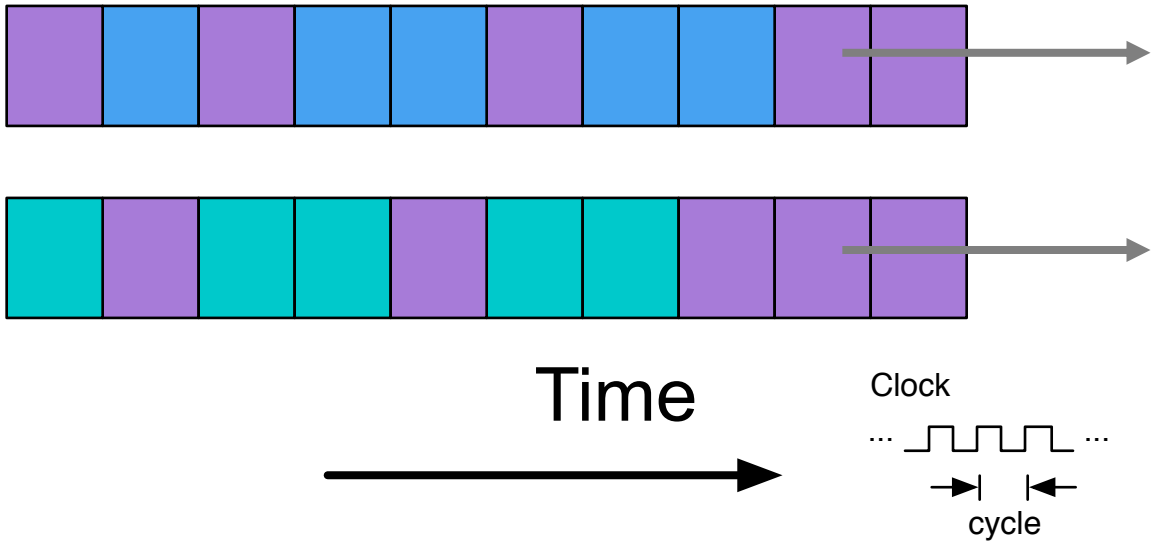
Same variable can be in two different caches

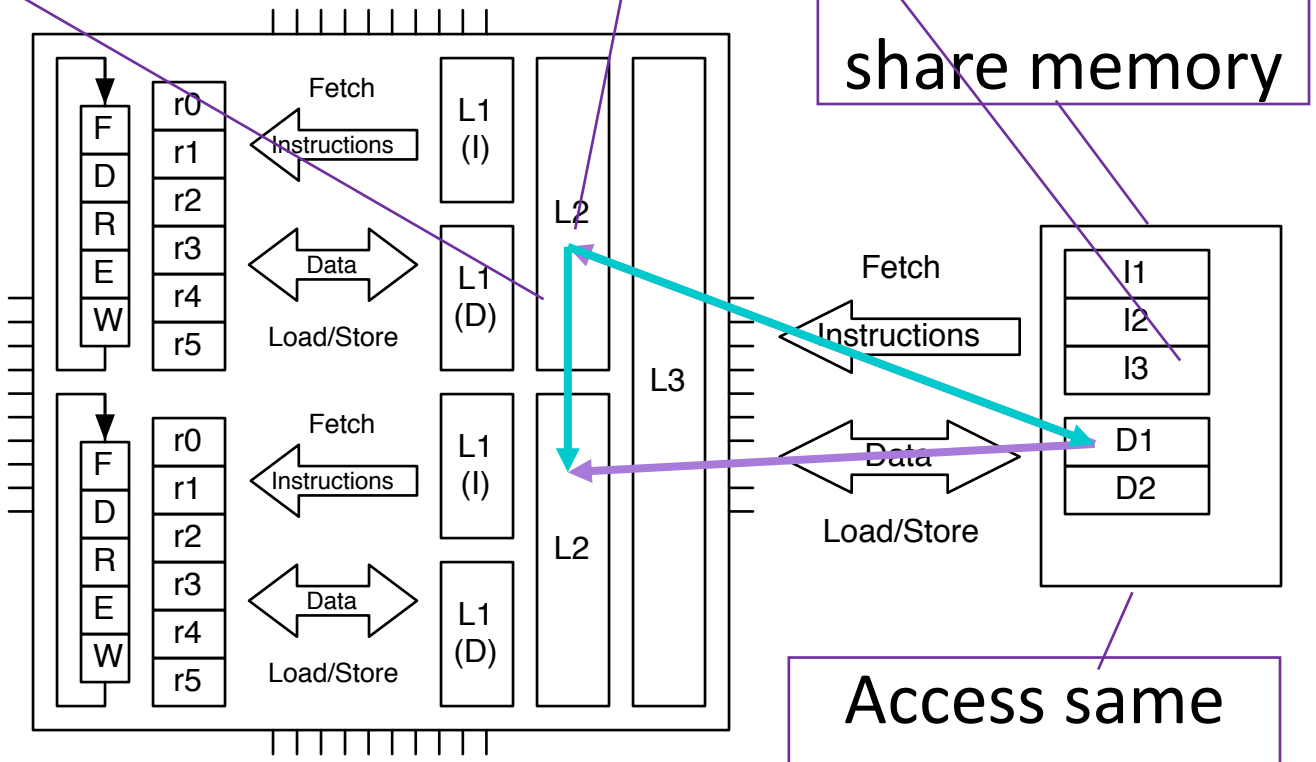
A multithreaded task can use multiple cores at a time

Cache coherence / memory consistency

What if one gets modified?

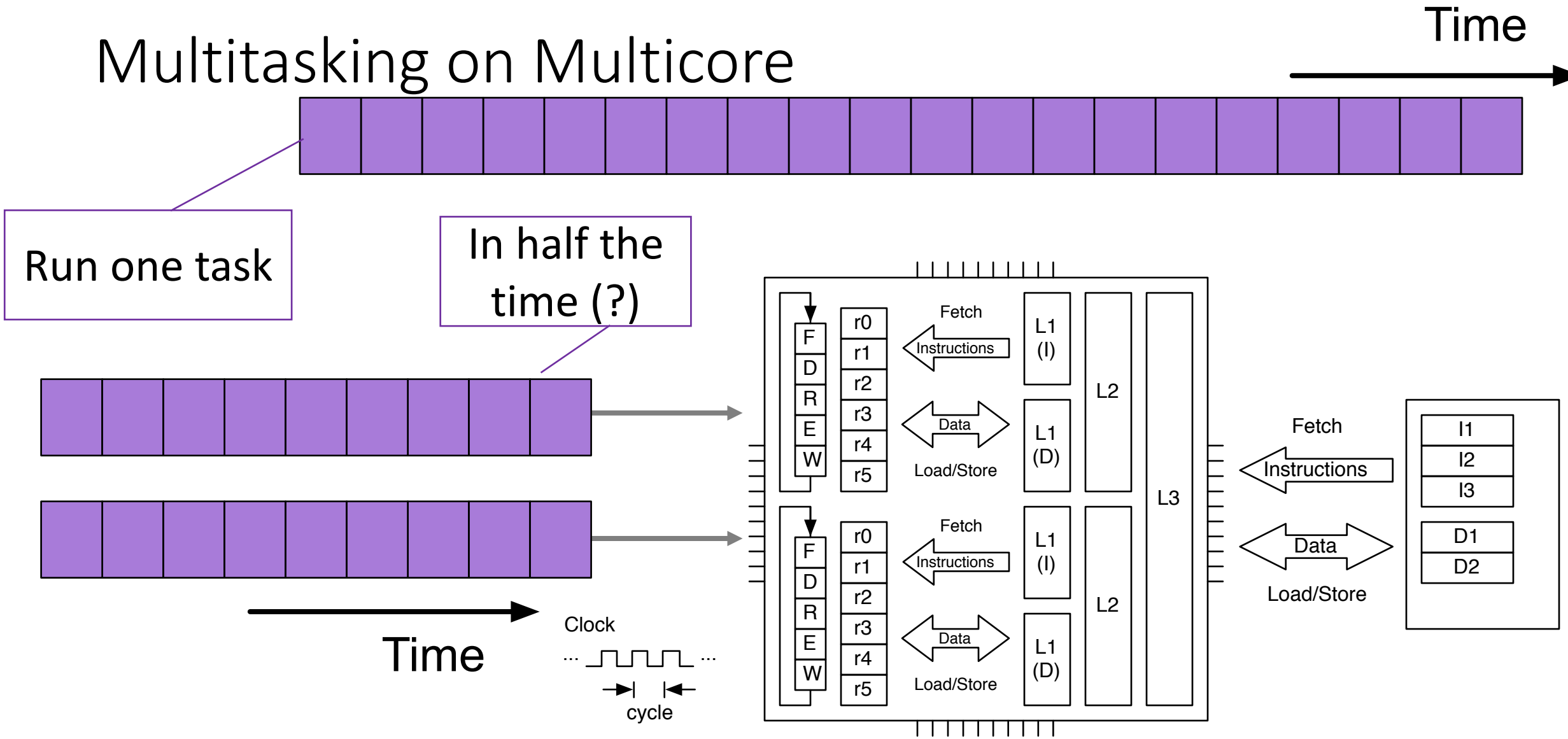
Threads can share memory



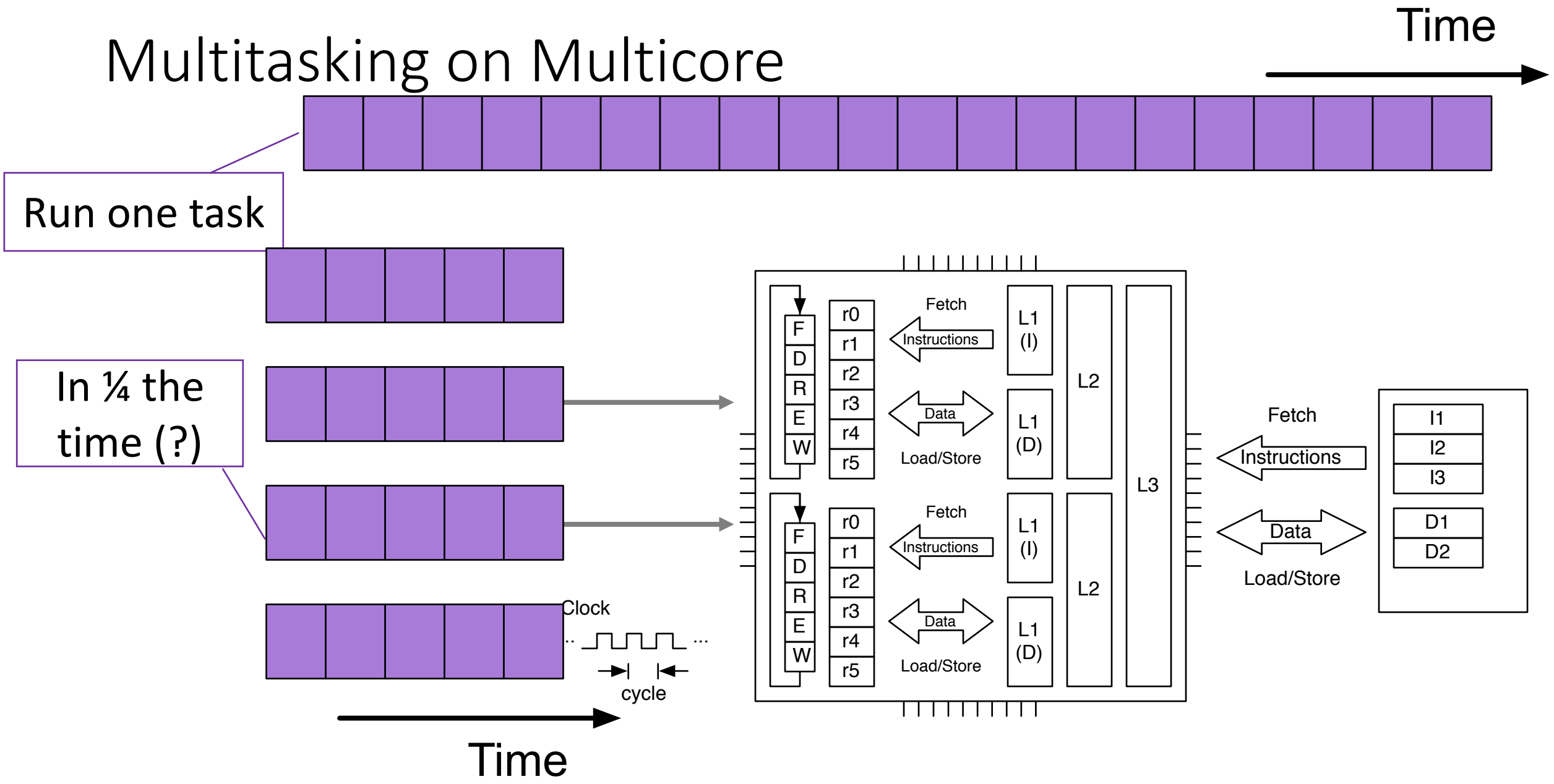


Access same variables

Multitasking on Multicore



Multitasking on Multicore



Multitasking on Multicore

Nonetheless, this is the essence of ***parallel*** computing

Parallel computation isn't done until all cores are done

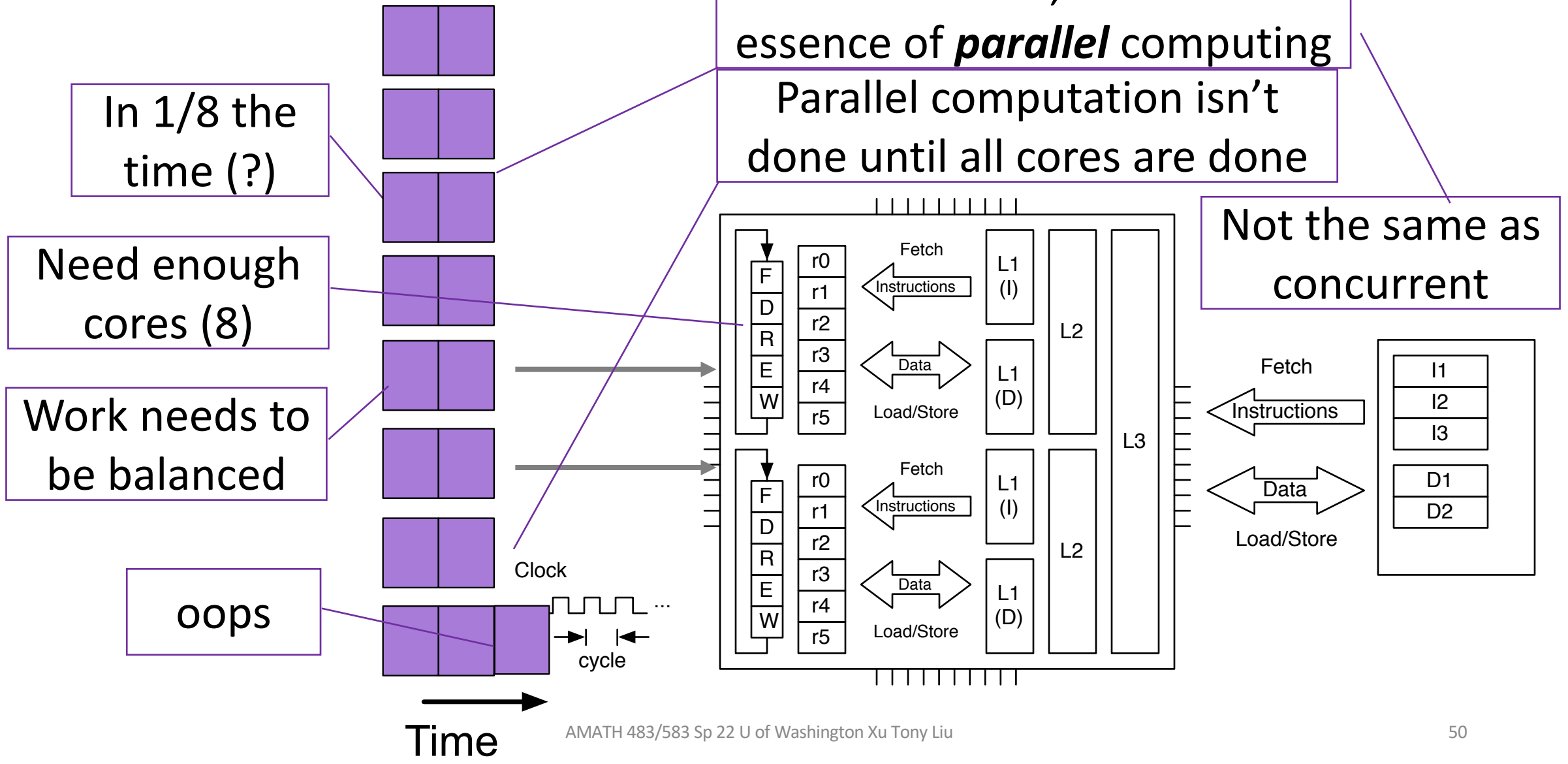
In 1/8 the time (?)

Need enough cores (8)

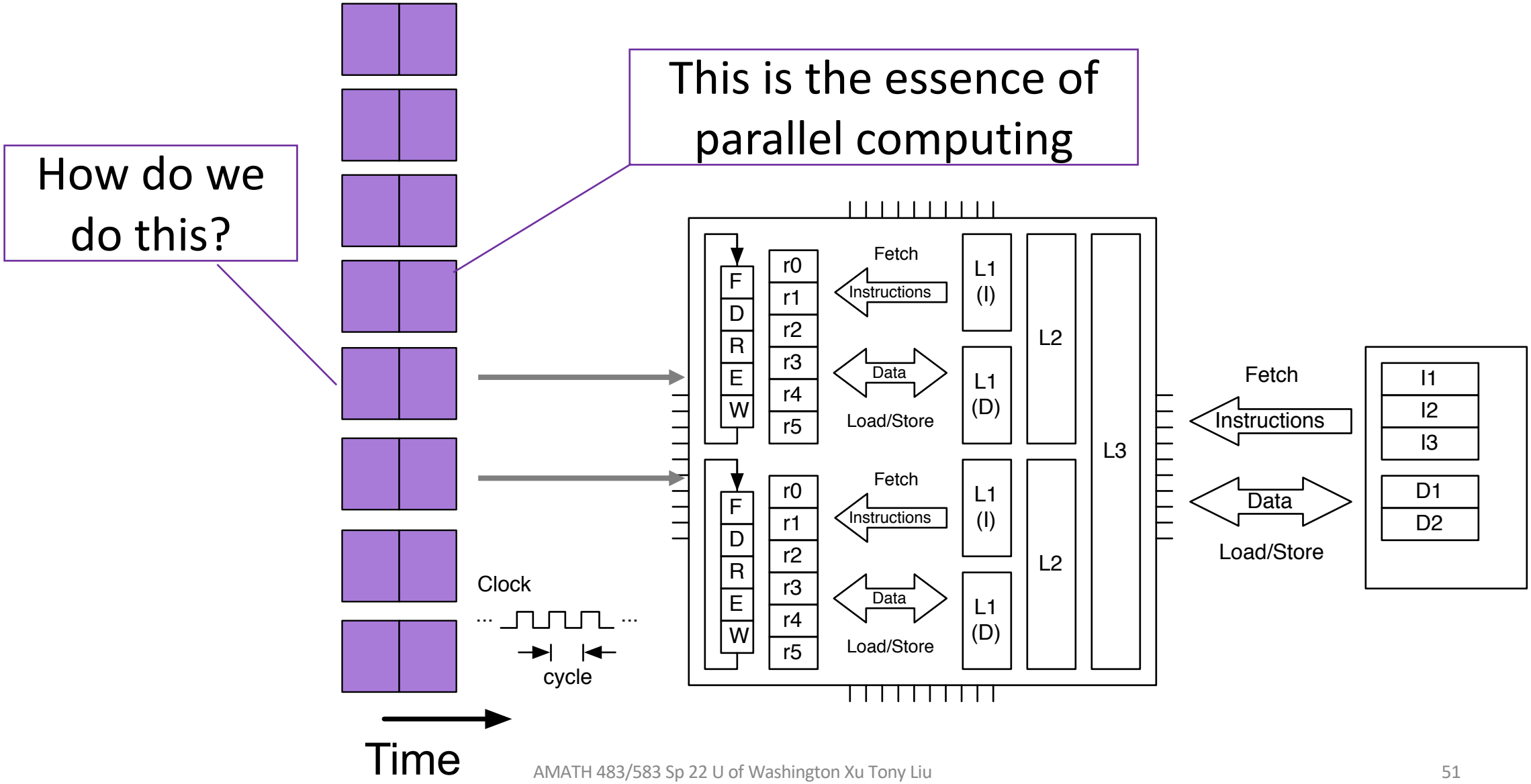
Work needs to be balanced

oops

Not the same as concurrent



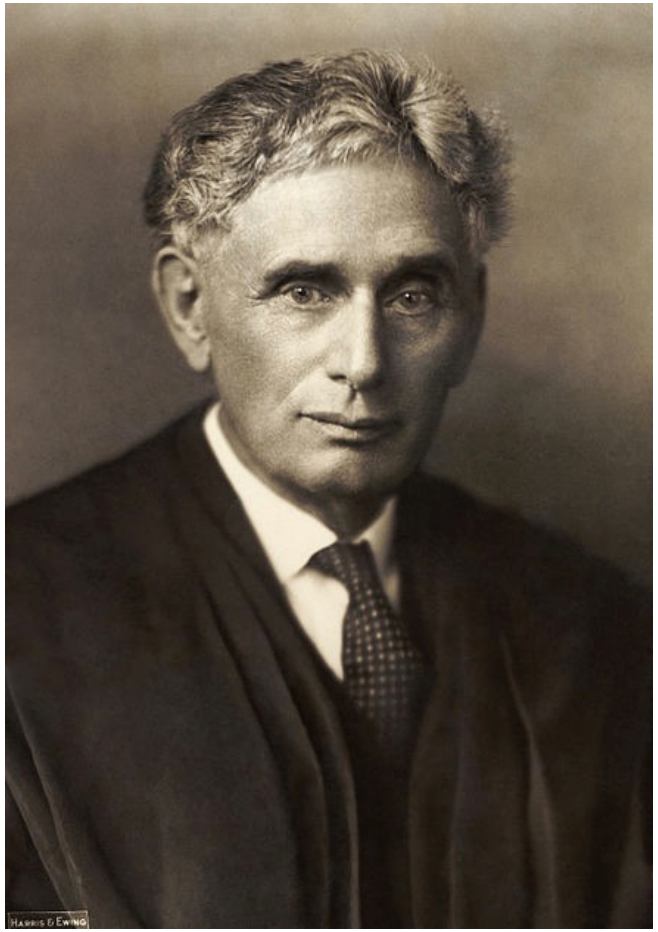
Multitasking on Multicore



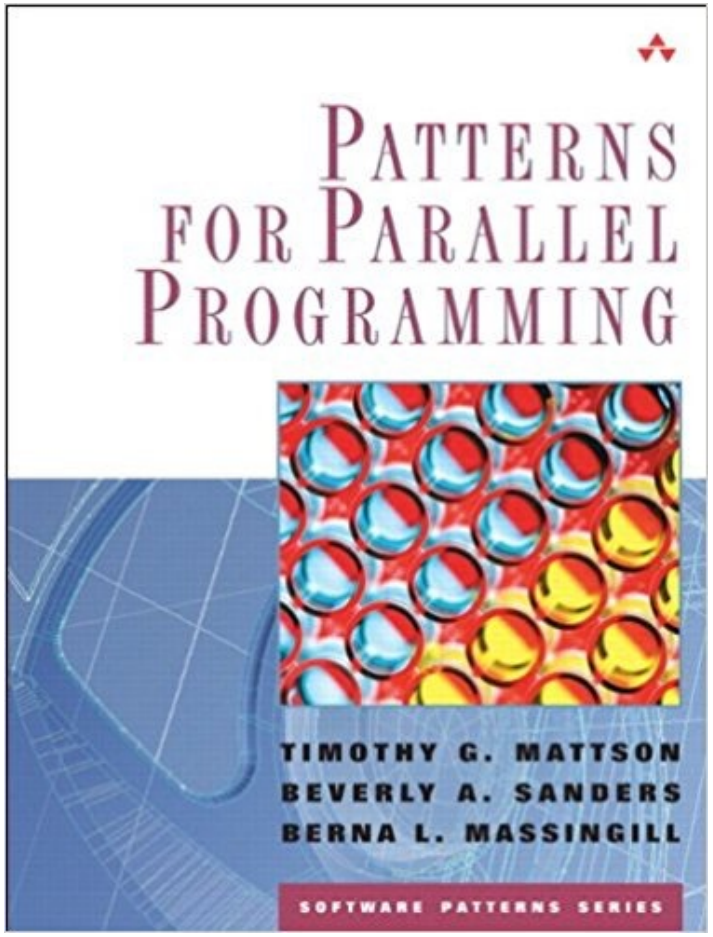
Processes vs Threads

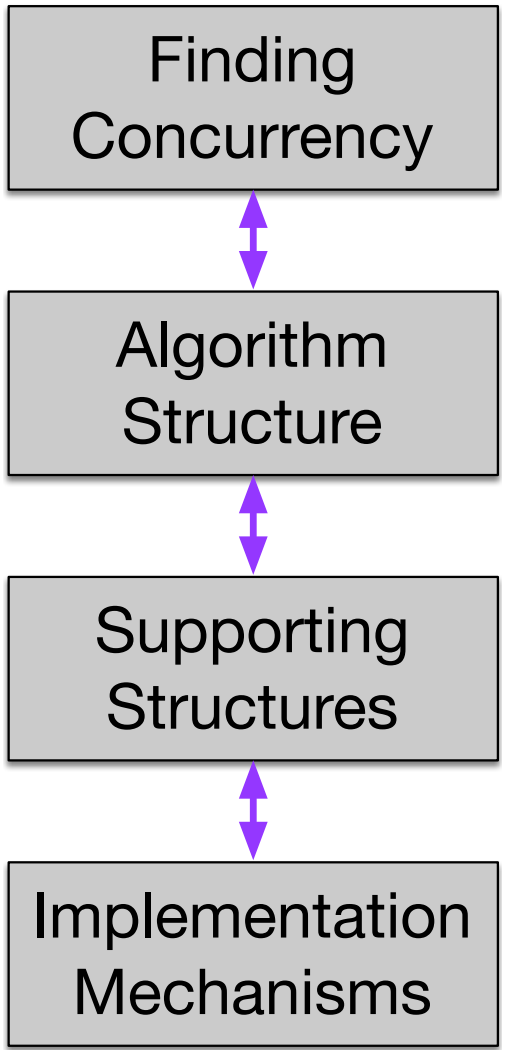
- A process is an abstraction for a collection of resources to represent a (running) program
  - Manage resources: CPU, Memory, Address space, etc.
  - Not light weighted
  - Expensive to create, and terminate

- A thread is an abstraction of execution (using the resources within a process)
  - The smallest sequence of programmed instructions
  - Threads can share an address space of a process
  - Light weighted
  - Less expensive to create and terminate



Parallelization Strategy





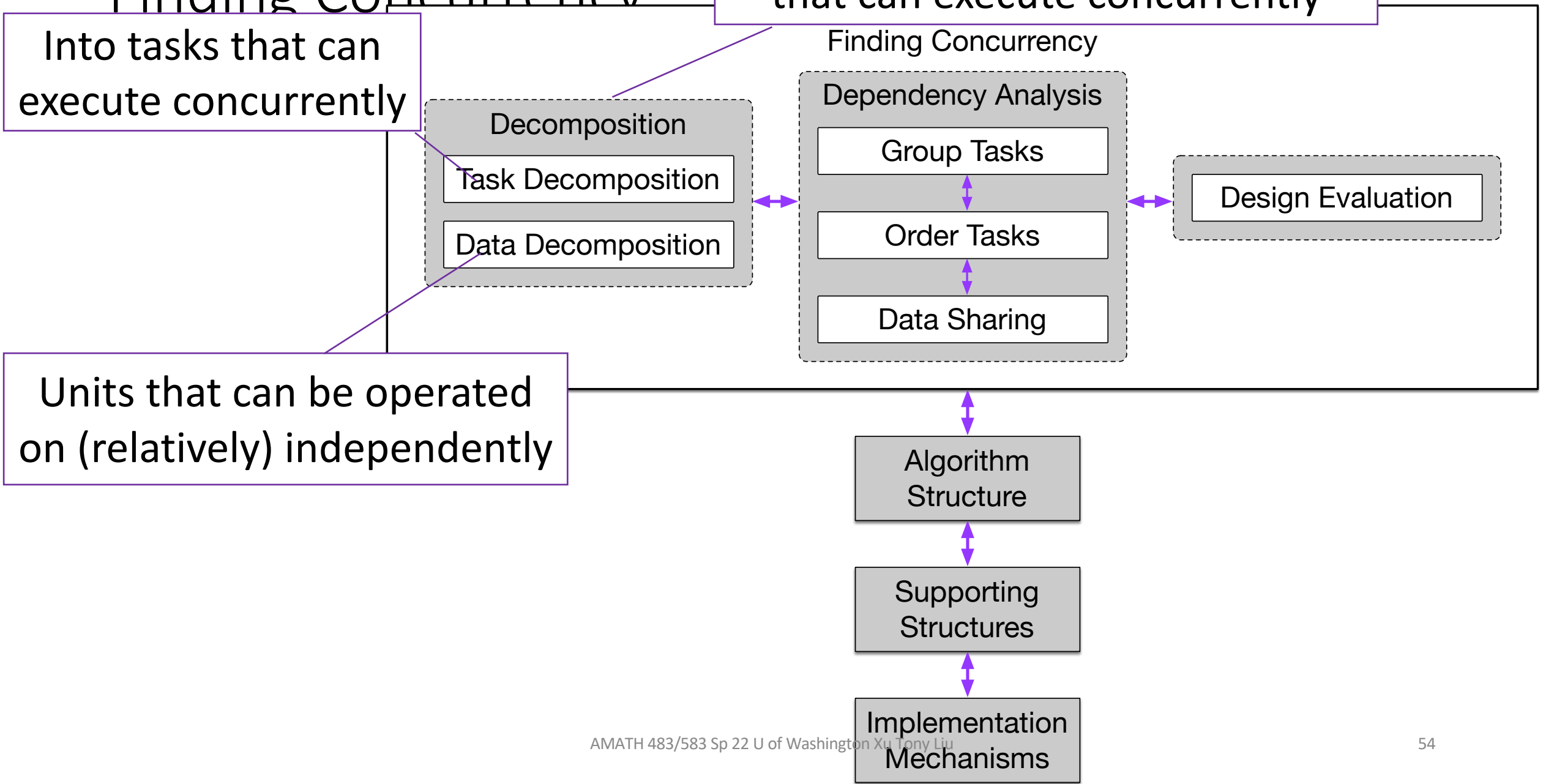
Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*(First ed.). Addison-Wesley Professional.

Finding Concurrency

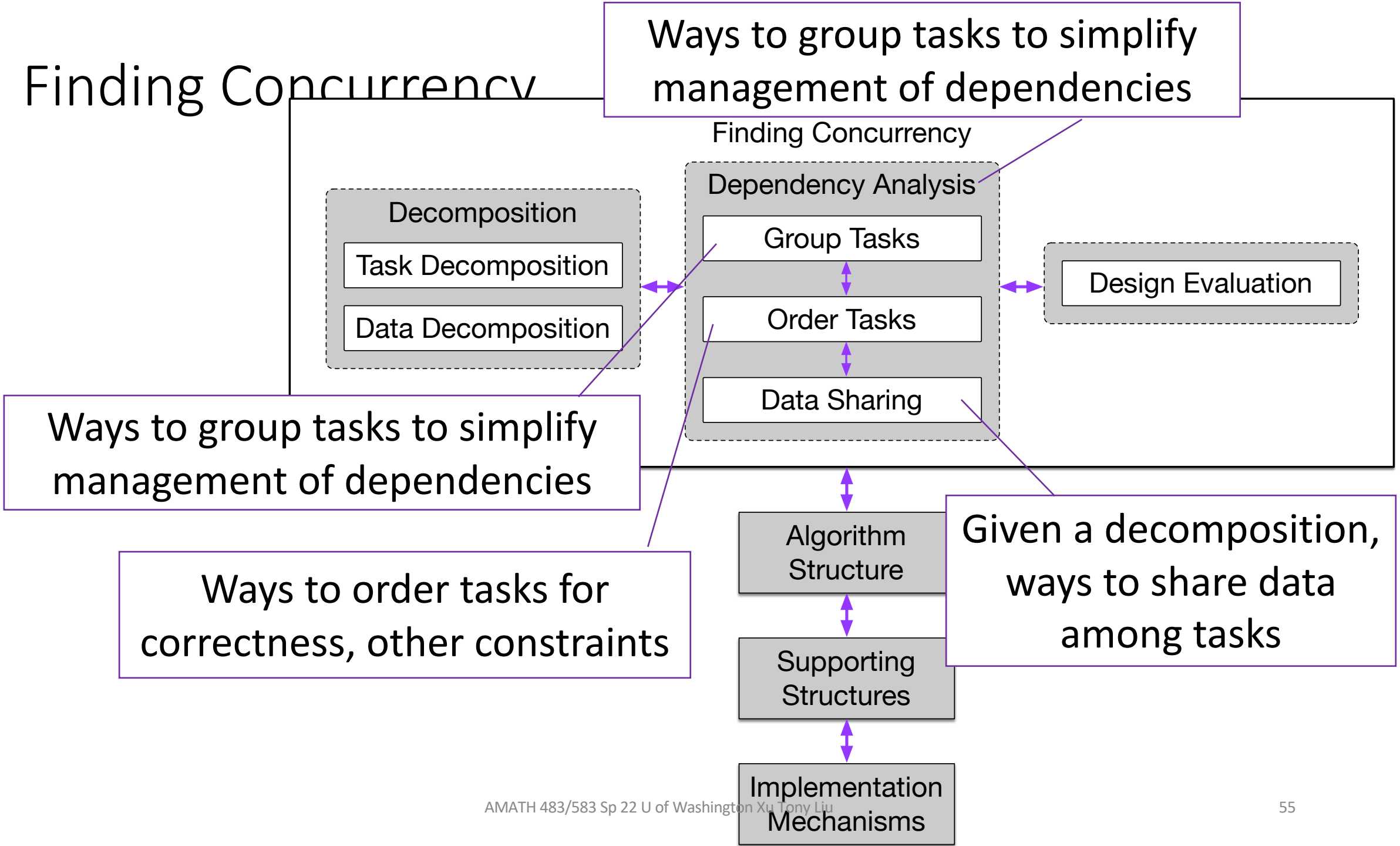
Decompose problem into pieces that can execute concurrently

Into tasks that can execute concurrently

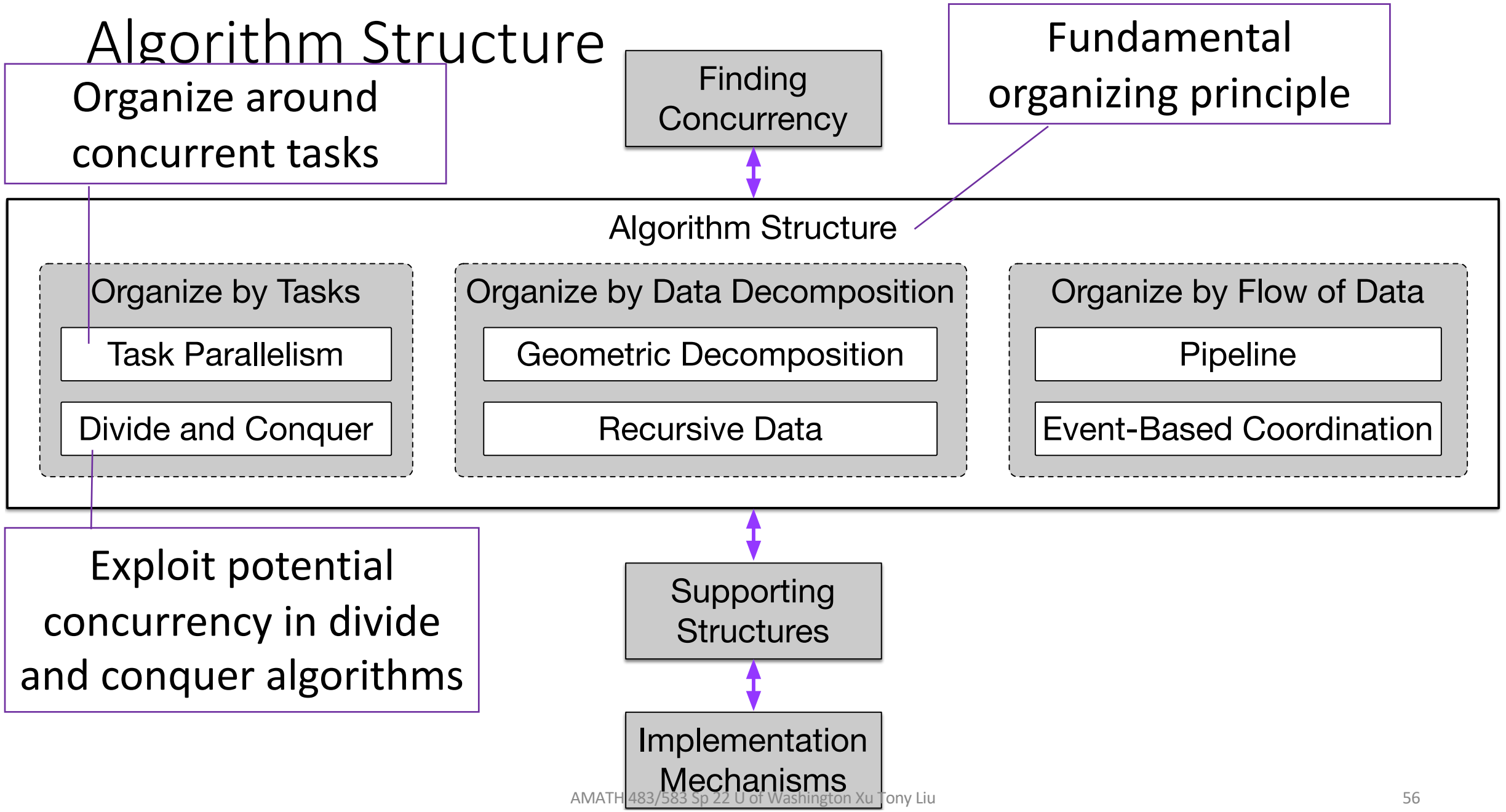
Units that can be operated on (relatively) independently



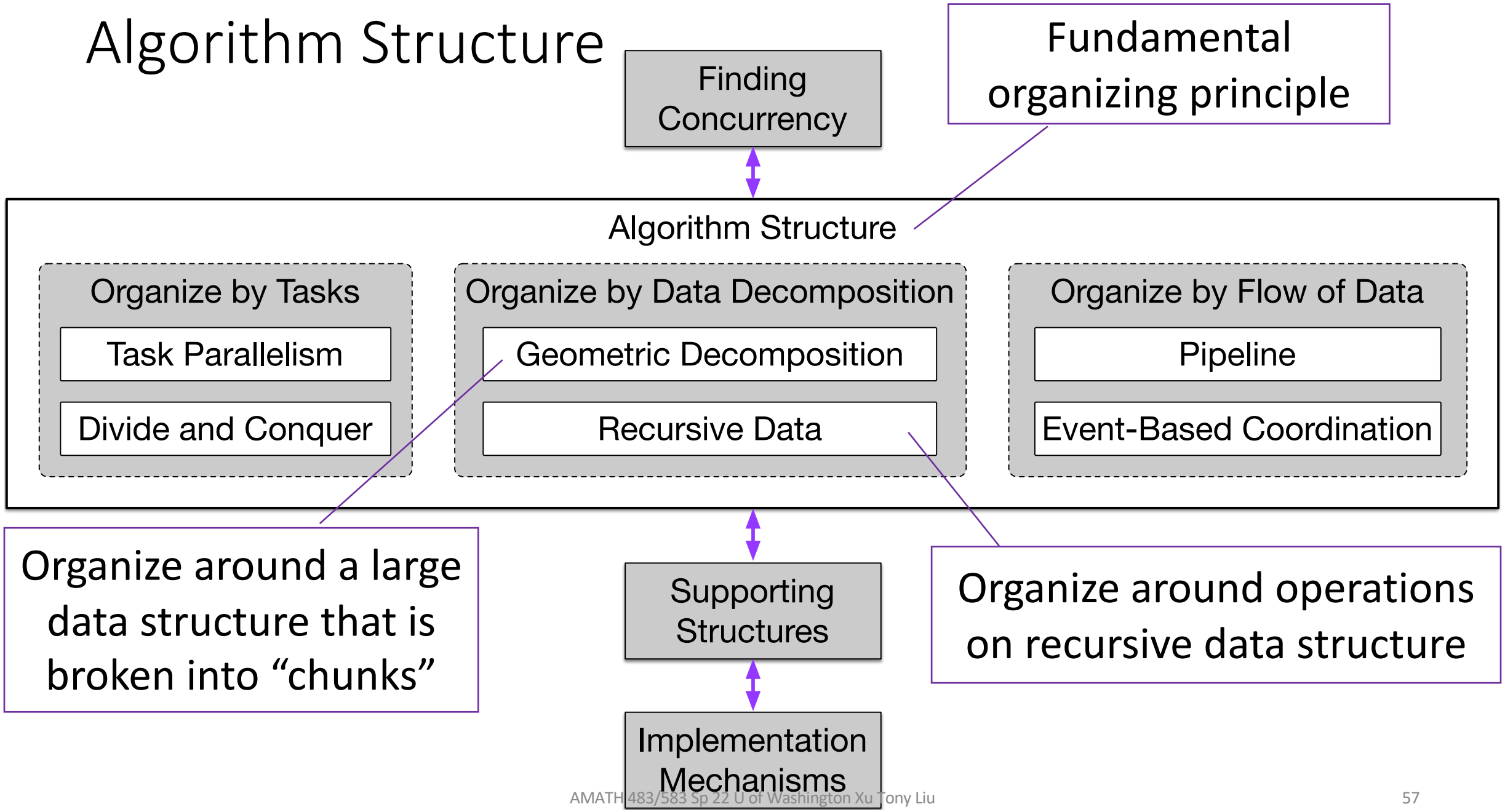
Finding Concurrency



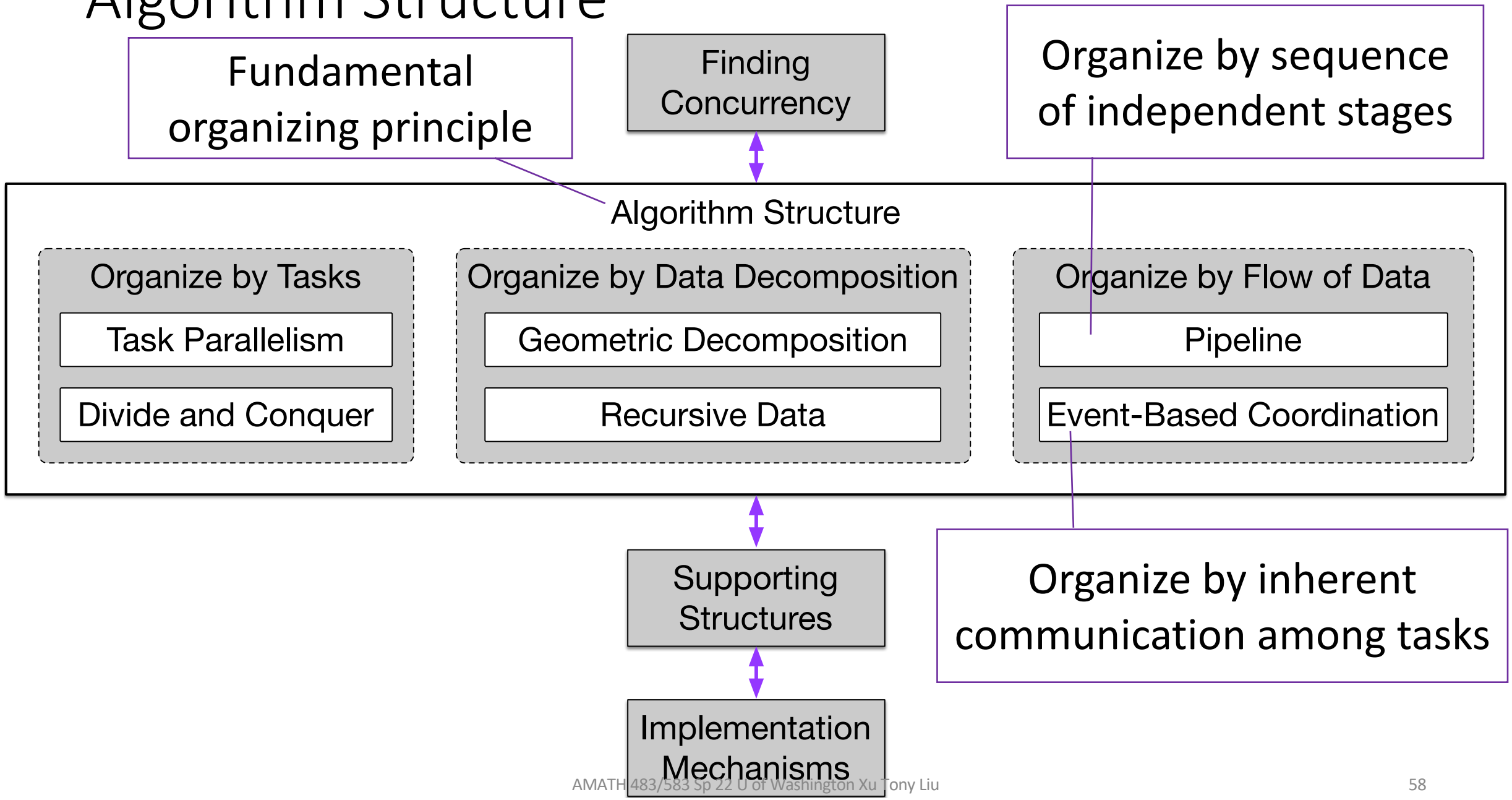
Algorithm Structure



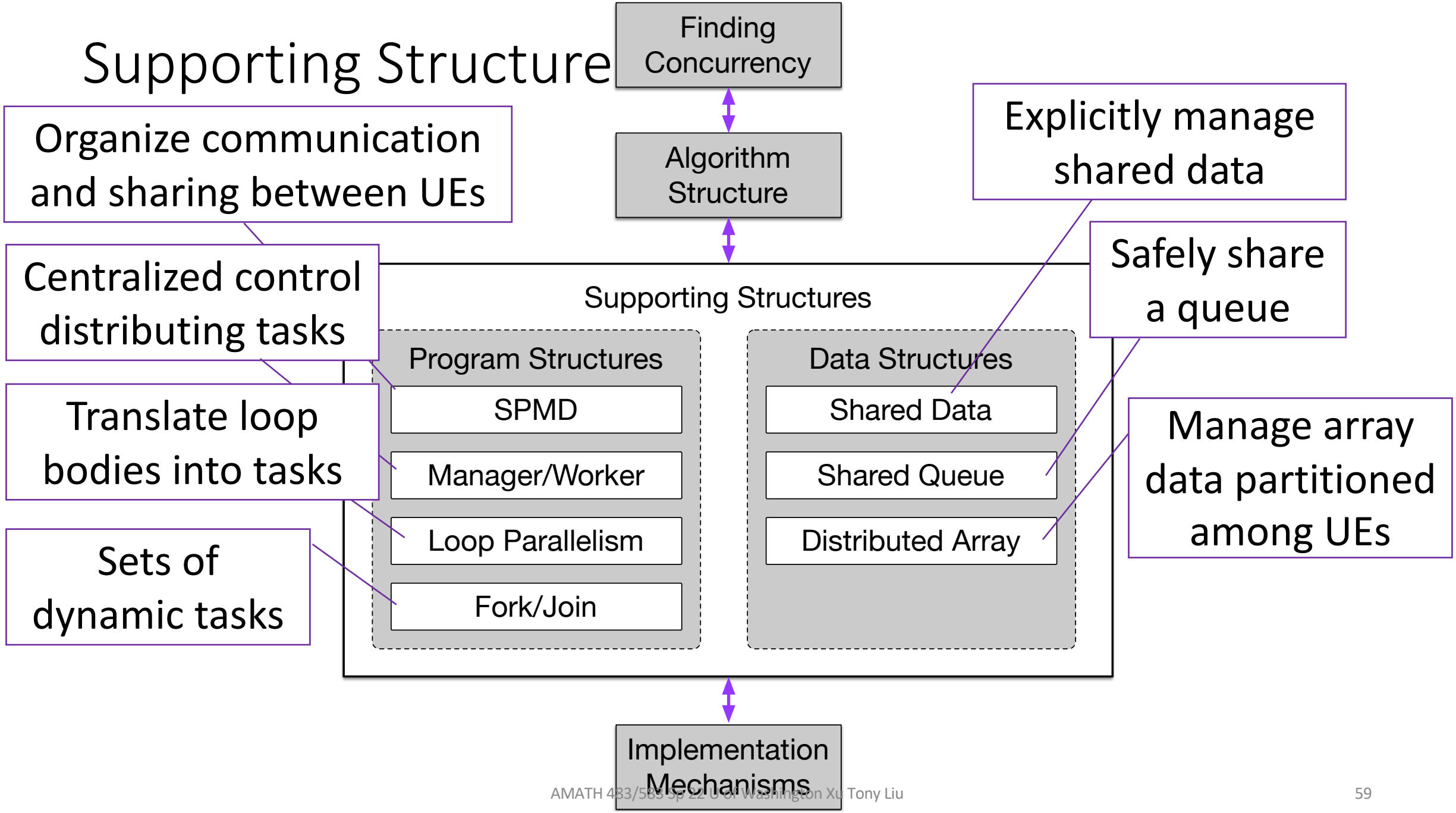
Algorithm Structure



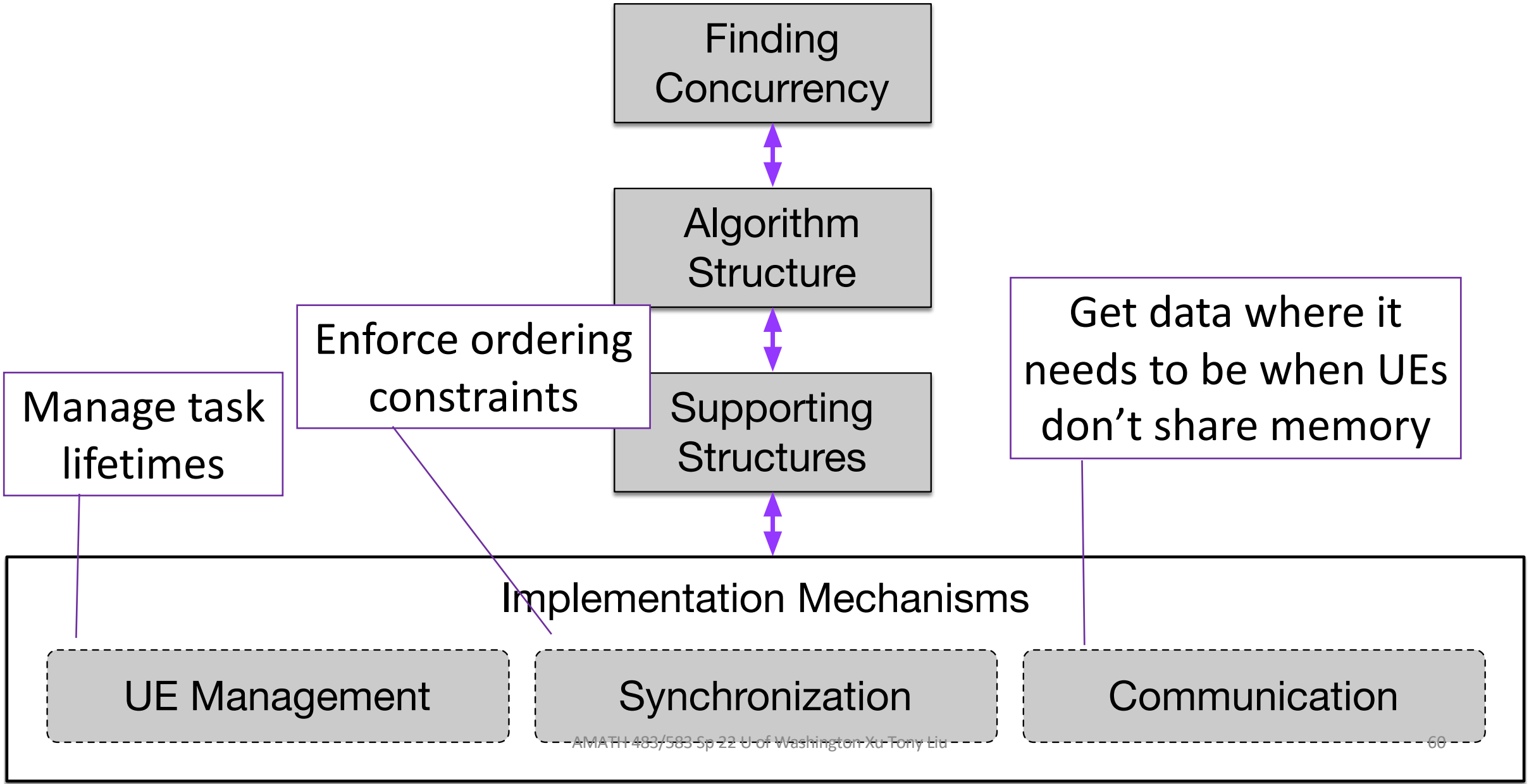
Algorithm Structure



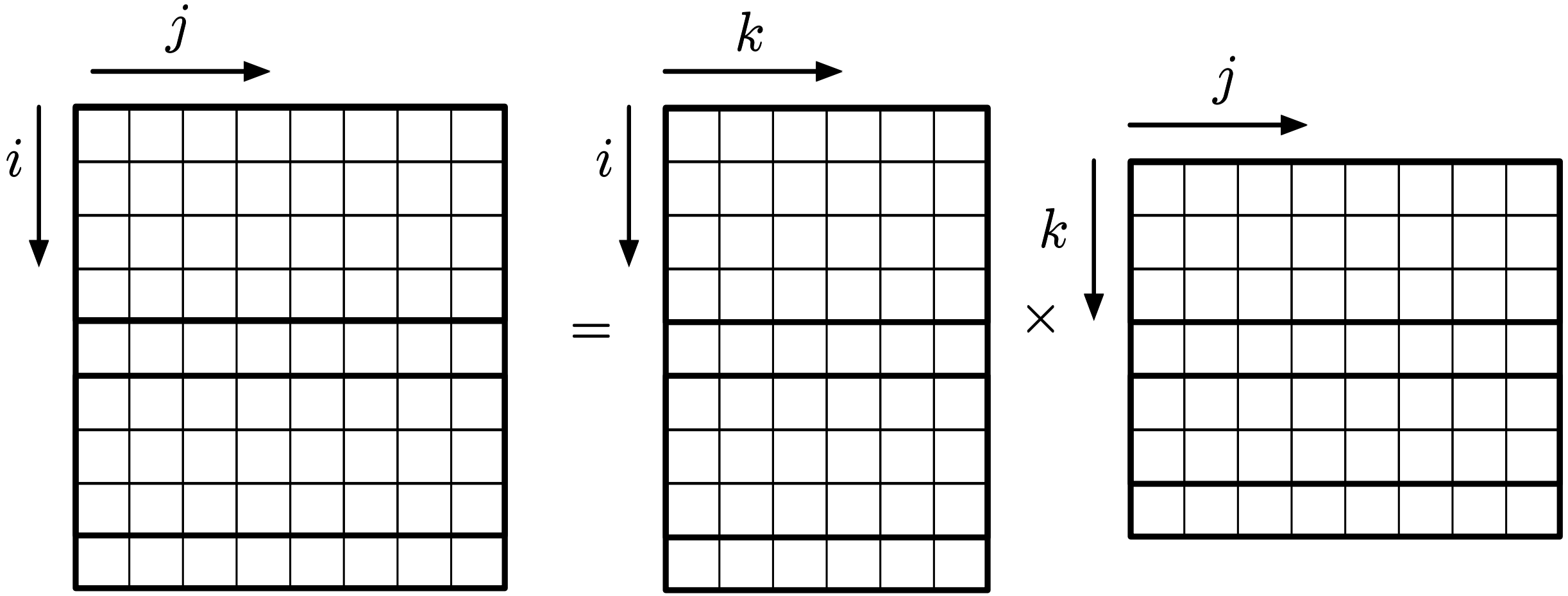
Supporting Structure



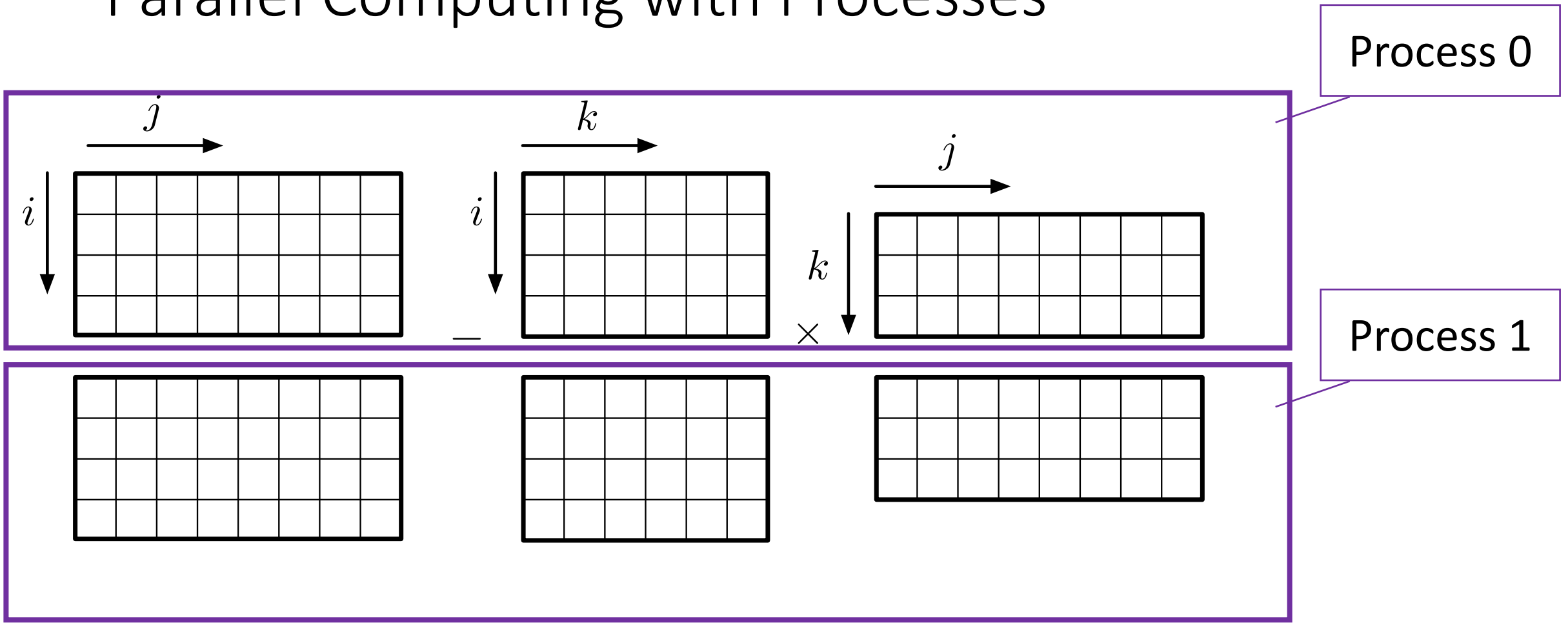
Implementation Mechanisms



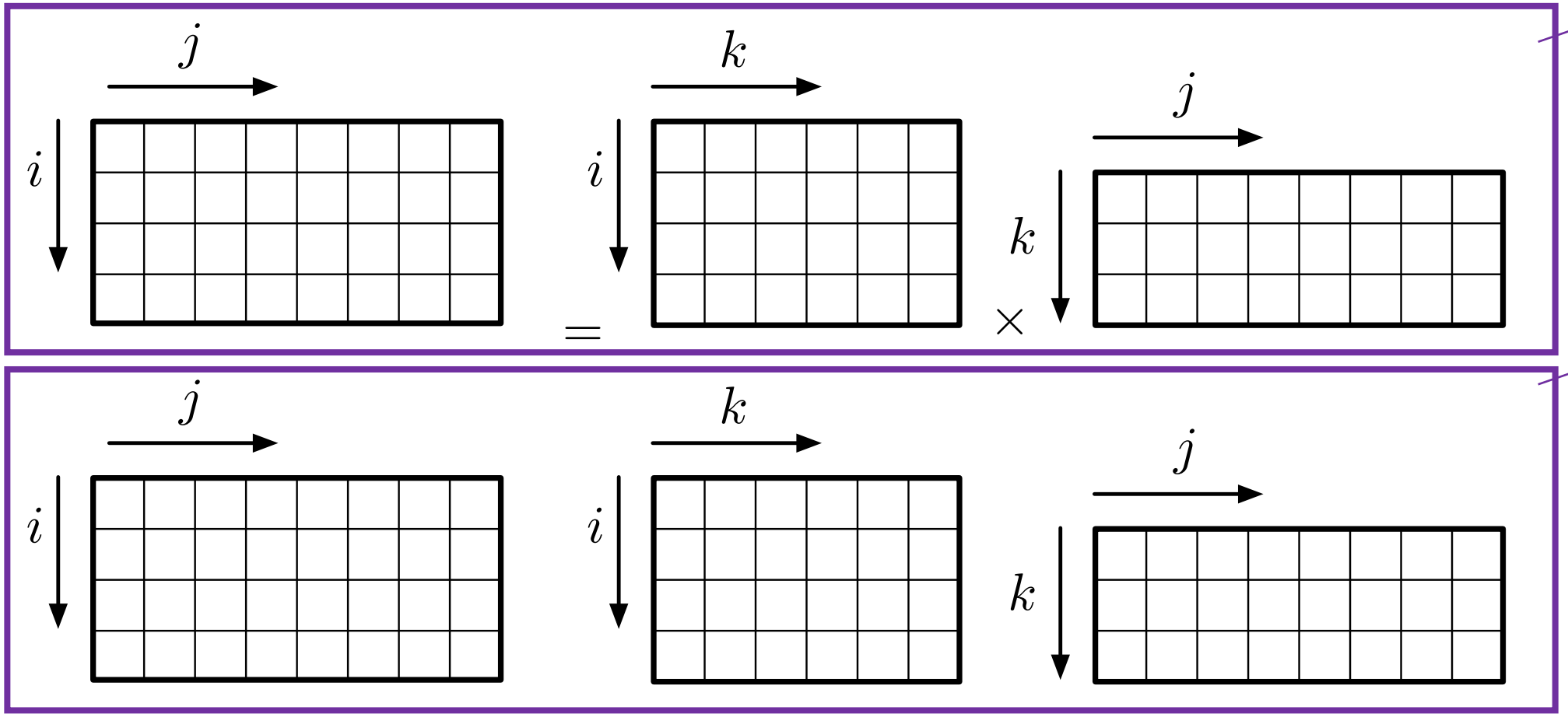
Parallel Computing with Processes



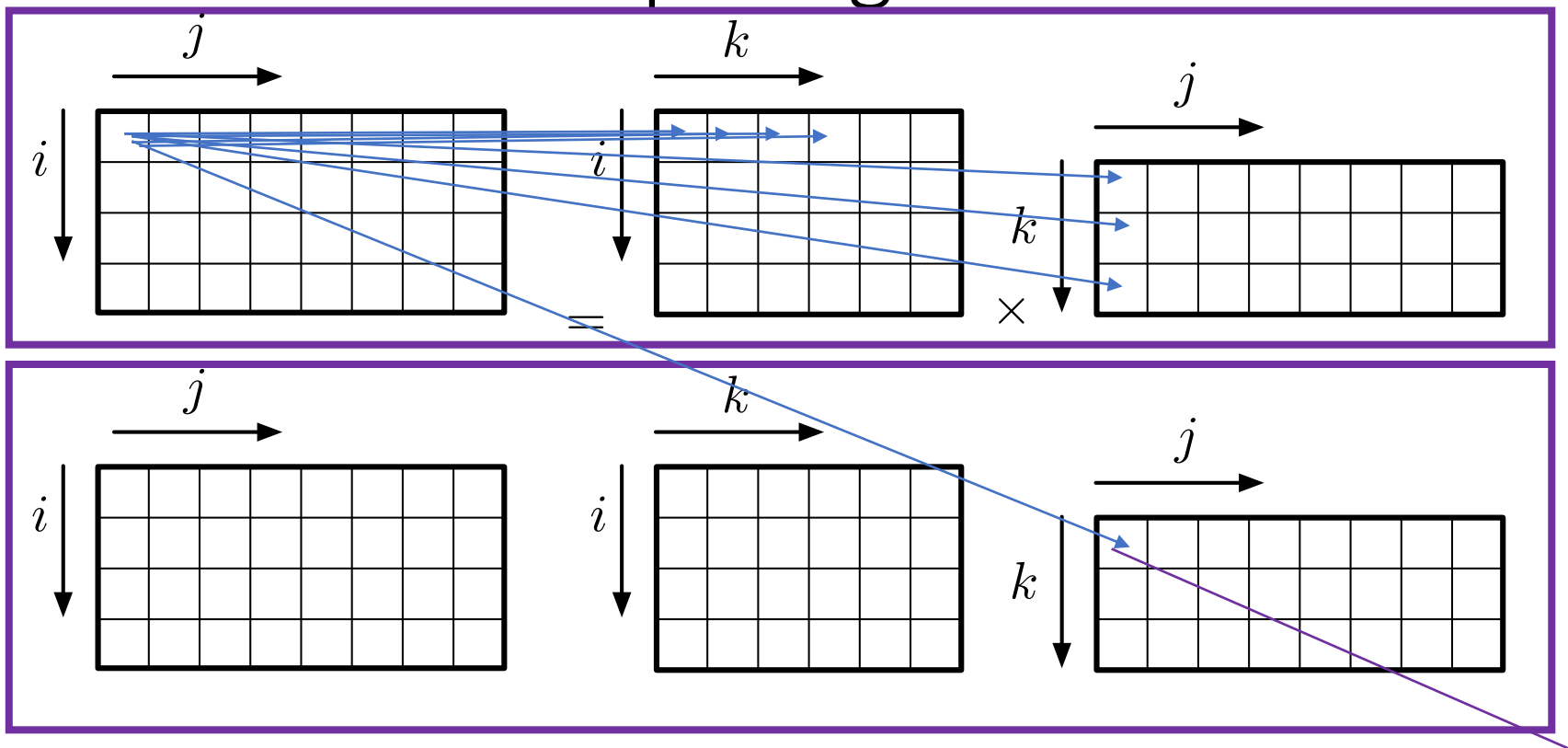
Parallel Computing with Processes



Parallel Computing with Processes



Parallel Computing with Processes



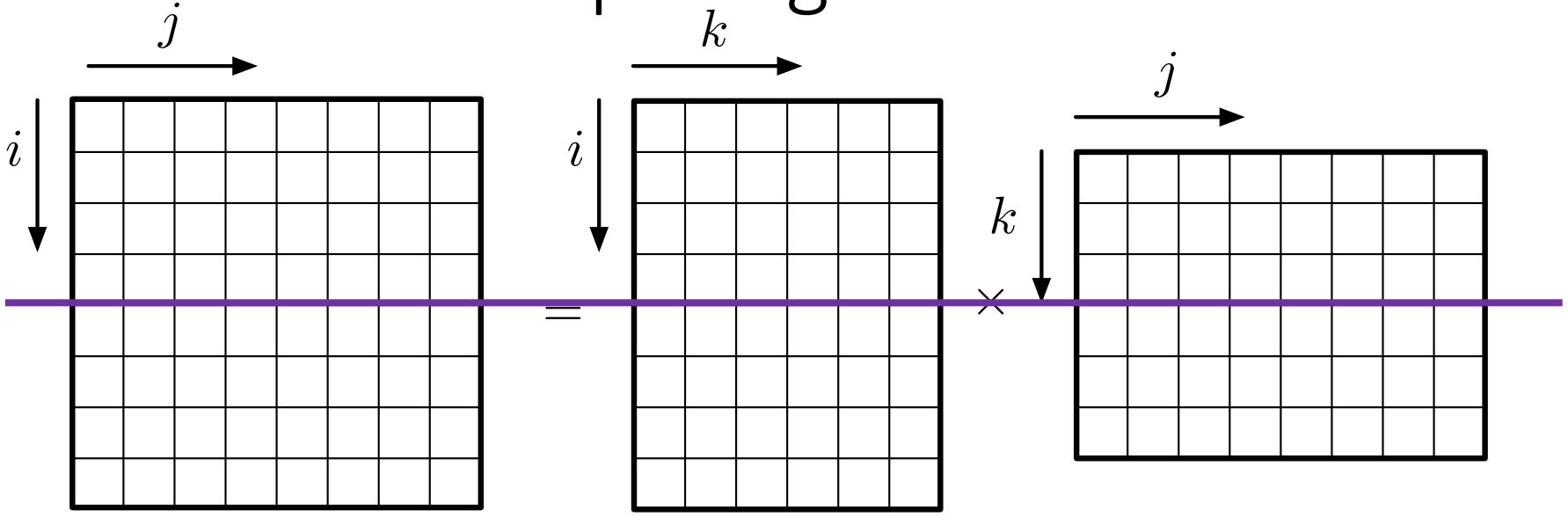

```
for (int i = 0; i < A.numRows(); ++i)
  for (int j = 0; j < B.numCols(); ++
    for (int k = 0; k < A.numCols();
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
for (int i = 0; i < A.numRows(); ++i)
  for (int j = 0; j < B.numCols(); ++
    for (int k = 0; k < A.numCols();
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
for (int k = 0; k < A.numCols(); ++k) {
  C(i,j) += A(i,k) * B(k,j);
}
```

Can't index from different process b/c different address space

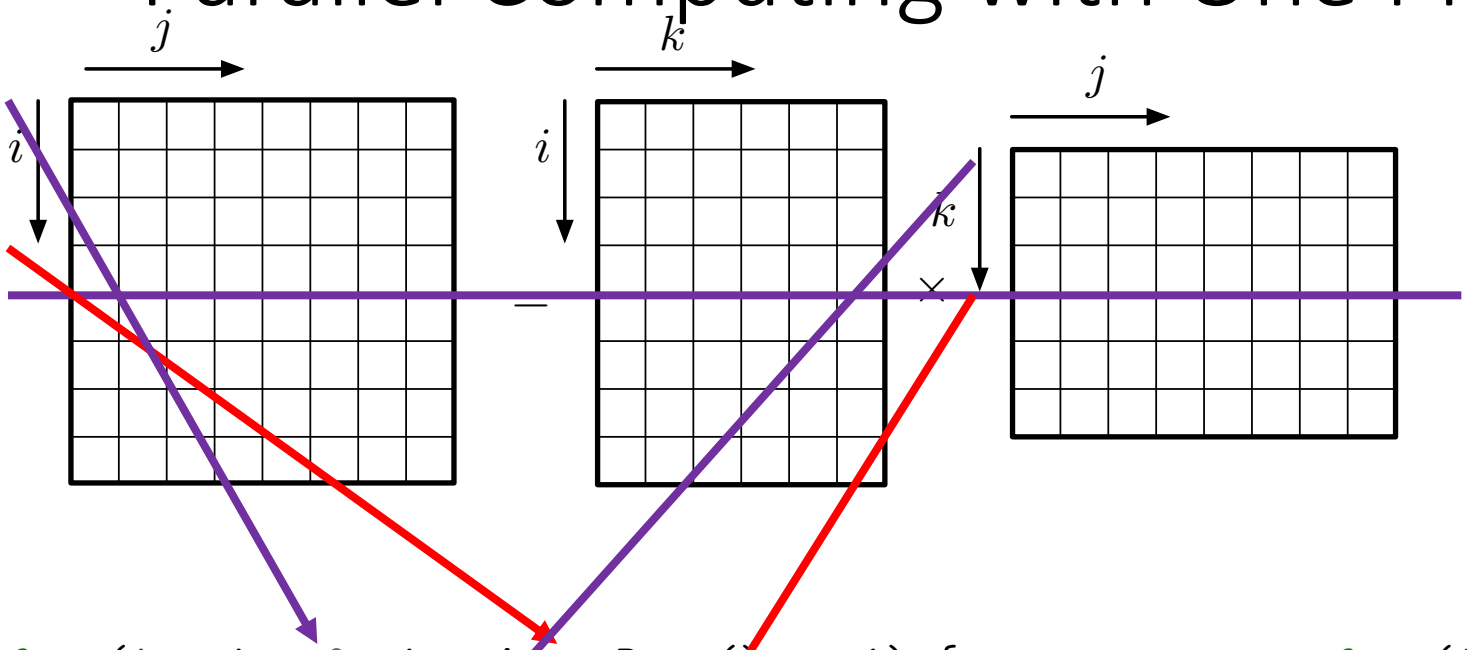
Parallel Computing with One Process




```
for (int i = 0; i < A.numRows(); ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = 0; k < A.numCols(); ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
for (int i = 0; i < A.numRows(); ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = 0; k < A.numCols(); ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

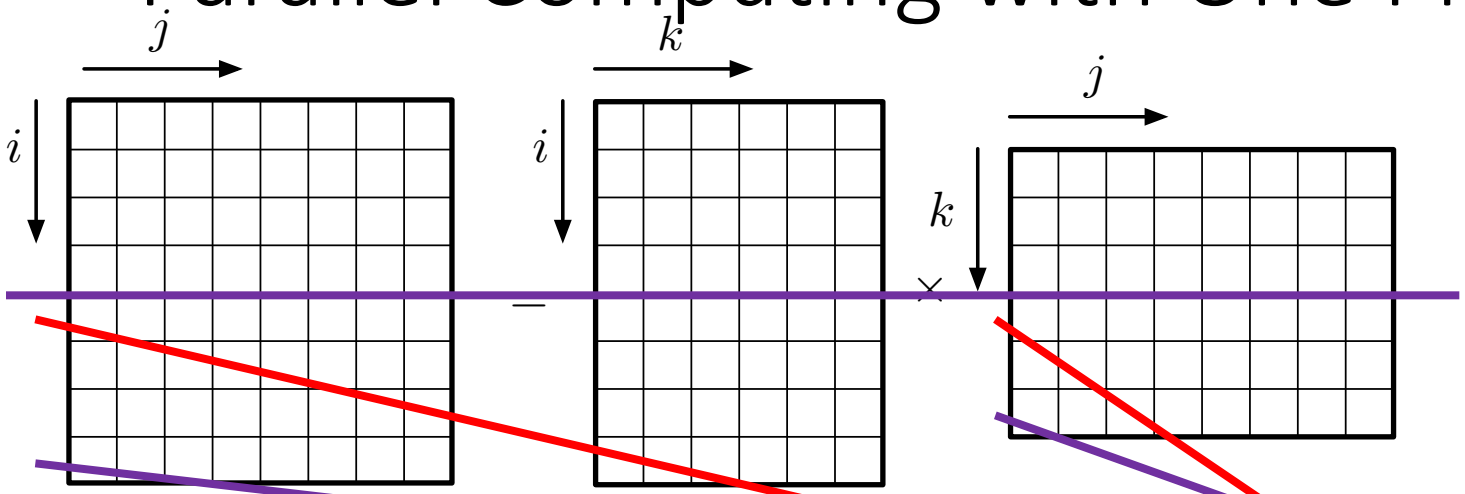
Parallel Computing with One Process




```
for (int i = 0; i < A.numRows(); ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = 0; k < A.numCols(); ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
for (int i = 0; i < A.numRows(); ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = 0; k < A.numCols(); ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

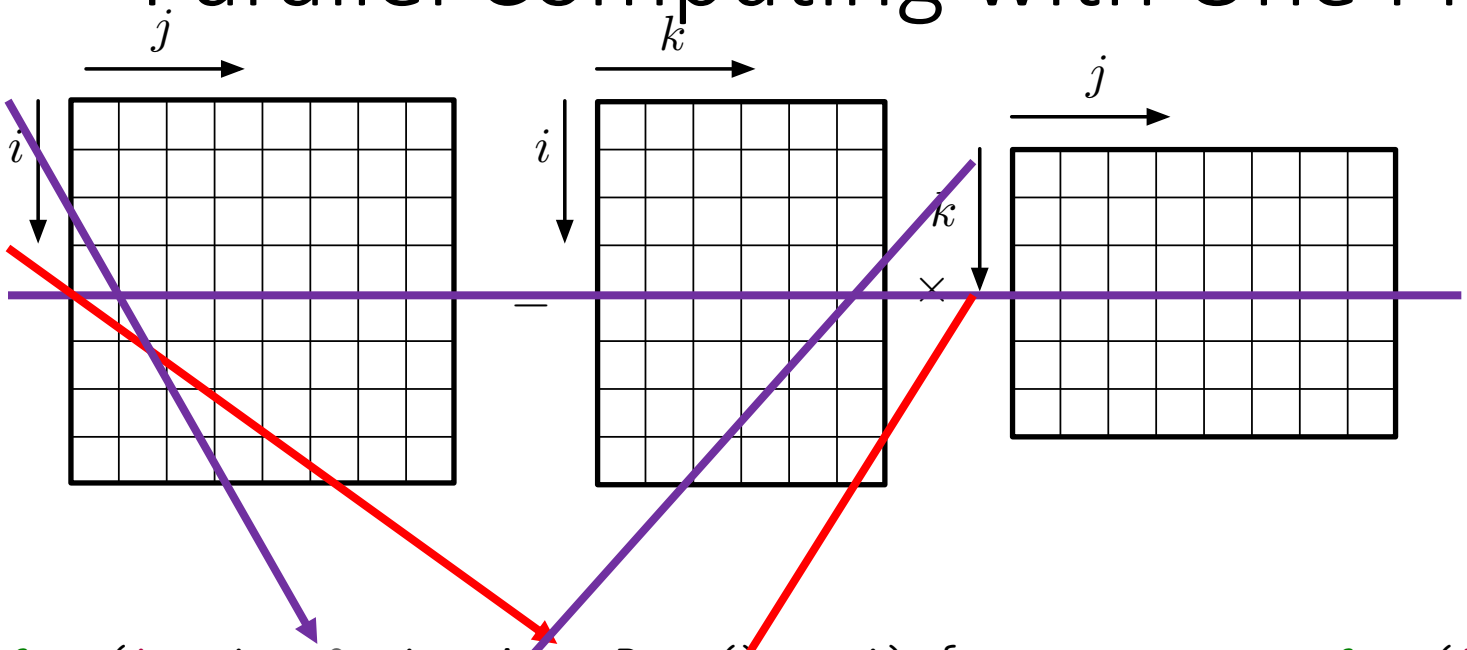
Parallel Computing with One Process




```
for (int i = 0; i < A.numRows(); ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = 0; k < A.numCols(); ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
for (int i = 0; i < A.numRows(); ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = 0; k < A.numCols(); ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

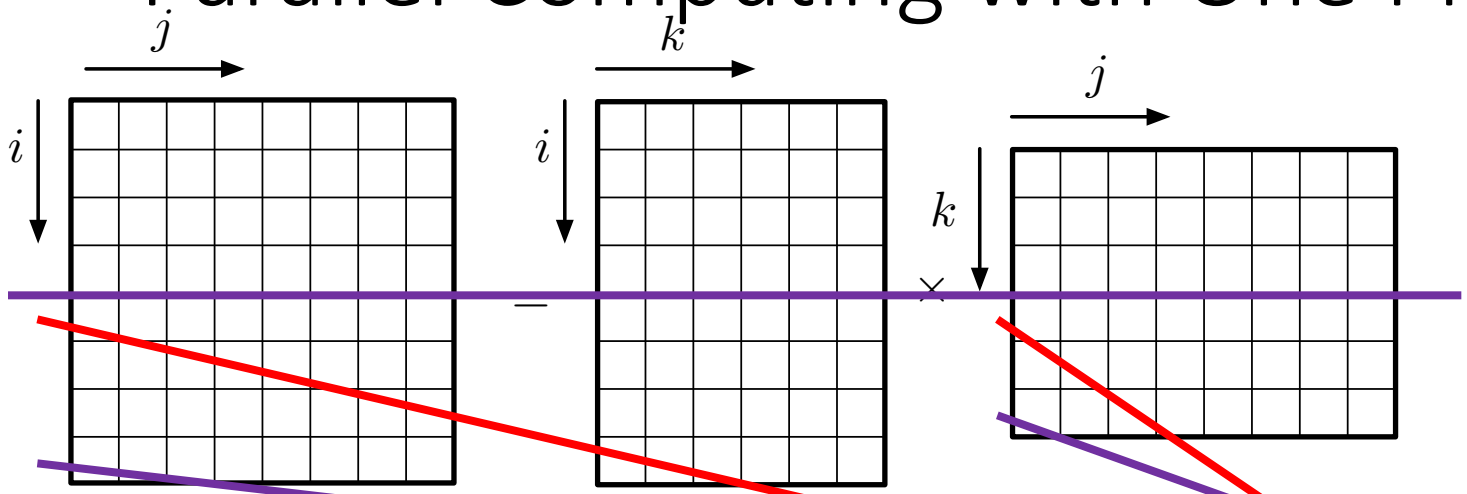
Parallel Computing with One Process




```
for (int i = 0; i < A.numRows(); ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = 0; k < A.numCols(); ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
for (int i = 0; i < A.numRows(); ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = 0; k < A.numCols(); ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

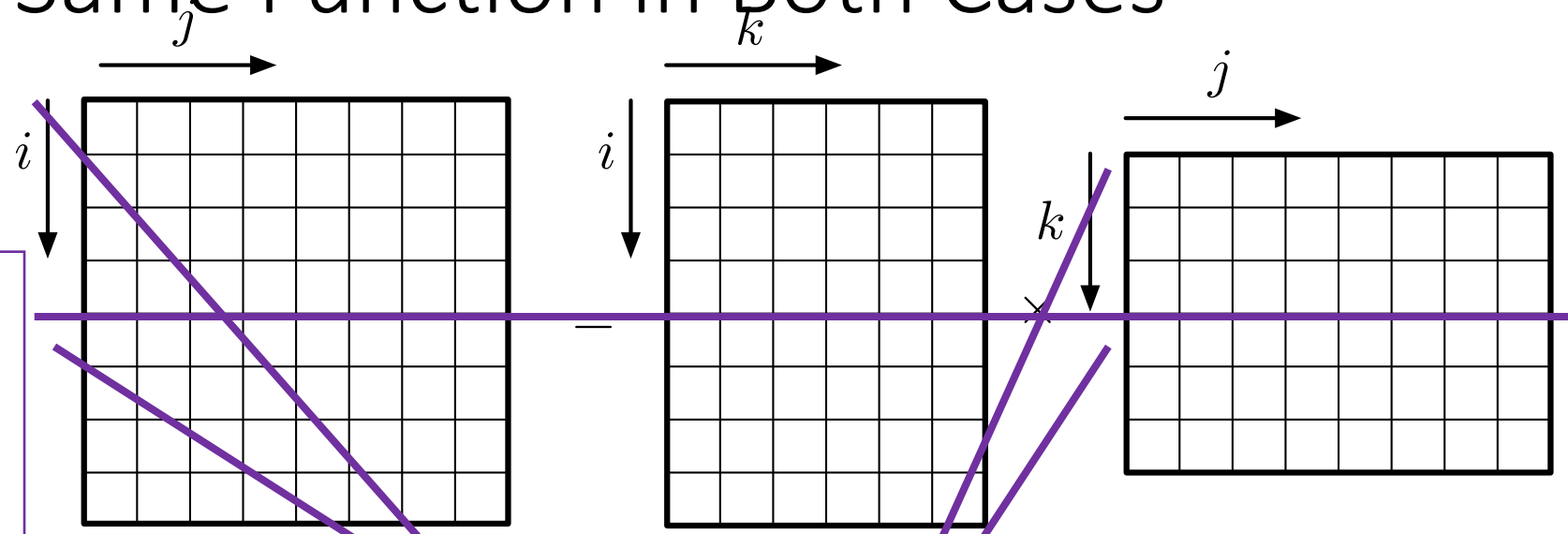
Parallel Computing with One Process




```
for (int i = 0; i < A.numRows()/2; ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = 0; k < A.numCols()/2; ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
for (int i = 0; i < A.numRows(); ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = 0; k < A.numCols(); ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

Use Same Function in Both Cases

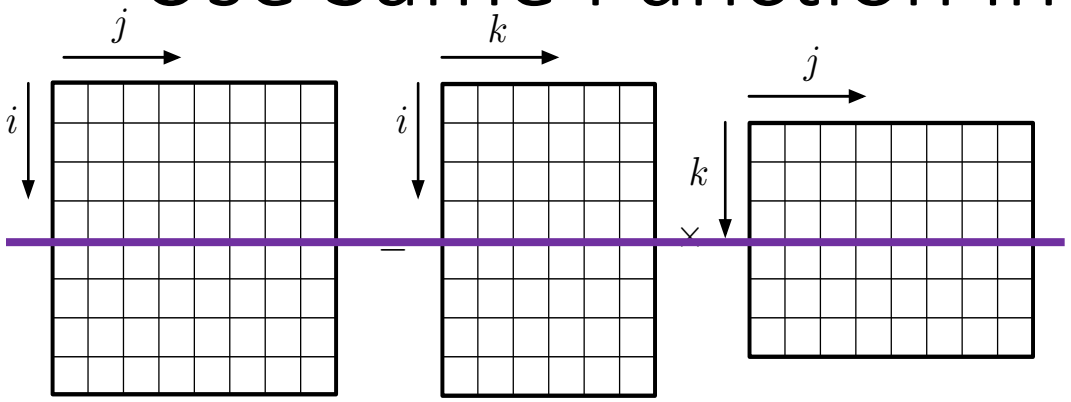


Still need to run two separate instances

Need to run them in ***parallel*** to get improved performance

```
for (int i = iStart; i < iStart + A.numRows()/2; ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = kStart; k < kStart + A.numCols()/; ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

Use Same Function in Both Cases



Run this

```
for (int i = iStart; i < iStart + A.numRows()/2; ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = kStart; k < kStart + A.numCols()/2; ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
int iStart = 0;
int kStart = 0;
```

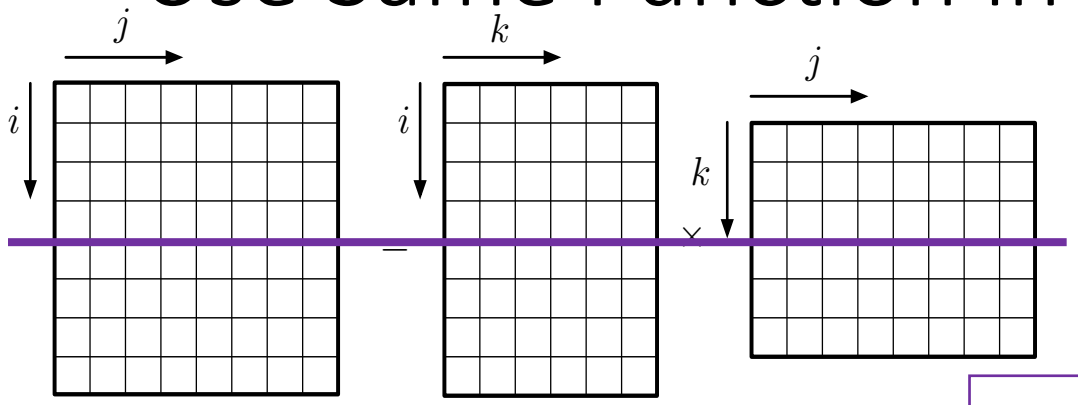
```
for (int i = iStart; i < iStart + A.numRows()/2; ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = kStart; k < kStart + A.numCols()/2; ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
int iStart = A.numRows()/2;
int kStart = A.numCols()/2;
```

Improved performance?

Then this

Use Same Function in Both Cases



At the same time

2X faster (?)

Run this

And this

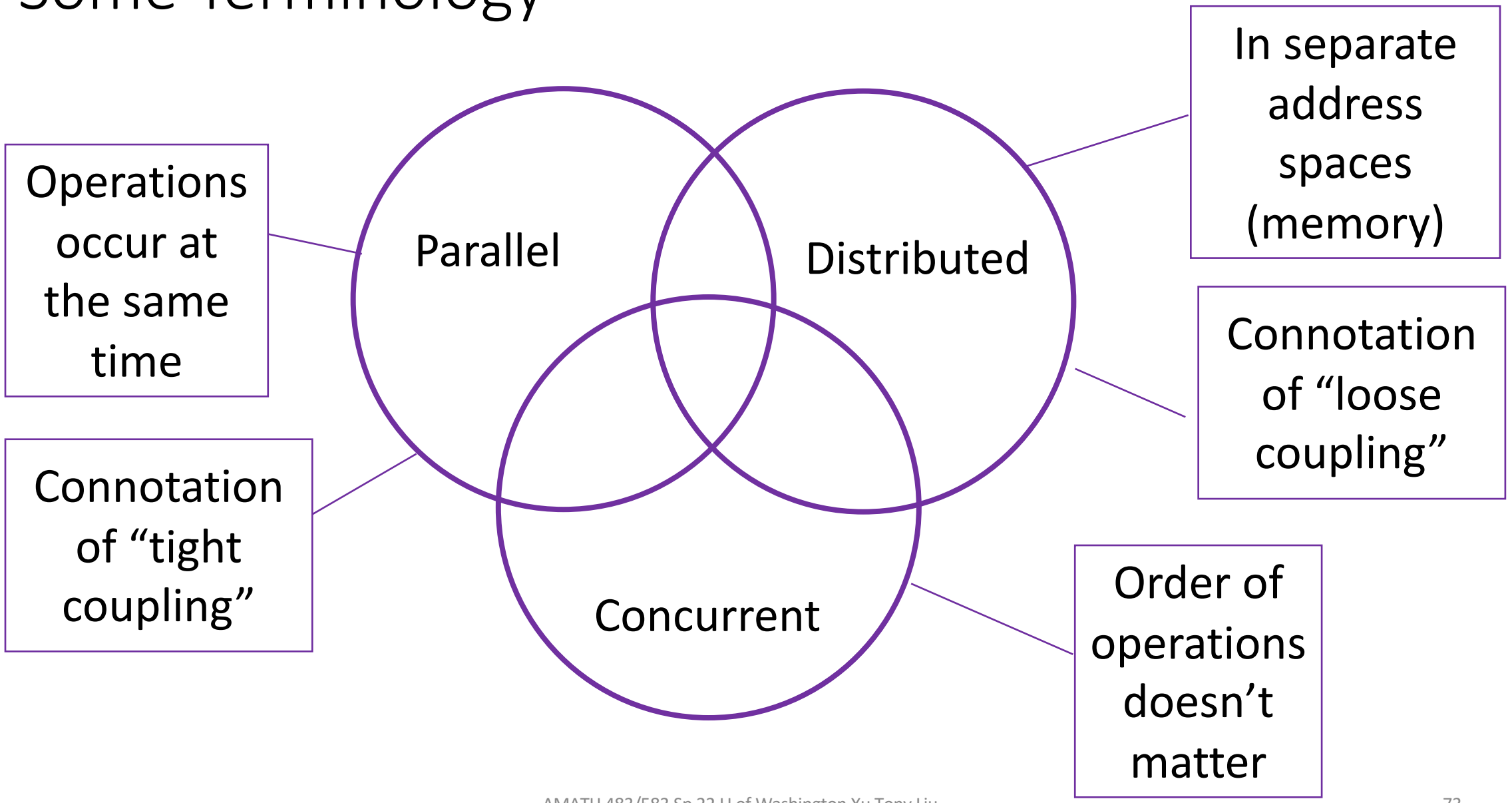
```
for (int i = iStart; i < iStart + A.numRows()/2; ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = kStart; k < kStart + A.numCols()/2; ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
int iStart = 0;
int kStart = 0;
```

```
for (int i = iStart; i < iStart + A.numRows()/2; ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = kStart; k < kStart + A.numCols()/2; ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
int iStart = A.numRows()/2;
int kStart = A.numCols()/2;
```

Some Terminology



Running Things “At the Same Time” vs Running Things At the Same Time

- Historically, threads evolved as a concurrency mechanism, not parallelism

- Enabled OS and processes to do multiple things “at the same time”
  - Running things “at the same time” – concurrency

- Can be used for performance if threads are executed in parallel
  - Running things at the same time (literally) - parallelism
  - Parallelism is the task of running one computation (part of it) simultaneously

Running Things At the Same Time in C++

```cpp
#include <iostream>
#include <thread>
using namespace std;

void sayHello() {
  cout << "Hello World" << endl;
}

int main() {

  thread helloThread(sayHello);

  helloThread.join();

  return 0;
}
```

Pull in thread library

Simple function

Create a thread

Join back to main thread

That runs this function

Multithreading

```
void sayHello(int tnum) {
  cout << "Hello World.  I am thread " << tnum << endl;
}

int main() {
  std::thread tid[16];

  for (int i = 0; i < 16; ++i)
    tid[i] = thread (sayHello, i);

  for (int i = 0; i < 16; ++i)
    tid[i].join();

  return 0;
}
```

Multithreading

```
void sayHello(int tnum) {
  cout << "Hello World.  I am thread " << tnum << endl;
}

int main() {
  std::thread tid[16];

  for (int i = 0; i < 16; ++i)
    tid[i] = thread (sayHello, i);

  for (int i = 0; i < 16; ++i)
    tid[i].join();

  return 0;
}
```

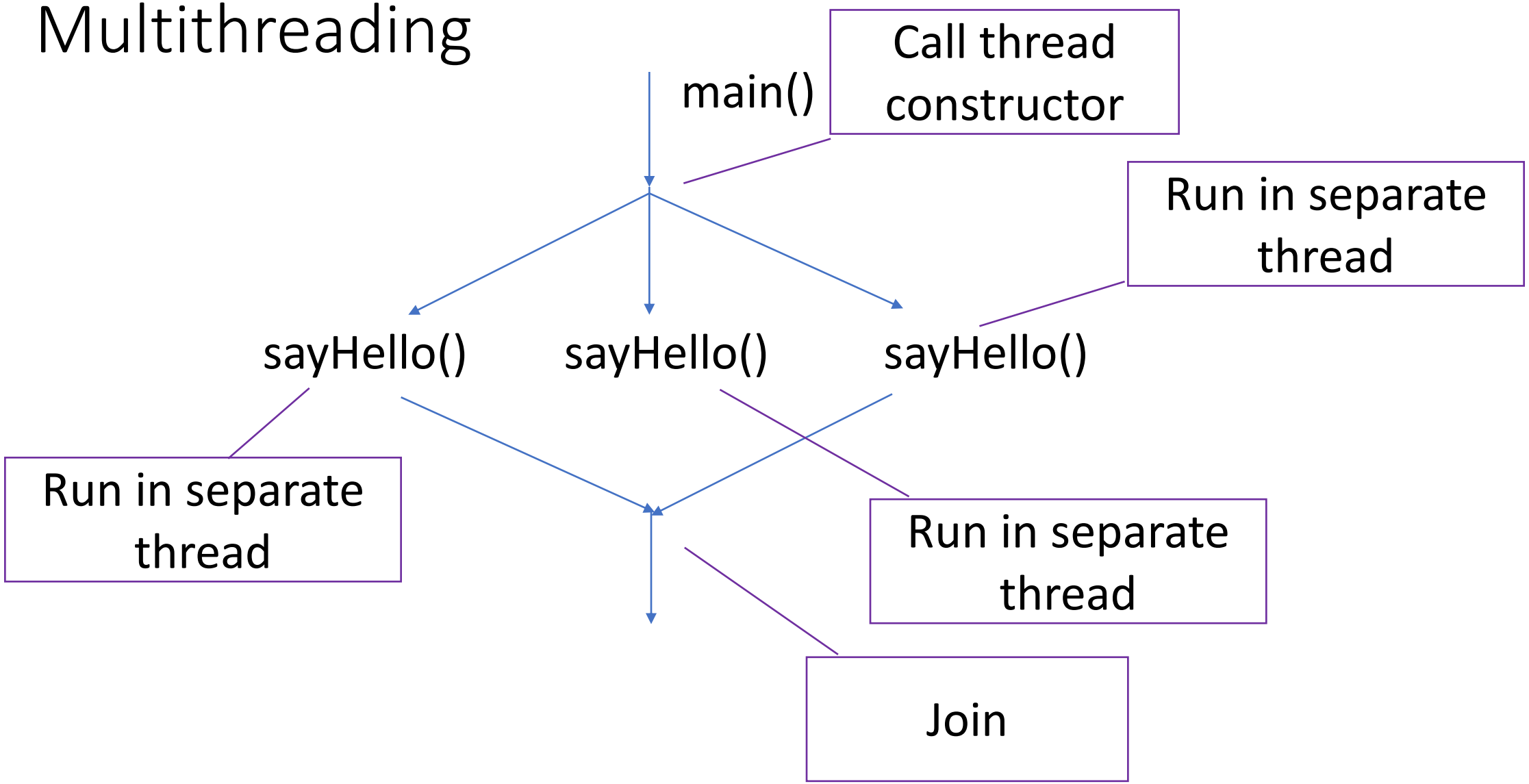
Program output

```
$ ./a.out
Hello World.  I am thread Hello World.  I am thread Hello
World.  I am thread Hello World.  I am thread Hello World.  I
am thread Hello World.  I am thread Hello World.  I am
thread Hello World.  I am thread Hello World.  I am thread
02Hello World.  I am thread Hello World.  I am thread 13Hello
World.  I am thread 5Hello World.  I am thread Hello World.  I
am thread 6Hello World.  I am thread 47Hello World.  I am
thread 8

910

111213
14

15
```

Concurrency?

Parallelism?

Multithreading



Why the Jumbled Output

```
void sayHello(int tnum) {
  cout << "Hello World.  I am thread " << tnum << endl;
}

int main() {
  std::thread tid[16];

  for (int i = 0; i < 16; ++i)
    tid[i] = thread (sayHello, i);

  for (int i = 0; i < 16; ++i)
    tid[i].join();

  return 0;
}
```

Concurrency!

$ ./a.out
Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread 02Hello World. I am thread Hello World. I am thread 13Hello World. I am thread 5Hello World. I am thread Hello World. I am thread 6Hello World. I am thread 47Hello World. I am thread 8

910

111213
14

15

Another Example

```
int value = 0;
```

```
int value = 0;

int main() {
  std::thread tid[16];

  for (int i = 0; i < 16; ++i)
    tid[i] = thread (sayHello, i);

  for (int i = 0; i < 16; ++i)
    tid[i].join();

  cout << "Final value is " << value << endl;

  return 0;
}
```

Example

./a.outHello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread 5302Hello World. I am thread Hello World. I am thread 64Hello World. I am thread Hello World. I am thread 1Hello World. I am thread 789Value is Value is Value is Hello World. I am thread Value is 1011Value is Value is 1213Value is 14Value is Value is Value is 000150Value is Value is 00Value is Value is 0Value is 000Value is 000000

Final value is 1 — ***Not Good!*** — Race condition

We will cover this next lecture

Review

- Process is an abstraction for resource allocation
- Thread is an abstraction for execution
- Concurrency vs Parallelism
- C++ threading library

Thank You!

Creative Commons BY-NC-SA 4.0 License