

AMATH 483/583

High Performance Scientific Computing

Lecture 10:

Processes, Threads, Concurrency, Parallelism

Andrew Lumsdaine
Northwest Institute for Advanced Computing
Pacific Northwest National Laboratory
University of Washington
Seattle, WA

Overview

- Multiple cores
- Concurrency
- Processes
- Threads
- Parallelization strategies
- Correctness

Supercomputers (HPC)



NORTHWEST INSTITUTE for ADVANCED COMPUTING

AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine


Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle
for the U.S. Department of Energy


UNIVERSITY of
WASHINGTON

Schematically

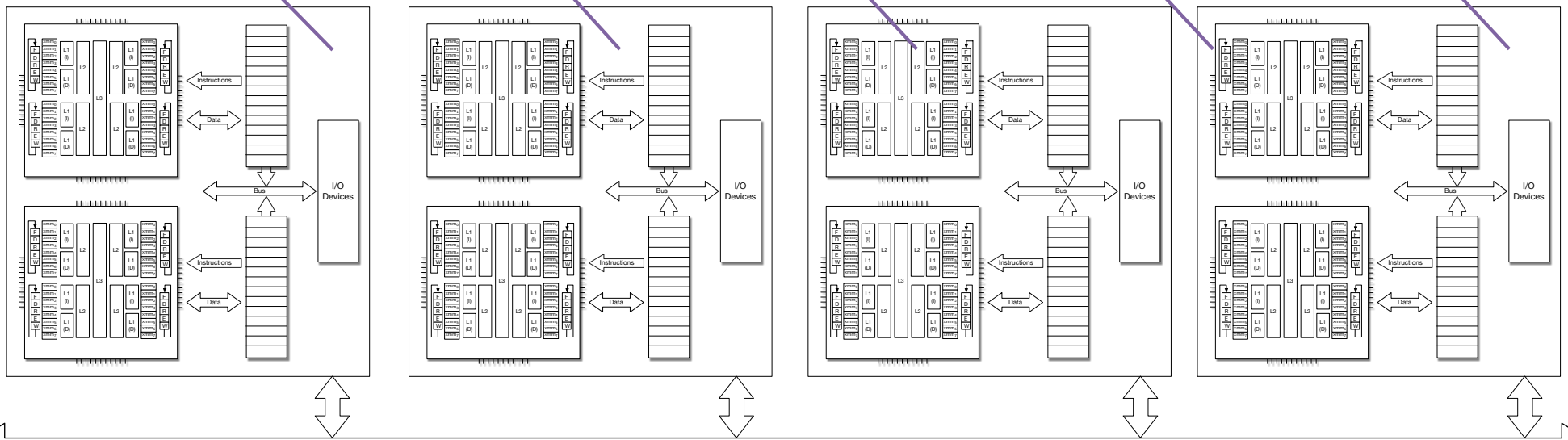
Put sockets
on a blade

Put blades
in a chassis

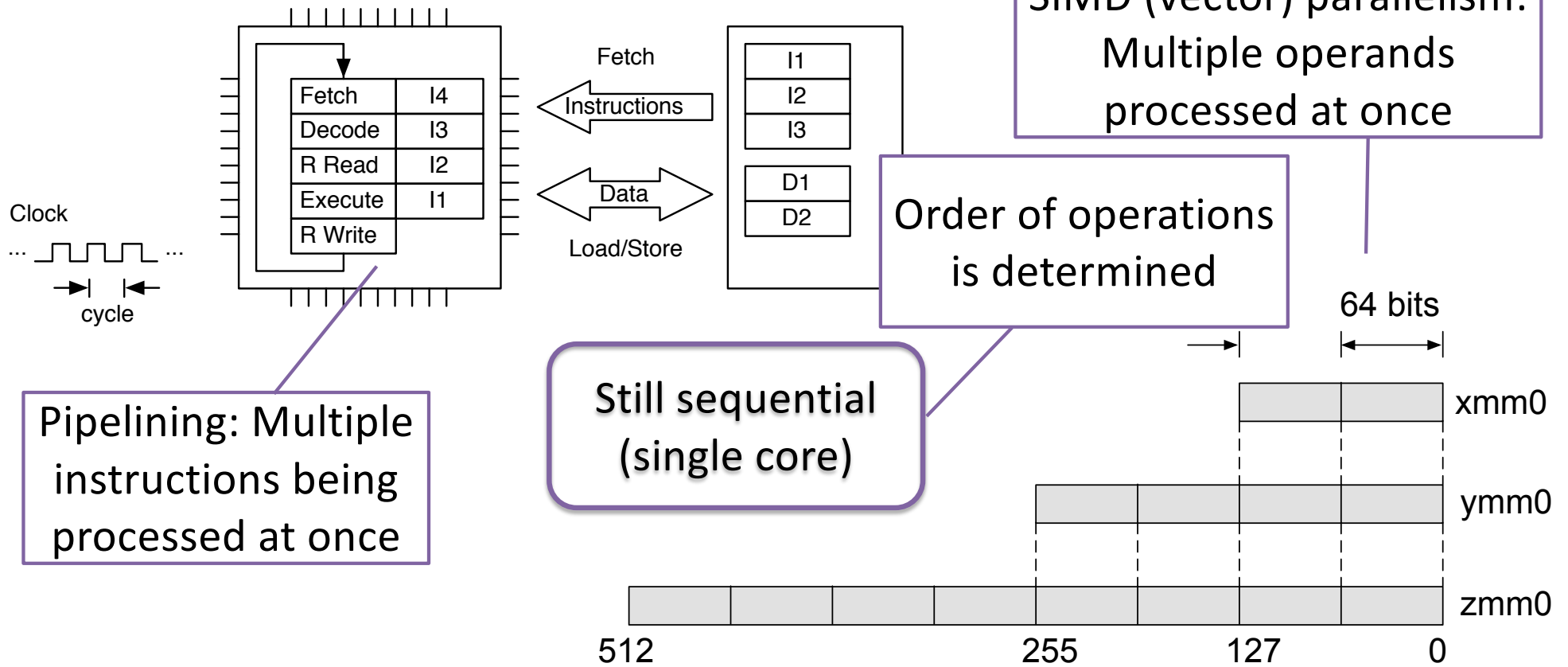
Put chassis
in a rack

Put racks in
a center

Put centers
in the cloud



Parallelism and HPC so far



General Performance Principles

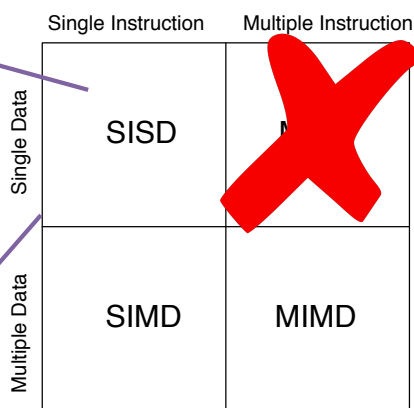
- Work harder
 - Faster core
 - Work smarter
 - Branch predictions, etc
 - Better compilation
 - Better algorithm
 - Better implementation
 - Get help
- Dennard scaling (ended 2005)
- What about this?
- We did this
- Parallel Computing

Flynn's Taxonomy (Aside)

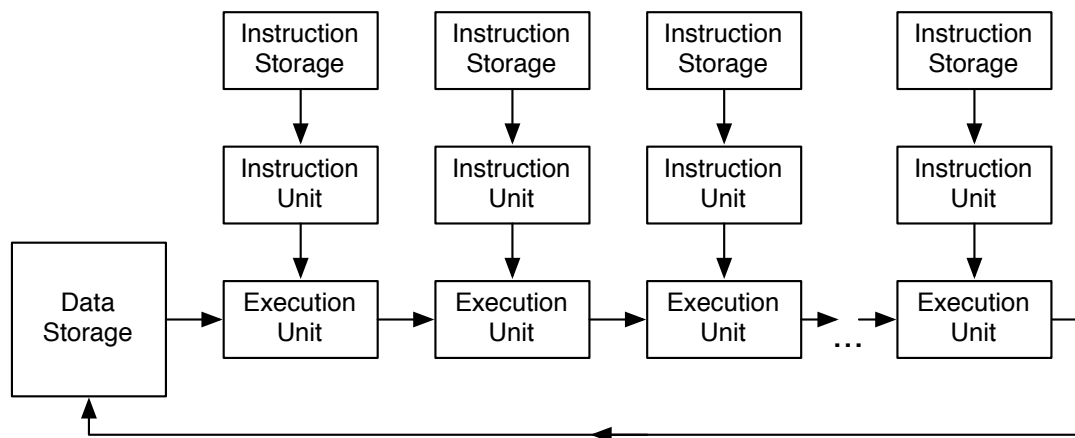
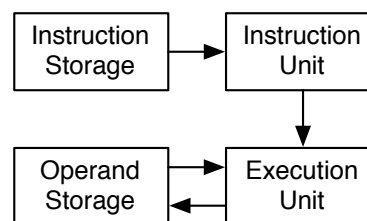
Anyone in HPC must know Flynn's taxonomy

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

Plain old sequential

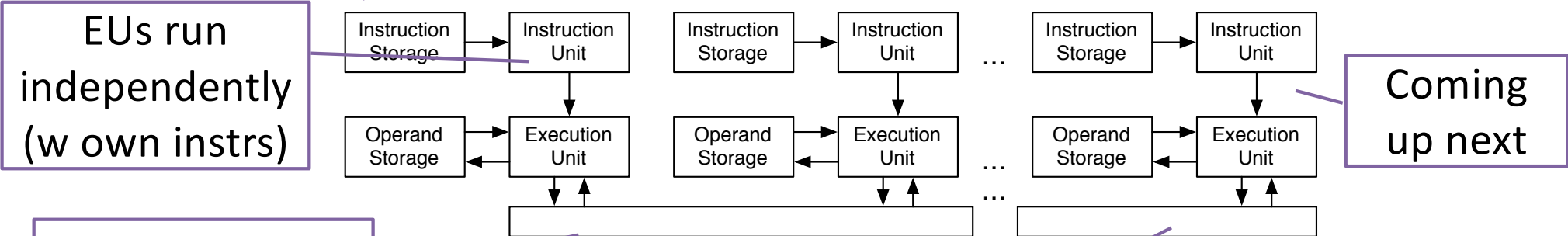
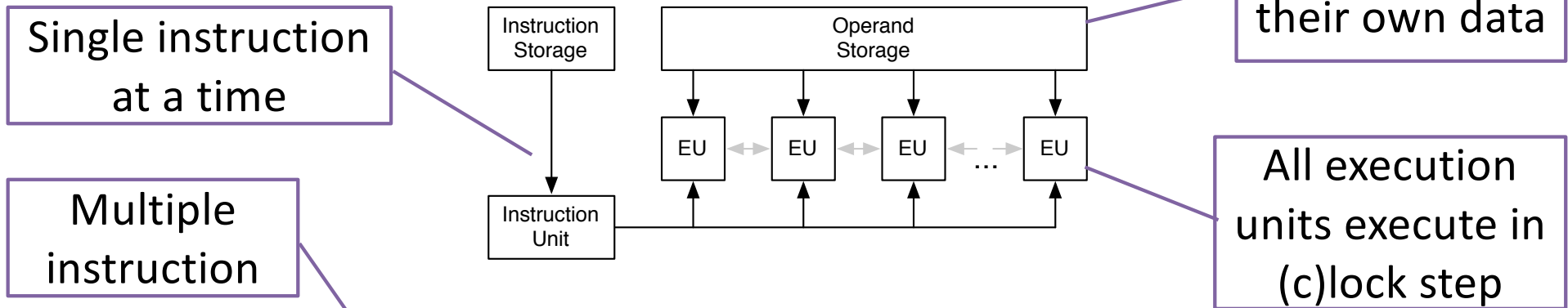


Based on multiplicity of instruction streams, data storage



SIMD and MIMD

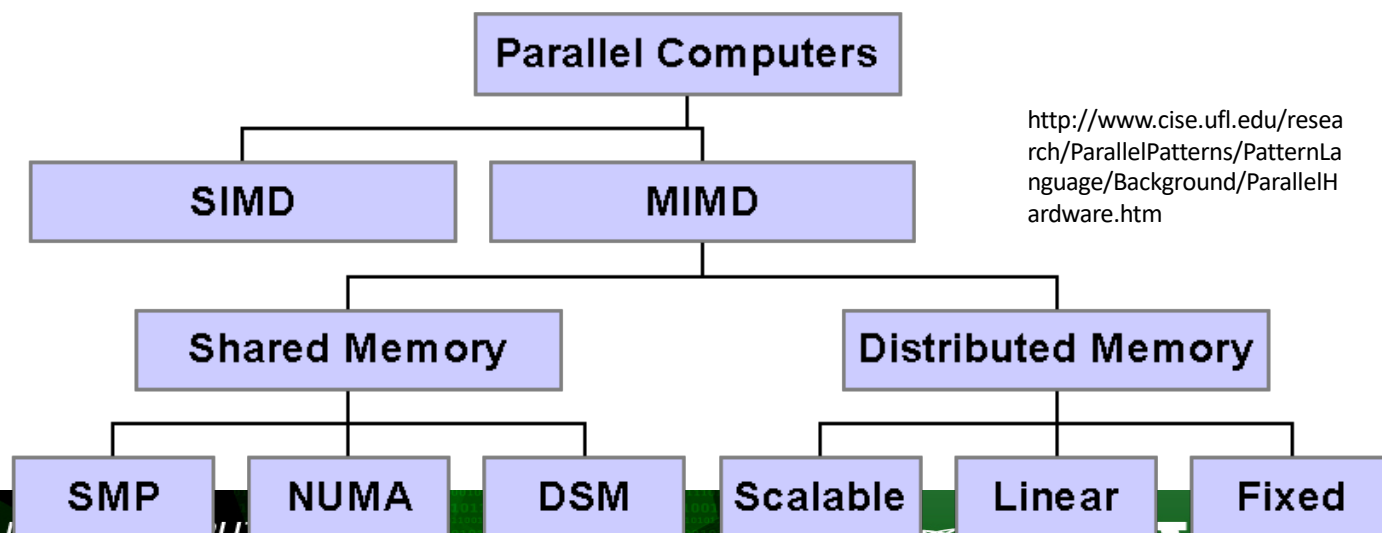
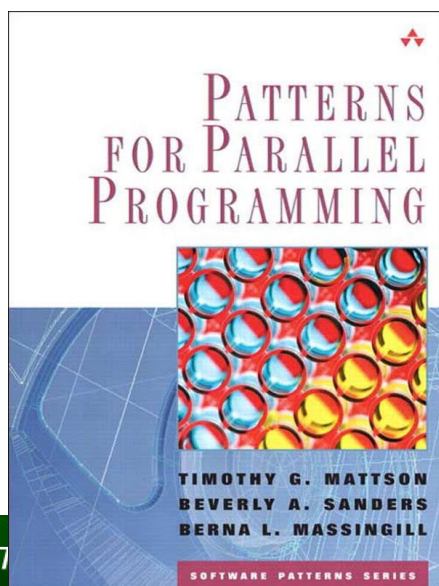
- Two principal parallel computing paradigms (multiple CPUs) But each have their own data



Shared Memory Not Shared

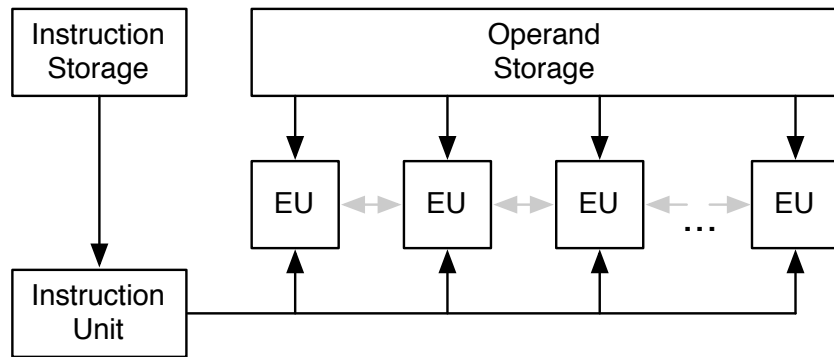
A More Refined (Programmer-Oriented) Taxonomy

- 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



<http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/ParallelHardware.htm>

SIMD in SSE/AVX

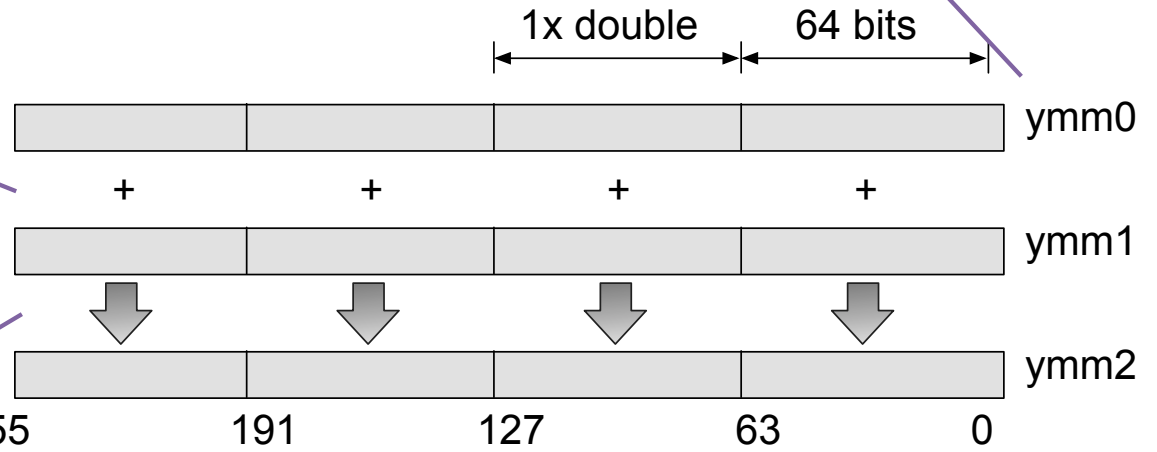


Flynn's original conceptual model

ymm are 256 bit registers

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

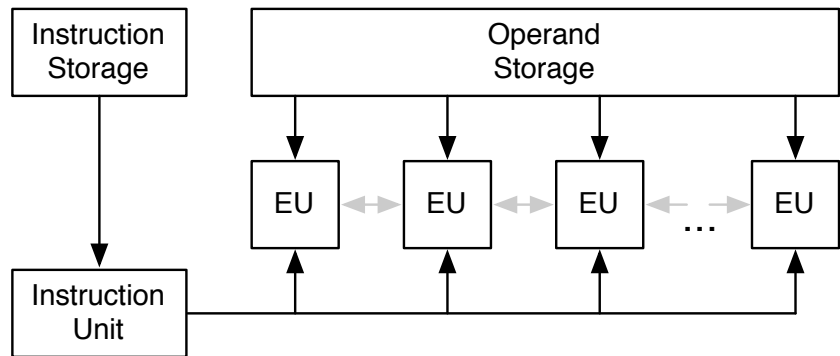
One machine instruction



Adds all four doubles *simultaneously*

SIMD in SSE/AVX

Flynn's original conceptual model

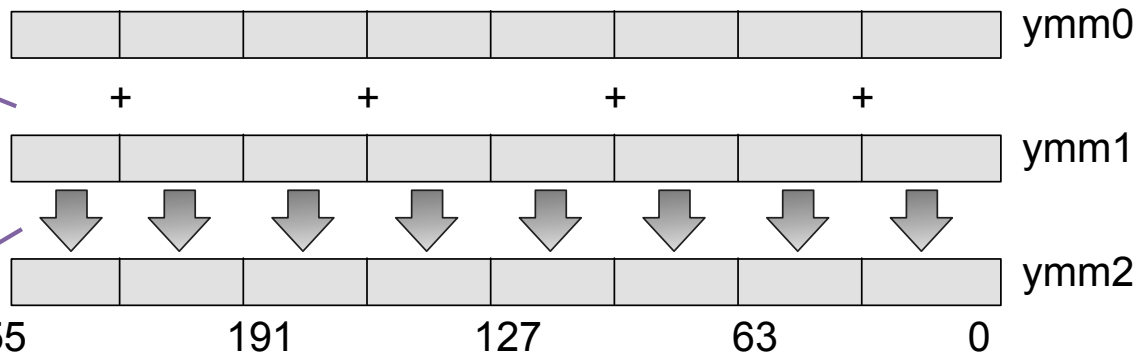


ymm are 256 bit registers



```
vfadd231ps %ymm0, %ymm1, %ymm2
```

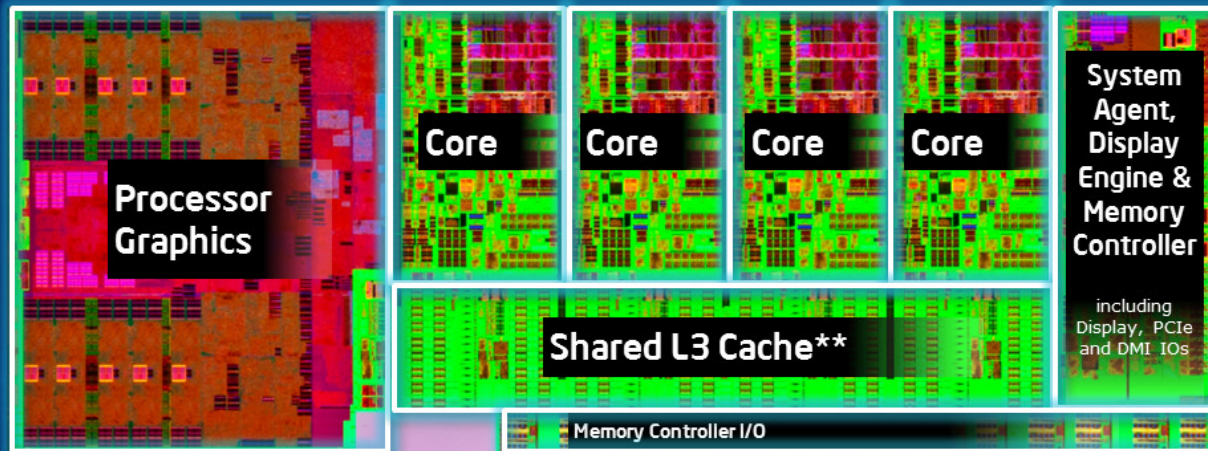
One machine instruction



Adds all eight floats *simultaneously*

Multicore Architecture

4th Generation Intel® Core™ Processor Die Map 22nm Haswell Tri-Gate 3-D Transistors



Quad core die shown above | Transistor count: 1.4Billion | Die size: 177mm²

** Cache is shared across all 4 cores and processor graphics

All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

UNDER EMBARGO UNTIL FURTHER NOTICE

INTEL CONFIDENTIAL



Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries. © 2013 Intel Corporation. Intel, the Intel logo, and other marks contained herein are trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Privately Operated by Battelle for the U.S. Department of Energy.

AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine

NORTHWEST INSTI

W
UNIVERSITY of
WASHINGTON

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

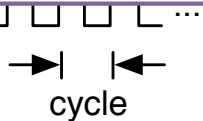
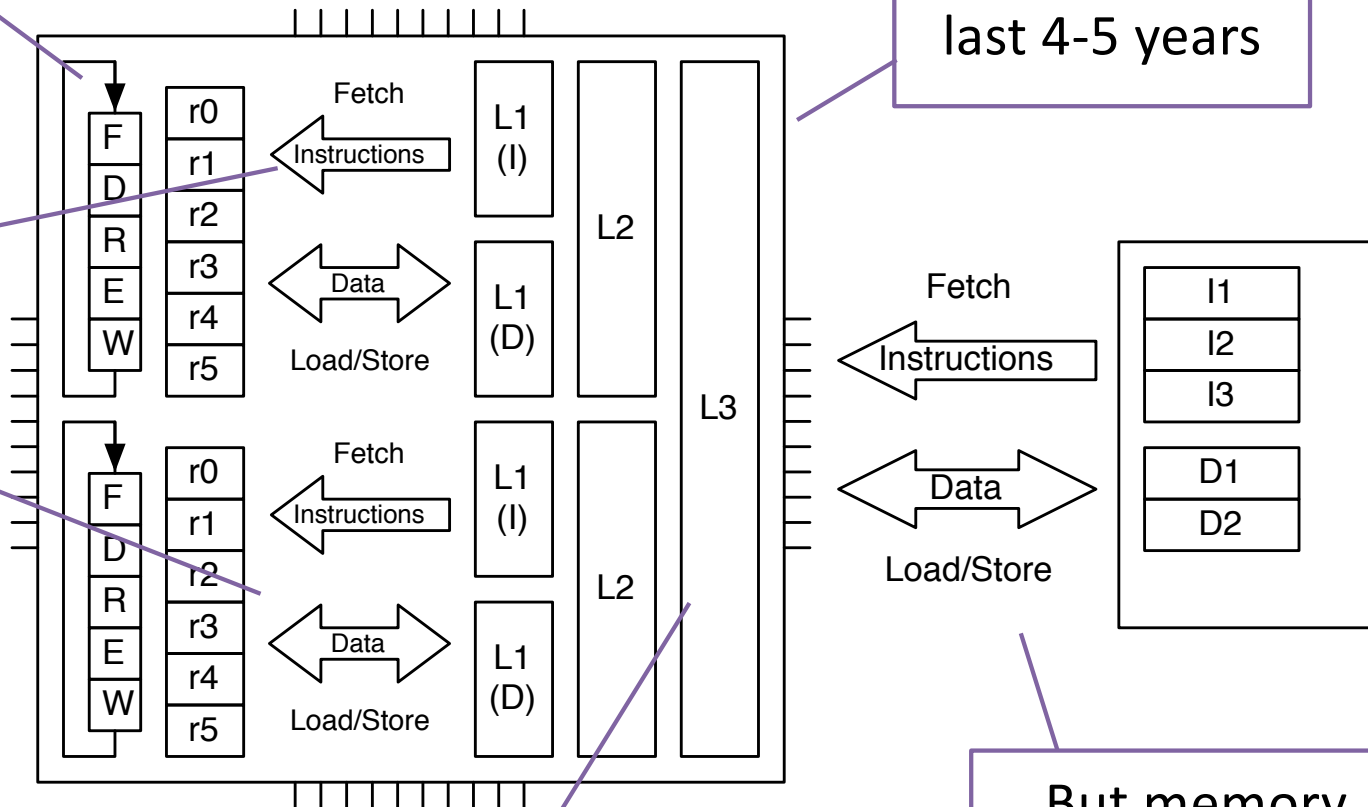
Multicore Architecture

Any CPU in the last 4-5 years

Core is a FDREW + regs

Each runs its own sequence of instructions

Each can access its own data



Each has memory hierarchy

But memory might be shared

Parallelization Example

- You are the TA for CSE 142 and have to 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 of the exams?



Parallelization Example

- You are the TA for CSE 142 and have to 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 of the exams?
- Describe your approach
 - List your assumptions



Parallelization Example

- You are the TA for CSE 142 and have to 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 of the exams?
- Describe your approach
- Describe another approach
- List your assumptions



Parallelization Example

- You are the TA for CSE 142 and have to 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 of the exams?
- Describe your approach



Parallelization Example

- You are the TA for CSE 142 and have to 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 of the exams?
- What if you had 368 friends? $368 * 22$?



Compare And Contrast

- Time for everyone grades one exam
- Time for everyone grades one question
- How (why) did you use the approaches you did?

How Do We Run Many Programs at the Same Time?

```
void hoistedMultiply(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); ++i) {  
        for (int j = 0; j < B.numCols(); ++j) {  
            double t = C(i,j);  
            for (int k = 0; k < A.numCols(); ++k) {  
                t += A(i,k) * B(k,j);  
            }  
            C(i,j) = t;  
        }  
    }  
}  
  
void tiledMultiply2x2(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); i += 2) {  
        for (int j = 0; j < B.numCols(); j += 2) {  
            for (int k = 0; k < A.numCols(); ++k) {  
                C(i, j) += A(i, k) * B(k, j);  
                C(i, j+1) += A(i, k) * B(k, j+1);  
                C(i+1, j) += A(i+1, k) * B(k, j);  
                C(i+1, j+1) += A(i+1, k) * B(k, j+1);  
            }  
        }  
    }  
}  
  
void tiledMultiply2x4(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); i += 2) {  
        for (int j = 0; j < B.numCols(); j += 4) {  
            for (int k = 0; k < A.numCols(); ++k) {  
                C(i, j) += A(i, k) * B(k, j);  
                C(i, j+1) += A(i, k) * B(k, j+1);  
                C(i, j+2) += A(i, k) * B(k, j+2);  
                C(i, j+3) += A(i, k) * B(k, j+3);  
                C(i+1, j) += A(i+1, k) * B(k, j);  
                C(i+1, j+1) += A(i+1, k) * B(k, j+1);  
                C(i+1, j+2) += A(i+1, k) * B(k, j+2);  
                C(i+1, j+3) += A(i+1, k) * B(k, j+3);  
            }  
        }  
    }  
}  
  
void tiledMultiply4x2(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); i += 4) {  
        for (int j = 0; j < B.numCols(); j += 2) {  
            for (int k = 0; k < A.numCols(); ++k) {  
                C(i, j) += A(i, k) * B(k, j);  
                C(i, j+1) += A(i, k) * B(k, j+1);  
                C(i+1, j) += A(i+1, k) * B(k, j);  
                C(i+1, j+1) += A(i+1, k) * B(k, j+1);  
                C(i+2, j) += A(i+2, k) * B(k, j);  
                C(i+2, j+1) += A(i+2, k) * B(k, j+1);  
                C(i+3, j) += A(i+3, k) * B(k, j);  
                C(i+3, j+1) += A(i+3, k) * B(k, j+1);  
            }  
        }  
    }  
}
```

```
lums658@WE31821:~$ cd ..  
lums658@WE31821:~/l7$ cp L7/L7.pptx L8  
lums658@WE31821:~/l7$ mv L7.pptx L8.pptx  
lums658@WE31821:~/l7$ open L8.pptx  
lums658@WE31821:~/l7$ ls  
L8.pptx  
lums658@WE31821:~/l7$ git add L8.pptx  
lums658@WE31821:~/l7$
```

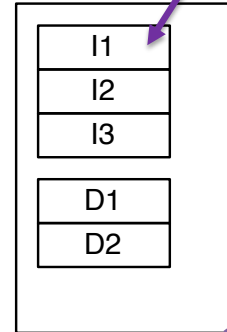
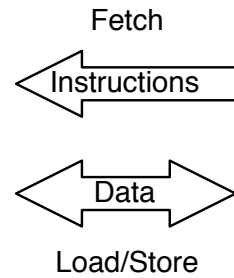
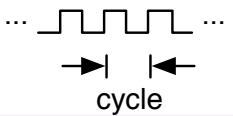
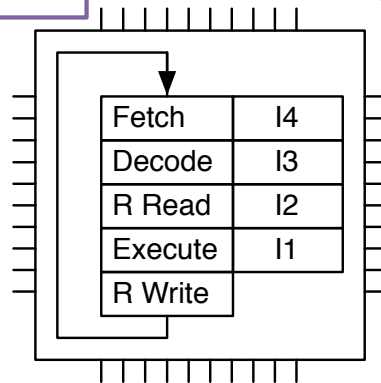
Running a Program

When a CPU is executing bytes from one program

It isn't executing bytes from another

Including from the OS (just another program)

Bytes from program stored in memory



```

.globl    __Z15hoistedMultiplyRK6MatrixS1_RS_
.p2align 4, 0x90
__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   8(%rdi), %rax
movq   (%rdi), %rcx
testq  %rax, %rcx
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)
movq   16(%rdx), %rdx
movq   8(%rdi), %rax
movq   16(%rdi), %r13
leaq  -1(%rcx), %rsi
movq   %rsi, -88(%rbp)
movl   %ecx, %esi
    
```

How does another program run?

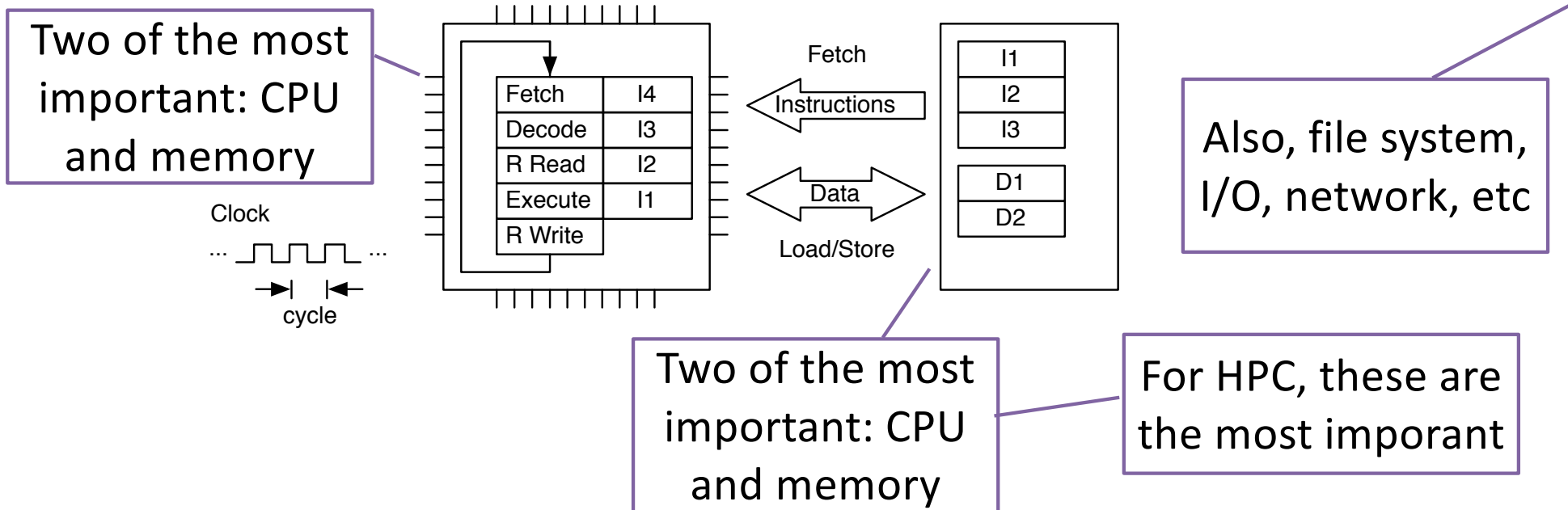
How did the bytes get here?

How Do We Run Many Programs at the Same Time?

The screenshot displays a multi-tasking environment. On the left, a code editor shows C++ code for matrix multiplication functions: `hoistedMultiply`, `tiltedMultiply2x2`, `tiltedMultiply2x4`, and `tiltedMultiply4x2`. The `hoistedMultiply` function uses a single loop for row `i` and a nested loop for columns `j` and `k`. The tiled functions use nested loops for rows `i` and columns `j`, with an inner loop for `k`. The terminal window on the right shows a sequence of commands: `cd ..`, `cp L7/L7.pptx L8`, `mv L7.pptx L8.pptx`, `open L8.pptx`, `ls`, and `git add L8.pptx`. The web browser in the background shows a search for 'douglas adams' on Google.

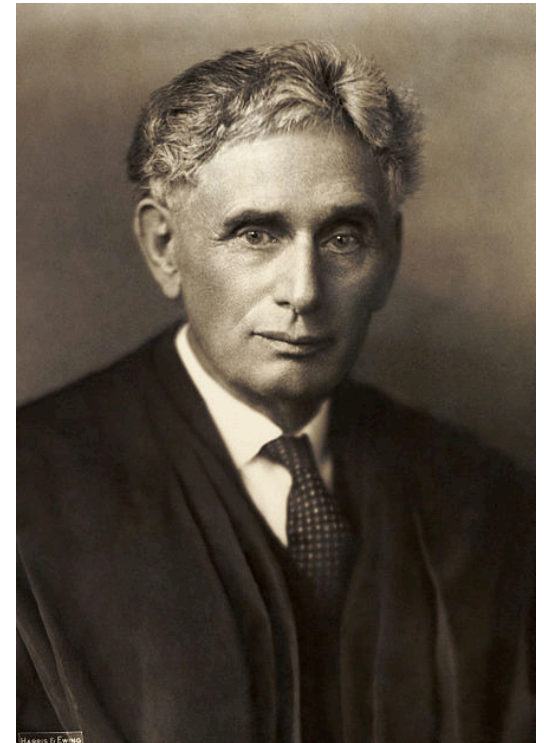
A Word About Operating Systems

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

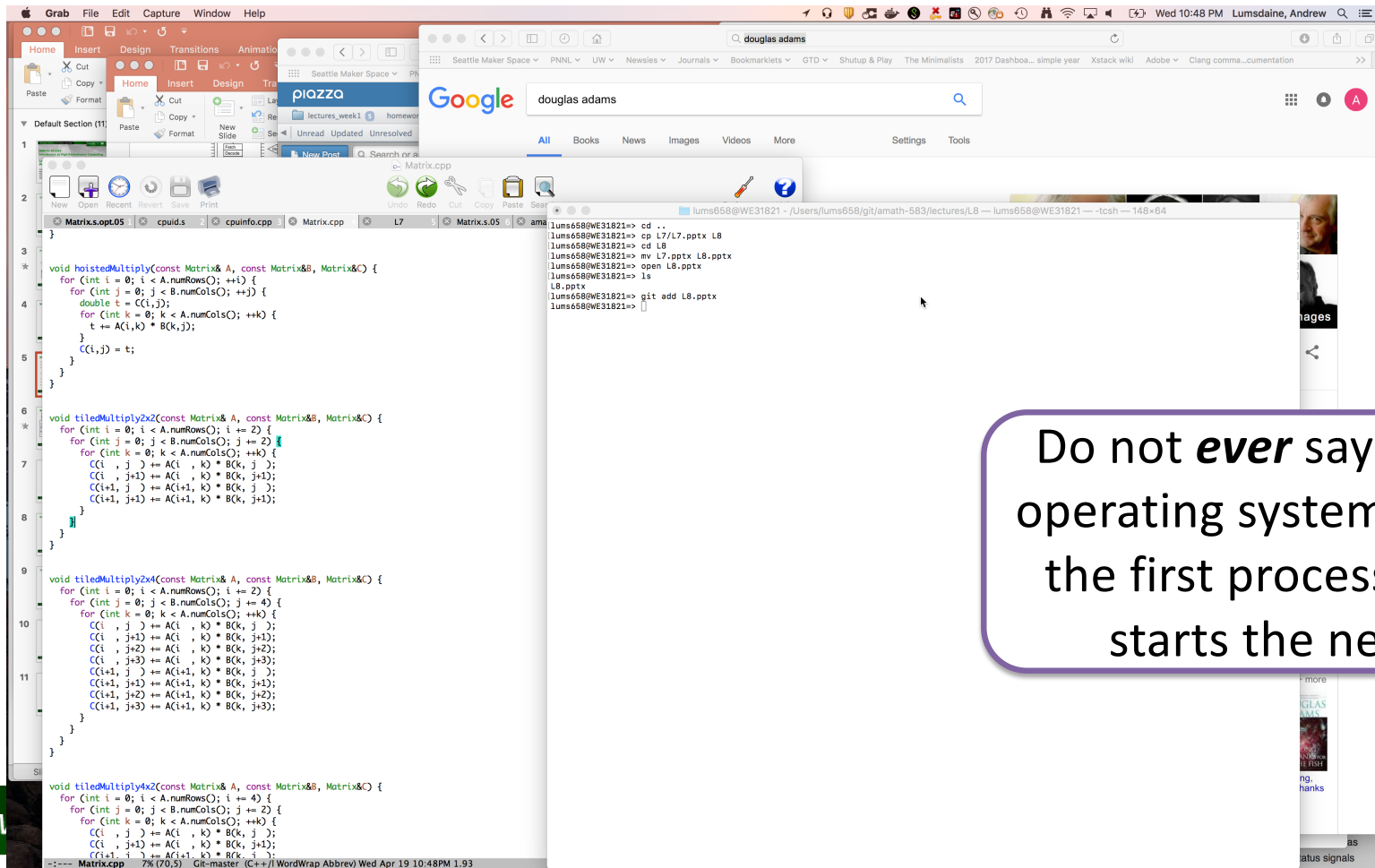


Processes and Threads

- 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



How Do We Run Many Programs ~~at the same time?~~?



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

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

Registers
Program counter
Program status word
Stack pointer
Process state
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

Memory management

Pointer to text segment
Pointer to data segment
Pointer to stack segment

File management

Root directory
Working directory
File descriptors
User ID
Group ID

What does program counter represent?

The Process Concept

\$ top -u

Process ID

```
lums658@WE31821 - /Users/lums658
Processes: 419 total, 2 running, 417 sleeping, 1988 threads
Load Avg: 1.93, 1.88, 1.87 CPU usage: 3.45% user, 3.69% sys,
MemRegions: 156549 total, 7076M resident, 141M private, 3629M
VM: 4328G vsize, 627M framework vsize, 71344832(64) swapins,
Disks: 57070556/1524G read, 36025949/792G written.
```

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	MEM	PUR
162	WindowServer	13.8	07:48:22	6	2	702+	537M+	93M
0	kernel_task	12.6	29:59:12	177/9	0	2	1809M+	0B
114	hidd	4.4	01:46:55	6	3	381+	3024K+	0B
8333	top	4.0	00:00.72	1/1	0	21	5016K	0B
8334	screencaptur	3.9	00:00.06	4	3	57	2500K+	20K
91791	LaTeXiT	2.3	09:45.97	6	2	255	42M	0B
67565	Terminal	2.0	01:50.53	13	8	346+	72M	0B
3288	Calendar	1.6	09:54.07	3	1	292	95M	185K
1234	com.docker.h	1.1	02:02:24	18	1	38	763M	0B
846	usernoted	1.1	03:13.97	5	4	139+	11M+	896K
83898	Slack Helper	1.0	01:40.81	19	2	149	189M+	0B
91742	splunkd	0.8	40:02.25	35	0	48	85M	0B
63334	Slack Helper	0.6	01:19.70	5	2	124	7780K	0B
184	mDNSResponde	0.5	22:51.68	5	1	103	5628K	0B
111-	NetworkMonit	0.4	12:37.75	28	27	49+	22M+	0B
883	CalNCService	0.3	19:18.74	5	3	182+	39M+	0B
853	SystemUIServ	0.2	02:45.35	5	3	371	33M+	28K-
63333	Slack	0.2	04:36.66	33	1	390	73M	0B

How much CPU

How many threads

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	MEM	PUR
162	WindowServer	13.8	07:48:22	6	2	702+	537M+	93M
0	kernel_task	12.6	29:59:12	177/9	0	2	1809M+	0B
114	hidd	4.4	01:46:55	6	3	381+	3024K+	0B
8333	top	4.0	00:00.72	1/1	0	21	5016K	0B
8334	screencaptur	3.9	00:00.06	4	3	57	2500K+	20K
91791	LaTeXiT	2.3	09:45.97	6	2	255	42M	0B
67565	Terminal	2.0	01:50.53	13	8	346+	72M	0B
3288	Calendar	1.6	09:54.07	3	1	292	95M	185K
1234	com.docker.h	1.1	02:02:24	18	1	38	763M	0B
846	usernoted	1.1	03:13.97	5	4	139+	11M+	896K
83898	Slack Helper	1.0	01:40.81	19	2	149	189M+	0B
91742	splunkd	0.8	40:02.25	35	0	48	85M	0B
63334	Slack Helper	0.6	01:19.70	5	2	124	7780K	0B
184	mDNSResponde	0.5	22:51.68	5	1	103	5628K	0B
111-	NetworkMonit	0.4	12:37.75	28	27	49+	22M+	0B
883	CalNCService	0.3	19:18.74	5	3	182+	39M+	0B
853	SystemUIServ	0.2	02:45.35	5	3	371	33M+	28K-
63333	Slack	0.2	04:36.66	33	1	390	73M	0B

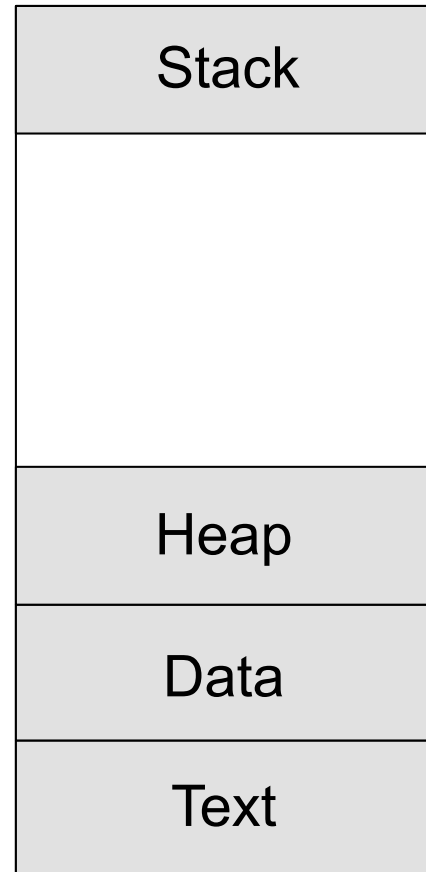
Process address space

Memory resources
for each process

All 32/48/64 bits

Address
Space

How can each
process use all the
address space?



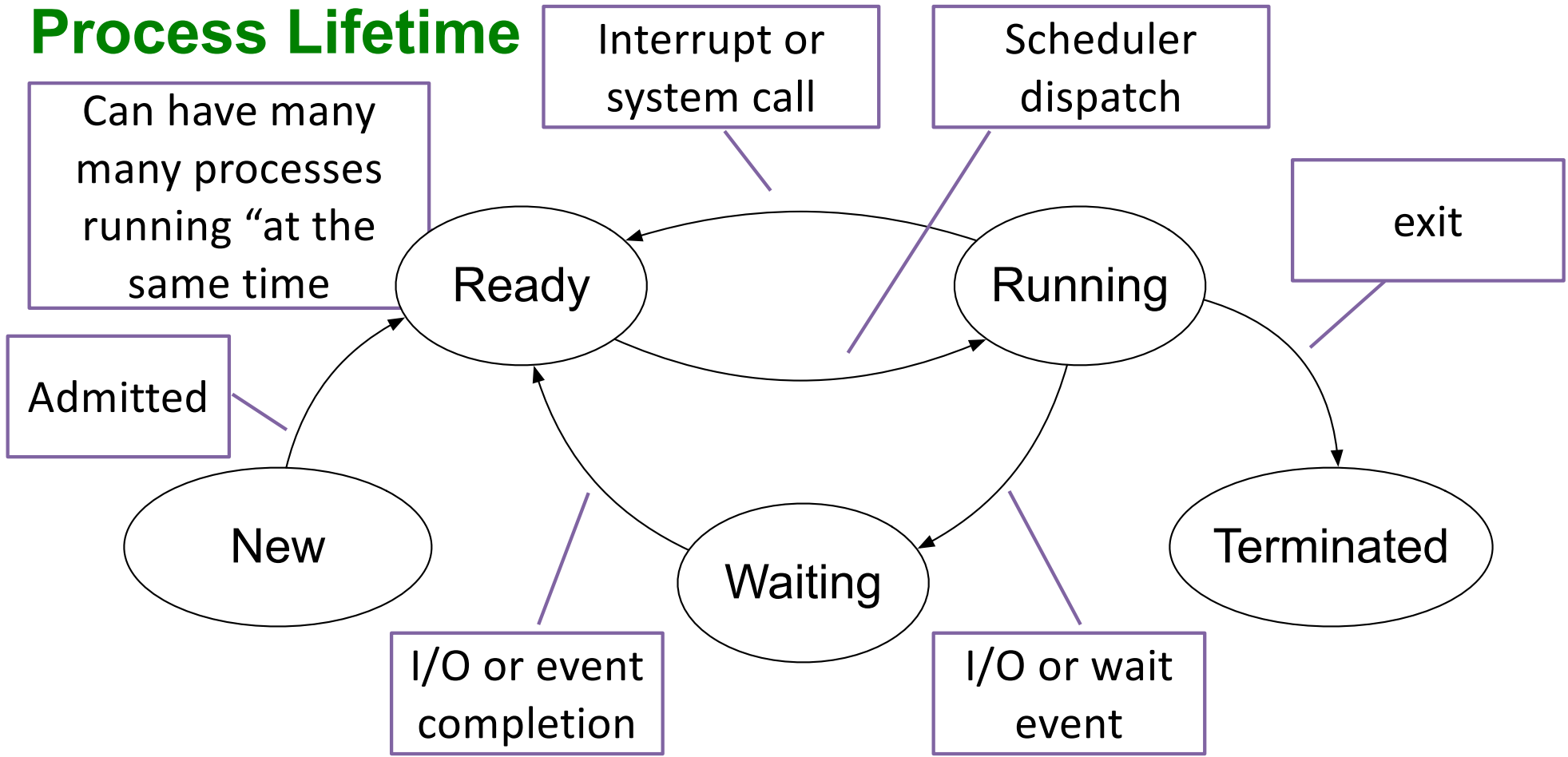
Created and
managed at run time

Created and
managed at run time

Compiled /
Linked

Stored
Program

Process Lifetime



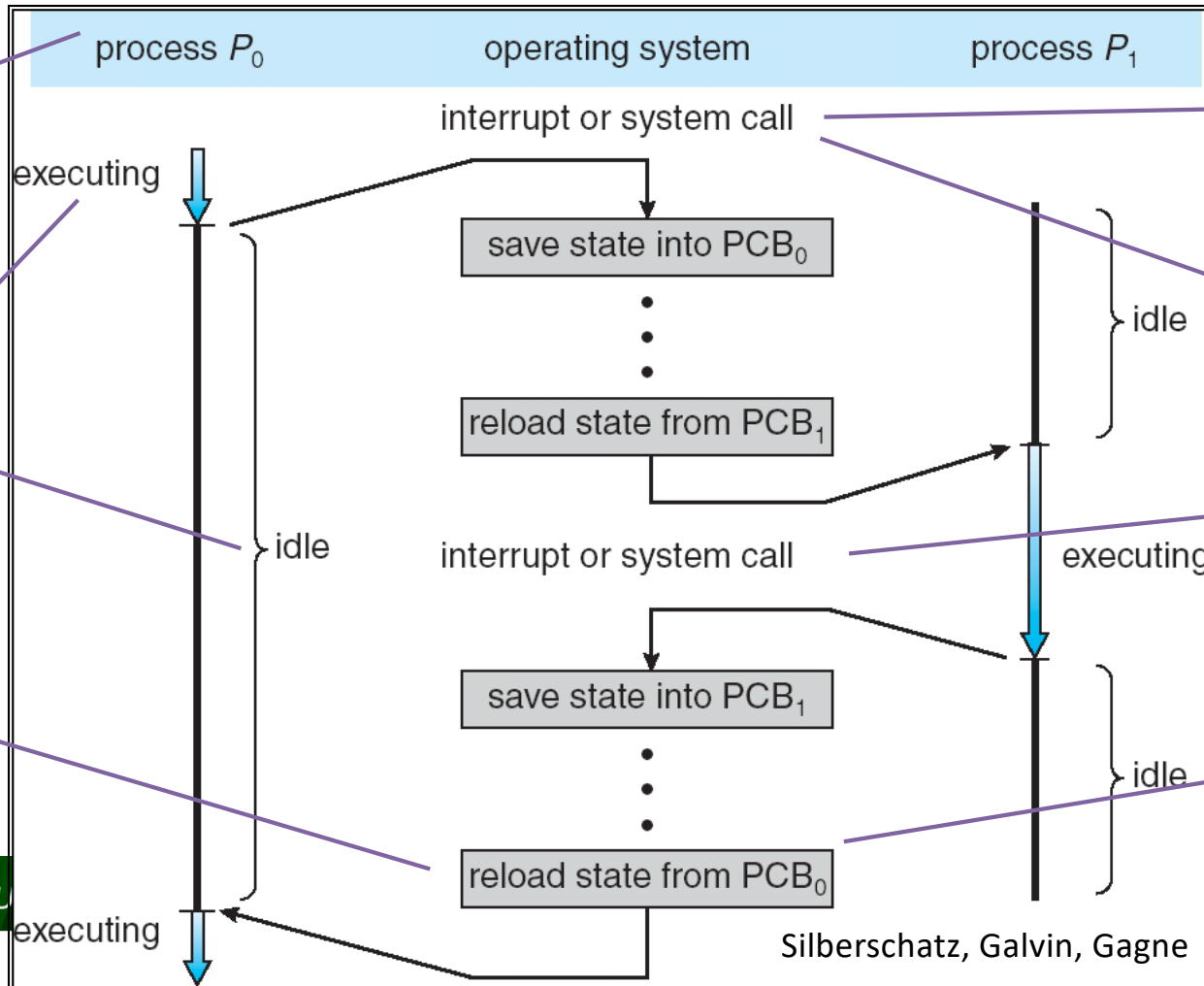
Context Switch

P0 and P1 are running processes

What does this mean?

And this?

PCB = Process Control Block



External to OS

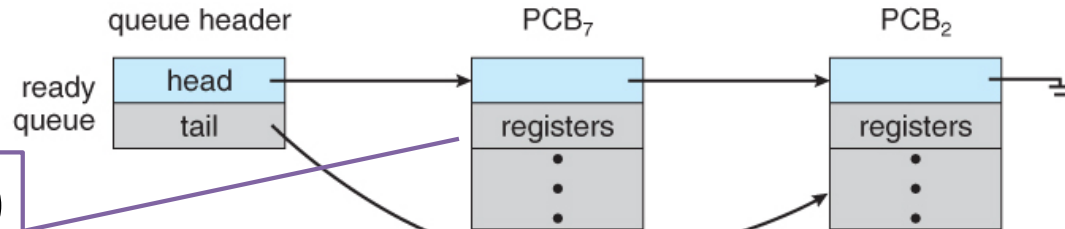
OS does not do this

External to OS

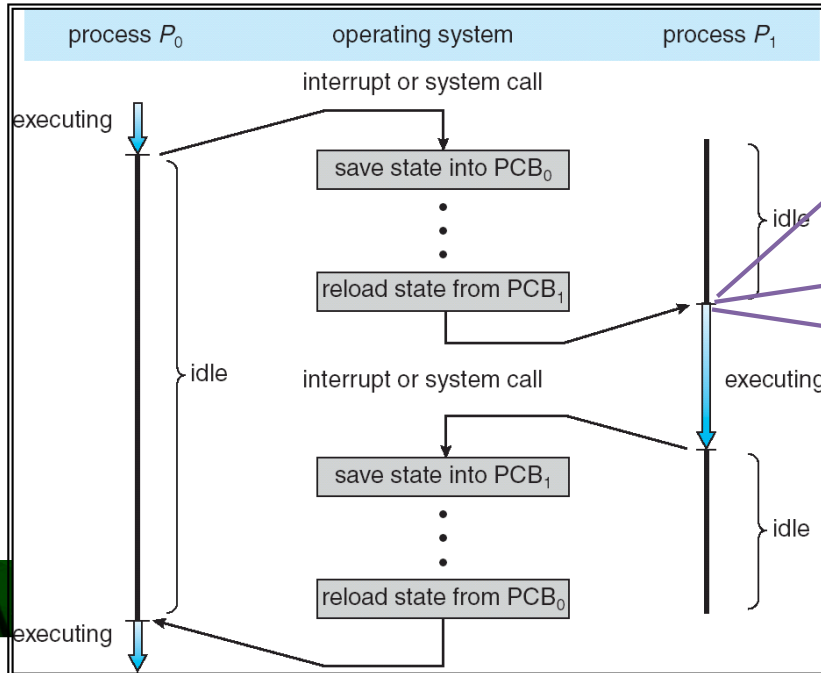
Expensive!

Process Queues

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



Program runs from start to finish



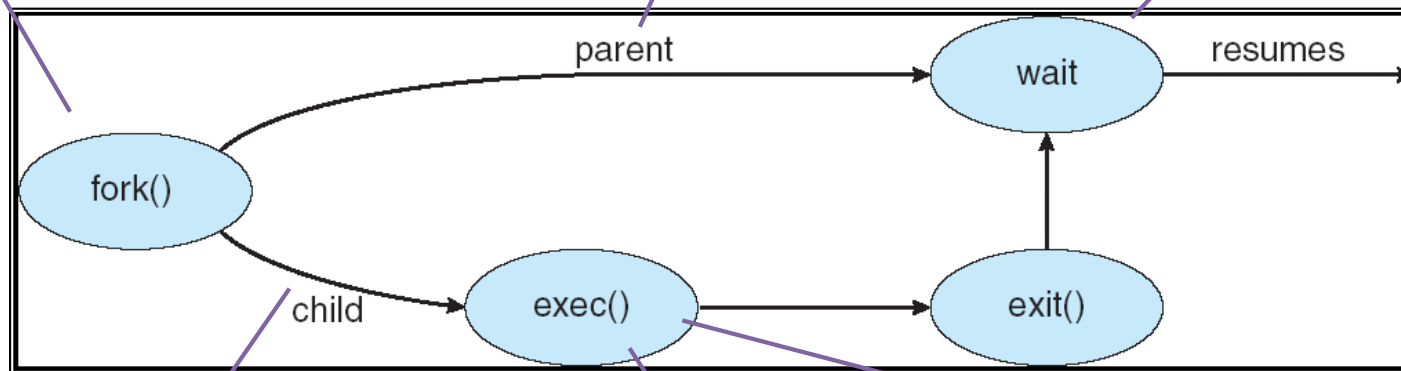
Context switches are not observable

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()

Two processes return from fork()

```
#include <unistd.h>

int main () {
    fork();
    return 0;
}
```

```
#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?

$i == 0$

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

Don't do this (ever)!

How many processes?

$i == 1$

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

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

$i == 2$

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

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

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

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

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 — 135x52
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).
RETURN 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.
ERRORS
    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.
COMPATIBILITY 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);  
    }  
}
```

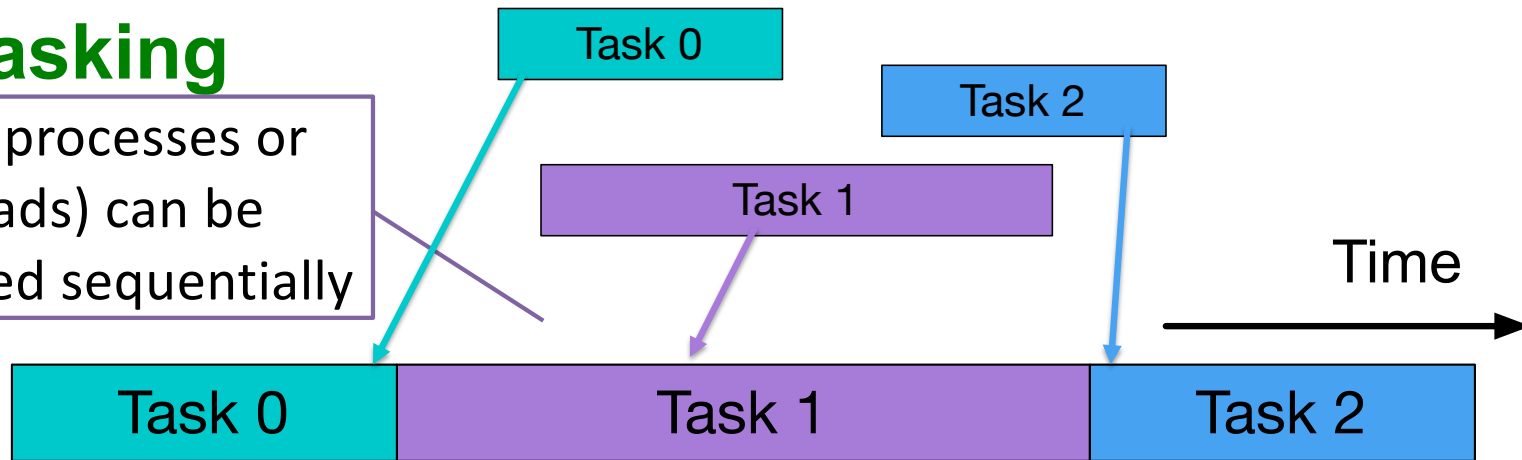
```
while (true) {  
    cout << "$ ";  
    cin >> command;  
    pid_t child = fork();  
    if (0 == child) {  
        execv(command, NULL);  
    } else {  
        wait(child);  
    }  
}
```


How Do We Run Multiple Programs Concurrently?

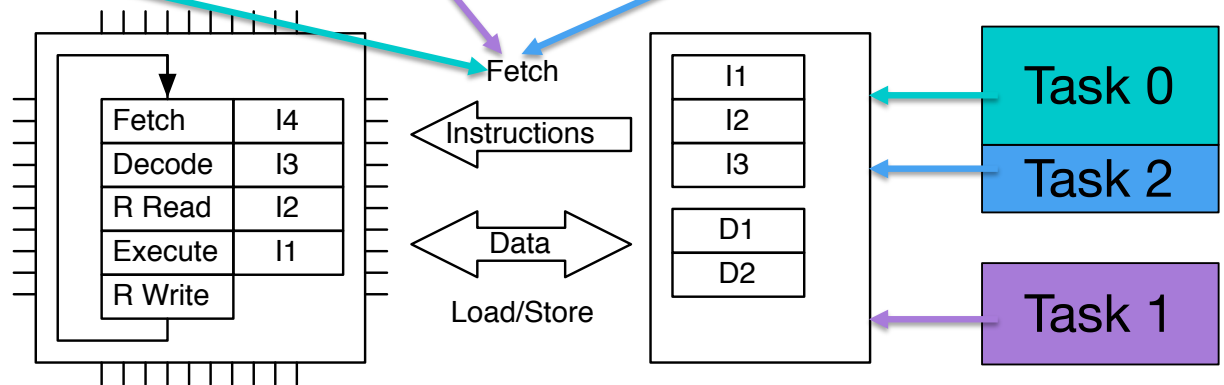
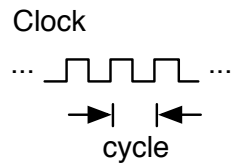
The screenshot displays a multi-tasking environment. In the foreground, a code editor shows C++ code for matrix multiplication. The code includes a standard sequential multiplication function and three tiled versions: `void tiledMultiply2x2`, `void tiledMultiply2x4`, and `void tiledMultiply4x2`. Each tiled function uses nested loops to process the matrix in smaller blocks. To the right, a terminal window shows a sequence of shell commands: `cd ..`, `cd L7/L7.pptx LB`, `cd L8`, `mv L7.pptx L8.pptx`, `open L8.pptx`, `ls`, and `git add L8.pptx`. In the background, a web browser window shows a Google search for 'douglas adams'. The desktop also features a presentation viewer with a slide titled 'piazza' and a sidebar with navigation icons.

Multitasking

Tasks (processes or threads) can be scheduled sequentially

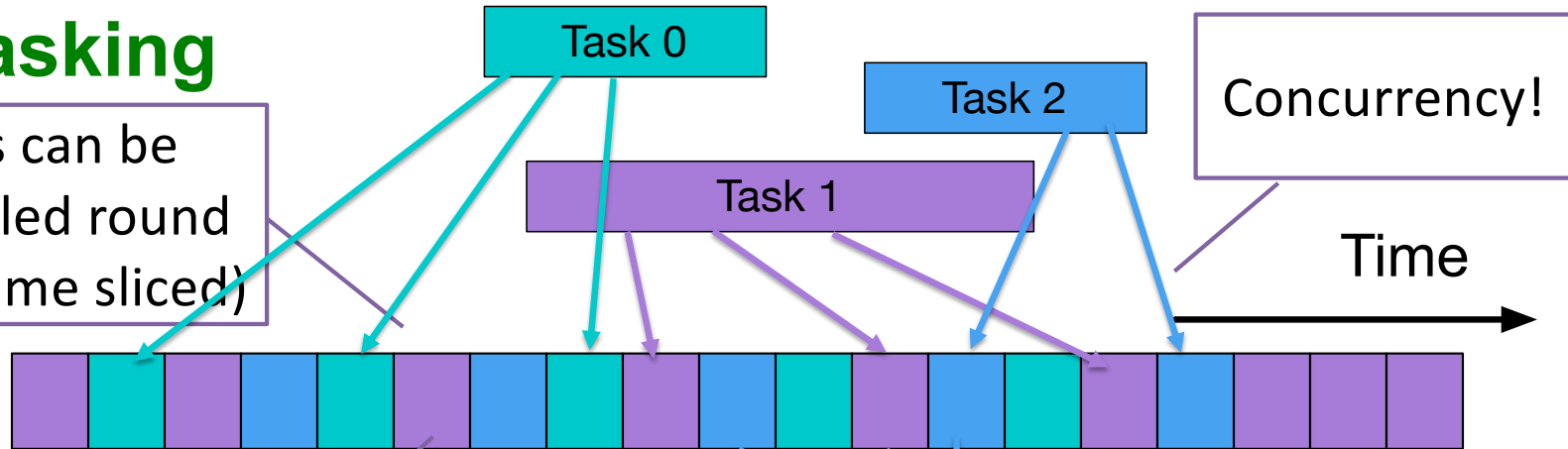


Run to completion

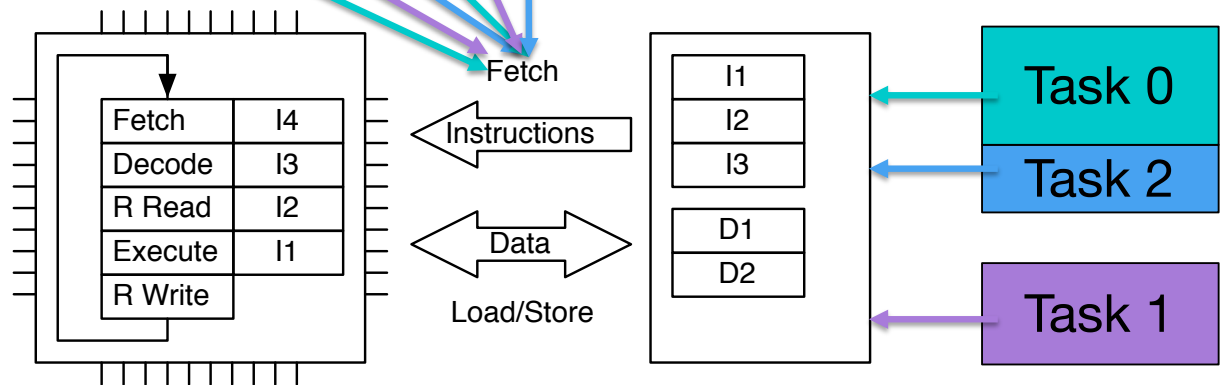
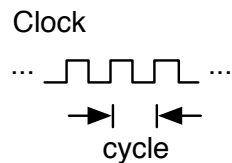


Multitasking

Tasks can be scheduled round robin (time sliced)



Run to context switch (system call or interrupt)

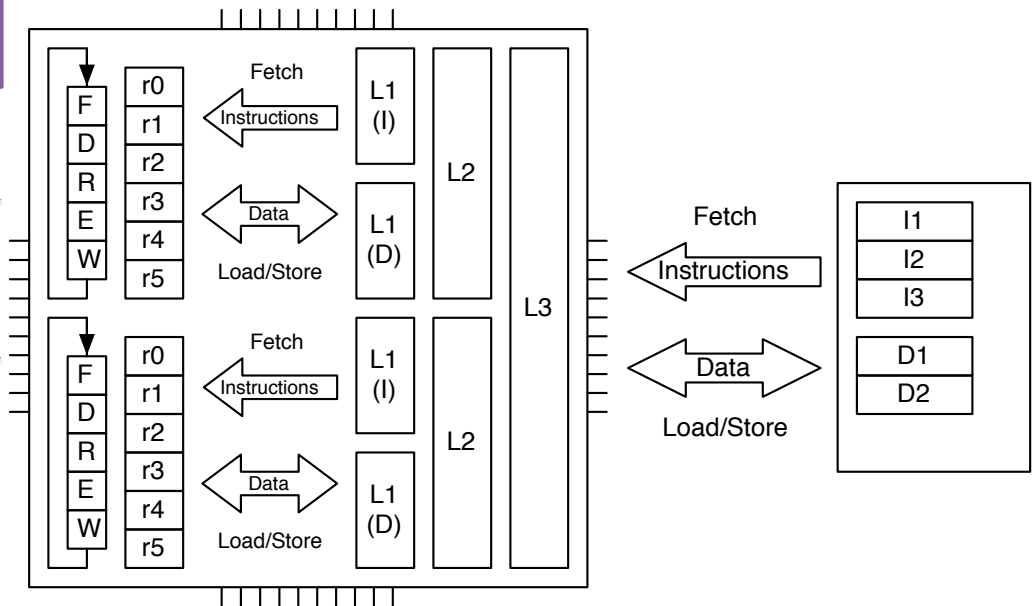
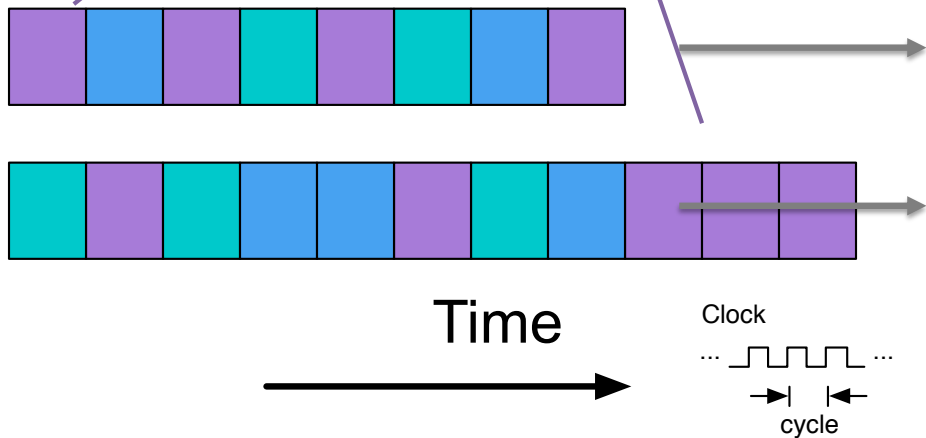


Multitasking on Multicore

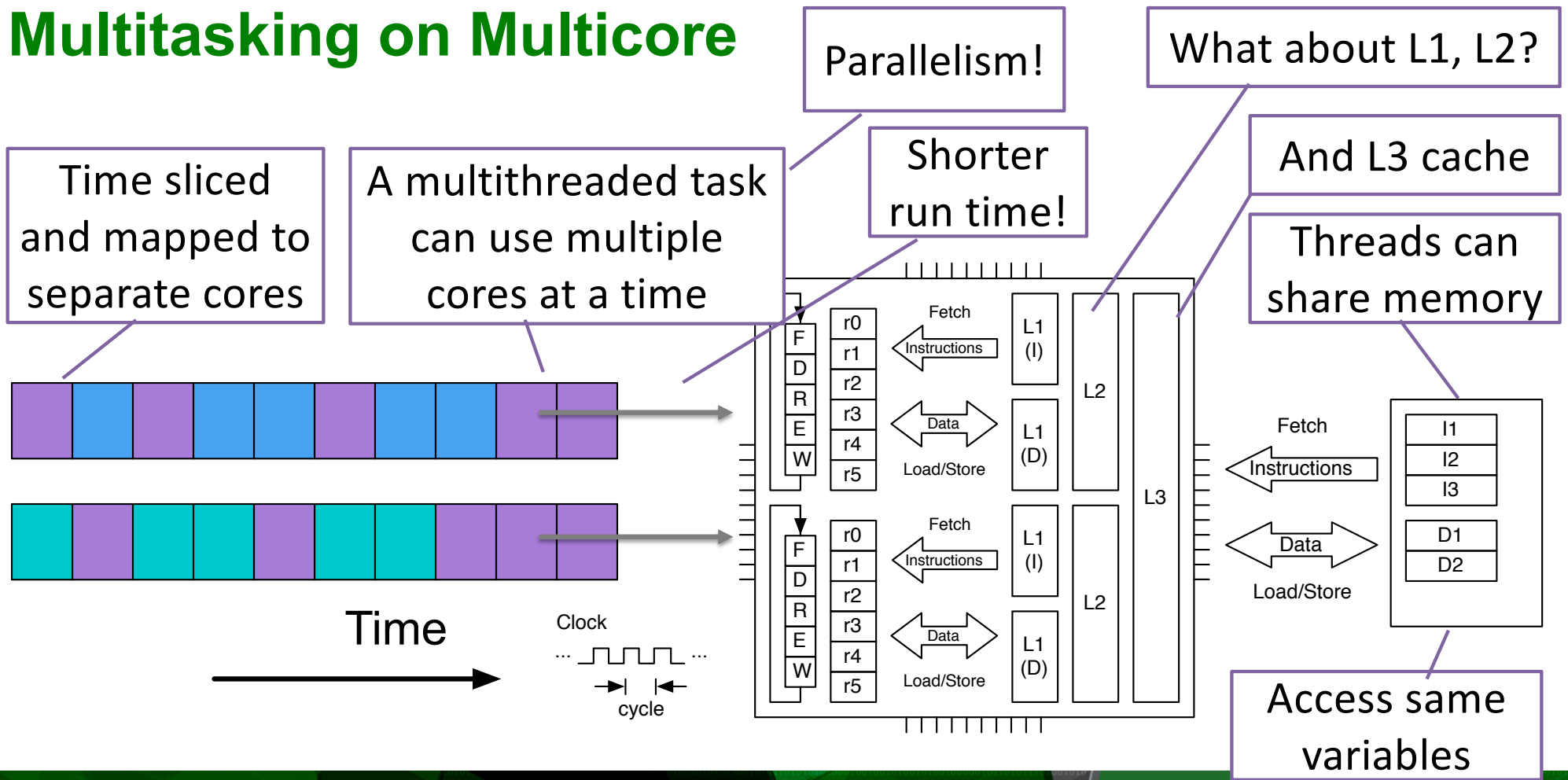
Concurrency!

Time sliced and mapped to separate cores

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



Multitasking on Multicore



Cache Coherence

Hardware managed

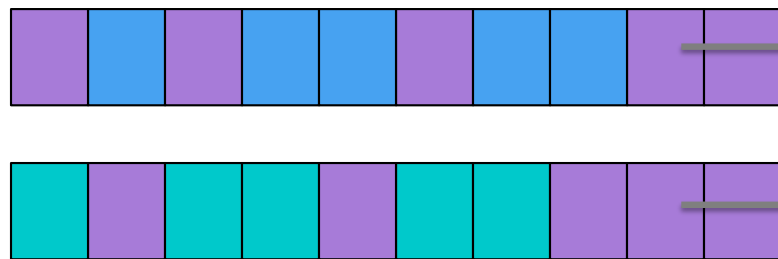
Same variable can be in two different caches

A multithreaded task can use multiple cores at a time

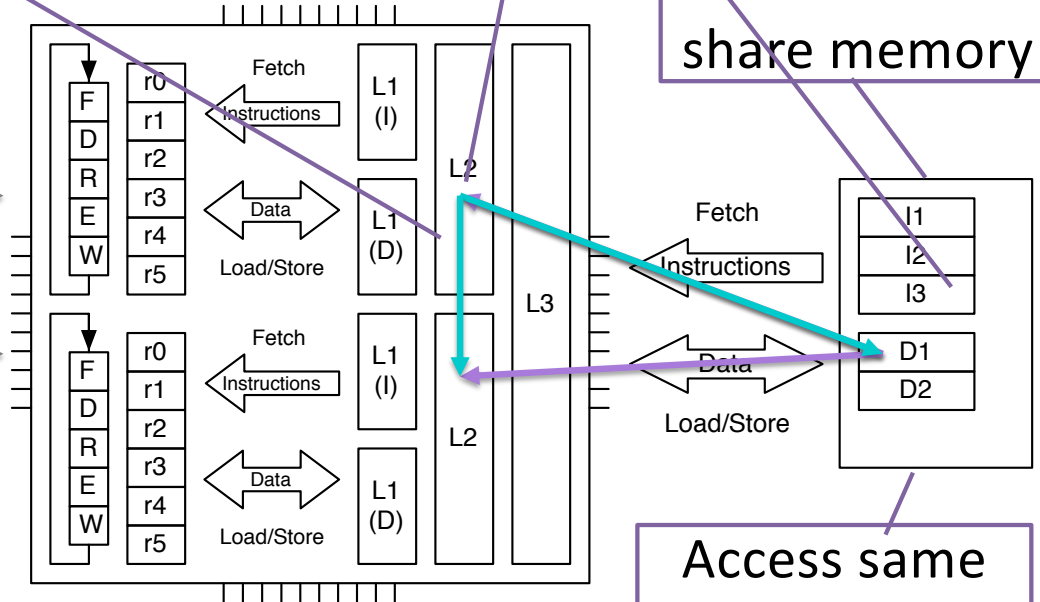
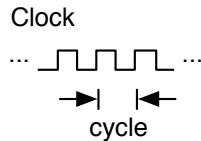
Cache coherence / memory consistency

What if one gets modified?

Threads can share memory

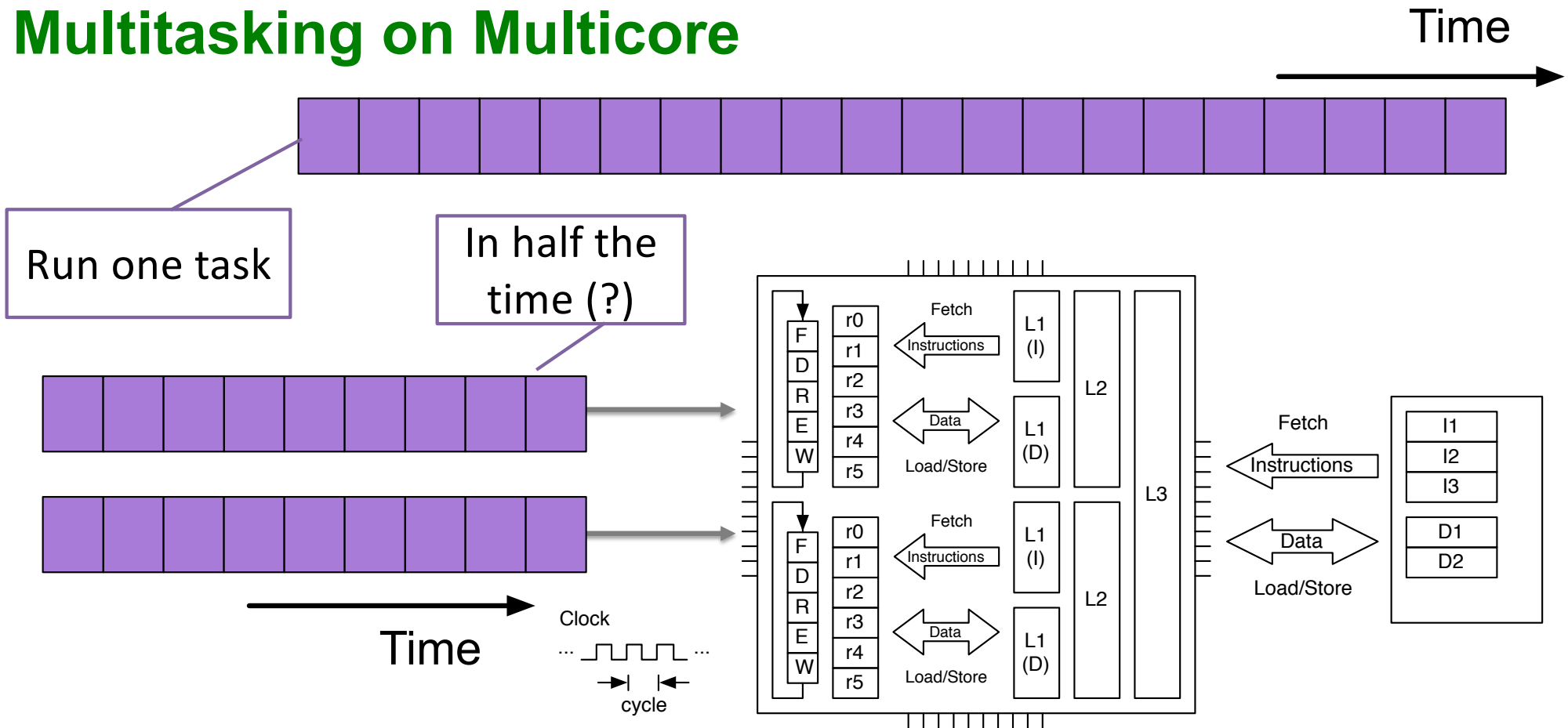


Time



Access same variables

Multitasking on Multicore

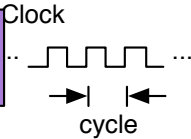
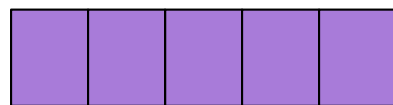
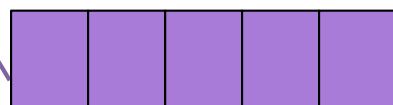
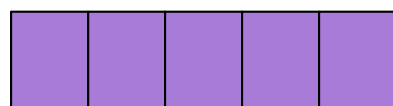


Multitasking on Multicore

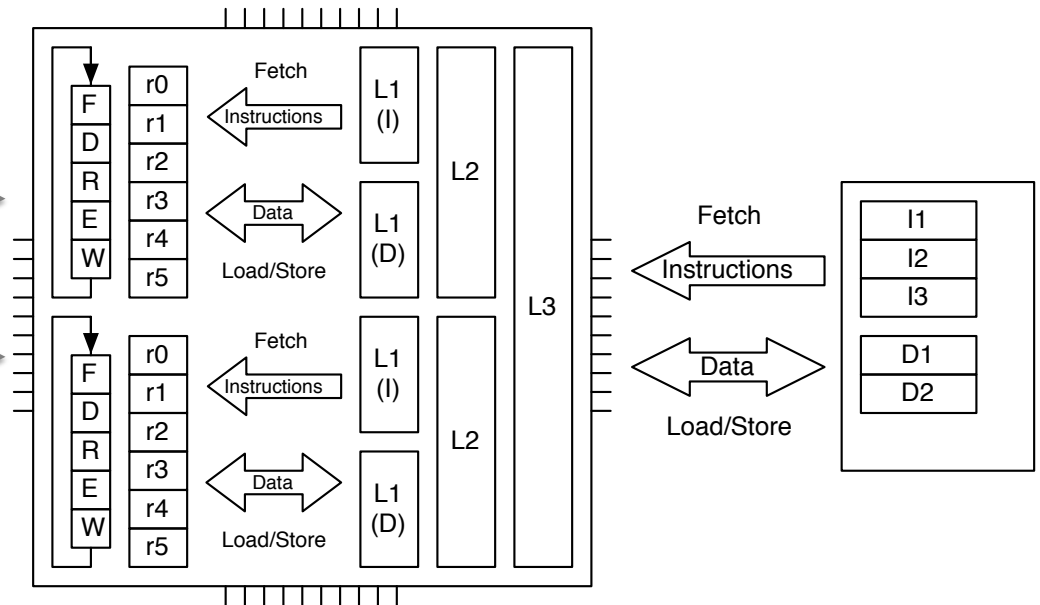
Time →

Run one task

In ¼ the time (?)



Time →



Multitasking on Multicore

Nonetheless, this is the essence of *parallel* computing

Parallel computation isn't done until all cores are done

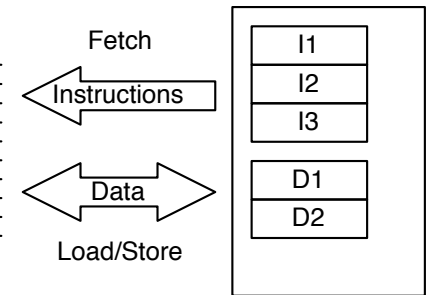
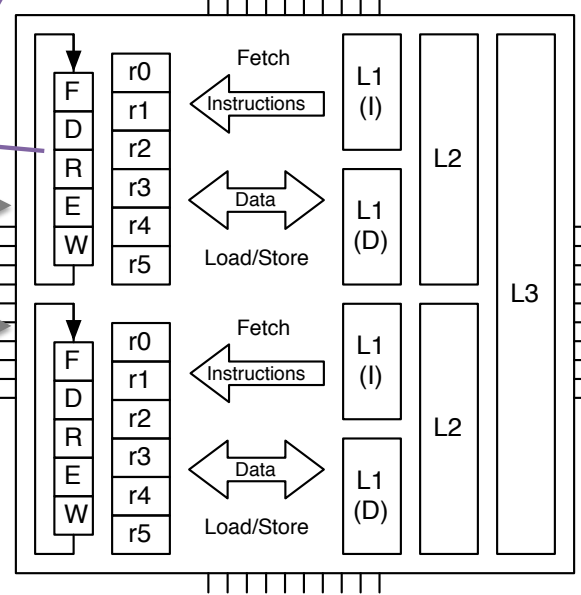
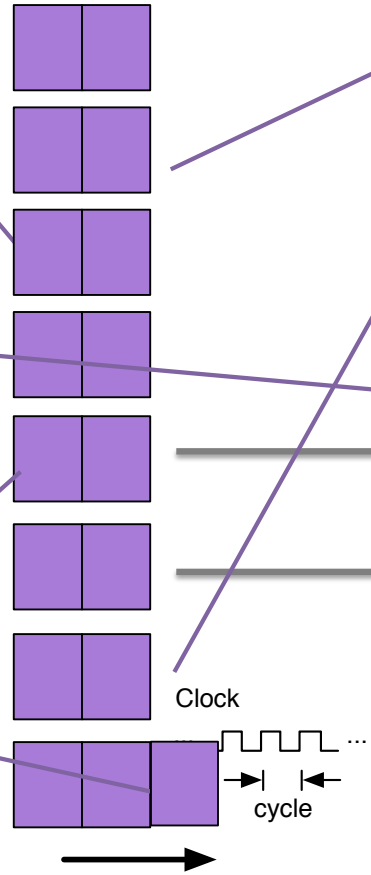
Not the same as concurrent

In 1/8 the time (?)

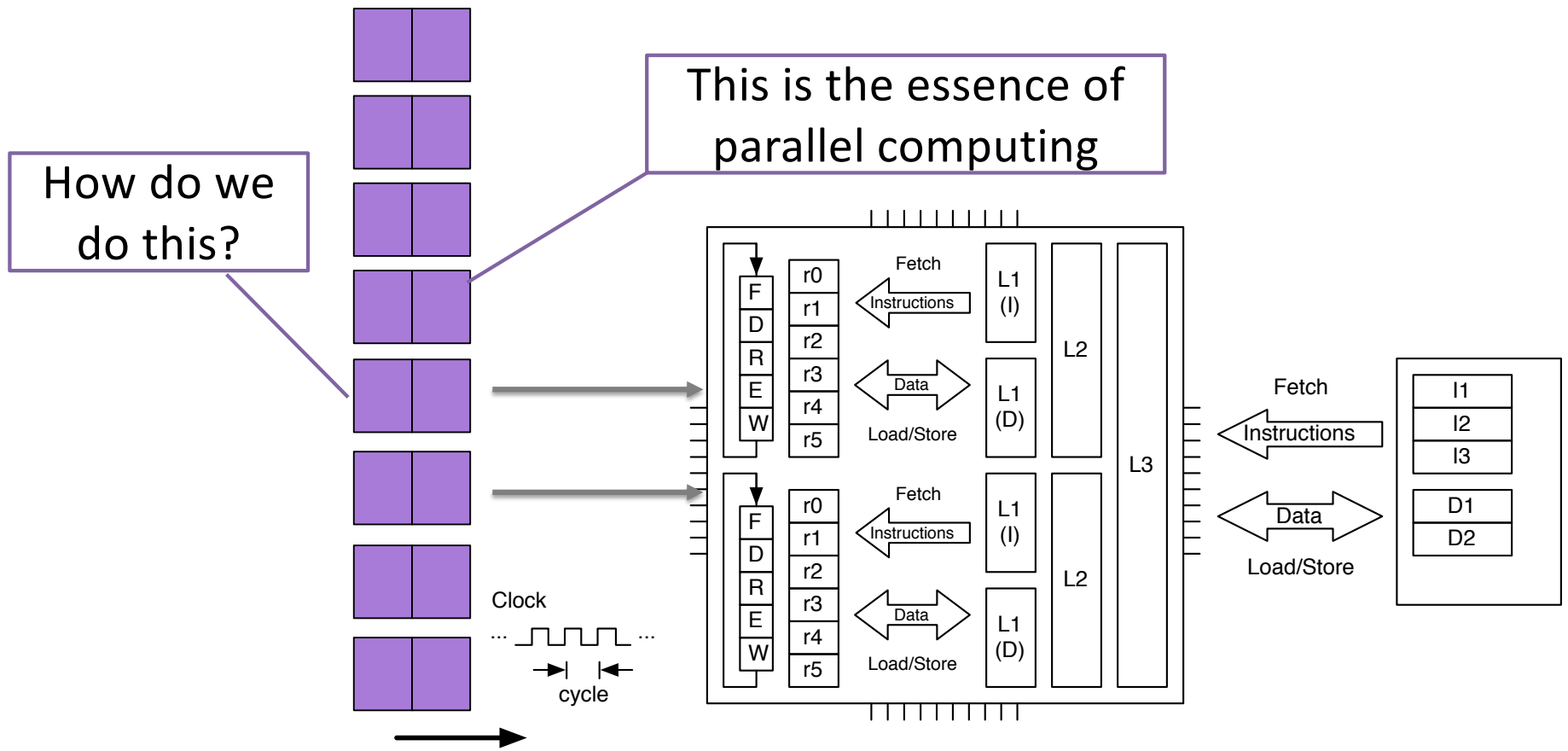
Need enough cores (8)

Work needs to be balanced

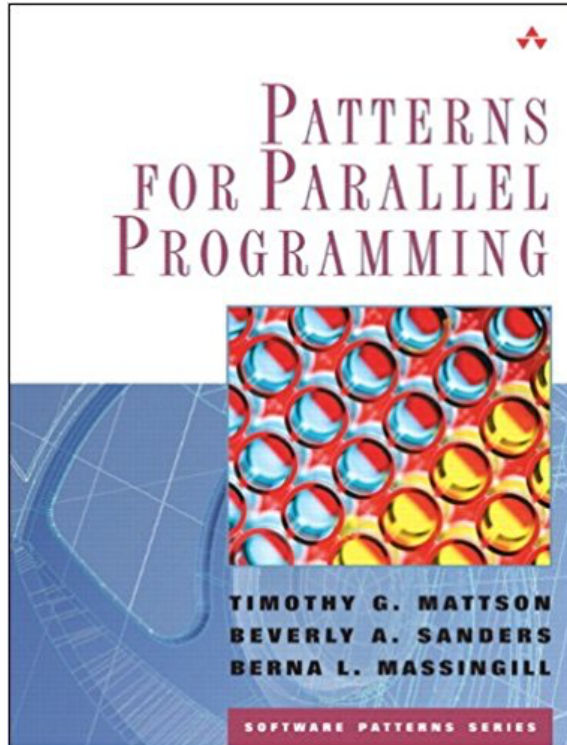
oops



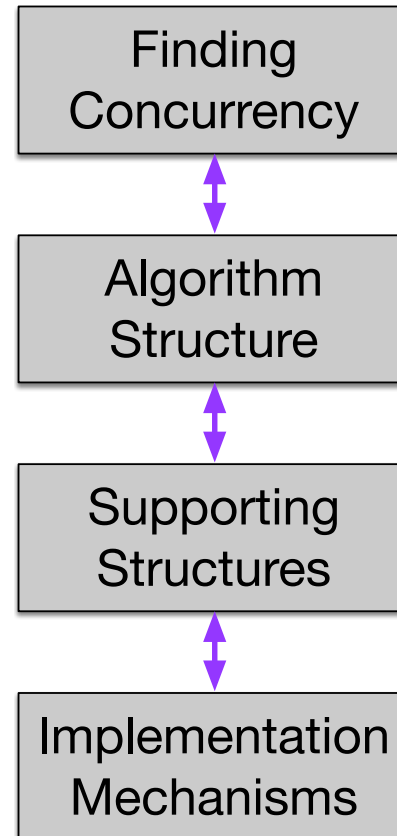
Multitasking on Multicore



Parallelization Strategy



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

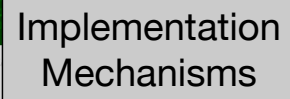
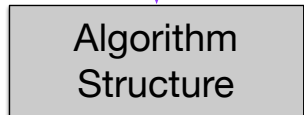
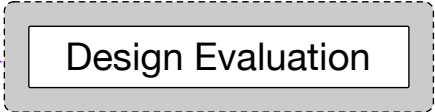
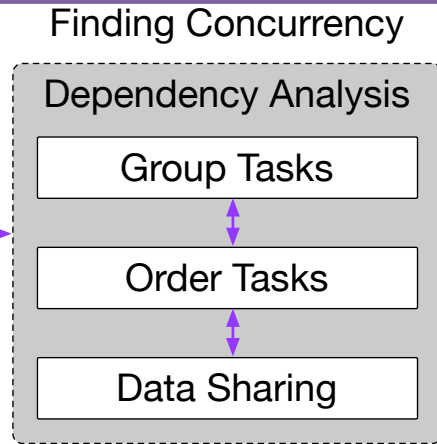
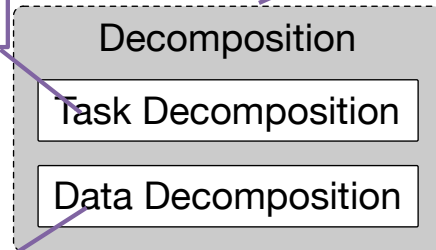


Finding Concurrency

Decompose problem into pieces that can execute concurrently

Into tasks that can execute concurrently

Units that can be operated on (relatively) independently



Finding Concurrency

Ways to group tasks to simplify management of dependencies

Finding Concurrency

Dependency Analysis

Group Tasks

Order Tasks

Data Sharing

Design Evaluation

Decomposition

Task Decomposition

Data Decomposition

Ways to group tasks to simplify management of dependencies

Ways to order tasks for correctness, other constraints

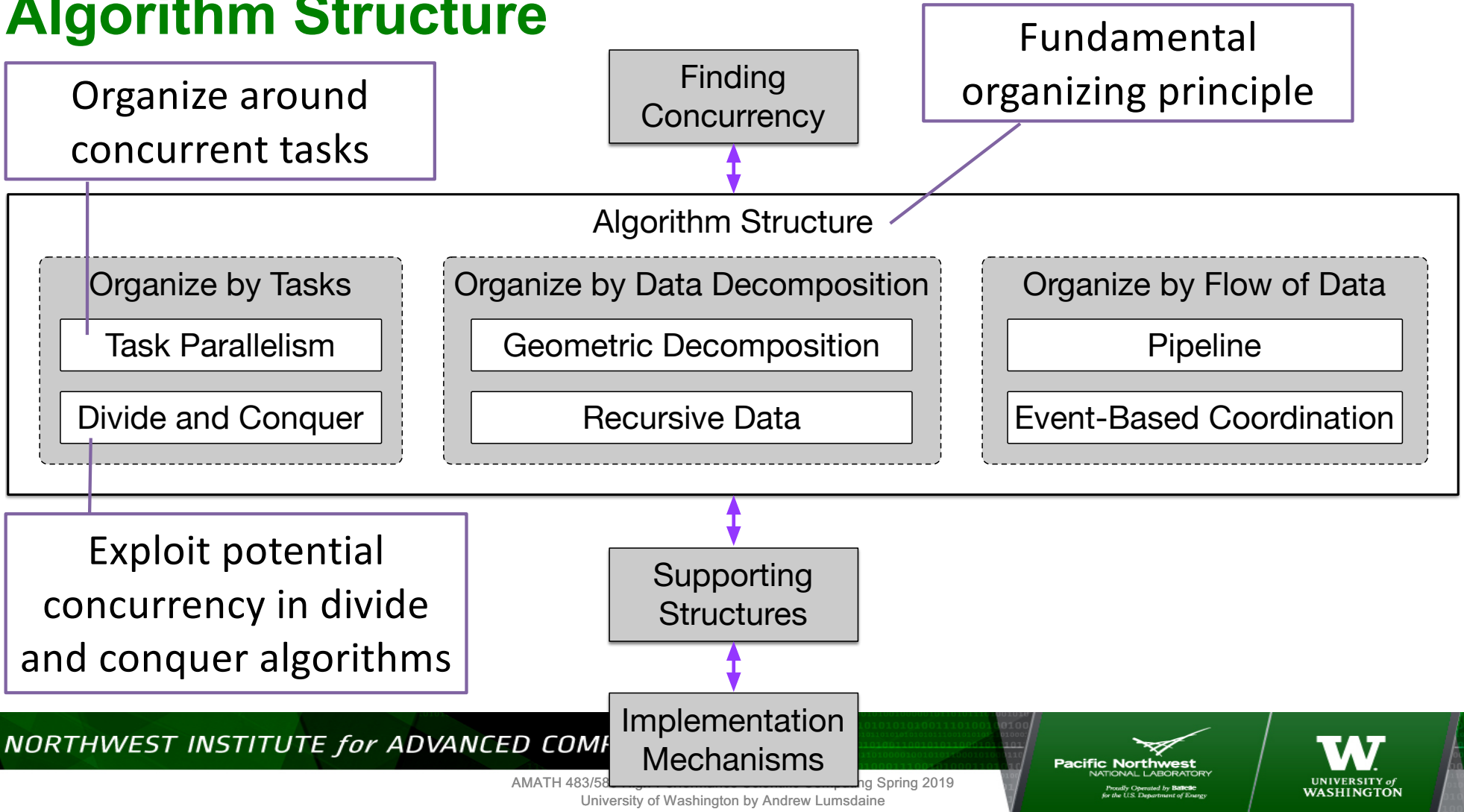
Given a decomposition, ways to share data among tasks

Algorithm Structure

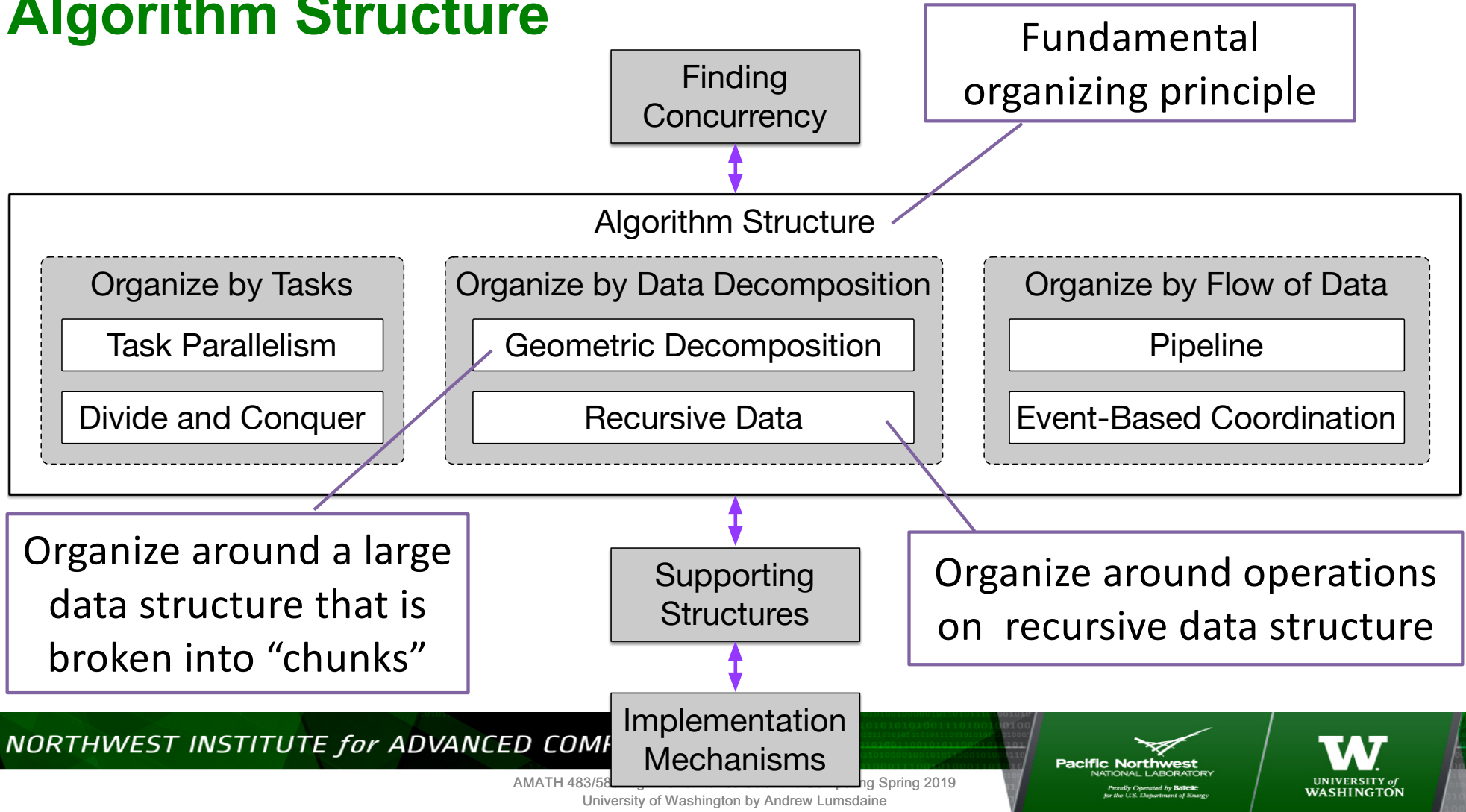
Supporting Structures

Implementation Mechanisms

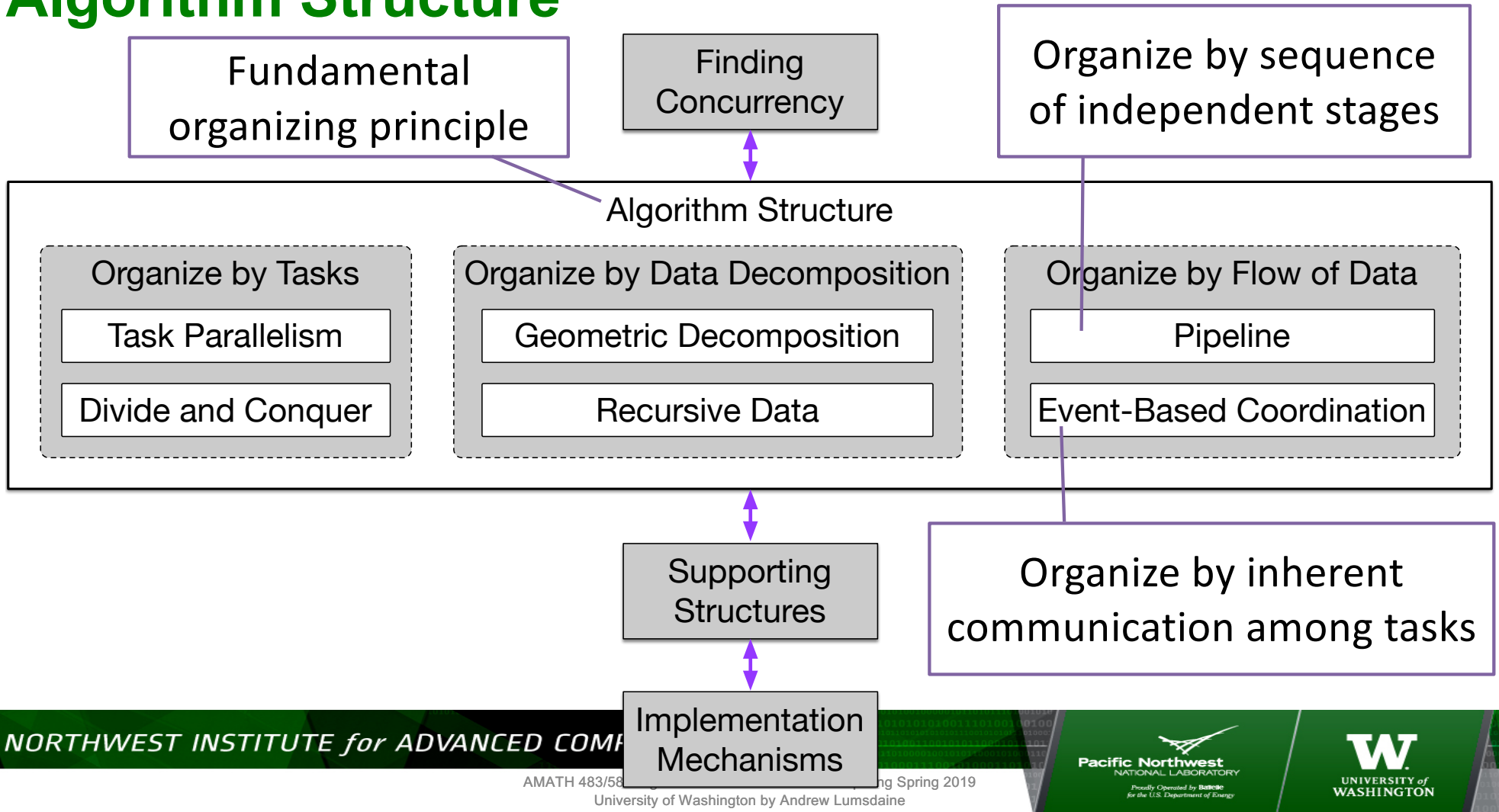
Algorithm Structure



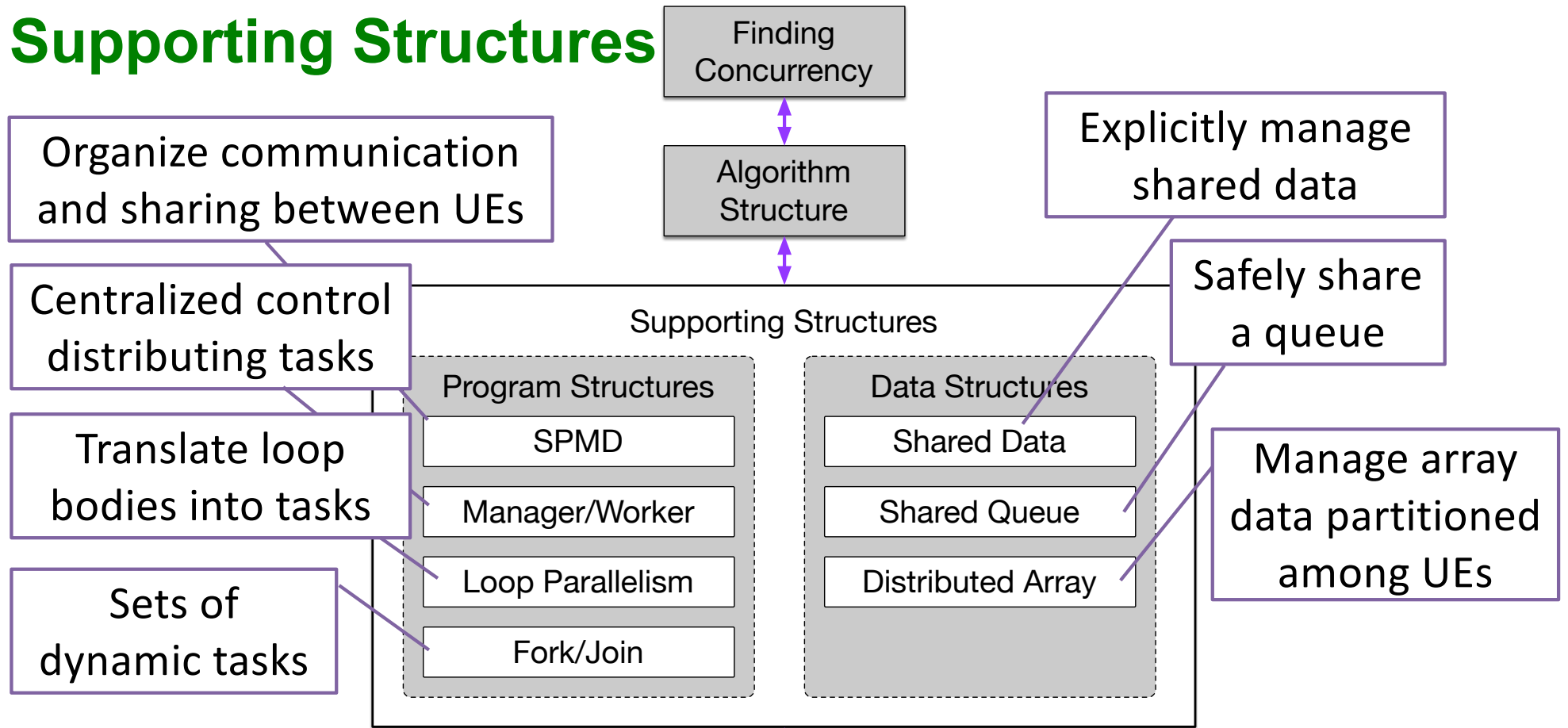
Algorithm Structure



Algorithm Structure



Supporting Structures



Implementation Mechanisms

Finding
Concurrency



Algorithm
Structure



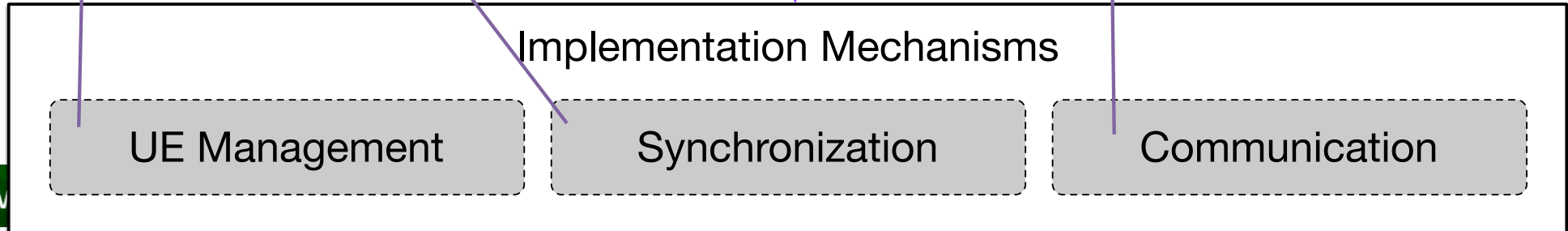
Supporting
Structures



Enforce ordering
constraints

Manage task
lifetimes

Get data where it
needs to be when UEs
don't share memory



Stay Tuned

- C++ threads
- C++ `async()`
- C++ atomics

Thank you!

NORTHWEST INSTITUTE for ADVANCED COMPUTING

AMATH 483/583 High-Performance Scientific Computing Spring 2019
University of Washington by Andrew Lumsdaine


Pacific Northwest
NATIONAL LABORATORY
Proudly Operated by Battelle
for the U.S. Department of Energy


UNIVERSITY of
WASHINGTON

Creative Commons BY-NC-SA 4.0 License



© Andrew Lumsdaine, 2017-2018

Except where otherwise noted, this work is licensed under

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

