CS 31: Introduction to Computer Systems

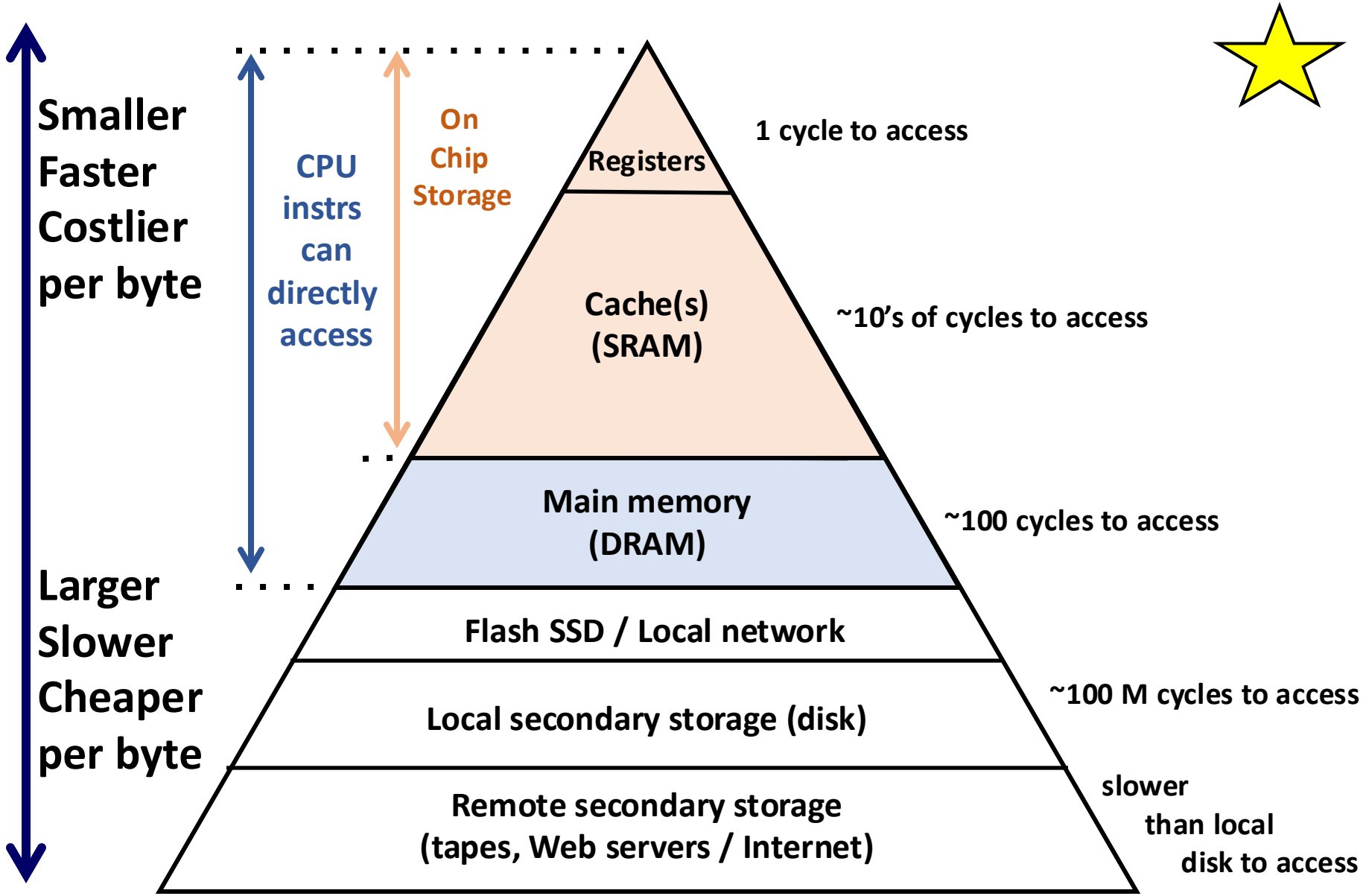
18 Caching and the OS

04-01-2025

Announcements

- CS 31 Final Exam: Singer 33: 9 – 12PM: May 12th
  - Final exam is inclusive of all the material in the course
  - You can request a reschedule if you have more than 2 exams in a 24 - 48 hour period.
  - All other travel related requests are not entertained ☹
- HW 6 Pushed out to next Monday
- CS 31 Midterm next Tuesday: Syllabus inclusive of Set-Associative Cache.
  - Practice Exam out today + topics (midterm 1 – midterm 2)
  - Please respond to accommodations requests for the midterm!!

Last class: The Memory Hierarchy



Memory Address Tells Us…

- Is the block containing the byte(s) you want already in the cache?
- If not, where should we put that block?
  - Do we need to kick out (“evict”) another block?
- Which byte(s) within the block do you want?

Memory Addresses for use with Cache

- Like everything else: series of bits (x86_64 has 64 bit addresses)

- Keep in mind:
  - N bits gives us 2^N unique values.

- 64-bit address:
  - 1011000101110010110101000101011010110001011100101101010001010110

Divide into regions, each with distinct meaning.

Address Division

- First section: Tag
  - Of all the addresses that map to this location, which one is here?
  - Number of bits for this section is any bits left over after index and offset.

- Second section: Index
  - Which location(s) in the cache should we check for the data with this address?
  - Number of bits for this section depends on the number of cache locations.

- Third section: Offset
  - If we find a block of bytes in the cache (on a hit) which byte offset within the block do we actually want?
  - Number of bits for this section depends on the block size – must be able to uniquely identify every byte in the block.

Address Division

10110001011100101101010001010110101100010111001011010100010101 10

Tag (X bits)	Index (Y bits)	Byte offset (Z bits)

- First section: **Tag**
  - Of all the addresses that map to this location, which one is here?
  - Uniquely identify the subset of memory contained within a cache line.
  - Number of bits for this section is any bits left over after index and offset.

- Second section: **Index**
  - Which location(s) in the cache should we check for the data with this address?
  - Number of bits for this section depends on the number of cache locations.

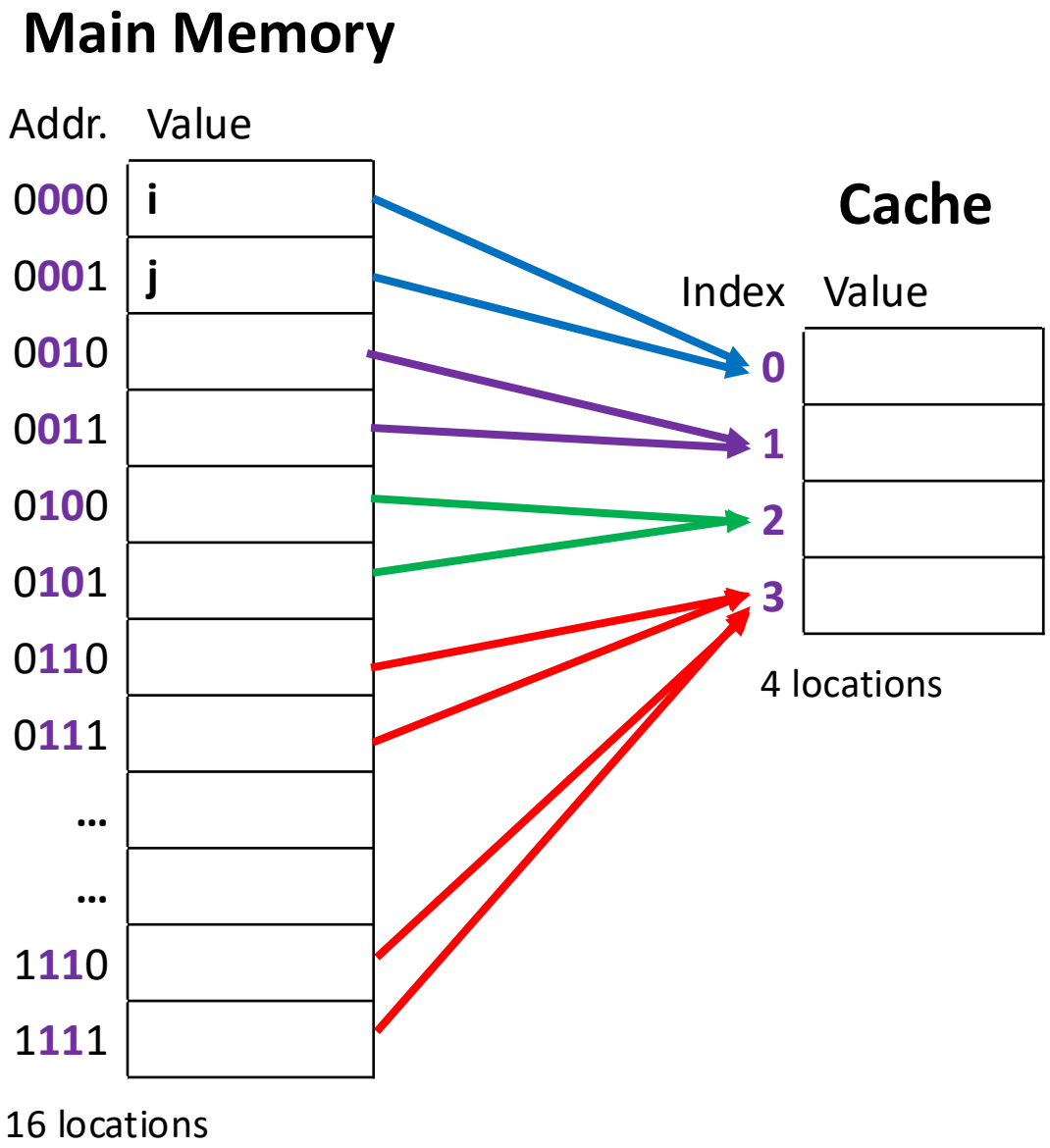
- Third section: **Offset**
  - If we find a block of bytes in the cache (on a hit) which *bytes within the block* do we actually want?
  - Number of bits for this section depends on the block size – must be able to uniquely identify every byte in the block.

A. **In exactly one place. ("Direct-mapped")**
   - **Every location in memory is directly mapped to one place in the cache. Easy to find data.**

B. In a few places. ("Set associative")
   - A memory location can be mapped to (2, 4, 8) locations in the cache. Middle ground.

A. Anywhere in the cache. ("Fully associative")
   - No restrictions on where memory can be placed in the cache. Fewer conflict misses, more searching.

Direct Mapped Cache



i and j map to the same cache line
and may constantly evict each other!

Direct-Mapped

- One place data can be.

- Example: let's assume some parameters:
  - 1024 cache locations (every block mapped to one)
  - Block size of 8 bytes

Suppose we had 8-bit addresses, a cache with 8 lines, and a block size of 4 bytes.

- How many bits would we use for:
  - Tag?
  - Index?
  - Offset?

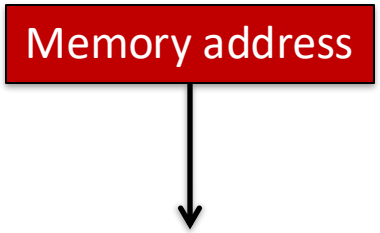
Suppose a system has 8-bit addresses, a DM cache with 8 lines, and 4-byte block size

- How many bits would be used for:
  - Byte-offset: 2 bits
    - 4 byte block size, can address each byte in 2 bits (00, 01, 10, 11)
  - Index: 3 bits
    - With 8 lines, need 3 bits to encode each cache line number
  - Tag: 3 bits
    - Bits left over in the address after byte-offset and index (8 – 2 – 3)

- Which bits would be used for:
  - Tag? high order
  - Index? middle (right after byte offset bits)
  - Byte offset? low order

ex. 01010011

How would the cache change if we performed the following memory operations?



Read 01000100 (Value: 5)

Read 11100010 (Value: 17)

Write 01110000 (Value: 7)

Read 10101010 (Value: 12)

Write 01101100 (Value: 2)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	011	9
2	0	0	101	15
3	1	1	001	8
4	1	0	011	4
5	0	0	111	6
6	0	0	101	32
7	1	0	110	3

How would the cache change if we performed the following memory operations?

Memory address

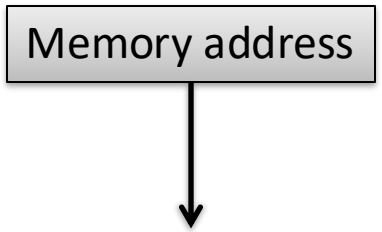
Tag: 3
Index: 3
Offset: 2

Read 01000100 (Value: 5) **MISS**
Read 11100010 (Value: 17)
Read 10111101 (Value: 2)
Write 01111100 (Value 10)
Write 01110000 (Value: 7)
Read 10101010 (Value: 12)
Write 01101100 (Value: 2)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	~~011~~ 010	~~9~~ 5
2	0	0	101	15
3	1	1	001	8
4	1	0	011	4
5	0	0	111	6
6	0	0	101	32
	1	0	110	3

At line 1, V=1 but tags don't match, so we have a MISS.
Dirty bit is 0, so we can safely overwrite it.
Write: V = 1; D = 0 (we're reading, not writing); tag, data (value)

How would the cache change if we performed the following memory operations?



Tag: 3
Index: 3
Offset: 2

Read 01000100 (Value: 5) **MISS**

Read 11100010 (Value: 17) **HIT**

Read 10111101 (Value: 2)

Write 01111100 (Value 10)

Write 01110000 (Value: 7)

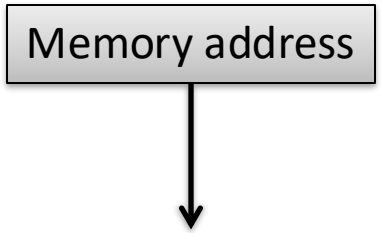
Read 10101010 (Value: 12)

Write 01101100 (Value: 2)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	~~011~~ 010	~~9~~ 5
2	0	0	101	15
3	1	1	001	8
4	1	0	011	4
5	0	0	111	6
6	0	0	101	32
	1	0	110	3

At line 1, V=1 and tags match, so we have a HIT.
What we wanted in cache is there, so we saved time!

How would the cache change if we performed the following memory operations?



Tag: 3
Index: 3
Offset: 2

Read 01000100 (Value: 5) **MISS**

Read 11100010 (Value: 17) **HIT**

Read 10111101 (Value: 2) **MISS**

Write 01111100 (Value 10)

Write 01110000 (Value: 7)

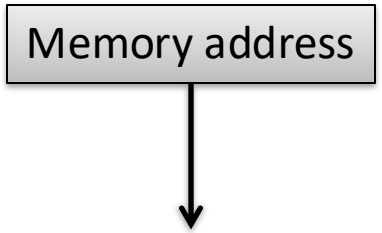
Read 10101010 (Value: 12)

Write 01101100 (Value: 2)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	~~011~~ 010	~~9~~ 5
2	0	0	101	15
3	1	1	001	8
4	1	0	011	4
5	0	0	111	6
6	0	0	101	32
	~~0~~ 1	0	~~110~~ 101	~~3~~ 2

At line 7, V=0 and tags don't match, so we have a MISS.
Dirty bit is 0, so we can safely overwrite it.
Write: V = 1; tag, data (value)

How would the cache change if we performed the following memory operations?



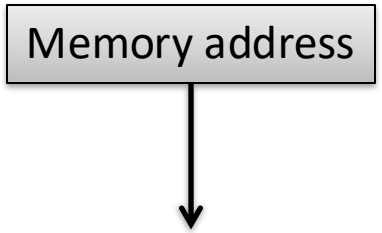
Tag: 3
Index: 3
Offset: 2

Read 01000100 (Value: 5) **MISS**
Read 11100010 (Value: 17) **HIT**
Read 10111101 (Value: 2) **MISS**
Write 01111100 (Value 10) **MISS**
Write 01110000 (Value: 7)
Read 10101010 (Value: 12)
Write 01101100 (Value: 2)

At line 7, V=1 but tags don't match, so we have a MISS. D=0, so we can safely overwrite it. Write: V = 1; D = 1 (we're updating cache but it is now out of sync with main memory); tag, data (value)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	~~011~~ 010	~~9~~ 5
2	0	0	101	15
3	1	1	001	8
4	1	0	011	4
5	0	0	111	6
6	0	0	101	32
7	~~0~~ ~~1~~ 1	~~0~~ 1	~~110~~ ~~101~~ 011	~~3~~ ~~2~~ 10

How would the cache change if we performed the following memory operations?



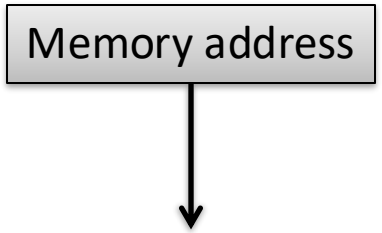
Tag: 3
Index: 3
Offset: 2

Read 01000100 (Value: 5) **MISS**
Read 11100010 (Value: 17) **HIT**
Read 10111101 (Value: 2) **MISS**
Write 01111100 (Value 10) **MISS**
Write 01110000 (Value: 7) **HIT**
Read 10101010 (Value: 12)
Write 01101100 (Value: 2)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	~~011~~ 010	~~9~~ 5
2	0	0	101	15
3	1	1	001	8
4	1	0	011	~~4~~ 7
5	0	0	111	6
6	0	0	101	32
7	~~0~~ ~~1~~ 1	~~0~~ 1	~~110~~ ~~101~~ 011	~~3~~ ~~2~~ 10

At line 4, V=1 and tags match, so we have a HIT. D=0. Write value 7. Set data (value); D = 1 (we're updating cache and it is now out of synch with main memory)

How would the cache change if we performed the following memory operations?



Tag: 3
Index: 3
Offset: 2

Read 01000100 (Value: 5) **MISS**
Read 11100010 (Value: 17) **HIT**
Read 10111101 (Value: 2) **MISS**
Write 01111100 (Value 10) **MISS**
Write 01110000 (Value: 7) **HIT**
Read 10101010 (Value: 12) **MISS**
Write 01101100 (Value: 2)

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	~~011~~ 010	~~9~~ 5
2	~~0~~ 1	0	~~101~~ 101	~~15~~ 12
3	1	1	001	8
4	1	0	011	~~4~~ 7
5	0	0	111	6
6	0	0	101	32
7	~~0~~ ~~1~~ 1	~~0~~ 1	~~110~~ ~~101~~ 011	3 ~~2~~ 10

At line 2, tags match but V=0, so we have a MISS. D=0, so no need to evict anything. Read value in from memory and store it in cache. Set V=1, D=0; tag; data (value).

How would the cache change if we performed the following memory operations?

Memory address

Tag: 3
Index: 3
Offset: 2

Read 01000100 (Value: 5) **MISS**
Read 11100010 (Value: 17) **HIT**
Read 10111101 (Value: 2) **MISS**
Write 01111100 (Value 10) **MISS**
Write 01110000 (Value: 7) **HIT**
Read 10101010 (Value: 12) **MISS**
Write 01101100 (Value: 2) **MISS**

At line 3, V=1 and tags don't match, so we have a MISS. D=1, so we need to save it to memory before we overwrite it. Set V=1, D=1; tag; data (value).

Line	V	D	Tag	Data (4 Bytes)
0	1	0	111	17
1	1	0	~~011~~ **010**	~~9~~ **5**
2	~~0~~ 1	0	~~101~~ **101**	~~15~~ **12**
3	~~1~~ **1**	~~1~~ **1**	~~001~~ **011**	~~8~~ **2**
4	1	0	011	~~4~~ **7**
5	0	0	111	6
6	0	0	101	32
7	~~0~~ ~~1~~ **1**	~~0~~ **1**	~~110~~ ~~101~~ **011**	3 ~~2~~ **10**

Associativity

- Problem: suppose we're only using a small amount of data (e.g., 8 bytes, 4-byte block size)

- Bad luck: (both) blocks map to same cache line
  - Constantly evicting one another
  - Rest of cache is going unused!

- Associativity: allow a set blocks to be stored at the same index. Goal: reduce conflict misses.

Direct-mapped vs N-way set associative Cache

Direct-mapped

- Tag tells you if you found the correct data.
- Offset specifies which byte within block.
- *Middle bits (index) tell you which 1 line to check.*

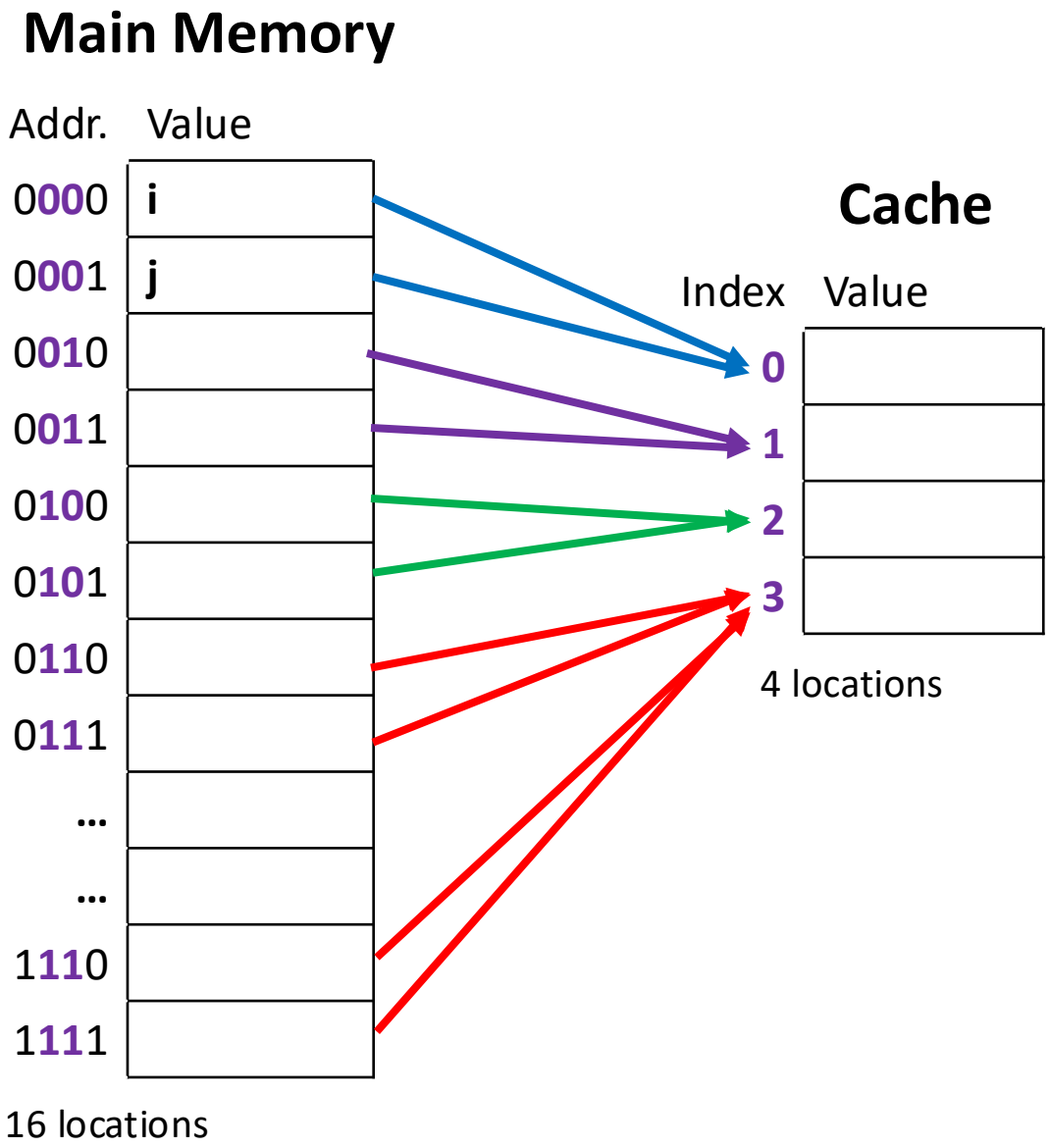
- (+) Low complexity, fast.
- (-) Conflict misses.

N-way set associative

- Tag tells you if you found the correct data.
- Offset specifies which byte within block.
- *Middle bits (set) tell you which N lines to check.*

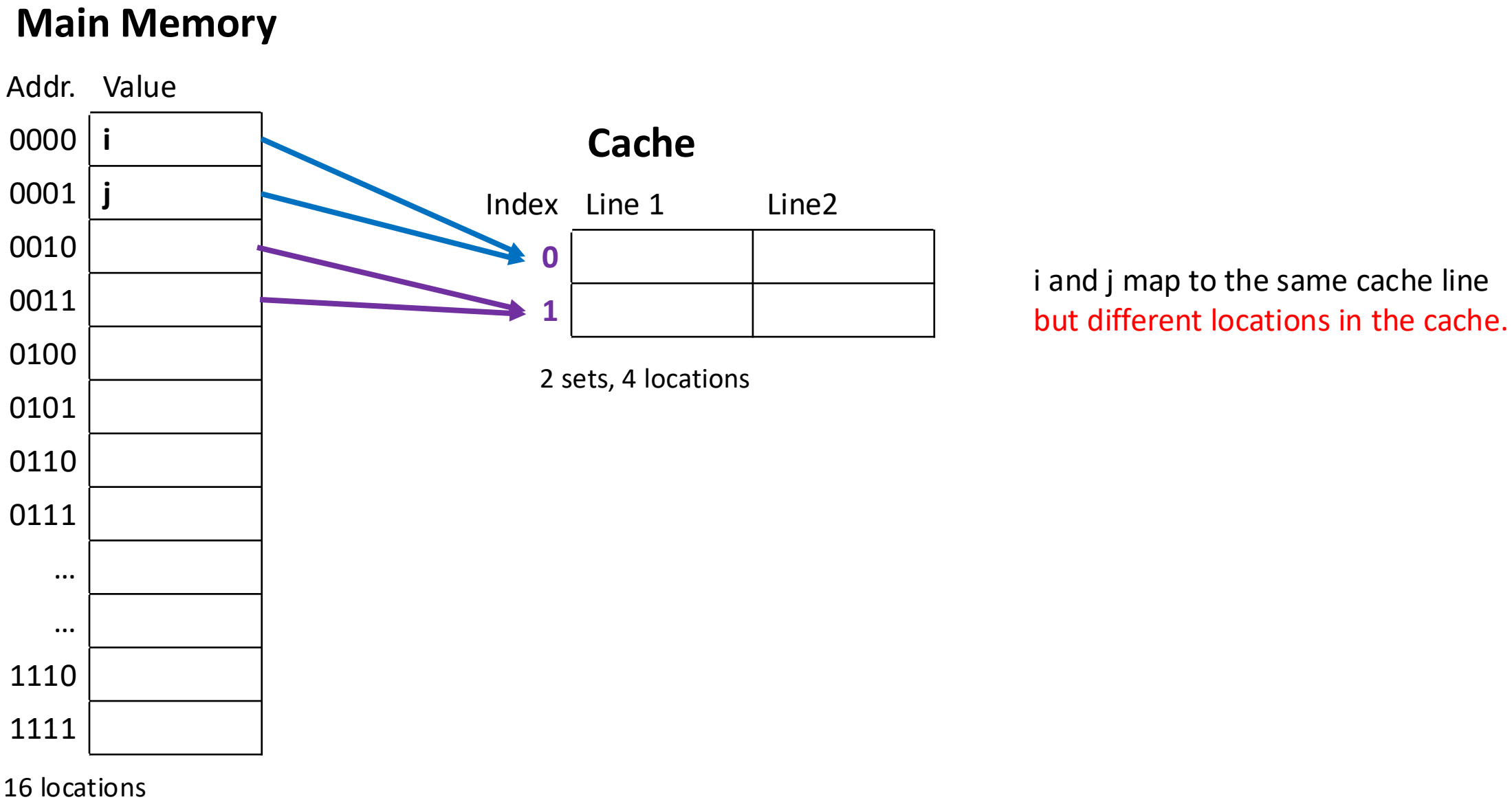
- (+) Fewer conflict misses.
- (-) More complex, slower, consumes more power.

Direct Mapped Cache



i and j map to the same cache line
and may constantly evict each other!

Set Associative Cache



Comparison: 1024 Lines

(For the same cache size, in bytes.)

Direct-mapped

- 1024 indices (10 bits)

2-way set associative

- 512 sets (9 bits)
  - Tag slightly (1 bit) larger.

Set #	V	D	Tag	Data (8 Bytes)	V	D	Tag	Data (8 Bytes)
0								
1								
2								
3								
4								
…			…				…	
508								
509								
510								
511								

2-Way Set Associative

Tag (52 bits)	Set (9 bits)	Byte offset (3 bits)
3941	4	

Same capacity as previous example:
1024 rows with 1 entry vs.
512 rows with 2 entries

Set #	V	D	Tag	Data (8 Bytes)	V	D	Tag	Data (8 Bytes)
0								
1								
2								
3								
4	1	1	4063		1	0	3941	
…			…				…	
508								
509								
510								
511								

2-Way Set Associative

Tag (52 bits)	Set (9 bits)	Byte offset (3 bits)
3941	4	

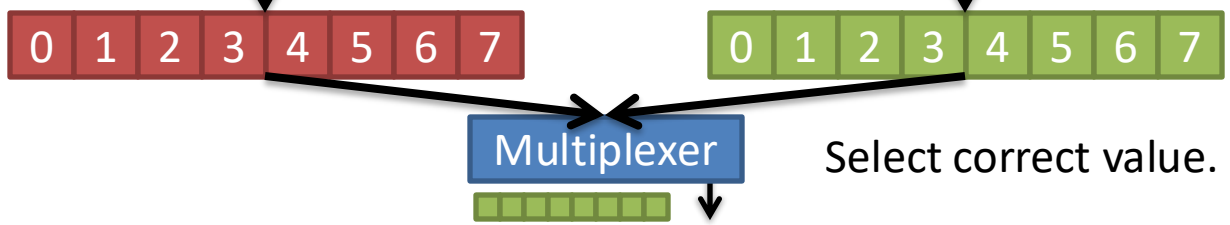
Set #	V	D	Tag	Data (8 Bytes)	V	D	Tag	Data (8 Bytes)
0								
1								
2								
3								
4	1	1	4063		1	0	3941	
…			…				…	
508								
509								
510								
511								

Check all locations in the set, in parallel.

2-Way Set Associative

Tag (52 bits)	Set (9 bits)	Byte offset (3 bits)
3941	4	

Set #	V	D	Tag	Data (8 Bytes)	V	D	Tag	Data (8 Bytes)
0								
1								
2								
3								
4	1	1	4063		1	0	3941	
…			…				…	
508								
509								
510								
511								



Eviction

- Mechanism is the same…
  - Overwrite bits in cache line: update tag, valid, data

- Policy: choose which line in the set to evict
  - Pick a random line in set
  - Choose an invalid line first
  - Choose the least recently used block
    - Has exhibited the least locality, kick it out!

Common combo in practice.

Least Recently Used (LRU)

- Intuition: if it hasn't been used in a while, we have no reason to believe it will be used soon.

- Need extra state to keep track of LRU info: which line is least recently used -> left or right?

Set #	LRU	V	D	Tag	Data (8 Bytes)	V	D	Tag	Data (8 Bytes)
0	0								
1	1								
2	1								
3	0								
4	1	1	1	4063		1	0	3941	
…		…				…			

Least Recently Used (LRU)

- Intuition: if it hasn't been used in a while, we have no reason to believe it will be used soon.

- Need extra state to keep track of LRU info.

- For perfect LRU info:
  - 2-way: 1 bit
  - 4-way: 8 bits
  - N-way: N * $\log_2$ N bits

Another reason why associativity often maxes out at 8 or 16.

These are metadata bits, not "useful" program data storage.

(Approximations make it not quite as bad.)

Suppose a system has 8-bit addresses, a two-way set associative cache with 8 lines, and 4-byte block size

- How many bits would we use for:
  - Tag?
  - Index?
  - Offset?

Suppose a system has 8-bit addresses, a two-way set associative cache with 8 lines, and 4-byte block size

- How many bits would we use for:
  - Tag? **3**
  - Set? **3**
  - Offset? **2**

How would the cache change if we performed the following memory operations? (2-way set)

Read 01000100 (Value: 5)

Read 11100010 (Value: 17)

Write 01100100 (Value: 7)

Read 01000110 (Value: 5)

Write 01100000 (Value: 2)

Tag:
Set:
Offset:

LRU = 0 means the **left line** in the set was least recently used.

LRU = 1 means the **right line** was used least recently.

Set #	LRU	V	D	Tag	Data (4 Bytes)	V	D	Tag	Data (4 Bytes)
0	0	0	0	111	4	1	0	001	17
1	1	1	1	111	9	1	0	010	5
2				…	…			…	…
3									
4									
5									
6									
7									

How would the cache change if we performed the following memory operations? (2-way set)

HIT → Read 01000100 (Value: 5)

Read 11100010 (Value: 17)

Write 01100100 (Value: 7)

Read 01000110 (Value: 5)

Write 01100000 (Value: 2)

Tag: 3
Set: 3
Offset: 2

LRU = 0 means the **left line** in the set was least recently used.

LRU = 1 means the **right line** was used least recently.

Set #	LRU	V	D	Tag	Data (4 Bytes)	V	D	Tag	Data (4 Bytes)
0	0	0	0	111	4	1	0	001	17
→ 1	~~1~~ **0**	1	1	111	9	1	0	**010**	**5**
2				…	…			…	…
3									
4									
5									
6									
7									

How would the cache change if we performed the following memory operations? (2-way set)

HIT

MISS → Read 01000100 (Value: 5)

Read 11100010 (Value: 17)

Write 01100100 (Value: 7)

Read 01000110 (Value: 5)

Write 01100000 (Value: 2)

Tag: 3
Index: 3
Offset: 2

LRU = 0 means the **left line** in the set was least recently used.

LRU = 1 means the **right line** was used least recently.

Set #	LRU	V	D	Tag	Data (4 Bytes)	V	D	Tag	Data (4 Bytes)
→ 0	~~0~~ 1	~~0~~ 1	0	**111**	~~4~~ 17	1	0	001	17
1	~~1~~ 0	1	1	111	9	1	0	**010**	**5**
2				…	…			…	…
3									
4									
5									
6									
7									

How would the cache change if we performed the following memory operations? (2-way set)

HIT Read 01000100 (Value: 5)

MISS Read 11100010 (Value: 17)

MISS → Write 01100100 (Value: 7)

Read 01000110 (Value: 5)

Write 01100000 (Value: 2)

Tag: 3
Index: 3
Offset: 2

LRU = 0 means the **left line** in the set was least recently used.

LRU = 1 means the **right line** was used least recently.

Set #	LRU	V	D	Tag	Data (4 Bytes)	V	D	Tag	Data (4 Bytes)
0	~~0~~ 1	~~0~~ 1	0	111	~~4~~ 17	1	0	001	17
→ 1	~~1~~ ~~0~~ 1	~~1~~ 1	~~1~~ 1	~~111~~ 011	~~9~~ 7	1	0	010	5
2				…	…			…	…
3									
4									
5									
6									
7									

How would the cache change if we performed the following memory operations? (2-way set)

HIT Read 01000100 (Value: 5)

MISS Read 11100010 (Value: 17)

MISS Write 01100100 (Value: 7)

HIT Read 01000110 (Value: 5)

Write 01100000 (Value: 2)

Tag: 3
Index: 3
Offset: 2

LRU = 0 means the **left line** in the set was least recently used.

LRU = 1 means the **right line** was used least recently.

Set #	LRU	V	D	Tag	Data (4 Bytes)	V	D	Tag	Data (4 Bytes)
0	~~0~~ 1	~~0~~ 1	0	111	~~4~~ 17	1	0	001	17
1	0 ~~1~~ 0	~~1~~ 1	~~1~~ 1	~~111~~ 011	~~9~~ 7	1	0	010	5
2				…	…			…	…
3									
4									
5									
6									
7									

How would the cache change if we performed the following memory operations? (2-way set)

HIT	Read 01000100 (Value: 5)
MISS	Read 11100010 (Value: 17)
MISS	Write 01100100 (Value: 7)
HIT	Read 01000110 (Value: 5)
➡	Write 01100000 (Value: 2)

Tag: 3
Index: 3
Offset: 2

LRU = 0 means the **left line** in the set was least recently used.

LRU = 1 means the **right line** was used least recently.

Set #	LRU	V	D	Tag	Data (4 Bytes)	V	D	Tag	Data (4 Bytes)
➡ 0	0 ~~0~~ ~~1~~ 1	~~0~~ 1	0	111	~~4~~ 17	~~1~~ 1	~~0~~ 1	~~001~~ 011	~~17~~ 2
1	0 ~~1~~ ~~0~~ 1	~~1~~ 1	~~1~~ 1	~~111~~ 011	~~9~~ 7	1	0	010	5
2				…	…			…	…
3									
4									
5									
6									
7									

Cache Conscious Programming

Knowing about caching and designing code around it can significantly effect performance

(ex) 2D array accesses

```
for(i=0; i < N; i++) {
  for(j=0; j< M; j++) {
      sum += arr[i][j];
}}
```

```
for(j=0; j < M; j++) {
  for(i=0; i< N; i++) {
     sum += arr[i][j];
}}
```

Algorithmically, both O(N * M).

Is one faster than the other?

Cache Conscious Programming

Knowing about caching and designing code around it can significantly effect performance

(ex) 2D array accesses

```
for(i=0; i < N; i++) {
  for(j=0; j< M; j++) {
      sum += arr[i][j];
}}
```

A. is faster.

```
for(j=0; j < M; j++) {
  for(i=0; i< N; i++) {
     sum += arr[i][j];
}}
```

B. is faster.

Algorithmically, both O(N * M).

Is one faster than the other?

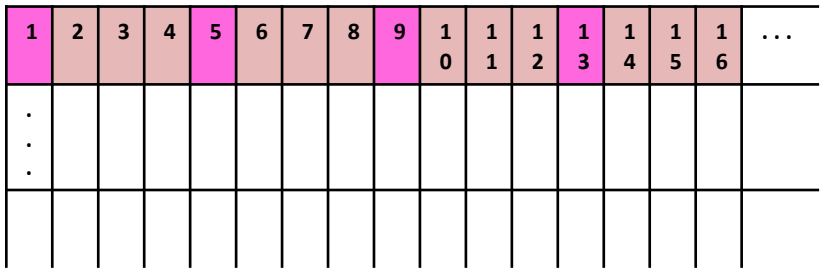
C. Both would exhibit roughly equal performance.

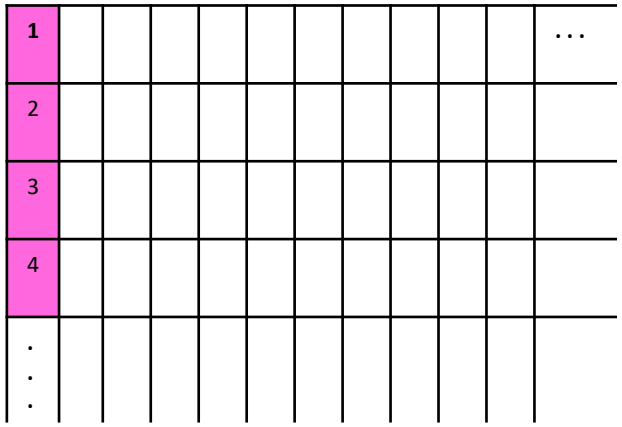
Cache Conscious Programming

The first nested loop is more efficient if the cache block size is larger than a single array bucket (for arrays of basic C types, it will be).

```
for(i=0; i < N; i++) {
  for(j=0; j< M; j++) {
      sum += arr[i][j];
}}
```

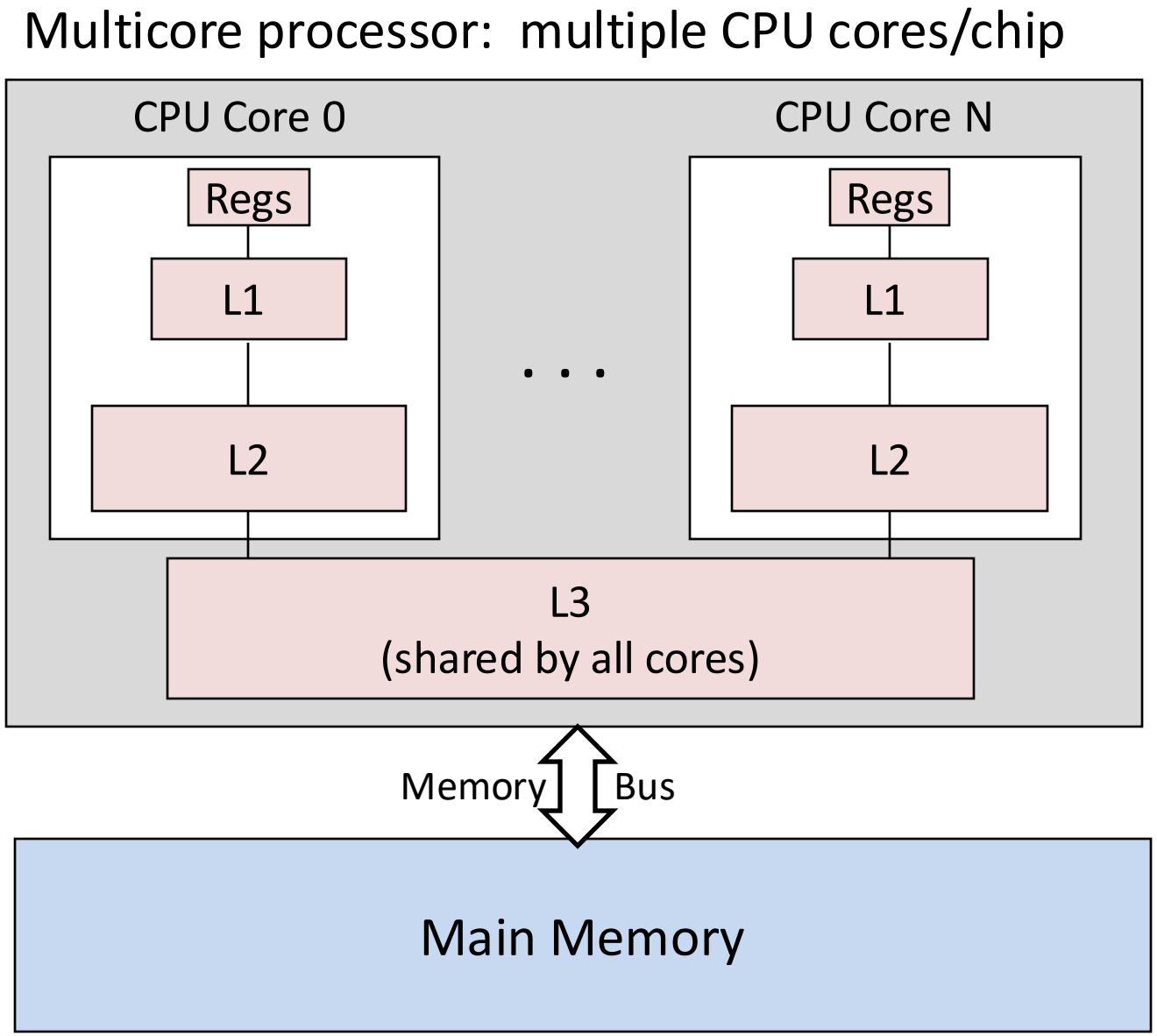
```
for(j=0; j < M; j++) {
  for(i=0; i< N; i++) {
     sum += arr[i][j];
}}
```





(ex) 1 miss every 4 buckets vs. 1 miss every bucket

FYI: Example Cache Organization



Example Sizes & Access Times

(KB: 2^{10} MB: 2^{20} GB: 2^{30})

L1: Size: 32 KB
Access: 4 cycles
write through
low associativity

L2: Size: 256 KB
Access: 10 cycles

L3: Size: 8 MB
Access: 40 cycles
write back
higher associativity

Main Memory (off chip):
Size: 16 GB
Access: 100 cycles

Program Efficiency and Memory

- Be aware of how your program accesses data
  - Sequentially, in strides of size X, randomly, …
  - How data is laid out in memory

- Will allow you to structure your code to run much more efficiently based on how it accesses its data

- Don’t go nuts…
  - Optimize the most important parts, ignore the rest
  - “Premature optimization is the root of all evil.” -Knuth

Amdahl's Law

Idea: an optimization can improve total runtime at most by the fraction it contributes to total runtime

> If program takes 100 secs to run*, and you optimize a portion of the code that accounts for 2% of the runtime,* the best your optimization can do is improve the runtime by 2 secs!

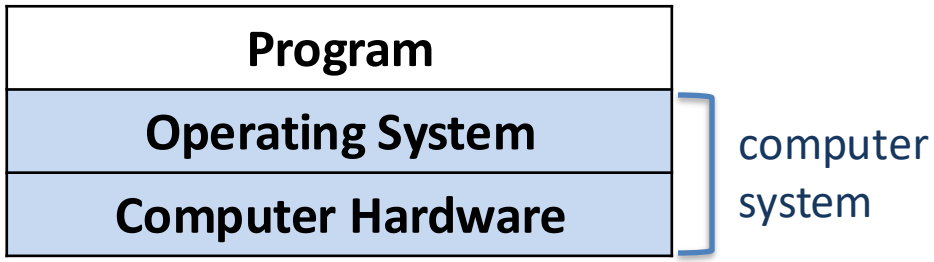
Amdahl's Law tells us to focus our optimization efforts on the code that matters:

> Speed-up what is accounting for the largest portion of runtime to get the largest benefit. And, don't waste time on the small stuff.

Up Next:

- Operating systems, Processes
- Virtual Memory

What do we know so far?



(1) How a Computer Runs a Program:

`./a.out` to x86 instructions being executed by HW

Program: how instructions and data are encoded to run on HW
HW: how implemented to run program instrs on program data

- CPU implementation and how it works (Fetch, Decode, Execute, Store)
- Memory Hierarchy, different storage types/devices
- CPU Cache implementation(DM and SA) and how it works

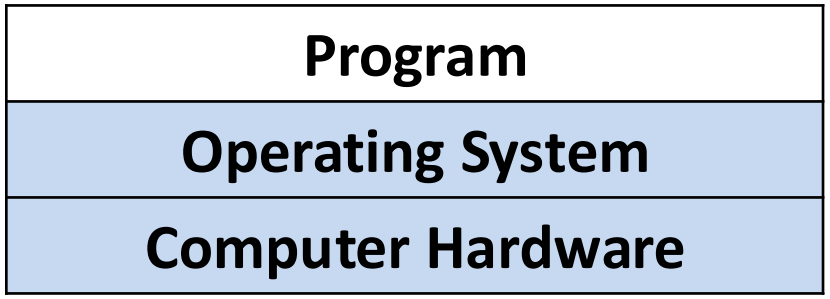
(2) How to Efficiently Run Programs

Compiler's role in producing efficient code (little bits of this)
The Memory Hierarchy and its effect performance

- Caching motivated by lots of locality in program execution

The Operating System

(1)Its role in how a Computer Runs a Program:



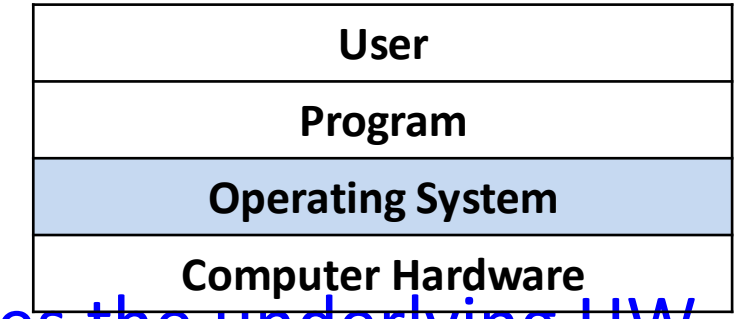
`./a.out` to IA32 instructions being executed by HW

(2) How to Efficiently Run Programs

- Compiler's role in producing efficient code
- The Memory Hierarchy and its effect performance
- OS abstractions for running programs efficiently

What is an Operating System?

Sits between the HW and the User/Program:



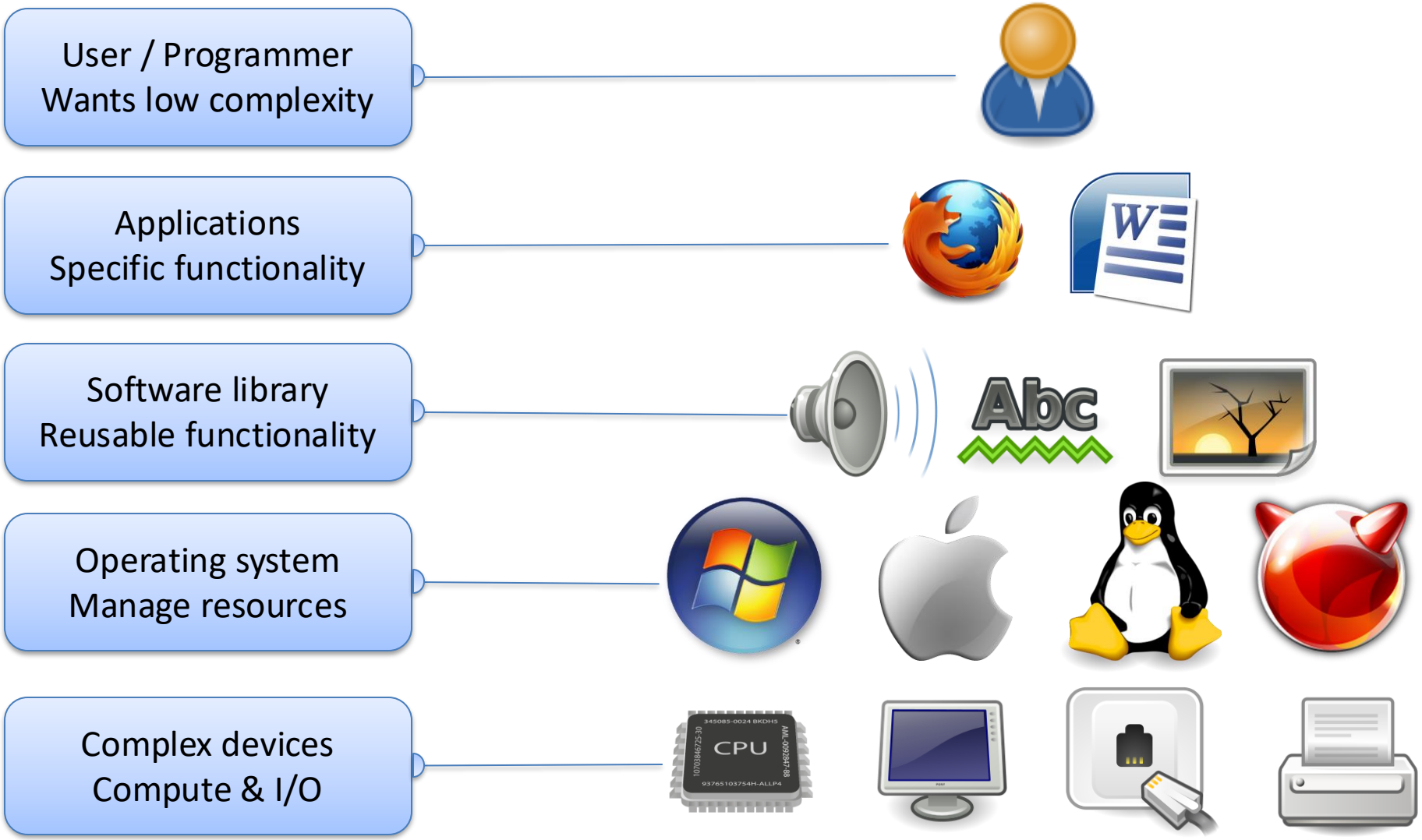
1. Manages the underlying HW
   - Coordinates shared access to HW

     `ps -A` (lots of processes sharing CPU, RAM, disk, …)
   - Efficiently schedules/manages HW resources
2. Provides easy-to-use interface to the HW
   - just type: `./a.out` to run a program

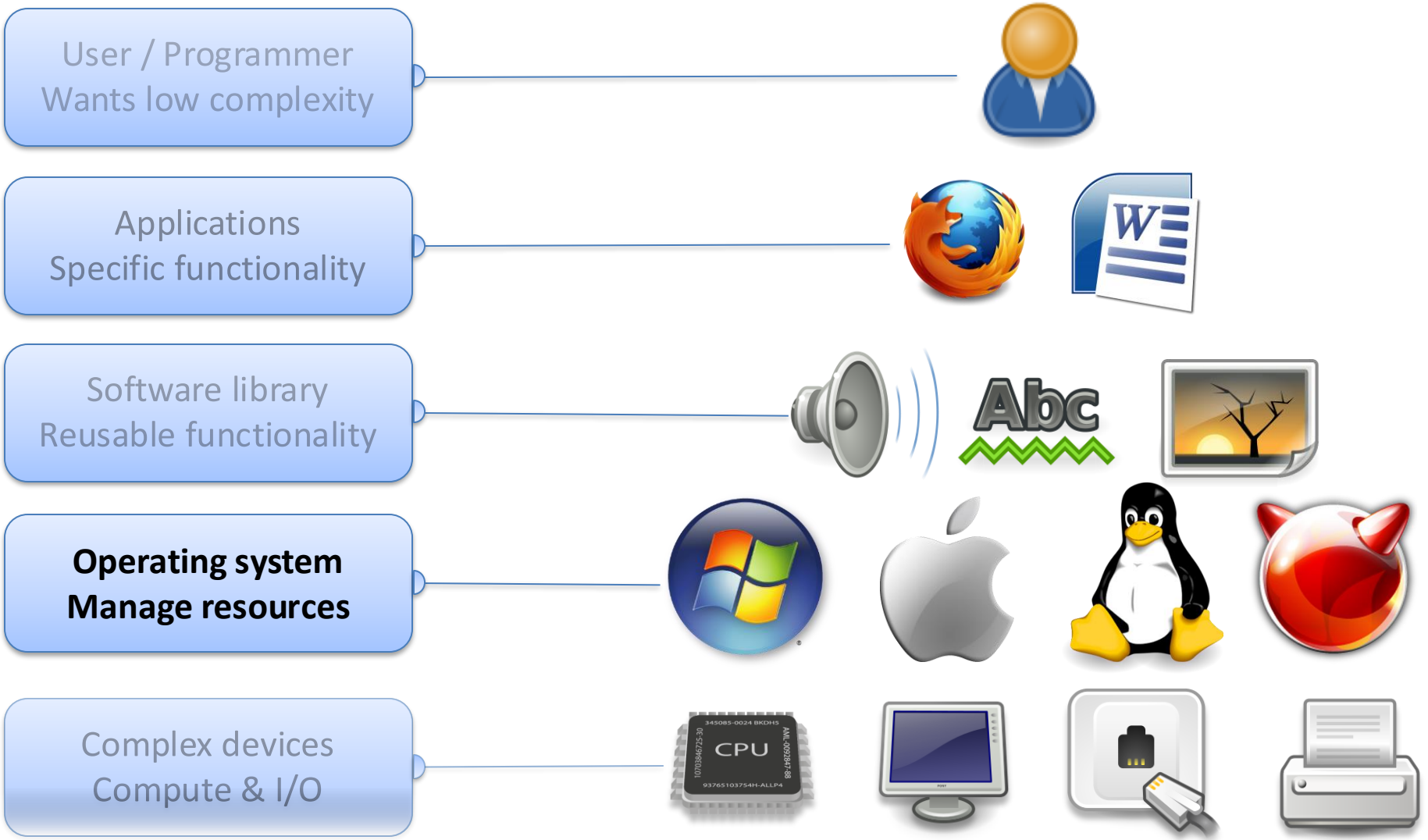
OS Big Picture Goals

- OS is a layer of code between user programs and hardware.
- Goal: Make life easier for users and programmers.
- How can the OS do that?

Abstraction



Abstraction



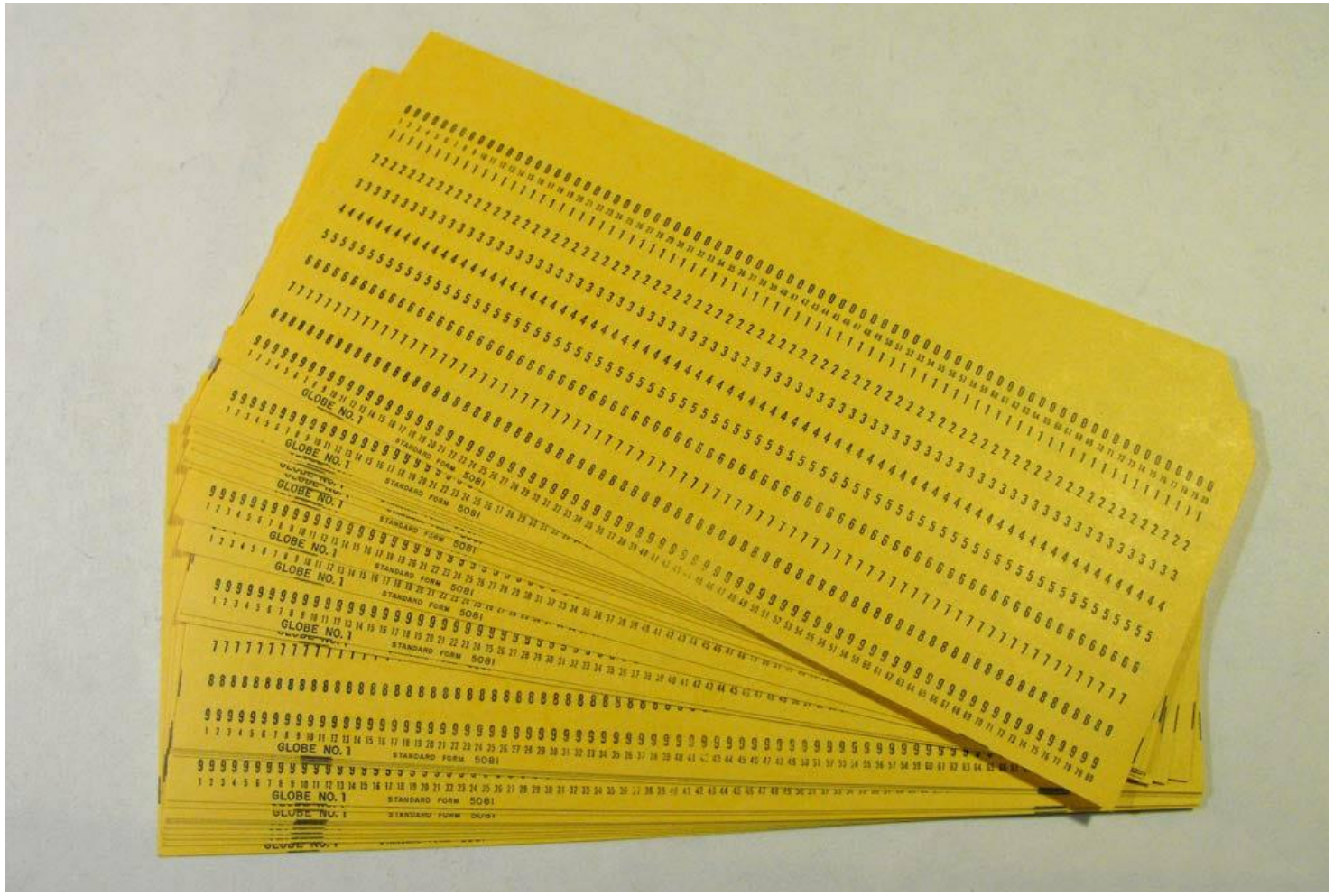
Key OS Responsibilities

1. Simplifying abstractions for programs
2. Resource allocation and/or sharing
3. Hardware gatekeeping and protection

OS: Turn undesirable into desirable

- Turn undesirable inconveniences: reality
  - Complexity of hardware
  - Single processor
  - Limited memory

Before Operating Systems



- One program executed at a time…

Why is it not ideal to have only a single program available to the hardware?

A. The hardware might run out of work to do.

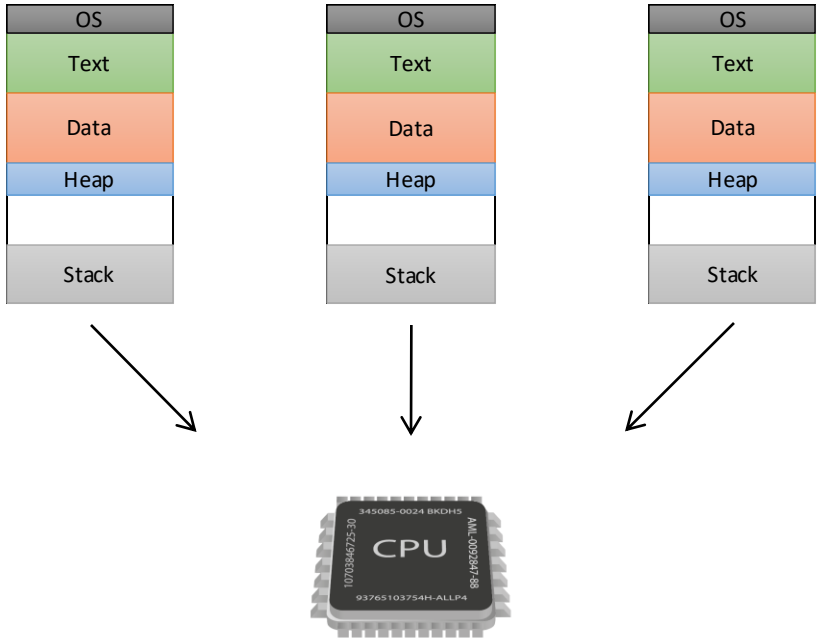
B. The hardware won't execute as quickly.

C. The hardware's resources won't be used as efficiently.

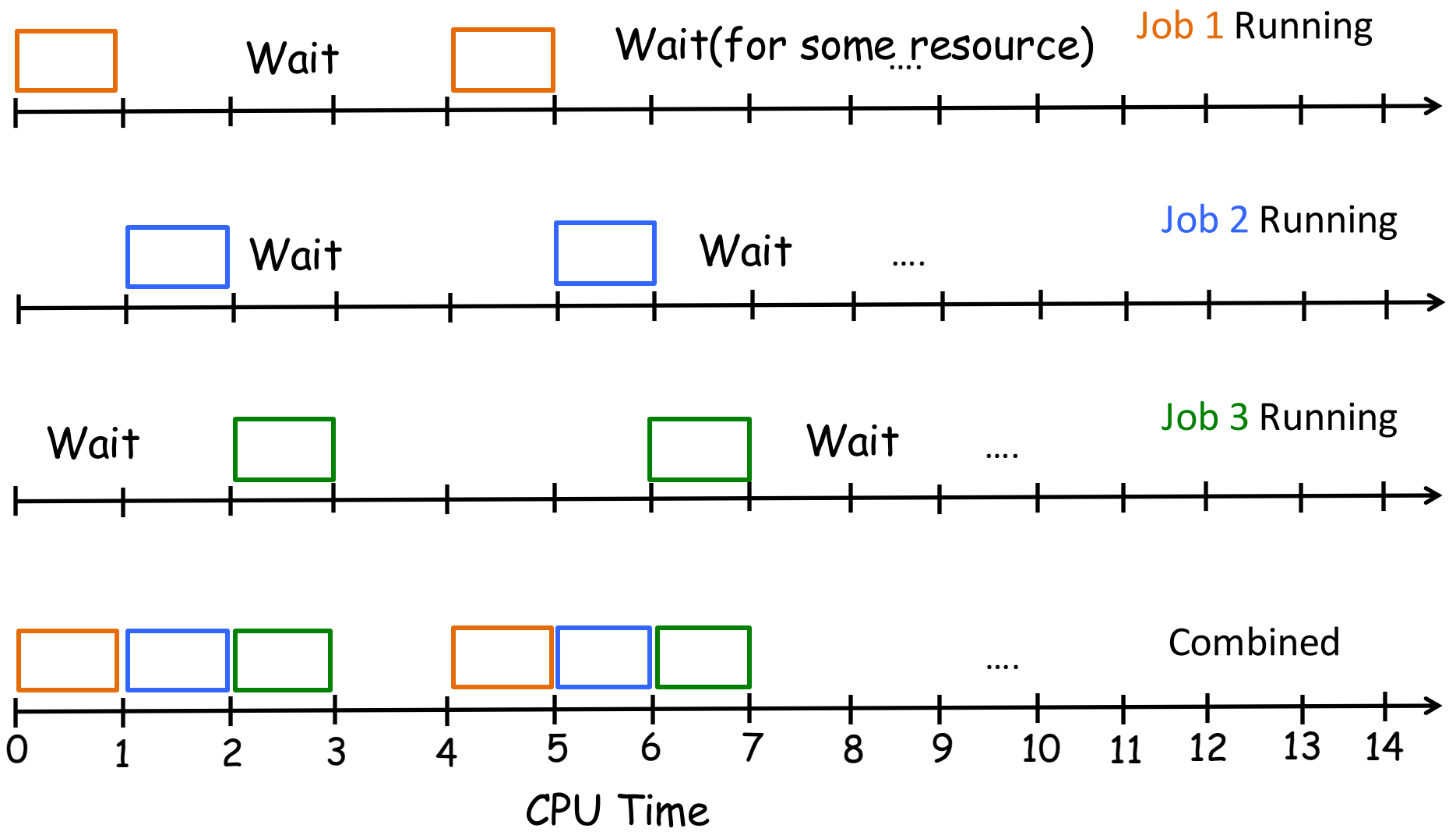
D. Some other reason(s). (What?)

Today: Multiprogramming

- Multiprogramming: have multiple programs available to the machine, even if you only have one CPU core that can execute them.



Multiprogramming on one core



How many programs do you think are running on a typical desktop computer?

A. 1-10

B. 20-40

C. 40-80

D. 80-160

E. 160+

Running multiple programs

More than 200 processes running on a typical desktop!

- Benefits: when I/O issued, CPU not needed
  - Allow another program to run
  - Requires yielding and sharing memory

- Challenges: what if one running program…
  - Monopolizes CPU, memory?
  - Reads/writes another's memory?
  - Uses I/O device being used by another?

OS: Turn undesirable into desirable

- Turn undesirable inconveniences: reality
  - Complexity of hardware
  - Single processor
  - Limited memory
- Into desirable conveniences: illusions
  - Simple, easy-to-use resources
  - Multiple/unlimited number of processors
  - Large/unlimited amount of memory

Virtualization

- Rather than exposing real hardware, introduce a “virtual”, abstract notion of the resource

- Multiple virtual processors
  - By rapidly switching CPU use
- Multiple virtual memories
  - By memory partitioning and re-addressing
- Virtualized devices
  - By simplifying interfaces, and using other resources to enhance function

We'll focus on the OS 'kernel'

- "Operating system" has many interpretations
  - E.g., all software on machine minus applications (user or even limited to 3rd party)

- Our focus is the *kernel*
  - What's necessary for everything else to work
  - Low-level resource control
  - Originally called the nucleus in the 60's

The Kernel

- All programs depend on it
  - Loads and runs them
  - Exports system calls to programs
- Works closely with hardware
  - Accesses devices
  - Responds to interrupts (hardware events)
- Allocates basic resources
  - CPU time, memory space
  - Controls I/O devices: display, keyboard, disk, network

Tron, 1982

Kernel provides common functions

- Some functions useful to many programs
  - I/O device control
  - Memory allocation
- Place these functions in central place (kernel)
  - Called by programs ("system calls")
  - Or accessed in response to hardware events
- What should functions be?
  - How many programs should benefit?
  - Might kernel get too big?

OS Kernel

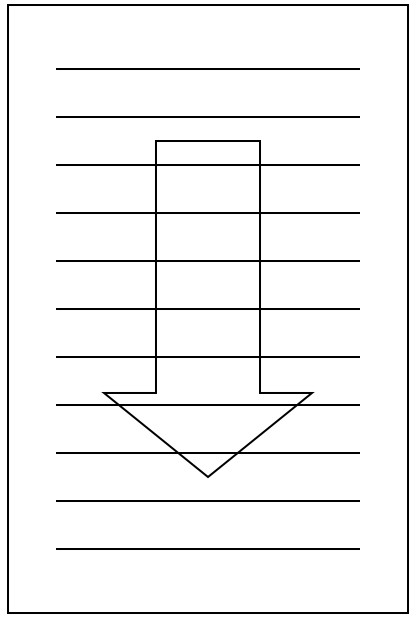
- Big Design Issue: How do we make the OS efficient, reliable, and extensible?

- General OS Philosophy: The design and implementation of an OS involves a constant tradeoff between simplicity and performance.

- As a general rule, strive for simplicity.
  - except when you have a strong reason to believe that you need to make a particular component complicated to achieve acceptable performance
  - (strong reason = simulation or evaluation study)

Main Abstraction: The Process

- Abstraction of a running program
  - "a program in execution"
- Dynamic
  - Has state, changes over time
  - Whereas a program is static
- Basic operations
  - Start/end
  - Suspend/resume



Basic Resources for Processes

- To run, process needs some basic resources:
  - CPU
    - Processing cycles (time)
    - To execute instructions
  - Memory
    - Bytes or words (space)
    - To maintain state
  - Other resources (e.g., I/O)
    - Network, disk, terminal, printer, etc.

What sort of information might the OS need to store to keep track of a running process?

- That is, what MUST an OS know about a process?
- (Discuss with your neighbors.)

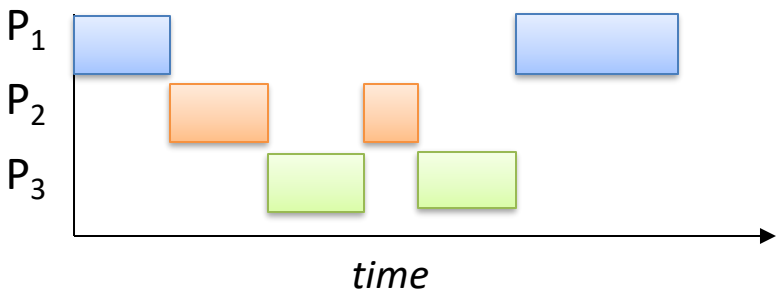
Machine State of a Process

- CPU or processor context
  - PC (program counter)
  - SP (stack pointer)
  - General purpose registers
- Memory
  - Code
  - Global Variables
  - Stack of activation records / frames
  - Other (registers, memory, kernel-related state)

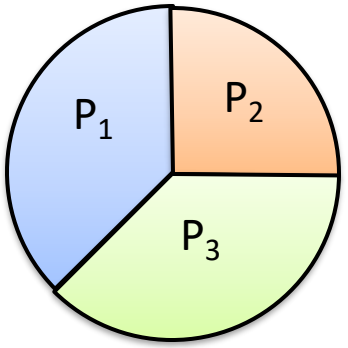
Must keep track of these for every running process !

Resource Sharing

CPU: Time



Memory: Space



Reality

- Multiple processes
- Small number of CPUs
- Finite memory

Abstraction

- Process is all alone
- Process is always running
- Process has all the memory

Resource: CPU

- Many processes, limited number of CPUs.
- Each process needs to make progress over time. Insight: processes don't know how quickly they *should* be making progress.
- Illusion: every process is making progress in parallel.

Timesharing: Sharing the CPUs

- Abstraction goal: make every process *think* it's running on the CPU all the time.
  - Alternatively: If a process was removed from the CPU and then given it back, it shouldn't be able to tell

- Reality: put a process on CPU, let it run for a short time (~10 ms), switch to another, … ("context switching")

How is Timesharing Implemented?

- Kernel keeps track of progress of each process
- Characterizes state of process's progress
  - Running: actually making progress, using CPU
  - Ready: able to make progress, but not using CPU
  - Blocked: not able to make progress, can't use CPU
- Kernel selects a ready process, lets it run
  - Eventually, the kernel gets back control
  - Selects another ready process to run, …

Multiprogramming

- Given a running process
  - At some point, it needs a resource, e.g., I/O device
  - If resource is busy (or slow), process can't proceed
  - "Voluntarily" gives up CPU to another process

- Mechanism: Context switching

Time Sharing / Multiprogramming

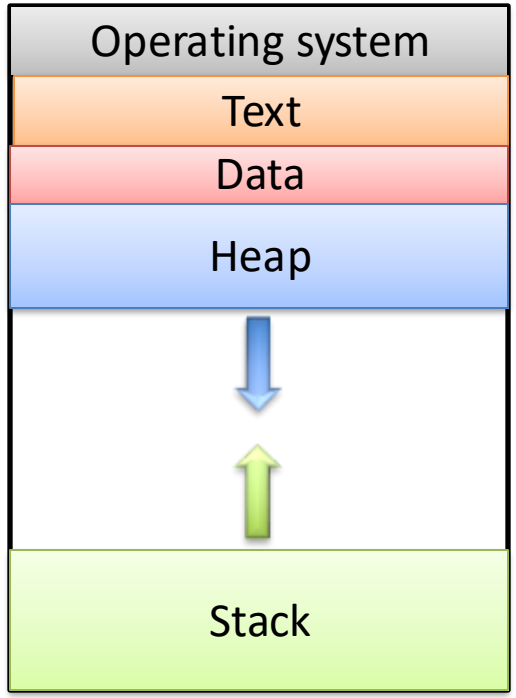
- Given a running process
  - At some point, it needs a resource, e.g., I/O device
  - If resource is busy (or slow), process can't proceed
  - "Voluntarily" gives up CPU to another process

- Mechanism: Context switching
- Policy: CPU scheduling

Resource: Memory

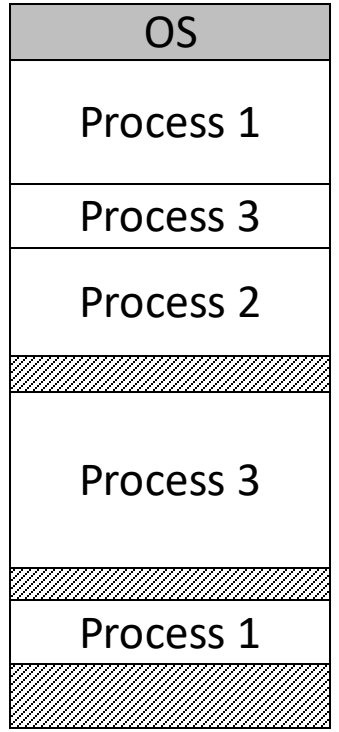
Abstraction goal: make every process think it has the same memory layout.

- MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.



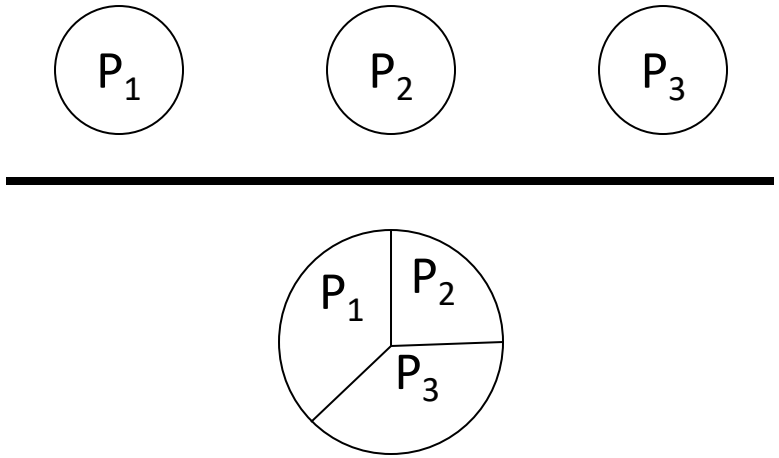
Memory

- Abstraction goal: make every process think it has the same memory layout.
  - MUCH simpler for compiler if the stack always starts at 0xFFFFFFFF, etc.
- Reality: there's only so much memory to go around
  - no two processes should use the same (physical) memory addresses (unless they're sharing).



OS (with help from hardware) will keep track of who's using each memory region.

Virtual Memory: Sharing Storage



- Like CPU cache, memory is a cache for disk.

- Processes never need to know where their memory truly is, OS translates virtual addresses into physical addresses for them.

Kernel Execution

- Great, the OS is going to somehow give us these nice abstractions.

- So…how / when should the kernel execute to make all this stuff happen?

The operating system kernel…

A. Executes as a process.

B. Is always executing, in support of other processes.

C. Should execute as little as possible.

D. More than one of the above. (Which ones?)

E. None of the above.

Process vs. Kernel

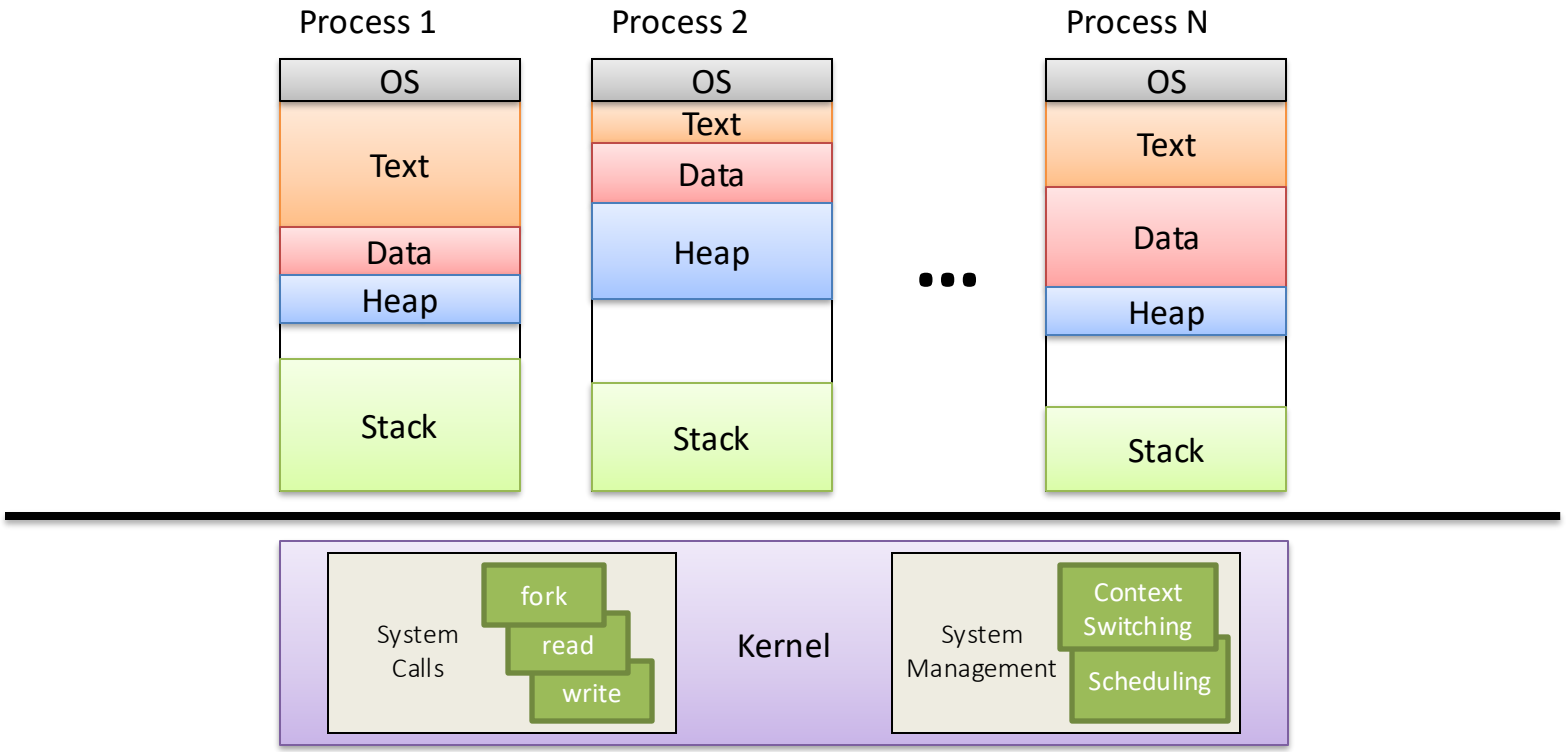
- Is the kernel itself a process?
  - No, it supports processes and devices

- OS only runs when necessary…
  - as an extension of a process making system call
  - in response to a device issuing an interrupt

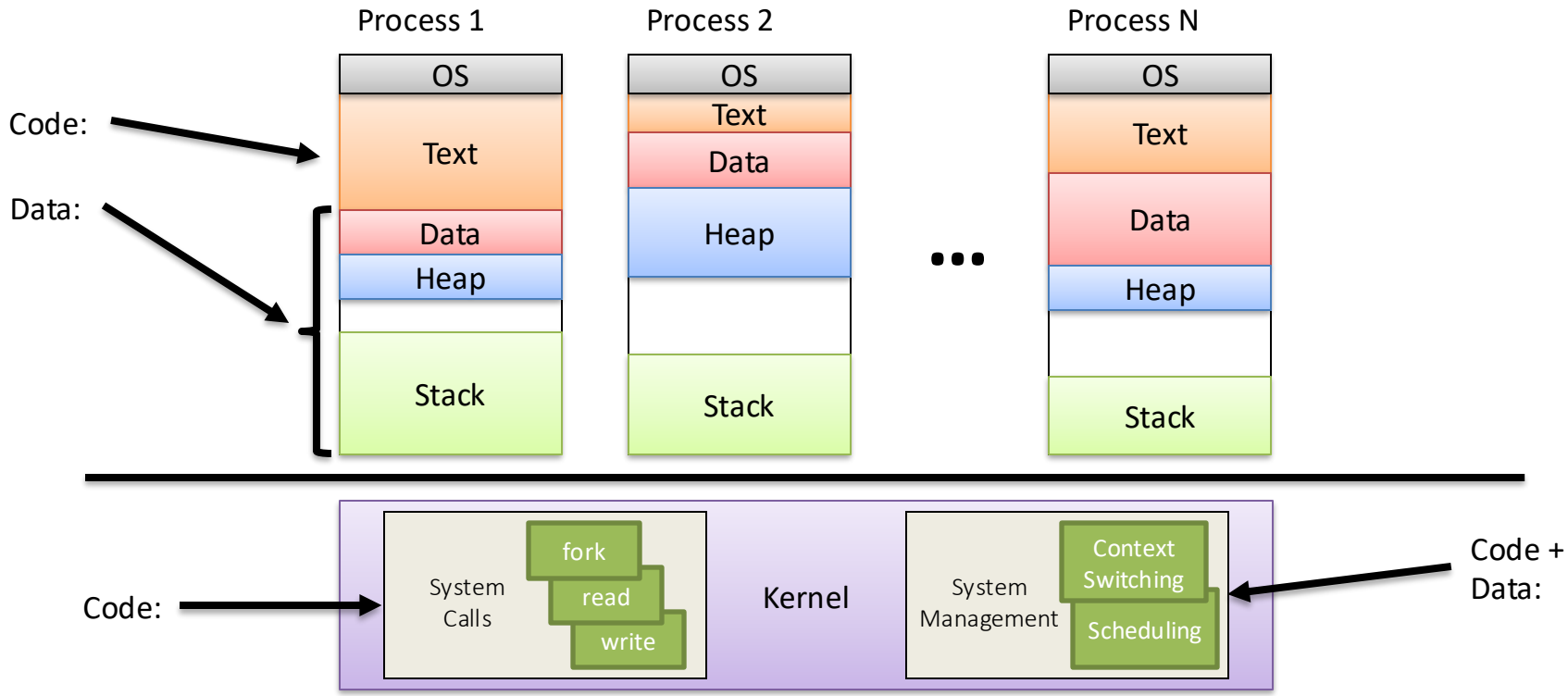
Process vs. Kernel

- The kernel is the code that supports processes
  - System calls: fork ( ), exit ( ), read ( ), write ( ), …
  - System management: context switching, scheduling, memory management

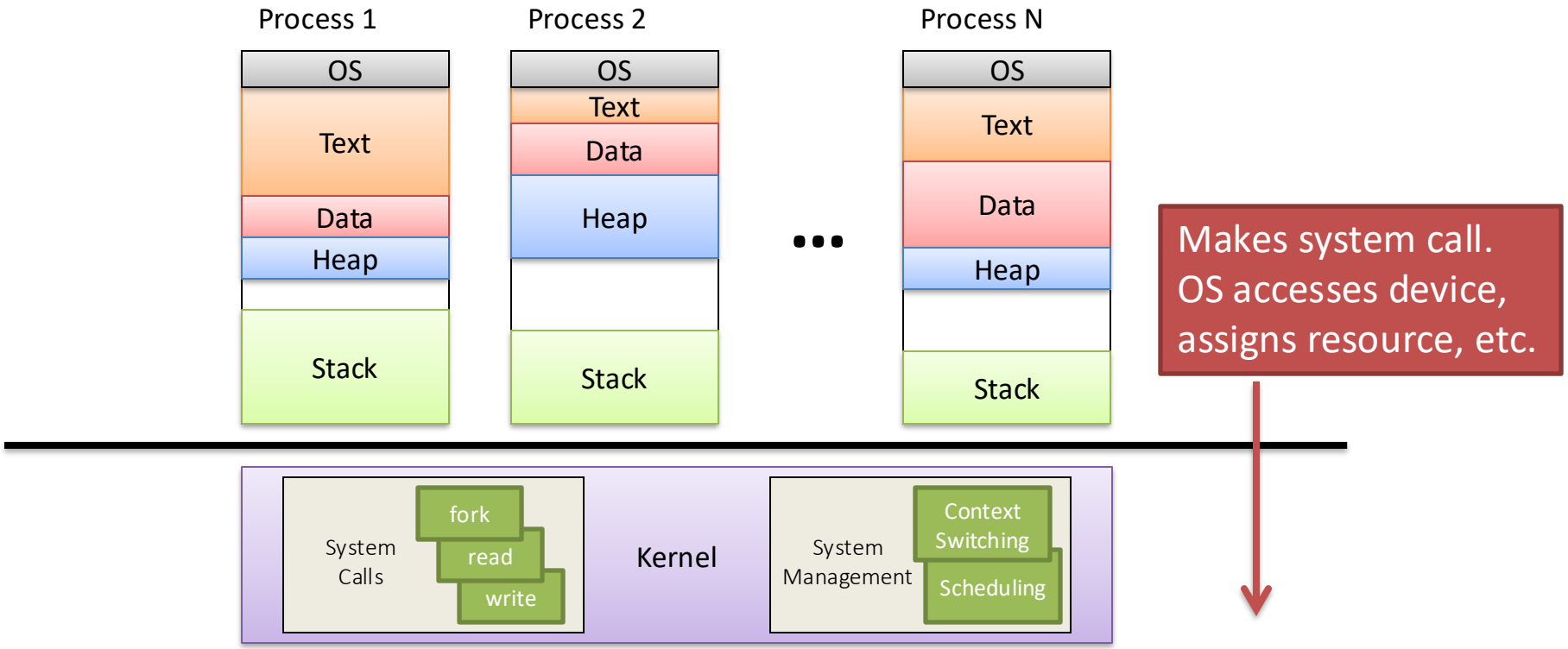
Kernel vs. Userspace: Model



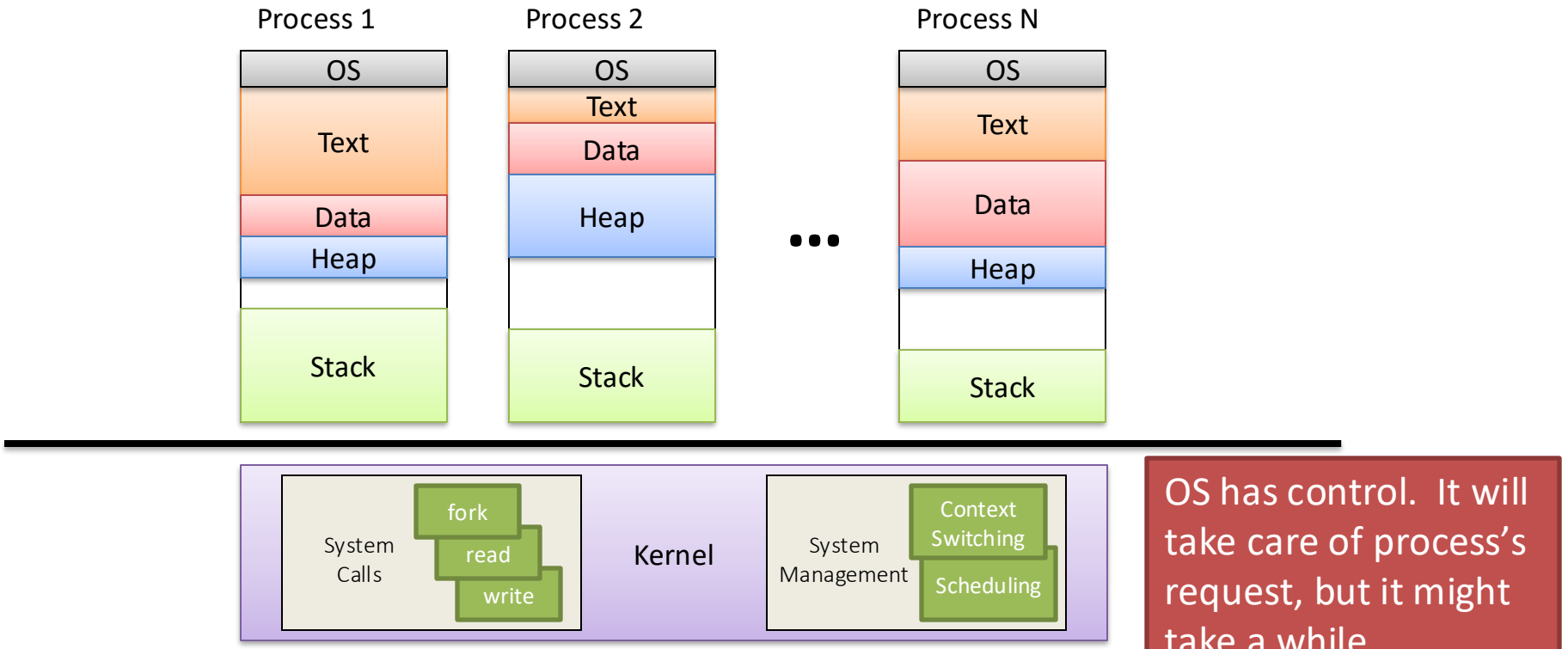
Kernel vs. Userspace: Model



Kernel vs. Userspace: Model



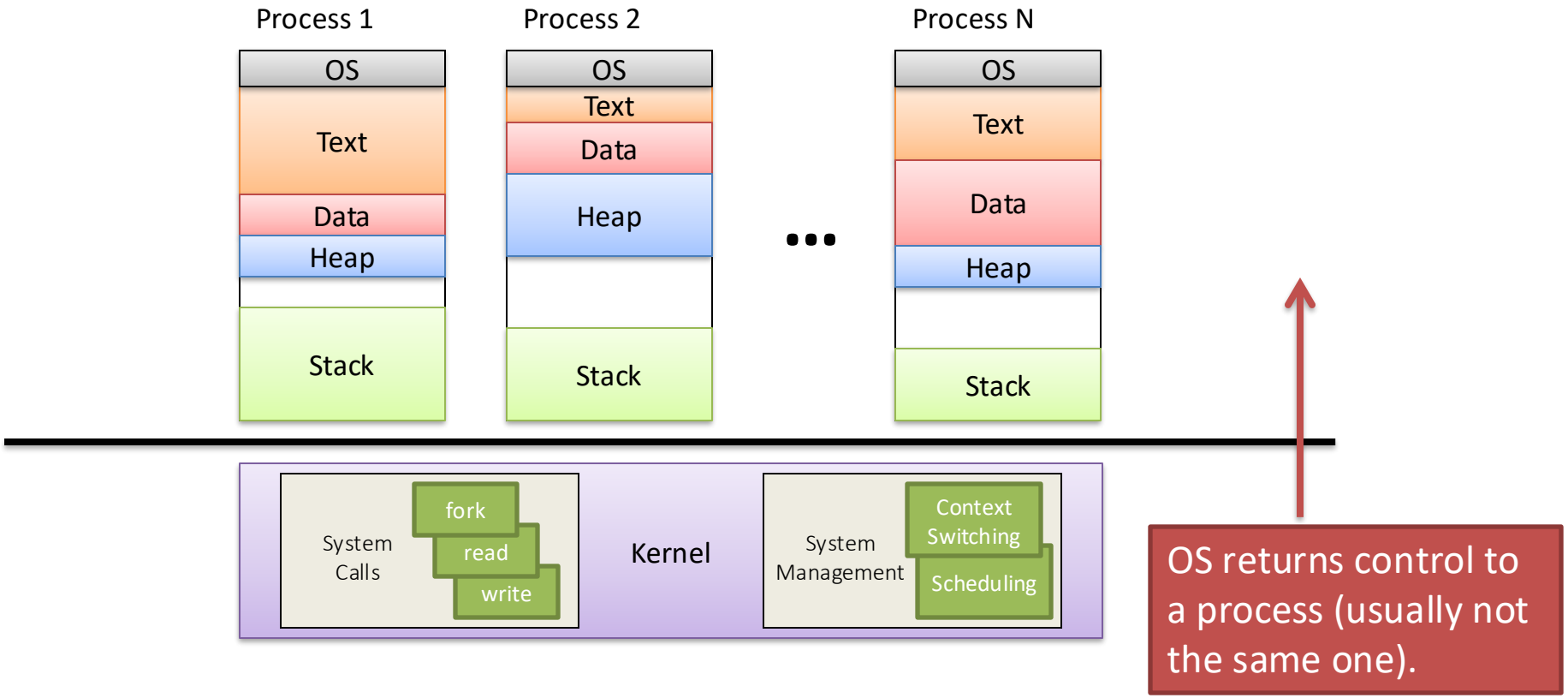
Kernel vs. Userspace: Model



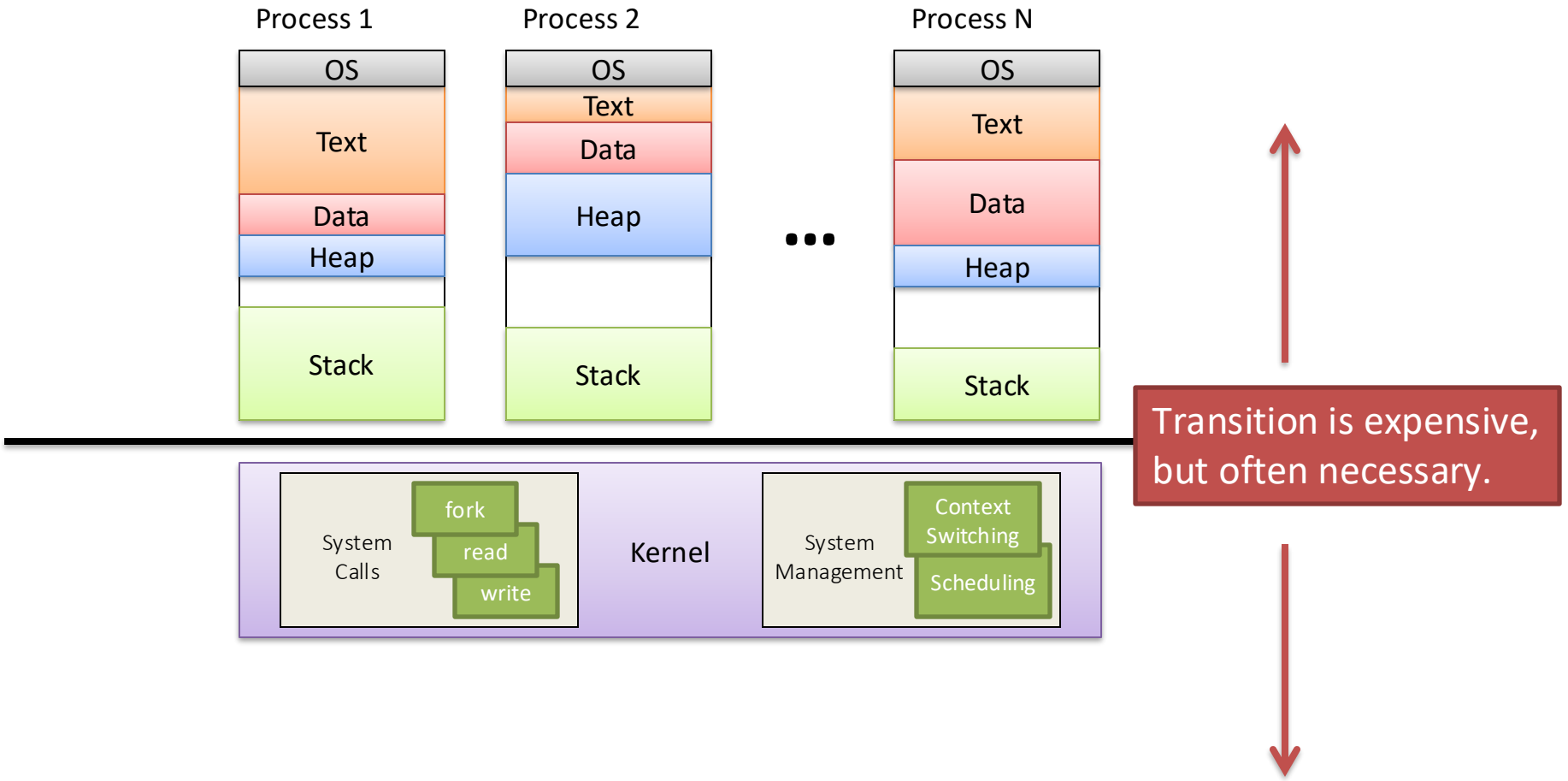
OS has control. It will take care of process's request, but it might take a while.

It can context switch (and usually does at this point).

Kernel vs. Userspace: Model

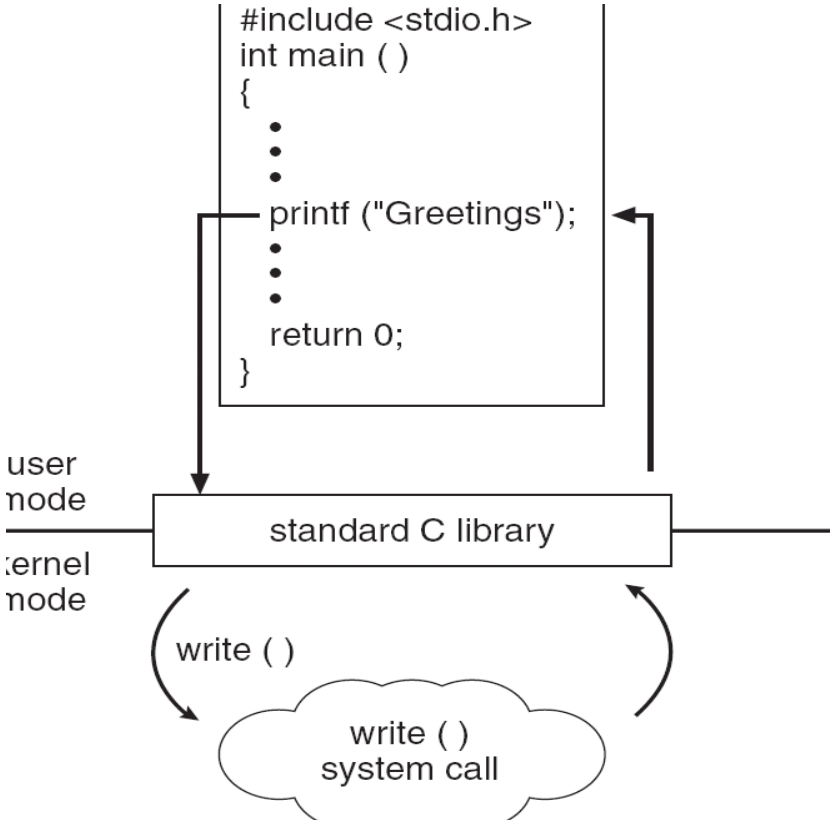


Kernel vs. Userspace: Model



Standard C Library Example

C program invoking printf() library call, which calls write() system call



Control over the CPU

- To context switch processes, kernel must get control:

1. Running process can give up control voluntarily
   - To block, call yield () to give up CPU
   - Process makes a blocking system call, e.g., read ()
   - Control goes to kernel, which dispatches new process

2. CPU is forcibly taken away: preemption

How might the OS forcibly take control of a CPU?

A. Ask the user to give it the CPU.

B. Require a program to make a system call.

C. Enlist the help of a hardware device.

D. Some other means of seizing control (how?).

CPU Preemption

1. While kernel is running, set a hardware timer.

2. When timer expires, a hardware interrupt is generated. (device asking for attention)

3. Interrupt pauses process on CPU, forces control to go to OS kernel.

4. OS is free to perform a context switch.