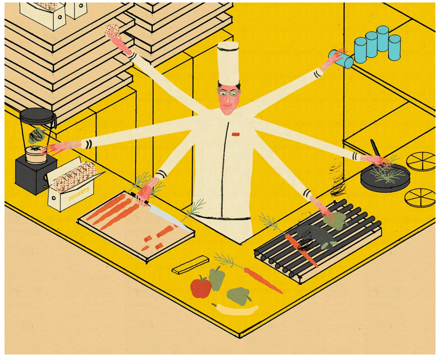
CSE 351 Section 8

Caches & Processes

Autumn 2021



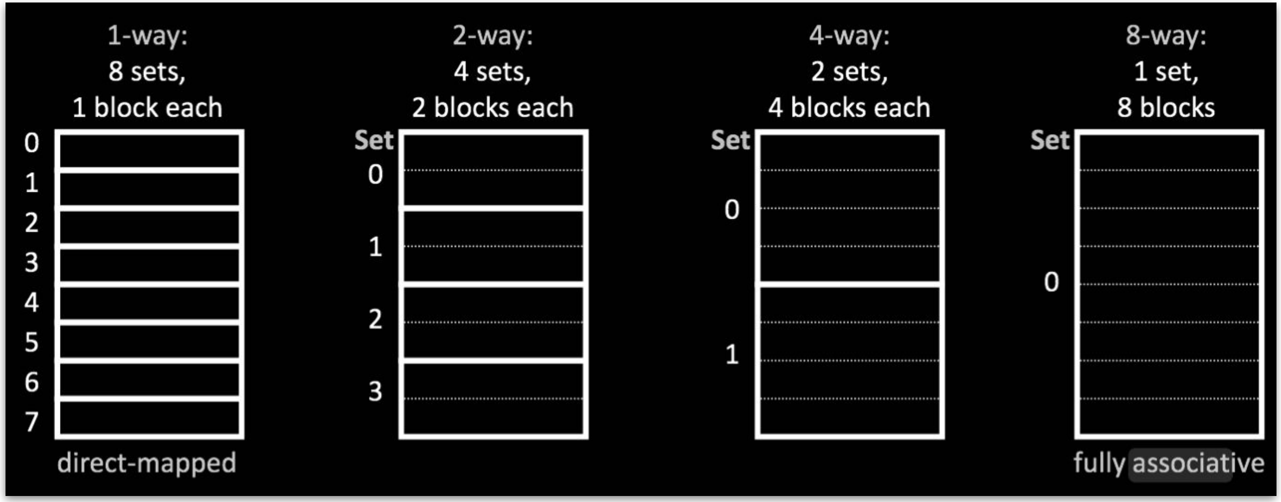
Administrivia

- **Homework 19**
  - Due Friday, Nov 18th
- **Homework 21**
  - Due Monday, Nov 21st
- **Lab 4**
  - Due Monday, Nov 28th (After Thanksgiving break)

Cache Review

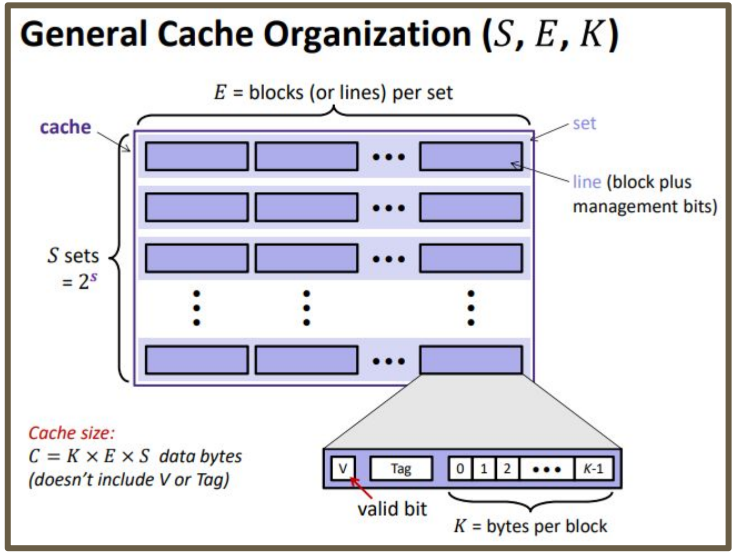
Associativity

- Caches are E-way associative
  - E = associativity = # of blocks per set = # of ways to fit blocks into a single set
- The number of sets = C/K/E
  - C: cache size
  - K: block size
  - E: associativity



Cache Review

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(C / K / E)$
t	# Tag Bits = m - k - s

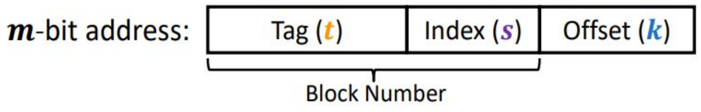

General Cache Organization (S, E, K)
E = blocks (or lines) per set
cache
set
line (block plus management bits)
S sets $= 2^s$
Cache size:
$C = K \times E \times S$ data bytes
(doesn't include V or Tag)
V
Tag
0
1
2
•••
K-1
valid bit
K = bytes per block

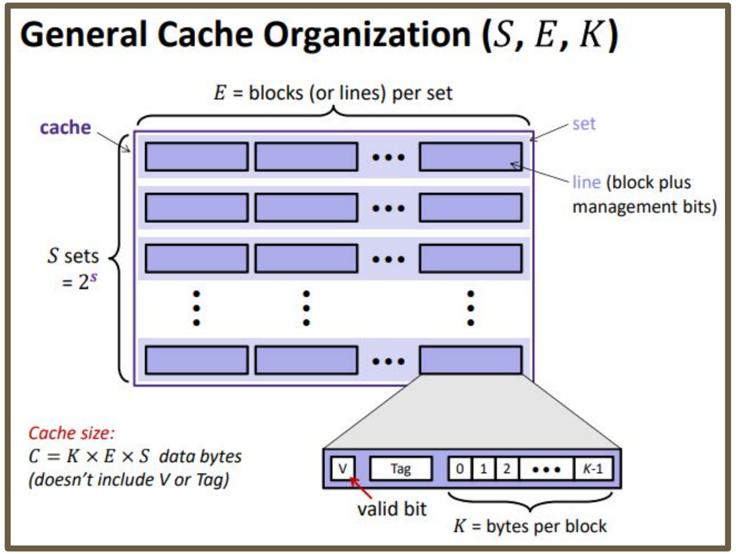
Replacement policy: Generally least recently used (LRU) or "not most recently used"

Cache cheatsheet: https://edstem.org/us/courses/7371/lessons/19530/slides/141929

Cache Organization Review

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
S	# Sets = (C / K) / E
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(S)$
t	# Tag Bits = m - k - s





General Cache Organization (S, E, K)
E = blocks (or lines) per set
cache
set
line (block plus management bits)
S sets = 2^s
Cache size:
$C = K \times E \times S$ data bytes
(doesn't include V or Tag)
V
Tag
0 1 2 ••• K-1
valid bit
K = bytes per block

1) Search the set indicated by the ***index*** field.
2) Check every line in the set for desired block: cache hit if ***valid*** bit = 1 and matching ***tag*** field.
3) If cache miss, place into invalid line or replace ***least recently used (LRU)*** line.
4) Read data starting from the ***offset***.

Dealing with Cache Misses

- ***Compulsory / Cold***
  - Occurs on the first access to a particular cache block.
  - Parameter fix: Increase the block size.
- ***Conflict***
  - Occurs when the cache is large enough but too many blocks map to the same set.
  - Parameter fix: Increase associativity (none in fully-associative caches).
- ***Capacity***
  - Occurs when the set of active cache blocks (the *working set*) is larger than the cache.
  - Parameter fix: Increase cache size.

Write Review

We’ve seen a lot of cache reads, but what about writes?

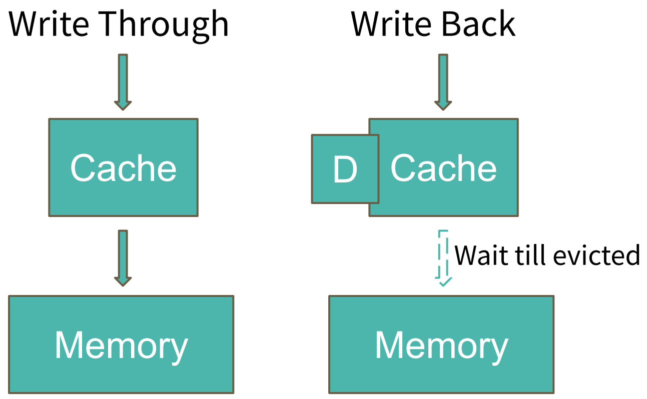
The cache typically stores a copy of the contents of memory (think about the memory hierarchy).

How do we know if and when we copy from the cache back to memory?

Let’s look closer at write policies:

Write Review: Hit!

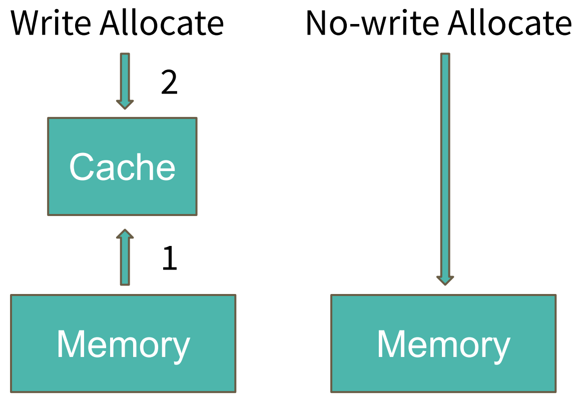
- Write Through
  - Write to "next level" directly
- Write Back
  - Defer writing until cache line we wrote to is evicted
  - We need to keep track of whether line has been modified
    - This requires we store additional information: the *dirty bit* D
    - We only write to memory if our block is replaced *and the dirty bit was set*

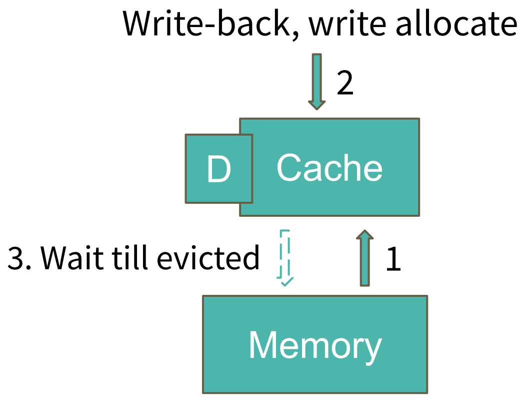


Write Review: Miss!

- Write Allocate (fetch on write)
    - Load data into cache first (akin to a read)
    - Then write to cache
    - Good for locality if adjacent writes or reads follow
- No-write Allocate (write around)
    - Write to "next level" directly

We will usually see ***write-back, write allocate***





Cache Exam Problem

Practice Exam Problem (a)

We have a 64 KiB address space. The cache is a 1 KiB, direct-mapped cache using 256-byte blocks and write-back and write-allocate policies.

What is the TIO address breakdown?

64 KiB = $\mathbf{2^{16}}$ B; 1 KiB = $\mathbf{2^{10}}$ B; 256 B = $\mathbf{2^{8}}$ B

6	2	8
Tag Bits	Index Bits	Offset Bits

Practice Exam Problem (b)

During some part of a running program, the cache’s management bits are in the following state. Four options for the next two memory accesses are given (R = read, W = write). Choose the option ***that results in data from the cache being written to memory***.

- When will data from the cache be written to memory?
  - a cache line is dirty and gets evicted
- You’ll need to do a TIO breakdown for all the addresses

Practice Exam Problem (b)

Will we write to memory?

```
    R 0x4C00, W 0x5C00
```

6	2	8
Tag Bits	Index Bits	Offset Bits

```
0x4C00 → 0100 1100 0000 0000

0x5C00 → 0101 1100 0000 0000
```

Set	Valid	Dirty	Tag
00	0	0	1000 01
01	1	1	0101 01
10	1	0	1110 00
11	0	0	0000 11

READ 0x4C00
Did we hit?
Is set 00 dirty?

Practice Exam Problem (b)

Will we write to memory?

```
R 0x4C00, W 0x5C00
```

6	2	8
Tag Bits	Index Bits	Offset Bits

```
0x4C00 → 0100 1100 0000 0000

0x5C00 → 0101 1100 0000 0000
```

Set	Valid	Dirty	Tag
00	1	0	0100 11
01	1	1	0101 01
10	1	0	1110 00
11	0	0	0000 11

```
WRITE 0x5C00
Did we hit?
Is set 00 dirty?
```

Practice Exam Problem (b)

Will we write to memory?

```
R 0x4C00, W 0x5C00
```

6	2	8
Tag Bits	Index Bits	Offset Bits

```
0x4C00 → 0100 1100 0000 0000

0x5C00 → 0101 1100 0000 0000
```

Set	Valid	Dirty	Tag
00	1	0	0101 11
01	1	1	0101 01
10	1	0	1110 00
11	0	0	0000 11

WRITE 0x5C00

Load 0x5C00 first

Practice Exam Problem (b)

Will we write to memory?

```
    R 0x4C00, W 0x5C00
```

6	2	8
Tag Bits	Index Bits	Offset Bits

```
0x4C00 → 0100 1100 0000 0000

0x5C00 → 0101 1100 0000 0000
```

Set	Valid	Dirty	Tag
00	1	1	0101 11
01	1	1	0101 01
10	1	0	1110 00
11	0	0	0000 11

Dirty bit set, but no memory write has occurred

You try!

Work on the rest of (b).

We will reconvene to discuss the answers!

Practice Exam Problem (b)

Will we write to memory?

```
    W 0x5500, W 0x7A00
```

Line	Valid	Dirty	Tag
00	0	0	1000 01
01	1	1	0101 01
10	1	0	1110 00
11	0	0	0000 11

```
            Tag     Index     Offset

0x5500 → 0101 0101 0000 0000

0x7A00 → 0111 1010 0000 0000
```

- First write is a hit; nothing is evicted.
- Second write evicts old data in set 10, but ***nothing is written*** to memory as the dirty bit was not set.

Practice Exam Problem (b)

Will we write to memory?

```
    W 0x2300, R 0x0F00
```

	Tag	Index	Offset
0x2300 →	0010 00	11	0000 0000
0x0F00 →	0000 11	11	0000 0000

Line	Valid	Dirty	Tag
00	0	0	1000 01
01	1	1	0101 01
10	1	0	1110 00
11	0	0	0000 11

- The write evicts line 3, loads it in, and sets the dirty bit.
- The read evicts line 3, but ***the dirty bit was set***, so we must ***write the changed value back to memory*** before we perform the read!

Practice Exam Problem (b)

Will we write to memory?

```
    R 0x3000, R 0x3000
```

```
                 Tag    Index    Offset
0x3000 → 0011 0000 0000 0000
```

Line	Valid	Dirty	Tag
00	0	0	1000 01
01	1	1	0101 01
10	1	0	1110 00
11	0	0	0000 11

- The first read evicts line 0, but it wasn't dirty so we don't write back to memory.
- The second read is a read hit. No writing occurs.

Practice Exam Problem (c)

Choose LEAP to produce a hit rate of 15/16.

Hint: |= is two accesses

```
#define ARRAY_SIZE 8192
char string[ARRAY_SIZE]; // &string = 0x8000
for (i = 0; i < ARRAY_SIZE; i += LEAP) {
  string[i] |= 0x20; // to lower
}
```

- Block size is 256B, want 16 accesses total with one miss
- |= is two accesses, so we want (256 / 16) / 2 = 8 loop iterations per block (note the access pattern)
- To get 8 iterations per block, LEAP must be 256 / 8 = **32**

Practice Exam Problem (d)

If LEAP is 64, how could we increase the hit rate?

```
#define ARRAY_SIZE 8192
char string[ARRAY_SIZE]; // &string = 0x8000
for (i = 0; i < ARRAY_SIZE; i += LEAP) {
  string[i] |= 0x20; // to lower
}
```

[✓ Bigger Blocks]	Bigger Cache	Add L2 Cache	Increase LEAP

This is the only option which reduces the miss rate, as it causes more to be loaded on each miss.

Practice Exam Problem (e)

What are the three kinds of cache misses, and which one is occurring here?

```
#define ARRAY_SIZE 8192
char string[ARRAY_SIZE]; // &string = 0x8000
for (i = 0; i < ARRAY_SIZE; i += LEAP) {
  string[i] |= 0x20; // to lower
}
```

[✓ Compulsory]	Conflict	Capacity

We miss because we are loading something new, not because of the size of our working set or conflicts.

Benedict Cumbercache

Given the following sequence of access results (addresses are given in decimal) on a cold/empty cache of size 16 bytes, what can we deduce about its properties? Assume an LRU replacement policy.

- `(0, Miss)`
- `(8, Miss)`
- `(0, Hit)`
- `(16, Miss)`
- `(8, Miss)`

Benedict Cumbercache

`(0, M) (8, M) (0, H) (16, M) (8, M)`

What can we say about the block size?

The block size must be no more than 8, because the initial miss at 0 will load in the aligned block from addresses (0) to (size - 1), but we miss when accessing 8 afterwards.

Benedict Cumbercache

`(0, M) (8, M) (0, H) (16, M) (8, M)`

If block size is 8, what about associativity?

DIRECT-MAPPED?

1st access misses (loads in block 0 [0 - 7])
2nd access misses (loads in block 1 [8 - 15])
3rd access hits (0 is already loaded in)
4th access misses (evicts block 0, loads in [16 - 23])
5th access HITS (8 is still loaded in)

So we can't have direct mapped!

Benedict Cumbercache

`(0, M) (8, M) (0, H) (16, M) (8, M)`

If block size is 8, what about associativity?

2-WAY ASSOCIATIVE?

1st access misses (loads in block 0 [0 - 7])
2nd access misses (loads in block 1 [8 - 15])
3rd access hits (0 is already loaded in)
4th access misses (evicts LRU block 1, loads in [16 - 23])
5th access misses (4th access evicted 8)

The cache could be 2-way associative!

Benedict Cumbercache

```
(0, M) (8, M) (0, H) (16, M) (8, M)
```

If block size is 8, what about associativity?

4-WAY ASSOCIATIVE?

The cache size is 16 B and the block size is 8 B, so we can't have a 4-way associative cache as one set would be bigger than the entire capacity!

Processes

What is a Process?

Processes are an abstraction which represent an instance of a running program. They are distinct from a “program” or a “processor.”

Exceptional control flow allows many processes to be run on a single processor at (what appears to be) the same time (concurrently). Exceptions include interrupts, traps, faults, and aborts.

When we switch running processes we perform a *context switch* and must preserve the *execution context* so we can restore the program state later!

It’s Forkin’ Time

We can create a clone of our currently running process with `fork()`. It’s a little special because it has two return values: 0 to the child, and the child’s PID (process ID) to the parent. This allows our code to distinguish the parent from the child.

We’ll focus on fork today, but there are many system calls to manage processes:

- `exec*()` - family of operations to replace current process
- `getpid()`
- `exit()`
- `wait()`, `waitpid()`

Multiple Processes

Can we predict the execution order of processes? Not really!

The OS will switch between running processes. Each process runs its instructions in order, but users won't be able to predict execution order of different processes.

Most machines these days have multiple *processors*... but we'll stick with just one for now!

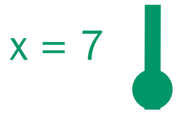
Exercise

What are all four possible outputs for this code?

```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

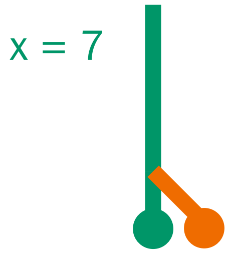
We can trace this program's execution with a diagram:

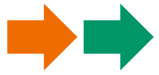


```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

We can trace this program's execution with a diagram:

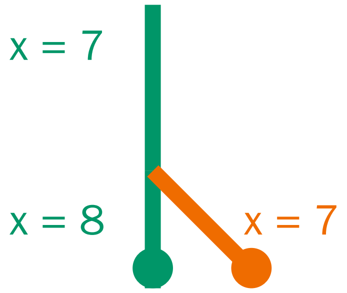




```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

We can trace this program's execution with a diagram:



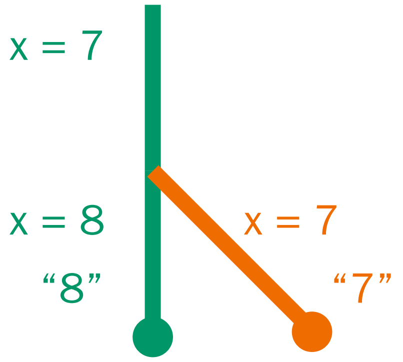
fork returned (PID) to parent

fork returned 0 to child

```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

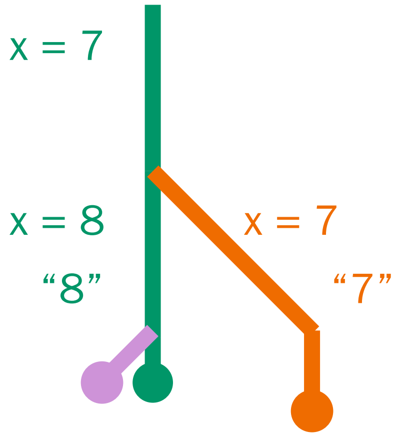
We can trace this program's execution with a diagram:



```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

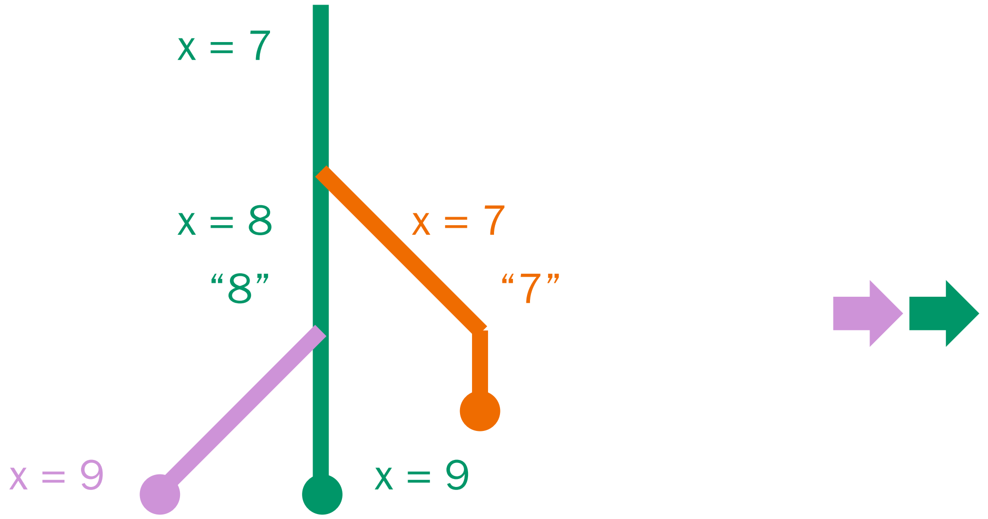
We can trace this program's execution with a diagram:



```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

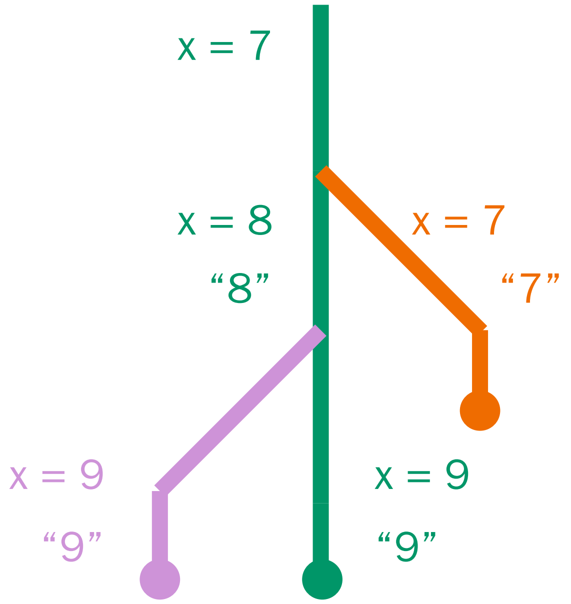
We can trace this program's execution with a diagram:

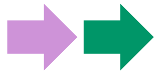


```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

We can trace this program's execution with a diagram:



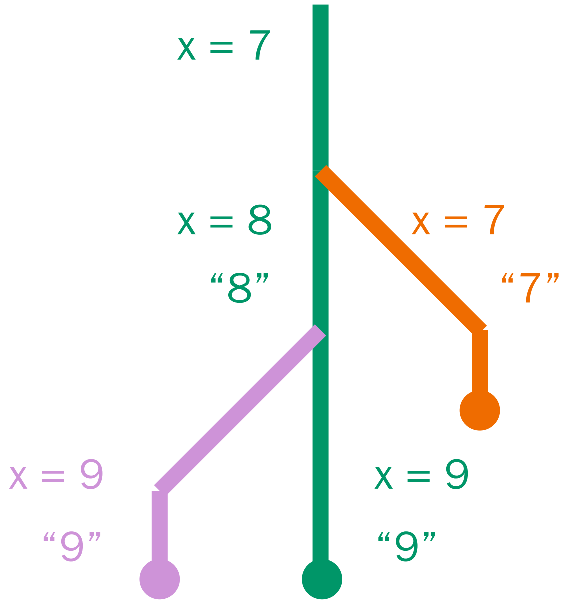


```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

We can trace this program's execution with a diagram:

What are the four possible outputs?



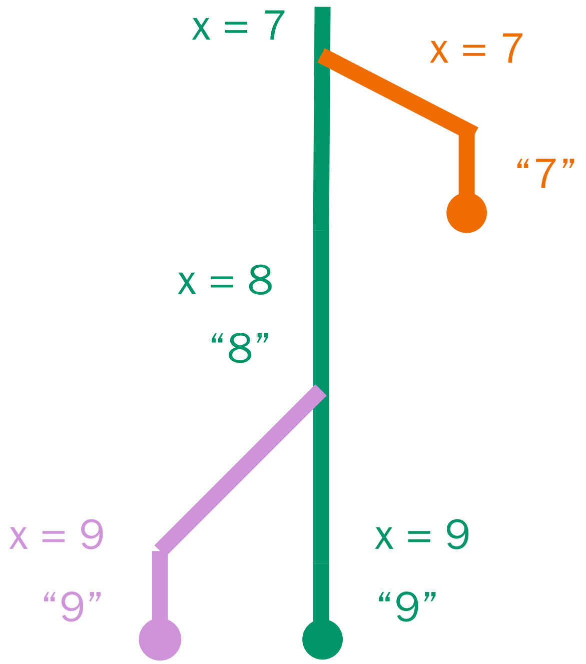
```
7899
8799
8979
8997
```

```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

We can trace this program's execution with a diagram:

What are the four possible outputs?



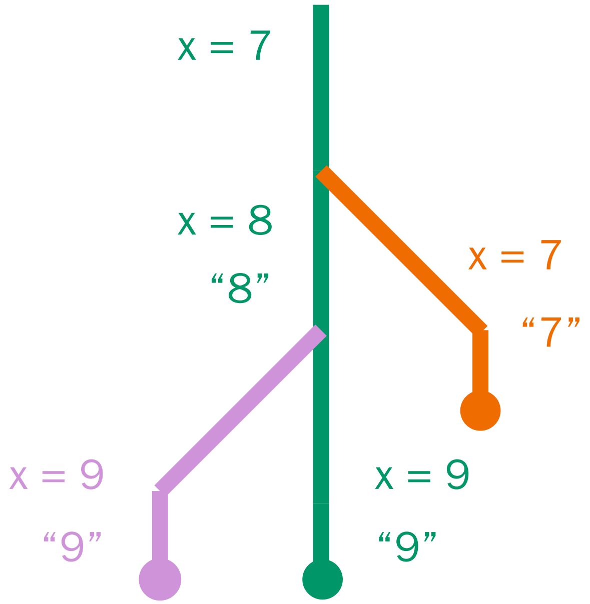
7899
8799
8979
8997

```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

We can trace this program's execution with a diagram:

What are the four possible outputs?



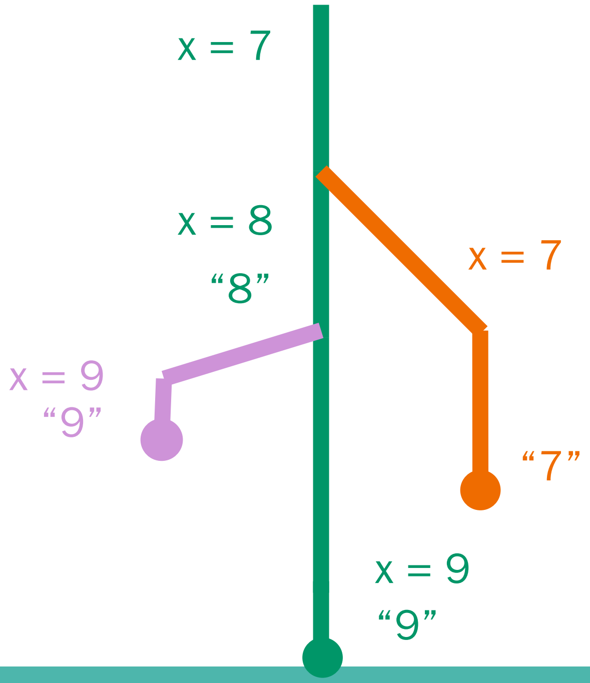
7899
8799
8979
8997

```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

We can trace this program's execution with a diagram:

What are the four possible outputs?



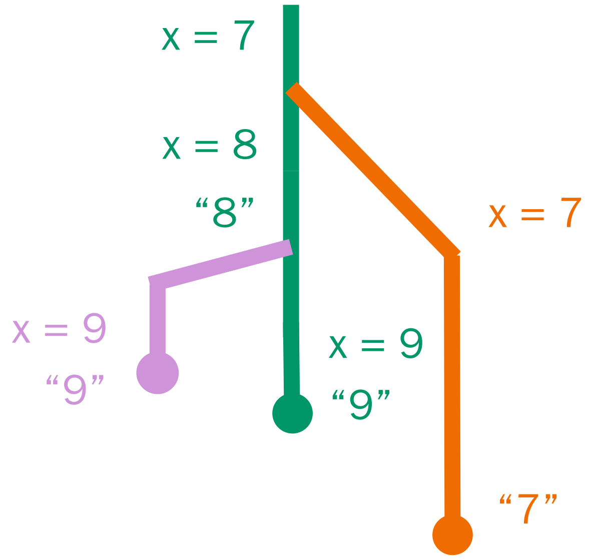
```
7899
8799
8979
8997
```

```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Process Graphs

We can trace this program's execution with a diagram:

What are the four possible outputs?



```
7899
8799
8979
8997
```

```
int x = 7;
if( fork() ) {
  x++;
  printf(" %d ", x);
  fork();
  x++;
  printf(" %d ", x);
} else {
  printf(" %d ", x);
}
```

Cache Simulator

Cache Simulator!

Link:
https://courses.cs.washington.edu/courses/cse351/cachesim/

The cache simulator can be a helpful tool for reasoning through cache problems and mechanisms, particularly on homework and in lab 4.

That’s All, Folks!

Thanks for attending section! Feel free to stick around for a bit if you have quick questions (otherwise post on Ed or go to OH).

See you all next week and good luck on lab 4.