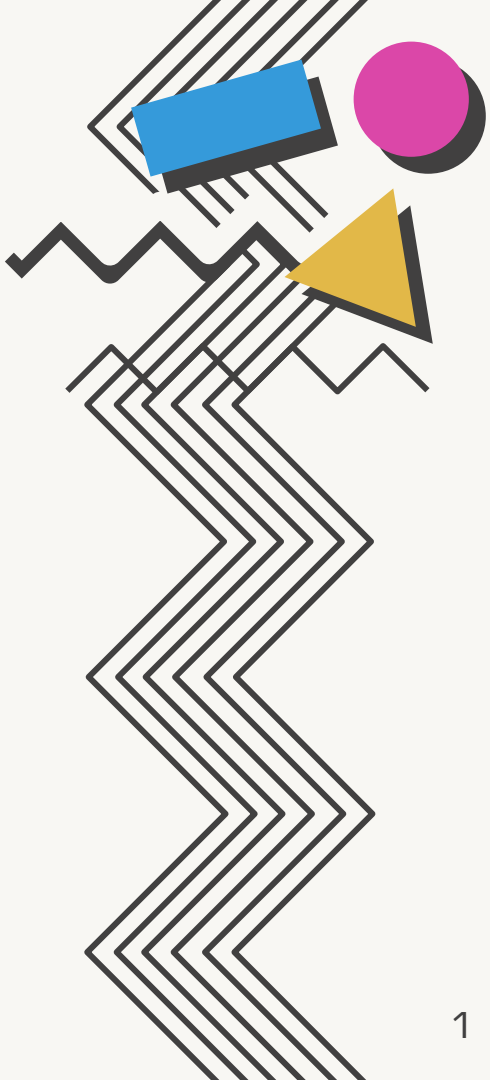
CSE 351
Section 8

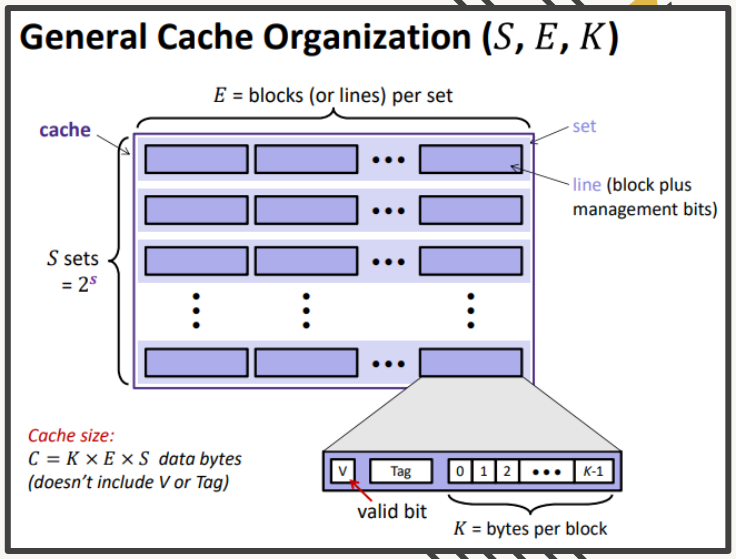
Caches and Processes



Cache Review

Cache Review

- Capacity (C) = total size of the cache in byte
- Block Size (K) = # of bytes in a cache line
- Associativity (E) = # of lines in a set
- # sets = C/K/E
- Replacement policy:
  - Generally least recently used (LRU) or “not most recently used”



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 policy:

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***
    - 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

Three Types of Cache Misses

- ***Compulsory***
  - aka cold miss, occurs on first access to a block
- ***Conflict***
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot
  - Decrease when associativity is increased (fully-associative caches have no conflict misses at all)
- ***Capacity***
  - Occurs when the set of active cache blocks (the ***working set***) is larger than the cache

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 = 2^{16} B; 1 KiB = 2^{10} B; 256 B = 2^{8} B

Tag	Index	Offset
6	2	8

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***.

Practice Exam Problem (b)

Will we write to memory?

R 0x4C00, W 0x5C00

0x4C00 →	0100 11	00	0000 0000
0x5C00 →	0101 11	00	0000 0000
	Tag	Ix	Offset

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

Tag	Index	Offset
6	2	8

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

Practice Exam Problem (b)

Will we write to memory?

R 0x4C00, W 0x5C00

	Tag	Ix	Offset
0x4C00 →	0100 11	00	0000 0000
0x5C00 →	0101 11	00	0000 0000

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

Tag	Index	Offset
6	2	8

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

Practice Exam Problem (b)

Will we write to memory?

R 0x4C00, W 0x5C00

0x4C00 →	0100 11	00	0000 0000
0x5C00 →	0101 11	00	0000 0000
	Tag	Ix	Offset

WRITE 0x5C00
Load 0x5C00 first

Tag	Index	Offset
6	2	8

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

Practice Exam Problem (b)

Will we write to memory?

R 0x4C00, W 0x5C00

0x4C00 →	0100	11	00 0000 0000
0x5C00 →	0101	11	00 0000 0000
	Tag	Ix	Offset

Dirty bit set, but no memory write has occurred

Tag	Index	Offset
6	2	8

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

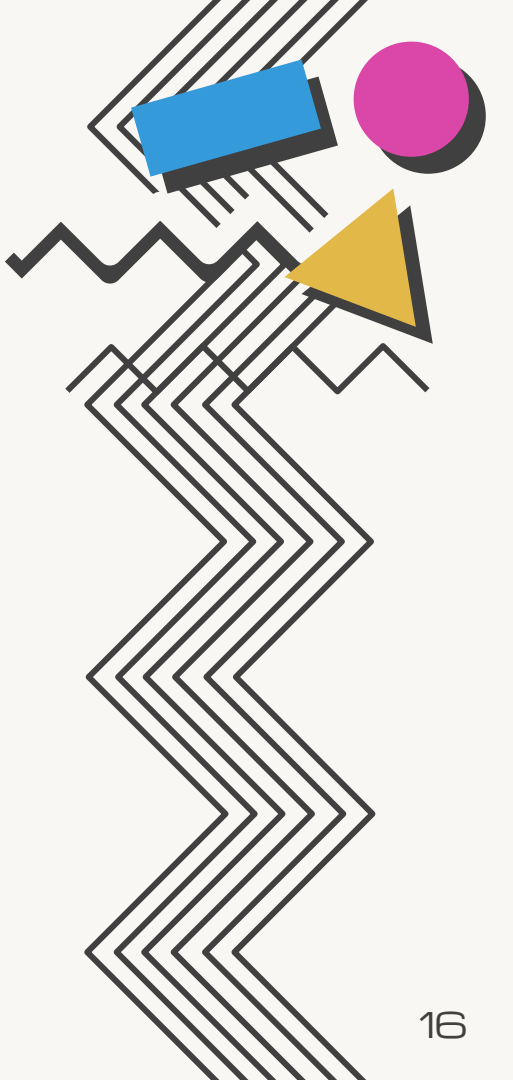
Processes

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 (perceptibly) 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!

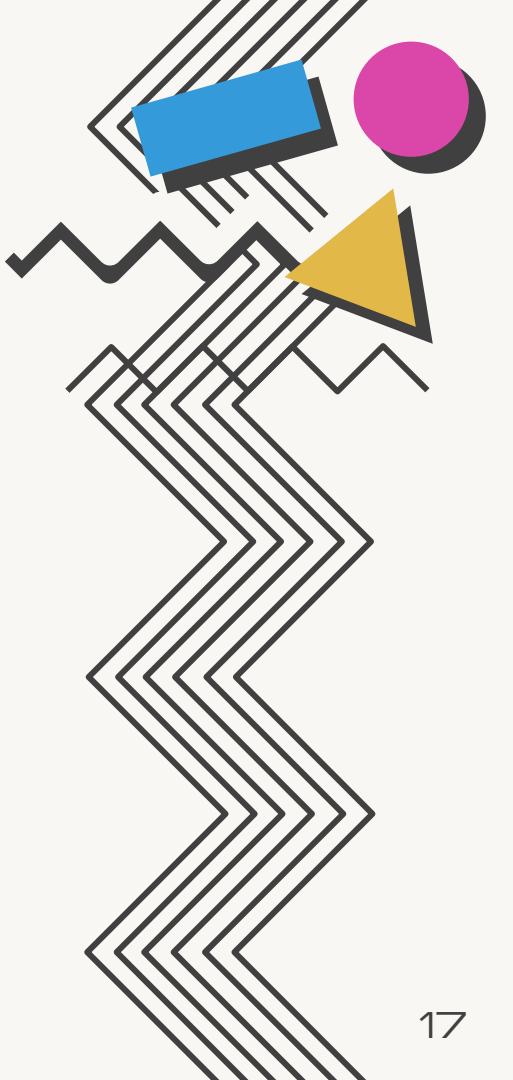


It’s Forkin’ Time

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 proc.
- `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!