Administrivia

pset2

  io61_eof() – **DON'T CHANGE IT**

No official class Tuesday

  TFs will be here to run impromptu office hours

Today's Agenda

  Processor Caches
  Eviction Policies
  Memory Mapped I/O (Feels like magic)

Storage Hierarchy (Fastest to slowest)

Eddie draws this on the board as a pyramid

- **Registers** (each 64-bits (8B) on 64-bit CPU)
  - On a Nehalem* CPU, you get 16 registers, 8B * 16 = 128B
  - *Nehalem is an Intel Processor, you don't need to know that.
- **CPU Cache**
  - 3 levels, each stores 64 blocks for the previous level's total size
  - L1 stores 128B*64 = 32KB
  - L2 stores 32KB*64 = 256 KB
  - L3 stores 256KB*64 = 8MB
- **Primary Memory**
  - RAM
- **Hard Drive Cache**
  - Cache on the physical drive
- **Hard Drive**
- **Backup Storage**
  - eg. Tape

Why have Processor caches?

Memory is slow, the bigger memory gets and the relatively slower it becomes

Let's practice some **cache lines** (boudaries between cache blocks)

```
void* a = 0x1234FABC;
void* b = 0x1234FAB0;
```

Is a and b the in the same L1 (64B) cache bucket? Why?

The cache bin can be found if the address return the same number from this math:

`a – a & 64`

is equivalent to

`a – (a & 63)` (Note: & means bitwise and)

Plain English: look at the last 6 binary places. Why 6 places? 64 = 2^**6**.

What math do coders really write?

```
    a & ~63
```

~ flips each bit

&, bitwise AND

```
& means bitwise and (0&0 = 0, 0&1 = 0, 1&0 = 0, 1&1 = 1)
```

Example:

```
0b 0101 1100   92
 & &&&& &&&&   bitwise AND
0b 1100 0000   ~63
 = ==== ====
0b 0100 0000   64
```

Practice matrix multiplication

Compare `ijk` order and `kji` order

Which method will cross more cache lines?

Matrix Multiplication

```
    for loop on i
        for loop on j
            for loop on k
                c[i*N+j] += a[i*N+k]*b[k*N+j];
```

```
/*    Why do we care about matrix multiplication?
BECAUSE BIG DATA AND AI IS MATRIX MULTIPLICATION, MWAHAHA.
*/
```

Types of Cache Misses

Cold Miss: when you have to fetch data and the cache is empty

Conflict Miss: when the cache is not smart enough

Capacity Miss: cache is out of space

Eviction Policies

FIFO - first in, first out

suffers from **belady's anomaly -** more slots, more misses

access pattern exists to make this slow

LRU - least recently used

throw away the block that was used the least recently

access pattern exists to make this slow

Magic Future - you magically know the future

you evict the block used least soon

immune to Gabriel crafting a slow access pattern

May actually be implemented!! Processor can receive 'hints' from the code to prioritize eviction.

Memory Mapped I/O

stdio cache ($) - standard block size 4KB. Single block cache.
buffer cache - cache the OS users to store file data. Standard block size 4KB. Total size is essentially all of memory

Why is stdio slow on strided access?

It only has one block. Strided access jumps to a new block on each call. This requires a system call for every request. (Note that the buffer cache may contain all data we need. We can only access it via a system call unfortunately)

How can we counter this?

Have our cache be the same size of the file we are working with?

No! Uses 2 bytes per byte in the file. Very inefficient.

Use the kernel's memory?

Wishful thinking??? Magic??? Tell OS

that's what happens. Your program tells the kernel to load your file in the kernel's memory using `mmap`
See `man 2 mmap` for how to use mmap

Let's run some experiments

Reading 1 byte at a time: 2.4 MB/s
Reading directly : ?? MB/s
Reading from mmap: 40 MB/s
Striding through the file 1.3 MB/s
stdio striding through the file 0.92 MB/s

stdio slows striding down because stdio is buffering the file

Striding with mmap: 40 MB/s

Why isn't this the same as doubling memory used?

Physically, there's only the buffer cache occupying space. mmap simply allows us to access it as if it weren't behind the system call wall.