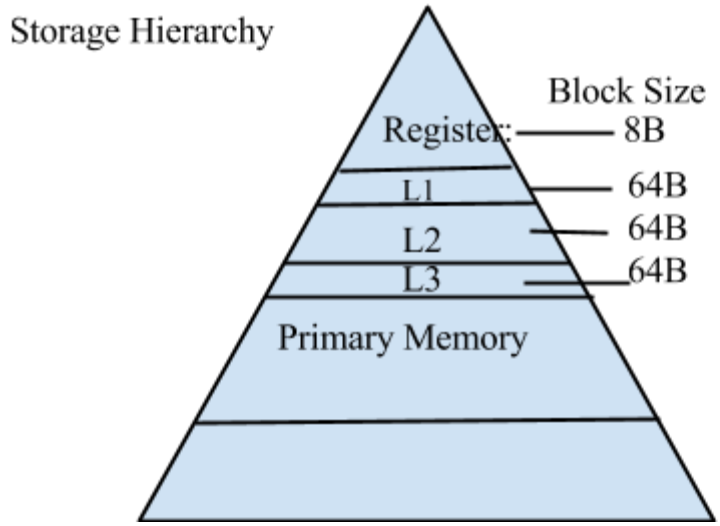
CS61 Processor cache, Eviction policy, Memory Mapped IO

Scribes: Jessica Xu, Eela Nagaraj, Shai Szulanski, Brian Wong



caches: organized in aligned units of 64 bytes

Registers are kind of cache → 64 bit machine has 64 bit register.

L1 - L3: caches in between registers and memory which get bigger and slower as we move from L1 → L3 (happiest in L1 cache)

Looking at Matrix Example

```
for (i = 0 to n)
        for (j = 0 to N)
                for (k = 0 to N)
                        c[i*N +  j] + = a[i*N + K] * b [k*N + j]
```

- exact same number of memory accesses.
- switching j and k loops would see no difference if they actually took the same amount of time, but there's actually **20x** difference.
- spherical postman analogy: gets more than just one piece of memory, so the more memory there is, the physically farther away it becomes

a block is called a cache line → 64 byte unit of memory

Bitwise arithmetic

In Decimal → 1947 - (1947%1000) = 1000
1947 - (1947%100) = 1900

Strategy for binary is the same:
a = 0x123FABC
0x123FAB0 // these hex values match up, so they fit the same 64 byte unit

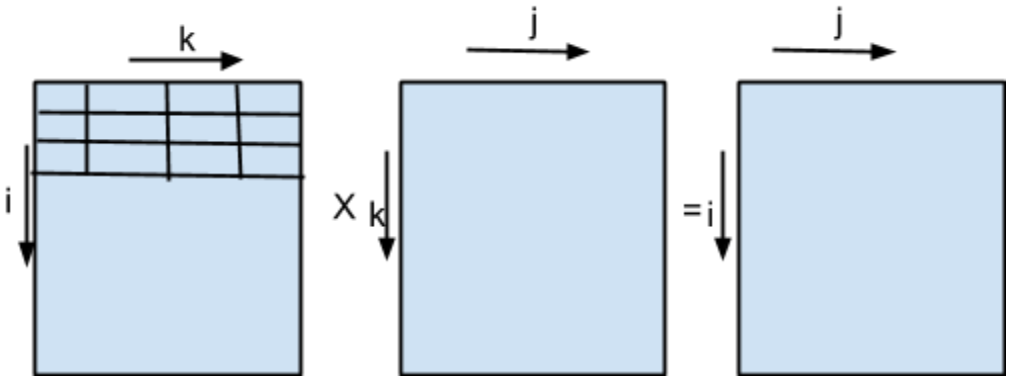
<u>all of the following are the same:</u>
a - a%64
a - (a&63) // & is a bitwise operator version of &&
a & ~63 // ~ *twiddle, flips bits, 1s←→ 0s; -1 = ~0 → -x = ~(x-1)

**Remember that 63 = 0111111, only one bit difference between 63 and 64

<u>How processor caches work by aligning units of 64-bytes:</u>
1st address in $ line containing a 0x123FA80
→ Threw away its lower 2 bits.
→ Units of 4 bits. Throw away lower ones, then 2 lowest of second most significant

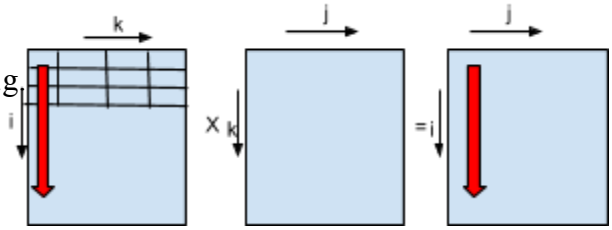
Matrix Fun:

- We have 3 Matrices filled with doubles
  - 1000 doubles on a side, 100 cache lines per side
- best (speediest) way to access cache lines → sequentially



<u>Evaluating speed of access orders</u>

- outermost two loops accessed in the same way, innermost loop determines whether we are attempting to access horizontally (sequentially, e.g. fast) or vertically (SLOW)
- best/good: i k j ~1 sec
- meh/bad: i j k ~10 sec
- worst: j k i OR k j i ~35 sec
  - i as iterator results in vertical memory access
  - one full round of inner loop = 1000 + 1000 + 1 = 2001 cache lines



Eviction policy:

Cold misses: misses resulting from an empty cache→ inevitable because we need to fill cache ("warm up the cache")

Conflict misses: misses resulting from the eviction policy selected

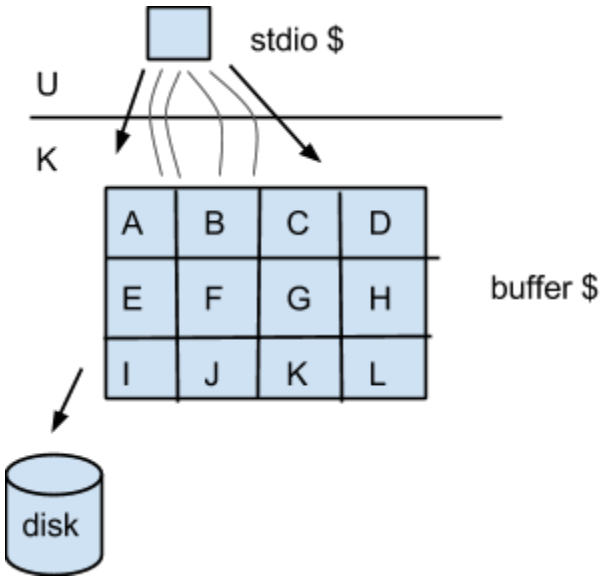
Capacity misses: cache is not big enough, resulting in misses

Belady's anomaly – adding slots to a cache *reduces* the hit ratio

- FIFO suffers from Belady's, but Least Recently Used does not
    - Best real policy: least recently used (is the one that gets evicted)
- Omniscient policy: evict the block that will be used the furthest in the future
    - The omniscient policy can sometimes be approximated using software hints
    - application gives hints; OS informs program's data's use → e.g. "will read file sequentially", allowing policy modification to better suit future data use
        - other example of hints: madvise: don't need

Multiple Processes running → all trying to use same cache, therefore we lose some benefits of the cache.

Memory Mapped IO



stdio cache: slot size = 4KB; single slot cache
→ so total size = 4096 bytes

- testing to find the total size of cache:
    - run experiments: ex)
        - write 4KB block
        - write another 4KB block
        - read the block
    - use strace

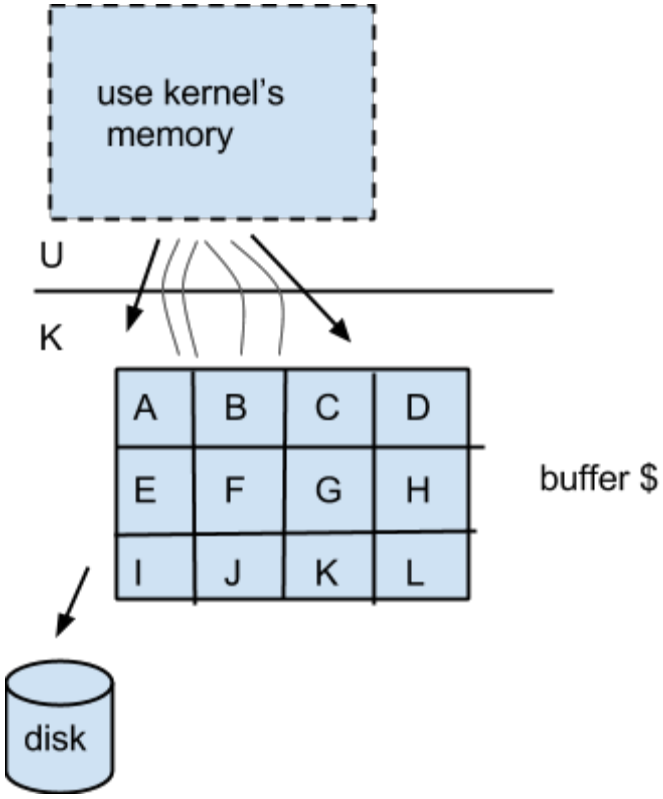
buffer $: slot size also = 4KB
→ total size = about 4GB;
(almost size of primary memory)

Stdio is slow for strided access patterns → relies upon system calls, which are slow

How can we speed up strided access calls?

- add slots to stdio cache
    - **how many can we add before it is too expensive to afford?**
        - if we have enough slots to contain entire buffer, then we end up using 2 bytes of memory per byte of file data → inefficient

- use kernel's memory via mmap
    - kernel puts portion of the buffer cache for the file and puts at the address of file
        - "creates space in memory that *is* the file"
    - has the same effect as reading a byte at a time, but with a single system call makes all bytes of a file available in memory to the application
        - unlike reading a byte at a time, functions well with strided access calls
    - Note: first call to load data into memory is expensive
    - OS manages buffer



Modified diagram with mmap