

# CS61 Section 7 Notes

Week 11/5-11/9

## [0. I/O and File Descriptors](#)

[I/O System calls, File Descriptors](#)

[Standard I/O \(stdio\)](#)

### [1. Locality](#)

### [2. Caches](#)

### [3. Request Costs](#)

### [4. Caching Strategies](#)

[Increasing Block Size When Writing](#)

[Batching Requests](#)

[Using stdio](#)

[Speculation and Read-Ahead \(Prefetching\)](#)

[Stride Access Patterns](#)

## 0. I/O and File Descriptors

### I/O System Calls, File Descriptors

For more information: `man 2 <system-call>`

```
int open(const char *path, int oflag, ...)
```

- system call; open a file with the given path and flags
- returns a ***file descriptor*** for the file (non-negative), or -1 on failure

oflag is obtained by or-ing these values together (see more flags in the man page)

O\_RDONLY open for reading only

O\_WRONLY open for writing only

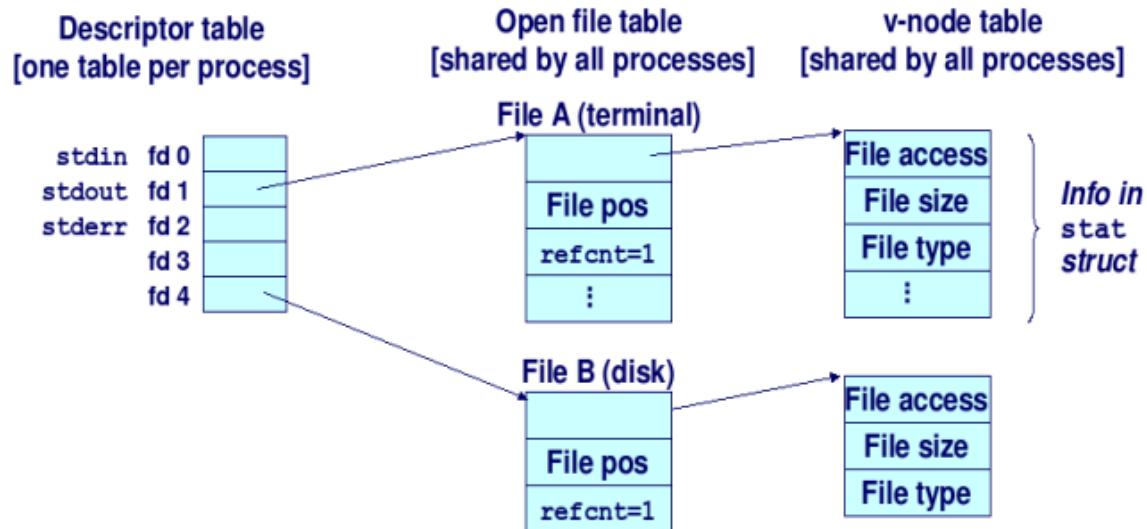
O\_RDWR open for reading and writing

O\_CREAT create file if it does not exist

O\_TRUNC truncate size to 0

O\_SYNC synchronous call--write directly to disk

The kernel maintains a ***descriptor table*** for each process to maintain a list of all opened resources for I/O, such as files. A ***file descriptor*** is an index into this table.



When the process starts, file descriptor 0 refers to *stdin* (the input stream, i.e. when the console prompts the user for input), file descriptor 1 refers to *stdout* (the output stream, i.e. where your `printf` statements print to), and file descriptor 2 refers to *stderr* (the error stream, used to report errors in code, and is usually printed out to console).

Whenever we call `open`, the kernel assigns a new file descriptor for the opened file. ***It is important to note that a file descriptor is NOT the file itself--it is an index into the kernel's descriptor table for the process.*** Using file descriptors, the kernel keeps track of data associated with the file, such as the size of the file, permissions of the file, where the current cursor is in the file, and so forth.

```
ssize_t read(int fildes, void *buf, size_t nbyte)
```

- system call
- attempts to read `nbyte` worth of bytes into `buf` from the given file descriptor `fildes`
- may actually return a value less than `nbyte` (if the call read fewer than `nbytes`), or return `-1` on error

```
ssize_t write(int fildes, const void *buf, size_t nbyte)
```

- system call
- attempts to write `nbyte` worth of bytes from `buf` to the given file descriptor `fildes`
- may actually return a value less than `nbyte` (if the write command wrote fewer than `nbytes`), or return `-1` on error

```
int close(int fildes)
```

- system call
- closes the file descriptor and returns `0` on success, `-1` on error
- it is important that we close all files that we do not need to allow the OS to reclaim resources

## Standard I/O (stdio)

The stdio library is a wrapper for I/O system calls that performs buffering. The purpose of the stdio library is to speed up I/O calls by (1) reducing the amount of system calls, and (2) caching data being read/written. See the man page for more information: `man 3 <library-call>`

```
FILE *fopen(const char *restrict filename, const char *restrict mode);
```

- Instead of returning a file descriptor, this returns a pointer to a FILE struct

```
size_t fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);
```

- reads `size * nitems` bytes into buffer `ptr` from the FILE pointer `stream`
- performs buffered I/O--the stdio library maintains a cache of data so that subsequent reads are faster
- returns the actual number of *items* read

```
size_t fwrite(const void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);
```

- writes `nitems * size` bytes from `ptr` into the FILE pointer `stream`
- performs buffered I/O--the stdio library batches write requests to minimize the number of system calls

```
int fclose(FILE *stream);
```

- stdio library version of close

## 1. Locality

**Locality of Reference** - property of code such that data access at time  $t$  is close to data accessed at  $t + 1$

**QUESTION. Does sequential access have good locality or bad locality?**

**QUESTION. Does random accessing an array have good locality or bad locality?**

**QUESTION. Throwback to the first lecture. Consider a list implemented as a resizable array (using the vector pattern), and a list implemented as a linked list. For iterating over the collection, which implementation provides better locality?**

## 2. Caches

**Cache** - Fast storage that stores temporary copies of data from slower storage

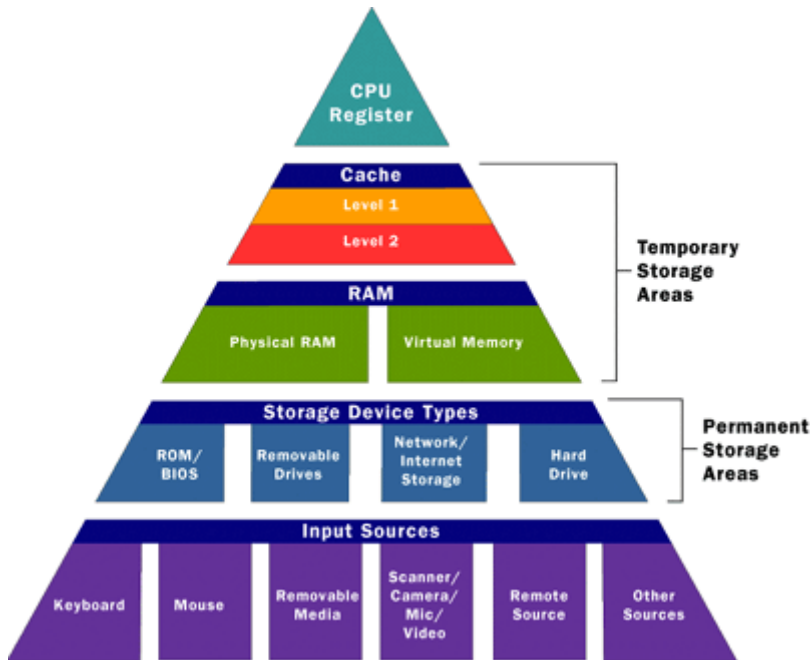
If we are looking for an item and it is in our cache, we say that we have a **cache hit**. If the item we are looking for is not in our cache, we say that we have a **cache miss**.

Caches allow us to store data closer to where it is actively being used, and therefore allow retrieval of the data to be faster. An analogy of a cache would be your backpack. You keep some books and other items in your bag so that you can quickly retrieve the items when you are at classes. It would be extremely inconvenient and slow to go back to your room every time just to pick up an item. **Thus, your backpack is a cache for your room.**

Taking this to another level, you can think of **your school dorm room as a cache for your room back in your hometown**. If you didn't live in Boston, it would be even more slow and inconvenient to have to go back to your hometown every time you needed an item--instead, you **speculate** about what you would need for the semester and take what you need with you when you move into your dorm room. Note that for each level of our "cache"--your backpack for your dorm room, and your dorm room for your hometown--the cache is smaller, but a lot faster to access.

The computer memory hierarchy acts in a very similar way. Below is a diagram of the **memory hierarchy** of a computer. The higher you go up, the faster it is to access items. The lower you go up, the cheaper it is to add more storage in that layer.

**KEY INSIGHT: Each level is a cache for the levels below!** Thus, registers can be thought of as a cache for primary storage (RAM). Primary storage can be thought of as a cache for disk.



### 3. Request Costs

Define:

$R$  = per request overhead [seconds / requests]

$U$  = per unit overhead, for some definition of "unit" [seconds / byte]

$C$  = total cost [seconds]

Then:  $C = \# \text{ requests} * R + \# \text{ bytes} * U$

Suppose we are writing a file of size  $N = 10^6$  bytes to disk. Similar to `w01-syncbyte.c`, we write each byte individually and synchronously to disk, and we observed the total time to write all  $N$  bytes to be **1.01s**. Similar to `w02-syncblock.c`, we write blocks of size 1000 to disk. We observed that the total time to write all  $N$  blocks to disk takes **0.011s**.

**QUESTION:** Write down an equation in terms of  $N$ ,  $R$ , and  $U$  that describes how long it took to write each byte individually to disk.

**QUESTION:** Write down an equation in terms of  $N$ ,  $R$ , and  $U$  that describes how long it took to write blocks of bytes of size 1000 to disk.

**QUESTION: What are the per request overhead (R) and per unit overhead (U)?**

**QUESTION: How long would it take if we instead wrote blocks of bytes of size 4000 to disk?**

## 4. Caching Strategies

### Increasing Block Size When Writing

In `w01-syncbyte.c` -> `w02-syncblock.c`

We go from: `write(fd, buf, 1)`  
to: `write(fd, buf, block_size)`

where we changed `block_size`, varying from 512 to  $2^{16}$  and  $10 * 2^{20}$  (10 megabytes). Also, note that both of these calls were ***synchronous*** (synchronous here means that each write call causes the operating system to write directly to disk for each write call). We noticed in lecture that `w01-syncbyte` was EXTREMELY slow in comparison to `w02-syncblock`.

**QUESTION. Why did changing the write call from 1 byte to a larger block size improve the rate at which we wrote to disk?**

**QUESTION. What are the disadvantages still about `w02-syncblock.c`?**

### Batching Requests

***Batching*** - group requests to reduce per request overhead

In our backpack cache example, batching would be like waiting until you go back to your room to get all the things that you need at once, rather than going back to your room every time you needed something back in your room.

**QUESTION: In terms of per unit overhead and per request overhead, when would batching be useful?**

Consider the changes in these file pairs:

w01-syncbyte.c -> w03-byte.c

w02-syncblock.c -> w04-block.c

In both of these file pairs, our only changes were removing the O\_SYNC flag in our call to open, thus making our system calls for write no longer synchronous.

We noticed that in each pair, removing the O\_SYNC flag dramatically increased the rate at which we wrote to disk.

**QUESTION: What is causing this speedup? Are we using a cache? If so, where is the cache located in the memory hierarchy, and who controls access to the cache?**

**QUESTION: Why is w04-block.c still faster than w03-byte.c?**

**QUESTION: What are still some disadvantages with w03-byte.c and w04-block.c?**

## Using stdio

Consider the changes in these file pairs:

w03-byte.c -> w05-stdiobyte.c

w04-block.c -> w06-stdioblock.c

In these file pairs, we replaced the system calls `open` and `write` with library calls to the `stdio` (Standard I/O) library functions `fopen` and `fwrite`. We noticed in lecture that there was an even greater rate at which we wrote to disk.

**QUESTION: What is causing this speedup? Are we using a cache? If so, where is the cache located in the memory hierarchy, and who controls access to the cache?**

**QUESTION: Why is `w06-stdioblock.c` still faster than `w05-stdiobyte.c`?**

**QUESTION: What are still some disadvantages with `w05-stdiobyte.c` and `w06-stdioblock.c`?**

## **Speculation and Read-Ahead (Prefetching)**

**Speculation** - performing work BEFORE an explicit request for the work; purpose is to speed up future requests

**Read-Ahead (Prefetching)** - read data in advance of explicit read requests for the data. Read-Ahead is an example of speculation.

Back in our backpack example--every day, you leave your room with a few select items in your bag. You **speculated** that you would need item X and item Y during your day BEFORE you made an explicit request for item X and item Y, hence you placed X and Y into your bag.

In `r01-byte.c` and `r02-block.c`, we make direct read system calls.

**QUESTION: What is the kernel doing to speed up read requests?**



**QUESTION: What are disadvantages with r01-byte.c and r02-block.c?**

In r04-stdiobyte.c and r05-stdioblock.c, we replaced the read system calls with stdio library calls to fread, and we noticed a higher rate of reading from disk.

**QUESTION: What is causing this speedup? Are we using a cache? If so, where is the cache located in the memory hierarchy, and who controls access to the cache?**

## **Stride Access Patterns**

**Sequential Access** - accessing memory in sequential order (contiguous addresses)

**QUESTION: Why is sequential access desirable?**

**Stride-k Access** - accessing every k-th memory address. Note that stride-1 access patterns is the same as sequential access.

**QUESTION: Where might we see stride access patterns?**

**QUESTION: What are the stride patterns of the functions below? How would you order these functions from that which displays the highest locality to that which displays the least?**

```
#define N 1000
typedef struct {
    int vel[3];
    int acc[3];
} point;
point p[N];
```

```

void clear1(point* p, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++)
            p[i].vel[j] = 0;
        for (j = 0; j < 3; j++)
            p[i].acc[j] = 0;
    }
}

```

```

void clear2(point* p, int n) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < 3; j++) {
            p[i].vel[j] = 0;
            p[i].acc[j] = 0;
        }
    }
}

```

```

void clear3(point* p, int n) {
    int i, j;
    for (j = 0; j < 3; j++) {
        for (i = 0; i < n; i++)
            p[i].vel[j] = 0;
        for (i = 0; i < n; i++)
            p[i].acc[j] = 0;
    }
}

```

In `r06-stridebyte.c` and `r07-strideblock.c`, instead of accessing memory sequentially, we use a stride-(1 MB) access pattern.

**QUESTION: Why is reading initially slow, and why does it eventually speed up?**

In `r08-stridestdiobyte.c` and `r09-stridestdioblock.c`, we replace the read system calls with calls to `fread`, and we noticed that the resulting code was much slower.

**QUESTION: Why is reading much slower using `stdio`?**

***Thrashing*** - excessive swapping of pages of data between memory and the hard disk. Often happens when the memory is full and there is poor locality in the program design.