Section Notes 6

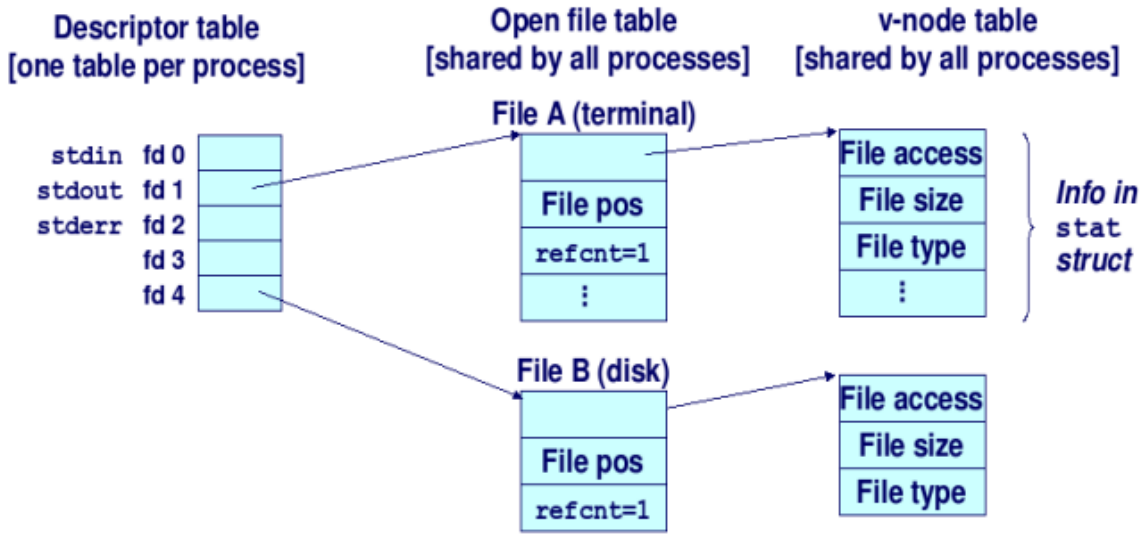
I/O and Advanced Caching

Overview

- I/O System Calls and File Descriptors
- Stdio vs. System Calls
- Data Structures with Caches
- Access Patterns
    - true/false
    - reference strings exercises
    - caching algorithms
- Cache Tags (and other pset questions!)

I/O System Calls, File Descriptors

The Linux kernel maintains a ***descriptor table*** containing an entry for each open I/O resource: a file, a network connection, a pipe (a communication channel between processes), a terminal, etc.
A ***file descriptor*** is an index into this table.



Logically, a file descriptor comprises a *file object*, which represents the underlying data (such as /home/kohler/grades.txt), and a *position*, which is an offset into the file. There can be many file

descriptors simultaneously open for the same file object, each with a different position. For disk files, the position can be explicitly changed: a process can rewind and re-read part of a file, for example, or skip around, as we saw with strided I/O patterns. These files are called *seekable*. However, not all types of file descriptor are seekable. Most communication channels between processes aren't, and neither are network channels.

When a process starts, three file descriptors are generated by the kernel for us. File descriptor 0 refers to ***stdin*** (the input stream, in a shell the terminal (your keyboard) is the default connection), file descriptor 1 refers to ***stdout*** (the output stream, the default is to print to your shell), and file descriptor 2 refers to ***stderr*** (the error stream, the default is to print to your shell).

If the program needs to open other streams it can do so with a number of system calls.
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 ORing the following values together. See more flags in the man page.

`O_RDONLY`	open for reading only	`0x0000`
`O_WRONLY`	open for writing only	`0x0001`
`O_RDWR`	open for reading and writing	`0x0002`
`O_CREAT`	create file if it does not exist	`0x0040`
`O_TRUNC`	truncate size to 0	`0x0200`
`O_SYNC`	synchronous call (write directly to disk)	`0x1000`

ssize_t read(int fd, void *buf, size_t count)

- system call
- attempts to read `count` bytes into `buf` from the given file descriptor `fd`
- returns number of bytes read. This is usually `count`, but may be less. 0 indicates end of file, and -1 indicates error.
- If N bytes are available, where 0 < N < `count`, and read() is reading from a "slow" file descriptor (i.e., a file descriptor where the writing end might never provide more bytes, such as a network connection or a pipe), then read() will return N (a short count).

ssize_t write(int fd, const void *buf, size_t count)

- system call
- attempts to write `count` bytes from `buf` to the given file descriptor `fd`
- returns number of bytes written. This is usually `count`, but may be less. -1 indicates

error.

int close(int fd)

- 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

Before we start working with the system calls you will notice that both read and write use a size_t as an argument but both return a ssize_t. This is a value with the same number of bits as size_t, but it is signed, rather than unsigned. **Why doesn't either call return a size_t?**

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 stdin library maintains a cache of data so that subsequent reads are faster
- returns the actual number of ***items*** read
- Where read() may return early if it temporarily runs out of bytes, fread() will not. It waits until size*nitems bytes are read or the file truly ends.

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 stdin library batches write requests to minimize the number of system calls

int fclose(FILE *stream);

- stdio library version of close

Stdio vs. system calls

Check out the s06/stdio directory in the cs61-sections repository. This contains four programs, two readers and two writers. The `f` versions use stdio, and the other versions don't. Each program works in a loop. It reads or writes a B-byte block, then waits for some delay and starts

again.

Try this:
% ./reader < reader.c
% ./reader -d 0.2 < reader.c
% ./writer | ./reader
% ./freader -d 0.2 < reader.c
% ./fwriter | ./freader

The ./fwriter | ./freader execution has different behavior than the ./writer | ./reader execution! Can you say why?

The stdio library functions are written in terms of the system calls the operating system supports. Stdio implements other functionality, such as buffering and caching, on top, just like you will in pset2. We can get an idea for how stdio is implemented by looking at the system calls it makes. The `strace` program runs another program, but prints to standard error a readable version of every system call that program makes. Let's use strace to investigate these programs' operating system behavior. Some things to try:

% ./writer 2>/dev/null | strace ./reader
What does `2>/dev/null` do?

% ./fwriter 2>/dev/null | strace ./freader
How are the freader system calls different?

% ./fwriter -n | ./freader
How is this behavior different? What do you think is going on?

How would you write a version of fread() that used the read() system call directly? We've supplied an incorrect skeleton version at the end of freader.c. What is wrong with this skeleton? How would you fix it? (Note: Don't worry about niceties like the file "error indicator"; just consider what bytes are read and what values are returned.)

Speeding up Data Structures with Caches

First step's first. Pull new section code. This week's code will be in directory s06/ in the cs61-sections repository.

Overview:

With 3 different data structures:

(A) numwave-list : Doubly Linked List
(B) numwave-vector : Vector

Given N:

(1) Insert N random integers into a data structure. Keep that data structure in sorted order at all times.
(2) N times, select a random integer in the data structure and delete it. Continue to hold the invariant that the data structure is sorted.

Some questions that we hope to address.
Which method is faster? Why? How does and doesn't caching play a role in each of these?

Doubly Linked List:

```
void list_add(struct link** list, value_t value) {
    struct link *newlink;
    while (*list && (*list)->value < value) // 1
      list = &(*list)->next;
    newlink = list_alloc(); // 2
    newlink->value = value;
    newlink->next = *list;
    *list = newlink;
}

void list_remove(struct link** list, unsigned position) {
    struct link* todelete;
    while (position > 0) { // 3
      list = &(*list)->next;
      --position;
    }
    todelete = *list;
    *list = todelete->next;
    list_free(todelete);
}
```

Q. Where is the location of the memory of LINE 2 relative to the previously added number?
Q. How many cache lines are accessed on LINE 1?
Q. How many cache lines are accessed on LINE 3?

Vector:

```
void vector_add(struct vector* v, value_t value) {
    unsigned position = 0;
    while (position != v->size && v->data[position] < value) //1
      ++position;
    if (v->size == v->capacity)
      vector_grow(v); // 2
    memmove(&v->data[position + 1], &v->data[position],
          sizeof(value_t) * (v->size - position)); // 3
    v->data[position] = value;
    ++v->size;
}

void vector_remove(struct vector* v, unsigned position) {
    unsigned cur_position = 0;
    while (cur_position != position) // 4
      ++cur_position;
    memmove(&v->data[cur_position], &v->data[cur_position + 1],
          sizeof(value_t) * (v->size - cur_position - 1)); // 5
    --v->size;
}
```

Q. How many cache lines are accessed on LINE 1?
Q. How many cache lines are accessed on LINE 4?
Q. How often is LINE 2 called relative to LINE 2 of the DLL Method(i.e. how many times is `vector_grow()` called vs. `list_alloc()`)?
Q. Is LINE 3 a “fast” or “slow” function call?
Q. How many cache lines are access on LINE 4?
Q. Is Line 5 a “fast” or “slow” function call?

Conclusion:

Basic principles for “good” memory usage:

- Don’t store data unnecessarily
- Keep data compact
- Access memory in a predictable manner.

Access Patterns

True/False

1. **A direct-mapped cache with N or more slots can handle any reference string containing ≤N distinct addresses with no misses except for cold misses.**

2. **A fully-associative cache with N or more slots can handle any reference string containing ≤N distinct addresses with no misses except for cold misses.**

3. **An operating system's buffer cache is generally fully associative.**

4. **The least-recently-used eviction policy is more useful for very large files that are read sequentially than it is for stacks.**

5. **Making a cache bigger can lower its hit rate for a given workload**

6. **x86 processor caches are coherent (i.e., always appear to contain the most up-to-date values).**

Reference Strings

Name	String
α	1
β	1, 2
γ	1, 2, 3, 4, 5
δ	2, 4
ε	5, 2, 4, 2

Which of the strings might indicate a sequential access pattern? Circle all that apply.

α β γ δ ε None of these

Which of the strings might indicate a strided access pattern with stride >1? Circle all that apply.

α β γ δ ε None of these

The remaining questions concern concatenated permutations of these five strings. For example, the permutation αβγδε refers to this reference string:

1, 1, 2, 1, 2, 3, 4, 5, 2, 4, 5, 2, 4, 2.
We pass such permutations through an initially-empty, fully-associative cache with 3 slots, and observe the numbers of hits.

How many cold misses might a permutation observe? Circle all that apply.

0 1 2 3 4 5 Some other number

How many hits does this permutation observe under FIFO eviction?

Give a permutation that will observe 8 hits under LRU eviction, which is the maximum for any permutation. There are several possible answers. (Write your answer as a permutation of αβγδε. For partial credit, find a permutation that has 7 hits, etc.)

Give a permutation that will observe 2 hits under LRU eviction, which is the minimum for any permutation. There is one unique answer. (Write your answer as a permutation of

αβγδε. For partial credit, find a permutation that has 3 hits, etc.)

Be a Caching Algorithm!

You are given the following access pattern and a 4 slot fully associative cache.

Time	1	2	3	4	5	6	7	8	9	10	11	12	13
Access	a	b	c	d	b	e	a	d	c	f	e	a	d

Cache

Slot	0	1	2	3
Data				

Question: What is in the cache after all 13 accesses are complete if we use the optimal replacement strategy?

Time	1	2	3	4	5	6	7	8	9	10	11	12	13
Access	a	b	c	d	b	e	a	d	c	f	e	a	d
Slot 0													
Slot 1													
Slot 2													
Slot 3													

Slot	0	1	2	3
Data				

Question: What is the hit ratio?

Question: List the time steps that were cold misses.

Question: If we chose to evict the least recently used (LRU) slot what would be in the cache at the end?

Time	1	2	3	4	5	6	7	8	9	10	11	12	13
Access	a	b	c	d	b	e	a	d	c	f	e	a	d
Slot 0													
Slot 1													
Slot 2													
Slot 3													

Slot	0	1	2	3
Data				

Question: What is the hit ratio under LRU?