CS241 System Programming Wikibook

Contents

Chapter 1

Introduction

1. Talk about how the book is organized
2. Talk about how the book goes along with lectures
3. Talk about the asides and the proofs
4. Thank people

Chapter 2

Interprocess Communication

TODO: Epigraph

In very simple embedded systems and early computers, processes directly access memory i.e. "Address 1234" corresponds to a particular byte stored in a particular part of physical memory. For example the IBM 709 had to read and write directly to a tape with no level of abstraction [1, P. 65]. Even in systems after that, it was hard to adopt virtual memory because virtual memory required the whole fetch cycle to be altered through hardware – a change many manufacturers still thought was expensive. In the PDP-10, a workaround was used by using different registers for each process and then virtual memory was added later. In modern systems, this is no longer the case. Instead each process is isolated, and there is a translation process between the address of a particular CPU instruction or piece of data of a process and the actual byte of physical memory ("RAM"). Memory addresses no longer map to physical addresses; the process runs inside virtual memory. Virtual memory not only keeps processes safe (because one process cannot directly read or modify another process's memory) it also allows the system to efficiently allocate and re-allocate portions of memory to different processes. The modern process of translating memory is as follows.

1. A process makes a memory request
2. The circuit first checks the Translation Lookaside Buffer (TLB) if the address page is cached into memory. It skips to the reading from/writing to phase if found otherwise the request goes to the MMU.
3. The Memory Management Unit (MMU) performs the address translation. If the translation succeeds (more on that later), the page get pulled from RAM – conceptually the entire page isn't loaded up. The result is cached in the TLB.

4. The CPU performs the operation by either reading from the physical address or writing to the address.

MMU and Translating Addresses

The Memory Management Unit is part of the CPU, and it converts a virtual memory address into a physical address. There is a sort of pseudocode associated with the MMU.

1. Receive address
2. Try to translate address according to the programmed scheme
3. If the translation fails, report an invalid address
4. Otherwise,

(a) If the page exists in memory, check if the process has permissions to perform the operation on the page meaning the process has access to the page, and it is reading from the page/writing to a page that is not marked as read only.
   i. If so then provide the address, cache the results in the TLB
   ii. Otherwise trigger a hardware interrupt. The kernel will most likely send a SIGSEGV or a Segmentation Violation.

(b) If the page doesn't exist in memory, generate an Interrupt.
   i. The kernel could realize that this page could either be not allocated or on disk. If it fits the mapping, allocate the page and try the operation again.
   ii. Otherwise, this is an invalid access and the kernel will most likely send a SIGSEGV to the process.

Imagine you had a 32 bit machine, meaning pointers are 32 bits. They can address 2^32 different locations or 4GB of memory where one address is one byte. Imagine we had a large table - here's the clever part - stored in memory! For every possible address (all 4 billion of them) we will store the 'real' i.e. physical address. Each physical address will need 4 bytes (to hold the 32 bits). This scheme would require 16 billion bytes to store all of entries. Oops - our lookup scheme would consume all of the memory that we could possibly buy for our 4GB machine. We need to do better than this. Our lookup table better be smaller than the memory we have otherwise we will have no space left for our actual programs and operating system data. The solution is to chunk memory into small regions called 'pages' and 'frames' and use a lookup table for each page.

A **page** is a block of virtual memory. A typical block size on Linux operating system is 4KB or 2^12 addresses, though you can find examples of larger blocks. So rather than talking about individual bytes we can talk about blocks of 4KBs, each block is called a page. We can also number our pages ("Page 0" "Page 1" etc). Let's do a sample calculation of how many pages are there assume page size of 4KB.

> For a 32 bit machine, 2^32 address / 2^12 (address/page) = 2^20 pages.
> For a 64 bit machine, 2^64 / 2^12 = 2^52, which is roughly 10^15 pages.

Terminology

A **frame** (or sometimes called a 'page frame') is a block of *physical memory* or RAM (=Random Access Memory). This kind of memory is occasionally called 'primary storage' in contrast with lower, secondary storage such as spinning disks that have lower access times. A frame is the same number of bytes as a virtual page. If a 32 bit machine has 2^32B of RAM, then there will be the same number of them in the addressable space of the machine. It's unlikely that a 64 bit machine will ever have 2^64 bytes of RAM.

A **page table** is a mapping between a page to the frame. For example Page 1 might be mapped to frame 45, page 2 mapped to frame 30. Other frames might be currently unused or assigned to other running processes, or used internally by the operating system.

A simple page table could be imagined as an array.

```
int frame_num = table[page_num];
```

For a 32 bit machine with 4KB pages, each entry needs to hold a frame number - i.e. 20 bits because we calculated there are 2^20 frames. That's 2.5 bytes per entry! In practice, we'll round that up to 4 bytes per entry and find a use for those spare bits. With 4 bytes per entry x 2^20 entries = 4 MB of

physical memory are required to hold the page table For a 64 bit machine with 4KB pages, each entry needs 52 bits. Let's round up to 64 bits (8 bytes) per entry. With 2^52 entries thats 2^55 bytes (roughly 40 peta bytes...) Oops our page table is too large In 64 bit architectures memory addresses are sparse, so we need a mechanism to reduce the page table size, given that most of the entries will never be used.

An **offset** take a particular page and looks up a byte by adding it to the start of the page. Remember our page table maps pages to frames, but each page is a block of contiguous addresses. How do we calculate which particular byte to use inside a particular frame? The solution is to re-use the lowest bits of the virtual memory address directly. For example, suppose our process is reading the following address- `VirtualAddress = 11110000111100001111000010101010 (binary)`

On a machine with page size 256 Bytes, then the lowest 8 bits (10101010) will be used as the offset. The remaining upper bits will be the page number (111100001111000011110000).

Multi-level page tables

Multi-level pages are one solution to the page table size issue for 64 bit architectures. We'll look at the simplest implementation - a two level page table. Each table is a list of pointers that point to the next level of tables, not all sub-tables need to exist. An example, two level page table for a 32 bit architecture is shown below-

```
VirtualAddress = 11110000111111110000000010101010 (binary)
                 |-Index1-||        ||          | 10 bit Directory index
                           |-Index2-||          | 10 bit Sub-table index
                                     |--offset--| 12 bit offset (passed directly to RAM)
```

In the above scheme, determining the frame number requires two memory reads: The topmost 10 bits are used in a directory of page tables. If 2 bytes are used for each entry, we only need 2KB to store this entire directory. Each subtable will point to physical frames (i.e. required 4 bytes to store the 20 bits). However, for processes with only tiny memory needs, we only need to specify entries for low memory address (for the heap and program code) and high memory addresses (for the stack). Each subtable is 1024 entries x 4 bytes i.e. 4KB for each subtable.

Thus the total memory overhead for our multi-level page table has shrunk from 4MB (for the single level implementation) to 3 frames of memory (12KB) ! Here's why: We need at least one frame for the high level directory and two frames for just two sub-tables. One sub-table is necessary for the low addresses (program code, constants and possibly a tiny heap), the other sub-table is for higher addresses used by the environment and stack. In practice, real programs will likely need more sub-table entries, as each subtable can only reference 1024*4KB = 4MB of address space but the main point still stands - we have significantly reduced the memory overhead required to perform page table look ups.

Page Table Disadvantages

Yes - Significantly ! (But thanks to clever hardware, usually no...) Compared to reading or writing memory directly. For a single page table, our machine is now twice as slow! (Two memory accesses are required) For a two-level page table, memory access is now three times as slow. (Three memory accesses are required)

To overcome this overhead, the MMU includes an associative cache of recently-used virtual-page-to-frame lookups. This cache is called the TLB ("translation lookaside buffer"). Everytime a virtual address needs to be translated into a physical memory location, the TLB is queried in parallel to the page table. For most memory accesses of most programs, there is a significant chance that the TLB has cached the results. However if a program does not have good cache coherence (for example is reading from random memory locations of many different pages) then the TLB will not have the result cache and now the MMU must use the much slower page table to determine the physical frame.

This may be how one splits up a multi level page table.

Advanced Frames and Page Protections

Frames be shared between processes! In addition to storing the frame number, the page table can be used to store whether a process can write or only read a particular frame. Read only frames can then be safely shared between multiple processes. For example, the C-library instruction code can be shared between all processes that dynamically load the code into the process memory. Each process can only read that memory. Meaning that if you try to write to a read-only page in memory you will get a `SEGFAULT`. That is why sometimes memory accesses segfault and sometimes they don't, it all depends on if your hardware says that you can access.

In addition, processes can share a page with a child process using the `mmap` system call. `mmap` is an interesting call because instead of tying each virtual address to a physical frame, it ties it to something else. That something else can be a file, a GPU unit, or any other memory mapped operation that you can think of! Writing to the memory address may write through to the device or the write may be paused by the operating system but this is a very powerful abstraction because often the operating system is able to perform optimizations (multiple processes memory mapping the same file can have the kernel create one mapping). In addition, it is common to store at least read-only, modification and execution information.

Read-only bit

The read-only bit marks the page as read-only. Attempts to write to the page will cause a page fault. The page fault will then be handled by the Kernel. Two examples of the read-only page include sharing the c runtime library between multiple processes (for security you wouldn't want to allow one process to modify the library); and Copy-On-Write where the cost of duplicating a page can be delayed until the first write occurs.

Dirty bit

Page Table The dirty bit allows for a performance optimization. A page on disk that is paged in to physical memory, then read from, and subsequently paged out again does not need to be written back to disk, since the page hasn't changed. However, if the page was written to after it's paged in, its dirty bit will be set, indicating that the page must be written back to the backing store. This strategy requires that the backing store retain a copy of the page after it is paged in to memory. When a dirty bit is not used, the backing store need only be as large as the instantaneous total size of all paged-out pages at any moment. When a dirty bit is used, at all times some pages will exist in both physical memory and the backing store.

Execution bit

The execution bit defines whether bytes in a page can be executed as CPU instructions. By disabling a page, it prevents code that is maliciously stored in the process memory (e.g. by stack overflow) from being easily executed. (further reading: background)

Page Faults

A page fault is when a running program tries to access some virtual memory in its address space that is not mapped to physical memory. Page faults will also occur in other situations. There are three types of Page Faults

1. **Minor** If there is no mapping yet for the page, but it is a valid address. This could be memory asked for by `sbrk(2)` but not written to yet meaning that the operating system can wait for the first write before allocating space. The OS simply makes the page, loads it into memory, and moves on.

2. **Major** If the mapping to the page is not in memory but on disk. What this will do is swap the page into memory and swap another page out. If this happens frequently enough, your program is said to *thrash* the MMU.
3. **Invalid** When you try to write to a non-writable memory address or read to a non-readable memory address. The MMU generates an invalid fault and the OS will usually generate a `SIGSEGV` meaning segmentation violation meaning that you wrote outside the segment that you could write to.

Pipes

Inter process communication is any way for one process to talk to another process. You've already seen one form of this virtual memory! A piece of virtual memory can be shared between parent and child, leading to communication. You may want to wrap that memory in `pthread_mutexattr_setpshared(&attrmutex, PTHREAD_PROCESS_SHARED);` mutex (or a process wide mutex) to prevent race conditions. There are more standard ways of IPC, like pipes! Consider if you type the following into your terminal.

```
$ ls -1 | cut -d'.' -f1 | uniq | sort | tee dir_contents
```

What does the following code do? Well it `ls`'s the current directory (the `-1` means that it outputs one entry per line). The `cut` command then takes everything before the first period. Uniq makes sure all the lines are uniq, sort sorts them and tee outputs to a file. The important part is that bash creates **5 separate processes** and connects their standard outs/stdins with pipes the trail lookssomething like this.

```
(0) ls (1)------>(0) cut (1)------->(0) uniq (1)------>(0) sort (1)------>(0) tee (1)
```

The numbers in the pipes are the file descriptors for each process and the arrow represents the redirect or where the output of the pipe is going. A POSIX pipe is almost like its real counterpart - you can stuff bytes down one end and they will appear at the other end in the same order. Unlike real pipes however, the flow is always in the same direction, one file descriptor is used for reading and the other for writing. The `pipe` system call is used to create a pipe. These file descriptors can be used with `read` and with `write`. A common method of using pipes is to create the pipe before forking in order to communicate with a child process

```c
int filedes[2];
pipe (filedes);
pid_t child = fork();
if (child > 0) {/* I must be the parent */
    char buffer[80];
    int bytesread = read(filedes[0], buffer, sizeof(buffer));
    // do something with the bytes read
} else {
    write(filedes[1], "done", 4);
}
```

One can use pipes inside of the same process, but there tends to be no added benefit. Here's an example program that sends a message to itself:

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
    int fh[2];
    pipe(fh);
    FILE *reader = fdopen(fh[0], "r");
    FILE *writer = fdopen(fh[1], "w");
    // Hurrah now I can use printf rather than using low-level read()
        write()
    printf("Writing...\n");
    fprintf(writer,"%d %d %d\n", 10, 20, 30);
    fflush(writer);

    printf("Reading...\n");
    int results[3];
    int ok = fscanf(reader,"%d %d %d", results, results + 1, results
        + 2);
    printf("%d values parsed: %d %d %d\n", ok, results[0],
        results[1], results[2]);

    return 0;
}
```

The problem with using a pipe in this fashion is that writing to a pipe can block meaning the pipe only has a limited buffering capacity. If the pipe is full the writing process will block! The maximum size of the buffer is system dependent; typical values from 4KB upto 128KB.

```c
int main() {
    int fh[2];
    pipe(fh);
    int b = 0;
    #define MESG "............................."
    while(1) {
        printf("%d\n",b);
        write(fh[1], MESG, sizeof(MESG))
        b+=sizeof(MESG);
    }
    return 0;
}
```

Pipe Gotchas

Here's a complete example that doesn't work! The child reads one byte at a time from the pipe and prints it out - but we never see the message! Can you see why?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main() {
    int fd[2];
    pipe(fd);
    //You must read from fd[0] and write from fd[1]
    printf("Reading from %d, writing to %d\n", fd[0], fd[1]);

    pid_t p = fork();
    if (p > 0) {
        /* I have a child therefore I am the parent*/
        write(fd[1],"Hi Child!",9);

        /*don't forget your child*/
        wait(NULL);
    } else {
        char buf;
        int bytesread;
        // read one byte at a time.
        while ((bytesread = read(fd[0], &buf, 1)) > 0) {
            putchar(buf);
        }
    }
    return 0;
}
```

The parent sends the bytes `H,i,(space),C...!` into the pipe (this may block if the pipe is full). The child starts reading the pipe one byte at a time. In the above case, the child process will read and print each character. However it never leaves the while loop! When there are no characters left to read it simply blocks and waits for more

The call `putchar` writes the characters out but we never flush the `stdout` buffer. i.e. We have transferred the message from one process to another but it has not yet been printed. To see the message we could flush the buffer e.g. `fflush(stdout)` (or `printf("\n")` if the output is going to a terminal). A better solution would also exit the loop by checking for an end-of-message marker,

```c
        while ((bytesread = read(fd[0], &buf, 1)) > 0) {
            putchar(buf);
            if (buf == '!') break; /* End of message */
        }
```

Processes receive the signal SIGPIPE when no process is listening! From the pipe(2) man page -

```
If all file descriptors referring to the read end of a pipe have been
    closed,
 then a write(2) will cause a SIGPIPE signal to be generated for the
     calling process.
```

Tip: Notice only the writer (not a reader) can use this signal. To inform the reader that a writer is closing their end of the pipe, you could write your own special byte (e.g. 0xff) or a message (`"Bye!"`)

Here's an example of catching this signal that does not work! Can you see why?

```c
#include <stdio.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void no_one_listening(int signal) {
    write(1, "No one is listening!\n", 21);
}

int main() {
    signal(SIGPIPE, no_one_listening);
    int filedes[2];

    pipe(filedes);
    pid_t child = fork();
    if (child > 0) {
        /* I must be the parent. Close the listening end of the pipe */
        /* I'm not listening anymore!*/
        close(filedes[0]);
    } else {
        /* Child writes messages to the pipe */
        write(filedes[1], "One", 3);
        sleep(2);
        // Will this write generate SIGPIPE ?
        write(filedes[1], "Two", 3);
        write(1, "Done\n", 5);
    }
    return 0;
}
```

The mistake in above code is that there is still a reader for the pipe! The child still has the pipe's first file descriptor open and remember the specification? All readers must be closed

When forking, *It is common practice* to close the unnecessary (unused) end of each pipe in the child and parent process. For example the parent might close the reading end and the child might close the writing end (and vice versa if you have two pipes)

Why is my pipe hanging?

Reads and writes hang on Named Pipes until there is at least one reader and one writer, take this

```
1$ mkfifo fifo
1$ echo Hello > fifo
# This will hang until I do this on another terminal or another
    process
2$ cat fifo
Hello
```

Any open is called on a named pipe the kernel blocks until another process calls the opposite open. Meaning, echo calls `open(.., O_RDONLY)` but that blocks until cat calls `open(.., O_WRONLY)`, then the programs are allowed to continue.

Race condition with named pipes

What is wrong with the following program?

```c
//Program 1

int main(){
    int fd = open("fifo", O_RDWR | O_TRUNC);
    write(fd, "Hello!", 6);
    close(fd);
    return 0;
}

//Program 2
int main() {
    char buffer[7];
    int fd = open("fifo", O_RDONLY);
    read(fd, buffer, 6);
    buffer[6] = '\0';
    printf("%s\n", buffer);
    return 0;
}
```

This may never print hello because of a race condition. Since you opened the pipe in the first process under both permissions, open won't wait for a reader because you told the operating system that you are a reader! Sometimes it looks like it works because the execution of the code looks something like this.

Process 1	Process 2
open(O_RDWR) & write()	
	open(O_RDONLY) & read()
close() & exit()	
	print() & exit()

Process 1	Process 2
open(O_RDWR) & write()	
close() & exit()	(Named pipe is destroyed)
(Blocks indefinitely)	open(O_RDONLY)

What is filling up the pipe? What happens when the pipe becomes full?

A pipe gets filled up when the writer writes too much to the pipe without the reader reading any of it. When the pipes become full, all writes fail until a read occurs. Even then, a write may partial fail if the pipe has a little bit of space left but not enough for the entire message

To avoid this, usually two things are done. Either increase the size of the pipe. Or more commonly, fix your program design so that the pipe is constantly being read from.

Are pipes process safe?

Yes! Pipe write are atomic up to the size of the pipe. Meaning that if two processes try to write to the same pipe, the kernel has internal mutexes with the pipe that it will lock, do the write, and return. The only gotcha is when the pipe is about to become full. If two processes are trying to write and the pipe can only satisfy a partial write, that pipe write is not atomic – be careful about that!

The lifetime of pipes

Unnamed pipes (the kind we've seen up to this point) live in memory (do not take up any disk space) and are a simple and efficient form of inter-process communication (IPC) that is useful for streaming data and simple messages. Once all processes have closed, the pipe resources are freed.

Want to use pipes with printf and scanf? Use fdopen!

POSIX file descriptors are simple integers 0,1,2,3... At the C library level, C wraps these with a buffer and useful functions like printf and scanf, so we that we can easily print or parse integers, strings etc. If you already have a file descriptor then you can 'wrap' it yourself into a FILE pointer using `fdopen` :

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
    char *name="Fred";
    int score = 123;
    int filedes = open("mydata.txt", "w", O_CREAT, S_IWUSR | S_IRUSR);

    FILE *f = fdopen(filedes, "w");
    fprintf(f, "Name:%s Score:%d\n", name, score);
    fclose(f);
```

For writing to files this is unnecessary - just use `fopen` which does the same as `open` and `fdopen` However for pipes, we already have a file descriptor - so this is great time to use `fdopen`

Here's a complete example using pipes that almost works! Can you spot the error? Hint: The parent never prints anything!

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
    int fh[2];
    pipe(fh);
    FILE *reader = fdopen(fh[0], "r");
    FILE *writer = fdopen(fh[1], "w");
    pid_t p = fork();
    if (p > 0) {
        int score;
        fscanf(reader, "Score %d", &score);
        printf("The child says the score is %d\n", score);
    } else {
        fprintf(writer, "Score %d", 10 + 10);
        fflush(writer);
    }
    return 0;
}
```

Note the unnamed pipe resource will disappear once both the child and parent have exited. In the above example the child will send the bytes and the parent will receive the bytes from the pipe. However, no end-of-line character is ever sent, so `fscanf` will continue to ask for bytes because it is waiting for the end of the line i.e. it will wait forever! The fix is to ensure we send a newline character, so that `fscanf` will return.

```
change:  fprintf(writer, "Score %d", 10 + 10);
to:      fprintf(writer, "Score %d\n", 10 + 10);
```

So do we need to fflush too?

Yes, if you want your bytes to be sent to the pipe immediately! At the beginning of this course we assumed that file streams are always *line buffered* i.e. the C library will flush its buffer everytime you send a newline character. Actually this is only true for terminal streams - for other filestreams the C library attempts to improve performance by only flushing when it's internal buffer is full or the file is closed.

When do I need two pipes?

If you need to send data to and from a child asynchronously, then two pipes are required (one for each direction). Otherwise the child would attempt to read its own data intended for the parent (and vice versa)!

Named Pipes

How do I create named pipes?

An alternative to *unamed* pipes is *named* pipes created using `mkfifo`.

From the command line: `mkfifo` From C: `int mkfifo(const char *pathname, mode_t mode);`

You give it the path name and the operation mode, it will be ready to go! Named pipes take up no space on the disk. What the operating system is essentially telling you when you have a named pipe is that it will create an unnamed pipe that refers to the named pipe, and that's it! There is no additional magic. This is just for programming convenience if processes are started without forking (meaning that there would be no way to get the file descriptor to the child process for an unnamed pipe)

Two types of files

On linux, there are two abstractions with files. The first is the linux `fd` level abstraction that means you can use * `open` * `read` * `write` * `close` * `lseek` * `fcntl` ...

And so on. The linux interface is very powerful and expressive, but sometimes we need portability (for example if we are writing for a mac or windows). This is where C's abstraction comes into play. On different operating systems, C uses the low level functions to create a wrapper around files you can use everywhere, meaning that C on linux uses the above calls. C has a few of the following * `fopen` * `fread` or `fgetc/fgets` or `fscanf` * `fwrite` or `fprintf` * `fclose` * `fflush`

But you don't get the expressiveness that linux gives you with system calls you can convert back and forth between them with `int fileno(FILE* stream)` and `FILE* fdopen(int fd...)`

Another important aspect to note is the C files are **buffered** meaning that their contents may not be written right away by default. You can can change that with C options.

How do I tell how large a file is?

For files less than the size of a long, using fseek and ftell is a simple way to accomplish this:

Move to the end of the file and find out the current position.

```
fseek(f, 0, SEEK_END);
long pos = ftell(f);
```

This tells us the current position in the file in bytes - i.e. the length of the file!

`fseek` can also be used to set the absolute position.

```
fseek(f, 0, SEEK_SET); // Move to the start of the file
fseek(f, posn, SEEK_SET); // Move to 'posn' in the file.
```

All future reads and writes in the parent or child processes will honor this position. Note writing or reading from the file will change the current position.

See the man pages for fseek and ftell for more information.

But try not to do this

Note: This is not recommended in the usual case because of a quirk with the C language. That quirk is that longs only need to be **4 Bytes big** meaning that the maximum size that `ftell` can return is a little under 2 Gigabytes (which we know nowadays our files could be hundreds of gigabytes or even terabytes on a distributed file system). What should we do instead? Use `stat`! We will cover stat in a later part but here is some code that will tell you the size of the file

```
struct stat buf;
if(stat(filename, &buf) == -1){
    return -1;
}
return (ssize_t)buf.st_size;
```

`buf.st_size` is of type `off_t` which is big enough for large files.

What happens if a child process closes a filestream using fclose or close?

Closing a file stream is unique to each process. Other processes can continue to use their own file-handle. Remember, everything is copied over when a child is created, even the relative positions of the files.

How about mmap for files?

One of the general uses for mmap is to map a file to memory. This does not mean that the file is malloc'ed to memory right away. Take the following code for example.

```
int fd = open(...); //File is 2 Pages
char* addr = mmap(..fd..);
addr[0] = '1';
```

The kernel may say, "okay I see that you want to mmap the file into memory, so I'll reserve some space in your address space that is the length of the file". That means when you write to addr[0] that you are actually writing to the first byte of the file. The kernel can actually do some optimizations too. Instead of loading the file into memory, it may only load pages at a time because if the file is 1024 pages; you may only access 3 or 4 pages making loading the entire file a waste of time. That is why page faults are so powerful! They let the operating system take control of how much you use your files.

For every mmap

Remember that once you are done `mmap`ping that you `munmap` to tell the operating system that you are no longer using the pages allocated, so the OS can write it back to disk and give you the addresses back in case you need to malloc later.

Bibliography

[1] International Business Machines Corporation (IBM). *IBM 709 Data Processing System Reference Manual*. International Business Machines Corporation (IBM). URL `http://archive.computerhistory.org/resources/text/Fortran/102653991.05.01.acc.pdf`.

Chapter 3

Appendix

The Hitchhiker's Guide to Debugging C Programs

This is going to be a massive guide to helping you debug your C programs. There are different levels that you can check errors and we will be going through most of them. Feel free to add anything that you found helpful in debugging C programs including but not limited to, debugger usage, recognizing common error types, gotchas, and effective googling tips.

In-Code Debugging

Clean code

Make your code modular using helper functions. If there is a repeated task (getting the pointers to contiguous blocks in the malloc MP, for example), make them helper functions. And make sure each function does one thing very well, so that you don't have to debug twice.

Let's say that we are doing selection sort by finding the minimum element each iteration like so,

```
void selection_sort(int *a, long len){
    for(long i = len-1; i > 0; --i){
        long max_index = i;
        for(long j = len-1; j >= 0; --j){
            if(a[max_index] < a[j]){
                max_index = j;
            }
        }
        int temp = a[i];
        a[i] = a[max_index];
        a[max_index] = temp;
    }

}
```

Many can see the bug in the code, but it can help to refactor the above method into

```
long max_index(int *a, long start, long end);
void swap(int *a, long idx1, long idx2);
void selection_sort(int *a, long len);
```

And the error is specifically in one function. In the end, we are not a class about refactoring/debugging your code. In fact, most kernel code is so atrocious that you don't want to read it. But for the sake of debugging, it may benefit you in the long run to adopt some of these practices.

Asserts!

Use assertions to make sure your code works up to a certain point – and importantly, to make sure you don't break it later. For example, if your data structure is a doubly linked list, you can do something like `assert(node->size == node->next->prev->size)` to assert that the next node has a pointer to the current node. You can also check the pointer is pointing to an expected range of memory address, not null, ->size is reasonable etc. The `NDEBUG` macro will disable all assertions, so don't forget to set that once you finish debugging. assert link

Here's a quick example with assert. Let's say that I'm writing code using memcpy

```
assert(!(src < dest+n && dest < src+n)); //Checks overlap
memcpy(dest, src, n);
```

This check can be turned off at compile time, but will save you **tons** of trouble debugging!

printfs

When all else fails, print like crazy! Each of your functions should have an idea of what it is going to do (ie find_min better find the minimum element). You want to test that each of your functions is doing what it set out to do and see exactly where your code breaks. In the case with race conditions, tsan may be able to help, but having each thread print out data at certain times could help you identify the race condition.

Valgrind

Valgrind is a suite of tools designed to provide debugging and profiling tools to make your programs more correct and detect some runtime issues. The most used of these tools is Memcheck, which can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behaviour (for example, unfreed memory buffers). To run Valgrind on your program:

```
valgrind --leak-check=full --show-leak-kinds=all myprogram arg1 arg2
```

Arguments are optional and the default tool that will run is Memcheck. The output will be presented in form of number of allocations, number of freed allocations, and the number of errors.Suppose we have a simple program like this:

```c
#include <stdlib.h>

void dummy_function() {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;      // error 1:as you can see here we write to an out
        of bound memory address
}                   // error 2: memory leak the allocated x not freed

int main(void) {
    dummy_function();
    return 0;
}
```

This program compiles and run with no errors. Let's see what Valgrind will output.

```
==29515== Memcheck, a memory error detector
==29515== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et
    al.
==29515== Using Valgrind-3.11.0 and LibVEX; rerun with -h for
    copyright info
==29515== Command: ./a
==29515==
==29515== Invalid write of size 4
==29515==   at 0x400544: dummy_function (in
    /home/rafi/projects/exocpp/a)
==29515==   by 0x40055A: main (in /home/rafi/projects/exocpp/a)
==29515== Address 0x5203068 is 0 bytes after a block of size 40
    alloc'd
==29515==   at 0x4C2DB8F: malloc (in
    /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==29515==   by 0x400537: dummy_function (in
    /home/rafi/projects/exocpp/a)
==29515==   by 0x40055A: main (in /home/rafi/projects/exocpp/a)
==29515==
==29515==
==29515== HEAP SUMMARY:
==29515==    in use at exit: 40 bytes in 1 blocks
==29515==  total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==29515==
==29515== LEAK SUMMARY:
==29515==   definitely lost: 40 bytes in 1 blocks
==29515==   indirectly lost: 0 bytes in 0 blocks
==29515==     possibly lost: 0 bytes in 0 blocks
==29515==   still reachable: 0 bytes in 0 blocks
==29515==        suppressed: 0 bytes in 0 blocks
==29515== Rerun with --leak-check=full to see details of leaked memory
==29515==
==29515== For counts of detected and suppressed errors, rerun with: -v
==29515== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from
    0)
```

Invalid write: It detected our heap block overrun, writing outside of allocated block.

Definitely lost: Memory leak — you probably forgot to free a memory block.

Valgrind is a very effective tool to check for errors at runtime. C is very special when it comes to such behavior, so after compiling your program you can use Valgrind to fix errors that your compiler may not catch and that usually happen when your program is running.

For more information, you can refer to the valgrind website.

TSAN

ThreadSanitizer is a tool from Google, built into clang and gcc, to help you detect race conditions in your code. For more information about the tool, see the Github wiki. Note, that running with tsan will slow your code down a bit. Consider the following code.

```
#include <pthread.h>
#include <stdio.h>

int global;

void *Thread1(void *x) {
    global++;
    return NULL;
}

int main() {
    pthread_t t[2];
    pthread_create(&t[0], NULL, Thread1, NULL);
    global = 100;
    pthread_join(t[0], NULL);
}
// compile with gcc -fsanitize=thread -pie -fPIC -ltsan -g
    simple_race.c
```

We can see that there is a race condition on the variable `global`. Both the main thread and the thread created with pthread_create will try to change the value at the same time. But, does ThreadSantizer catch it?

```
$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=28888)
  Read of size 4 at 0x7f73ed91c078 by thread T1:
    #0 Thread1 /home/zmick2/simple_race.c:7 (exe+0x000000000a50)
    #1  :0 (libtsan.so.0+0x00000001b459)

  Previous write of size 4 at 0x7f73ed91c078 by main thread:
    #0 main /home/zmick2/simple_race.c:14 (exe+0x000000000ac8)

  Thread T1 (tid=28889, running) created by main thread at:
    #0  :0 (libtsan.so.0+0x00000001f6ab)
    #1 main /home/zmick2/simple_race.c:13 (exe+0x000000000ab8)

SUMMARY: ThreadSanitizer: data race /home/zmick2/simple_race.c:7
    Thread1
==================
ThreadSanitizer: reported 1 warnings
```

If we compiled with the debug flag, then it would give us the variable name as well.

GDB

Introduction to gdb

Setting breakpoints programmatically A very useful trick when debugging complex C programs with GDB is setting breakpoints in the source code.

```c
int main() {
    int val = 1;
    val = 42;
    asm("int $3"); // set a breakpoint here
    val = 7;
}
```

```
$ gcc main.c -g -o main && ./main
(gdb) r
[...]
Program received signal SIGTRAP, Trace/breakpoint trap.
main () at main.c:6
6     val = 7;
(gdb) p val
$1 = 42
```

Checking memory content Memory Content

For example,

```c
int main() {
    char bad_string[3] = {'C', 'a', 't'};
    printf("%s", bad_string);
}
```

```
$ gcc main.c -g -o main && ./main
$ Cat ZVQï£¡- $
```

```
(gdb) l
1 #include <stdio.h>
2 int main() {
3     char bad_string[3] = {'C', 'a', 't'};
4     printf("%s", bad_string);
5 }
(gdb) b 4
Breakpoint 1 at 0x100000f57: file main.c, line 4.
(gdb) r
[...]
Breakpoint 1, main () at main.c:4
4     printf("%s", bad_string);
(gdb) x/16xb bad_string
0x7fff5fbff9cd: 0x63 0x61 0x74 0xe0 0xf9 0xbf 0x5f 0xff
0x7fff5fbff9d5: 0x7f 0x00 0x00 0xfd 0xb5 0x23 0x89 0xff

(gdb)
```

Here, by using the `x` command with parameters `16xb`, we can see that starting at memory address `0x7fff5fbff9c` (value of `bad_string`), `printf` would actually see the following sequence of bytes as a string because we provided a malformed string without a null terminator.

What do you actually use to run your program? A shell! A shell is a programming language that is running inside your terminal. A terminal is merely a window to input commands. Now, on POSIX we usually have one shell called `sh` that is linked to a POSIX compliant shell called `dash`. Most of the time, you use a shell called `bash` that is not entirely POSIX compliant but has some nifty built in features. If you want to be even more advanced, `zsh` has some more powerful features like tab complete on programs and fuzzy patterns, but that is neither here nor there.

Shell

A shell is actually how you are going to be interacting with the system. Before user friendly operating systems, when a computer started up all you had access to was a shell. This meant that all of your commands and editing had to be done this way. Nowadays, our computers start up in desktop mode, but one can still access a shell using a terminal.

```
(Stuff) $
```

It is ready for your next command! You can type in a lot of unix utilities like `ls`, `echo Hello` and the shell will execute them and give you the result. Some of these are what are known as `shell-builtins` meaning that the code is in the shell program itself. Some of these are compiled programs that you run. The shell only looks through a special variable called path which contains a list of `:` separated paths to search for an executable with your name, here is an example path.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:
/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

So when the shell executes `ls`, it looks through all of those directories, finds `/bin/ls` and executes that.

```
$ ls
...
$ /bin/ls
```

You can always call through the full path. That is always why in past classes if you want to run something on the terminal you've had to do `./exe` because typically the directory that you are working in is not in the `PATH` variable. The `.` expands to your current directory and your shell executes `<current_dir>/exe` which is a valid command.

Shell tricks and tips

- The up arrow will get you your most recent command
- `ctrl-r` will search commands that you previously ran
- `ctrl-c` will interrupt your shell's process
- Add more!

Alright then what's a terminal?

A terminal is just an application that displays the output form the shell. You can have your default terminal, a quake based terminal, terminator, the options are endless!

Common Utilities

1. `cat` concatenate multiple files. It is regularly used to print out the contents of a file to the terminal but the original use was concatenation.

```
$ cat file.txt
...
$ cat shakespeare.txt shakespeare.txt > two_shakes.txt
```

2. `diff` tells you the difference of the two files. If nothing is printed, then zero is returned meaning the files are the same byte for byte. Otherwise, the longest common subsequence difference is printed

```
$ cat prog.txt
hello
world
$ cat adele.txt
hello
it's me
$ diff prog.txt prog.txt
$ diff shakespeare.txt shakespeare.txt
2c2
< world
---
> it's me
```

3. `grep` tells you which lines in a file or standard in match a POSIX pattern.

```
$ grep it adele.txt
it's me
```

4. `ls` tells you which files are in the current directory.
5. `cd` this is a shell builtin but it changes to a relative or absolute directory

```
$ cd /usr
$ cd lib/
$ cd -
$ pwd
/usr/
```

6. `man` every system programmers favorite command, tells you more about all your favorite functions!
7. `make` executes programs according to a makefile.

Syntactic

Shells have many useful utilities like saving output to a file using redirection >. This overwrites the file from the beginning. If you only meant to append to the file, you can use ». Unix also allows file descriptor swapping. This means that you can take the output going to one file descriptor and make it seem like its coming out of another. The most common one is 2>1 which means take the stderr and make it seem like it is coming out of standard out. This is important because when you use > and » they only write the standard output of the file. There are some examples below.

```
$ ./program > output.txt # To overwrite
$ ./program >> output.txt # To append
$ ./program 2>&1 > output_all.txt # stderr & stdout
$ ./program 2>&1 > /dev/null # don't care about any output
```

The pipe operator has a fascinating history. The UNIX philosophy is writing small programs and chaining them together to do new and interesting things. Back in the early days, hard disk space was limited and write times were slow. Brian Kernighan wanted to maintain the philosophy while also not having to write a bunch of intermediate files that take up hard drive space. So, the UNIX pipe was born. A pipe take the `stdout` of the program on its left and feeds it to the `stdin` of the program on its write. Consider the command `tee`. It can be used as a replacement for the redirection operators because tee will both write to a file and output to standard out. It also has the added benefit that it doesn't need to be the last command in the list. Meaning, that you can write an intermediate result and continue your piping.

```
$ ./program | tee output.txt # Overwrite
$ ./program | tee -a output.txt # Append
$ head output.txt | wc | head -n 1 # Multi pipes
$ ((head output.txt) | wc) | head -n 1 # Same as above
$ ./program | tee intermediate.txt | wc
```

The and || operator are operators that execute a command sequentially. only executes a command if the previous command succeeds, and || always executes the next command.

```
$ false && echo "Hello!"
$ true && echo "Hello!"
$ false || echo "Hello!"
```

TODO: Shell Tricks

Tips and tricks

TODO: Shell Tricks

What are environment variables?

Well each process gets its own dictionary of environment variables that are copied over to the child. Meaning, if the parent changes their environment variables it won't be transferred to the child and vice versa. This is important in the fork-exec-wait trilogy if you want to exec a program with different environment variables than your parent (or any other process).

For example, you can write a C program that loops through all of the time zones and executes the `date` command to print out the date and time in all locals. Environment variables are used for all sorts of programs so modifying them is important.

Stack Smashing

Each thread uses a stack memory. The stack 'grows downwards' - if a function calls another function, then the stack is extended to smaller memory addresses. Stack memory includes non-static automatic (temporary) variables, parameter values and the return address. If a buffer is too small some data (e.g. input values from the user), then there is a real possibility that other stack variables and even the return address will be overwritten. The precise layout of the stack's contents and order of the automatic variables is architecture and compiler dependent. However with a little investigative work we can learn how to deliberately smash the stack for a particular architecture.

The example below demonstrates how the return address is stored on the stack. For a particular 32 bit architecture Live Linux Machine, we determine that the return address is stored at an address two pointers (8 bytes) above the address of the automatic variable. The code deliberately changes the stack value so that when the input function returns, rather than continuing on inside the main method, it jumps to the exploit function instead.

```c
// Overwrites the return address on the following machine:
// http://cs-education.github.io/sys/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void breakout() {
    puts("Welcome. Have a shell...");
    system("/bin/sh");
}
void input() {
  void *p;
  printf("Address of stack variable: %p\n", &p);
  printf("Something that looks like a return address on stack: %p\n",
      *((&p)+2));
  // Let's change it to point to the start of our sneaky function.
  *((&p)+2) = breakout;
}
int main() {
    printf("main() code starts at %p\n",main);

    input();
    while (1) {
        puts("Hello");
        sleep(1);
    }

    return 0;
}
```

There are a lot of ways that computers tend to get around this.

System Programming Jokes

`0x43 0x61 0x74 0xe0 0xf9 0xbf 0x5f 0xff 0x7f 0x00`

Warning: Authors are not responsible for any neuro-apoptosis caused by these "jokes." - Groaners are allowed.

Light bulb jokes

Q. How many system programmers does it take to change a lightbulb?

A. Just one but they keep changing it until it returns zero.

A. None they prefer an empty socket.

A. Well you start with one but actually it waits for a child to do all of the work.

Groaners

Why did the baby system programmer like their new colorful blankie? It was multithreaded.

Why are your programs so fine and soft? I only use 400-thread-count or higher programs.

Where do bad student shell processes go when they die? Forking Hell.

Why are C programmers so messy? They store everything in one big heap.

System Programmer (Definition)

A system programmer is...

Someone who knows `sleepsort` is a bad idea but still dreams of an excuse to use it.

Someone who never lets their code deadlock... but when it does, causes more problems than everyone else combined.

Someone who believes zombies are real.

Someone who doesn't trust their process to run correctly without testing with the same data, kernel, compiler, RAM, filesystem size,file system format, disk brand, core count, CPU load, weather, magnetic flux, orientation, pixie dust, horoscope sign, wall color, wall gloss and reflectance, motherboard, vibration, illumination, backup battery, time of day, temperature, humidity, lunar position, sun-moon, co-position...

A system program ...

Evolves until it can send email.

Evolves until it has the potential to create, connect and kill other programs and consume all possible CPU,memory,network,... resources on all possible devices but chooses not to. Today.