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

C Programming Language

If you want to teach systems, don't drum up the programmers, sort the issues, and make PRs. Instead, teach them to yearn for the vast and endless C.

Antoine de Saint-Exupéry (Kinda)

C is the de-facto programming language to do serious system serious programming. Why? Most kernels are written in largely in C. The Linux kernel [4] and the XNU kernel Inc. [1] of which Mac OS X is based off. The Windows Kernel uses C++, but doing system programming on that is much harder on windows that UNIX for beginner system programmers. Most of you have some experience with C++, but C is a different beast entirely. You don't have nice abstractions like classes and RAII to clean up memory. You are going to have to do that yourself. C gives you much more of an opportunity to shoot yourself in the foot but lets you do thinks at a much finer grain level.

History of C

C was developed by Dennis Ritchie and Ken Thompson at Bell Labs back in 1973 [5]. Back then, we had gems of programming languages like Fortran, ALGOL, and LISP. The goal of C was two fold. One, to target the most popular computers at the time liek the PDP-7. Two, try and remove some of the lower level constructs like managing registers, programming assembly for jumps and instead create a language that had the power to express programs procedurally (as opposed to mathematically like lisp) with more readable code all while still having the ability to interface with the operating system. It sounded like a tough feat. At first, it was only used internally at Bell Labs along with the UNIX operating system.

The first "real" standardization is with Brian Kerninghan and Dennis Ritchies book [3]. It is still widely regarded today as the only portable set of C instructions. The K&R book is known as the de-facto standard for learning C. There were different standards of C from ANSI to ISO after the Unix guides. The one that we will be mainly focusing on is the POSIX C library. Now to get the elephant out of the room, the Linux kernel is not entirely POSIX compliant. Mostly, it is because they didn't want to pay the fee for compliance

TODO:

Features

- Fast. There is nothing separating you and the system.

- Simple. C and its standard library pose a simple set of portable functions.
- Memory Management. C let's you manage your memory. This can also bite you if you have memory errors.
- It's Everywhere. Pretty much every computer that is not embedded has some way of interfacing with C. The standard library is also everywhere. C has stood the test of time as a popular language, and it doesn't look like it is going anywhere.

Crash course intro to C

The only way to start learning C is by starting with hello world. As per the original example that Kernighan and Ritchie proposed way back when, the hello world hasn't changed that much.

```c
#include <stdio.h>
int main(void) {
    printf("Hello World\n");
    return 0;
}
```

1. The `#include` directive takes the file `stdio.h` (which stands for **st**an**d**ard **i**nput and **o**utput) located somewhere in your operating system, copies the text, and substitutes it where the `#include` was.
2. The `int main(void)` is a function declaration. The first word `int` tells the compiler what the return type of the function is. The part before the parens (`main`) is the function name. In C, no two functions can have the same name in a single compiled program, shared libraries are a different touchy subject. Then, what comes after is the paramater list. When we give the parameter list for regular functions `(void)` that means that the compiler should error if the function is called with any arguments. For regular functions having a declaration like `void func()` means that you are allowed to call the function like `func(1, 2, 3)` because there is no delimiter [2]. In the case of `main`, it is a special function. There are many ways of declaring `main` but the ones that you will be familiar with are `int main(void)`, `int main()`, and `int main(int argc, char *argv[])`.
3. `printf("Hello World");` is what we call a function call. `printf` is defined as a part of `stdio.h`. The function has been compiled and lives somewhere else on our machine. All we need to do is include the header and call the function with the appropriate parameters (a string literal `"Hello World"`). If you don't have the newline, the buffer will not be flushed. It is by convention that buffered IO is not flushed until a newline. [2]
4. `return 0;`. `main` has to return an integer. By convention, `return 0` means success and anything else means failure [2].

```
$ gcc main.c -o main
$ ./main
Hello World
$
```

1. `gcc` is short for the GNU-Compiler-Collection which has a host of compilers ready for use. The compiler infers from the extension that you are trying to compile a .c file
2. `./main` tells your shell to execute the program in the current directory called main. The program then prints out hello world

Preprocessor

What is the preprocessor? Preprocessing is an operation that the compiler performs **before** actually compiling the program. It is a copy and paste command. Meaning that if I do the following.

```
#define MAX_LENGTH 10
char buffer[MAX_LENGTH]
// After
char buffer[10]
```

But there are side effects to the preprocessor. Does the following code print out.

```
#define min(a,b) ((a)<(b) ? (a) : (b))
int x = 4;
if(min(x++, 100)) printf("%d is six", x);
```

Macros are simple text substitution so the above example expands to `x++ < 100 ? x++ : 100` (parenthesis omitted for clarity). Now for this case, it will probably still print 6 but consider the edge case when `x = 99`. Also consider the edge case when operator precedence comes into play.

```
#define min(a,b) a<b ? a : b
int x = 99;
int r = 10 + min(99, 100); // r is 100!
```

Macros are simple text substitution so the above example expands to `10 + 99 < 100 ? 99 : 100`

C Preprocessor logical gotcha

```
#define ARRAY_LENGTH(A) (sizeof((A)) / sizeof((A)[0]))
int static_array[10]; // ARRAY_LENGTH(static_array) = 10
int* dynamic_array = malloc(10); // ARRAY_LENGTH(dynamic_array) = 2
    or 1
```

What is wrong with the macro? Well, it works if we have a static array like the first array because `sizeof` a static array returns the bytes that array takes up, and dividing it by the `sizeof(an_element)` would give you the number of entries. But if we use a pointer to a piece of memory, taking the sizeof the pointer and dividing it by the size of the first entry won't always give us the size of the array.

Language Facilities

Keywords

C has an assortment of keywords. Here are some constructs that you should know briefly as of C99.

1. `break` is a keyword that is used in case statements or looping statements. When used in a case statement, the program jumps to the end of the block.

   ```
   switch(1) {
     case 1: /* Goes to this switch */
       puts("1");
       break; /* Jumps to the end of the block */
     case 2: /* Ignores this program */
       puts("2");
       break;
   } /* Continues here */
   ```

   In the context of a loop, it breaks out of the inner-most loop. The loop can be either a `for`, `while`, or `do-while` construct

   ```
   while(1) {
     while(2) {
       break; /* Breaks out of while(2) */
     } /* Jumps here */
     break; /* Breaks out of while(1) */
   } /* Continues here */
   ```

2. `const` is a language level construct that tells the compiler that this data should not be modified. If one tries to change a const variable, the program will not even compile. `const` works a little differently when put before the type, the compiler flips the first type and const. Then the compiler uses a left associativity rule. Meaning that whatever is left of the pointer is constant. This is known as const-correctedness.

```
const int i = 0; // Same as "int const i = 0"
char *str = ...; // Mutable pointer to a mutable string
const char *const_str = ...; // Mutable pointer to a constant
    string
char const *const_str2 = ...; // Same as above
const char *const const_ptr_str = ...;
// Constant pointer to a constant string
```

But, it is important to know that this is a compiler imposed restriction only. There are ways of getting around this and the program will run fine with defined behavior. In systems programming, the only type of memory that you can't write to is system write-protected memory.

```
const int i = 0; // Same as "int const i = 0"
(*((int *)&i)) = 1; // i == 1 now
const char *ptr = "hi";
*ptr = '\0'; // Will cause a Segmentation Violation
```

3. `continue` is a control flow statement that exists only in loop constructions. Continue will skip the rest of the loop body and set the program counter back to the start of the loop before.

```
int i = 10;
while(i--) {
  if(1) continue; /* This gets triggered */
  *((int *)NULL) = 0;
} /* Then reaches the end of the while loop */
```

4. `do {} while();` is another loop constructs. These loops execute the body and then check the condition at the bottom of the loop. If the condition is zero, the loop body is not executed and the rest of the program is executed. Otherwise, the loop body is executed.

```
int i = 1;
do {
  printf("%d\n", i--);
} while (i > 10) /* Only executed once */
```

5. `enum`

6. `extern`

7. `float`

8. `for`

9. `goto`

10. `if` `else` `else-if` are control flow keywords. There are a few ways to use these

11. `inline`

12. `register`

13. `restrict`

14. `return` is a control flow operator that exits the current function. If the function is `void` then it simply exits the functions. Otherwise another parameter follows as the return value.

```
```

15. `signed` is a modifier which is rarely used, but it forces an type to be signed instead of unsigned. The reasont that this is so rarely used is because types are signed by default and need to have the `unsigned` modifier to make them unsigned but it may be useful in cases where you want the compiler to default a signed type like.

```c
int count_bits_and_sign(signed representation) {
  //...
}
```

16. `sizeof` is an operator that is evaluated at compile time, which evaluates to the number of bytes that the expression contains. Meaning that when the compiler infers the type the following code

changes.

```c
char a = 0;
printf("%zu", sizeof(a++));
```

```c
char a = 0;
printf("%zu", 1);
```

Which then the compiler is allowed to operate on further. A note that you must have a complete definition of the type at compile time or else you may get an odd error. Consider the following

```
// file.c
struct person;

printf("%zu", sizeof(person));

// file2.c

struct person {
  // Declarations
}
```

This code will not compile because sizeof is not able to compile `file.c` without knowing the full declaration of the `person` struct. That is typically why we either put the full declaration in a header file or we abstract the creation and the interaction away so that users cannot access the internals of our struct.

`sizeof(ary)`: `ary` is an array. Returns the number of bytes required for the entire array (5 chars + zero byte = 6 bytes) `sizeof(ptr)`: Same as sizeof(char *). Returns the number bytes required for a pointer (e.g. 4 or 8 for a 32 bit or 64-bit machine) `sizeof` is a special operator. Really it's something the compiler substitutes in before compiling the program because the size of all types is known at compile time. When you have `sizeof(char*)` that takes the size of a pointer on your machine (8 bytes for a 64-bit machine and 4 for a 32 bit and so on). When you try `sizeof(char[])`, the compiler looks at that and substitutes the number of bytes that the **entire** array contains because the total size of the array is known at compile time.

```
char str1[] = "will be 11";
char* str2 = "will be 8";
sizeof(str1) //11 because it is an array
sizeof(str2) //8 because it is a pointer
```

Be careful, using sizeof for the length of a string!

17. `static`

18. `struct`

19. `switch case default` Switches are essentially glorified jump statements. Meaning that you take either a byte or an integer and the control flow of the program jumps to that location.

```
switch(/* char or int */) {
  case INT1: puts("1");
  case INT2: puts("2");
  case INT3: puts("3");
}
```

If we give a value of 2 then

```
switch(2) {
  case 1: puts("1"); /* Doesn't run this */
  case 2: puts("2"); /* Runs this */
  case 3: puts("3"); /* Also runs this */
}
```

The break statement

20. `typedef` declares an alias for a type. Often used with structs to reduce the visual clutter of having to write 'struct' as part of the type.

```
typedef float real;
real gravity = 10;
// Also typedef gives us an abstraction over the underlying
    type used.
// In the future, we only need to change this typedef if we
// wanted our physics library to use doubles instead of floats.

typedef struct link link_t;
//With structs, include the keyword 'struct' as part of the
    original types
```

In this class, we regularly typedef functions. A typedef for a function can be this for example

```
typedef int (*comparator)(void*,void*);

int greater_than(void* a, void* b){
    return a > b;
}
comparator gt = greater_than;
```

This declares a function type comparator that accepts two `void*` params and returns an integer.

21. `union`

22. `unsigned`

23. `void` is a two folded keyword. When used in terms of function or parameter definition then it means that it returns no value or accepts no parameter specifically. The following declares a function that accepts no parameters and returns nothing.

```c
void foo(void);
```

The other use of void is when you are defining. A void * pointer is just a memory address. It is specified as an incomplete type meaning that you cannot dereference it but it can be promoted to any time to any other type. Pointer arithmetic with these pointer is undefined behavior.

```c
int *array = void_ptr; // No cast needed
```

24. volatile is a compiler keyword. This means that the compiler should not optimize its value out. Consider the following simple function.

```c
int flag = 1;
pass_flag(&flag);
while(flag) {
    // Do things unrelated to flag
}
```

The compiler may, since the internals of the while loop have nothing to do with the flag, optimize it to the following even though a function may alter the data.

```c
while(1) {
    // Do things unrelated to flag
}
```

If you put the

volatile keyword then it forces the compiler to keep the variable in and perform that check. This is particularly useful for cases where you are doing multi-process or multi-threading programs so that we can

25. while represents the traditional while loop. There is a condition at the top of the loop. While that condition evaluates to a non-zero value, the loop body will be run.

C data types

1. char Represents exactly one byte of data. The number of bits in a byte might vary. unsigned char and signed char mean the exact same thing. This must be aligned on a boundary (meaning you cannot use bits in between two addresses). The rest of the types will assume 8 bits in a byte.

2. short (short int) must be at least two bytes. This is aligned on a two byte boundary, meaning that the address must be divisble by two.

3. int must be at least two bytes. Again aligned to a two byte boundary. Page 34. On most machines this will be 4 bytes.

4. `long (long int)` must be at least four bytes, which are aligned to a four byte boundary. On some machines this can be 8 bytes.

5. `long long` must be at least eight bytes, aligned to an eight byte boundary.

6. `float` represents an IEEE-754 single percision floating point number tightly specified by IEEE. This will be four bytes aligned to a four byte boundary on most machines.

7. `double` represents an IEEE-754 double percision floating point number specified by the same standard, which is aligned to the nearest eight byte boundary.

Operators

Common C Functions

To find more information about any functions, use the man pages. Note the man pages are organized into sections. Section 2 are System calls. Section 3 are C libraries. On the web, Google `man 7 open`. In the shell, `man -S2 open` or `man -S3 printf`

Input/Output

`printf` is the function with which most people are familiar. The first parameter is a format string that includes placeholders for the data to be printed. Common format specifiers are %s treat the argument as a c string pointer, keep printing all characters until the NULL-character is reached; %d print the argument as an integer; %p print the argument as a memory address.

A simple example is shown below:

```c
char *name = ... ; int score = ...;
printf("Hello %s, your result is %d\n", name, score);
printf("Debug: The string and int are stored at: %p and %p\n", name,
    &score );
// name already is a char pointer and points to the start of the
    array.
// We need "&" to get the address of the int variable
```

By default, for performance, `printf` does not actually write anything out (by calling write) until its buffer is full or a newline is printed.

How else can I print strings and single characters?

Use `puts( name )` and `putchar( c )` where name is a pointer to a C string and c is just a `char`

How do I print to other file streams?

Use `fprintf( _file_ , "Hello %s, score: %d", name, score);` Where _file_ is either predefined 'stdout' 'stderr' or a FILE pointer that was returned by `fopen` or `fdopen`

Can I use file descriptors?

Yes! Just use `dprintf(int fd, char* format_string, ...);` Just remember the stream may be buffered, so you will need to assure that the data is written to the file descriptor.

How do I print data into a C string?

Use `sprintf` or better `snprintf`.

```
char result[200];
int len = snprintf(result, sizeof(result), "%s:%d", name, score);
```

snprintf returns the number of characters written excluding the terminating byte. In the above example, this would be a maximum of 199.

The following code is vulnerable to buffer overflow. It assumes or trusts that the input line will be no more than 10 characters, including the terminating byte.

```
char buf[10];
gets(buf); // Remember the array name means the first byte of the
    array
```

Should I Use Gets?

`gets` is deprecated in C99 standard and has been removed from the latest C standard (C11). Programs should use `fgets` or `getline` instead.

Where each has the following structure respectively:

```
char *fgets (char *str, int num, FILE *stream);

ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Here's a simple, safe way to read a single line. Lines longer than 9 characters will be truncated:

```
char buffer[10];
char *result = fgets(buffer, sizeof(buffer), stdin);
```

The result is NULL if there was an error or the end of the file is reached. Note, unlike `gets`, `fgets` copies the newline into the buffer, which you may want to discard-

```
if (!result) {return; /* no data - don't read the buffer contents */}

int i = strlen(buffer) - 1;
if (buffer[i] == '\n')
    buffer[i] = '\0';
```

One of the advantages of `getline` is that will automatically (re-) allocate a buffer on the heap of sufficient size.

```
// ssize_t getline(char **lineptr, size_t *n, FILE *stream);

 /* set buffer and size to 0; they will be changed by getline */
char *buffer = NULL;
size_t size = 0;

ssize_t chars = getline(&buffer, &size, stdin);

// Discard newline character if it is present,
if (chars > 0 && buffer[chars-1] == '\n')
    buffer[chars-1] = '\0';

// Read another line.
// The existing buffer will be re-used, or, if necessary,
// It will be `free''d and a new larger buffer will `malloc''d
chars = getline(&buffer, &size, stdin);

// Later... don't forget to free the buffer!
free(buffer);
```

Parsing

string.h

Why are `memcpy` and `memmove` both in `<string.h>`? Because strings are essentially raw memory with a null byte at the end of them! `void *memcpy(void *dest, const void *src, size_t n)` moves `n` bytes starting at `src` to `dest`. **Be careful**, there is undefined behavior when the memory regions overlap. This is one of the classic works on my machine examples because many times valgrind won't be able to pick it up because it will look like it works on your machine. When the autograder hits, fail. Consider the safer version which is.

`void *memmove(void *dest, const void *src, size_t n)` does the same thing as above, but if the memory regions overlap then it is guaranteed that all the bytes will get copied over correctly.

stdlib.h

To allocate space on the heap, use malloc. There's also realloc and calloc. Typically used with sizeof. e.g. enough space to hold 10 integers

```
int *space = malloc(sizeof(int) * 10);
```

Conventions/Errno

System Calls

What is a system call?

The interplay with library functions

Does printf call write or does write call printf?

printf calls write. printf includes an internal buffer so, to increase performance printf may not call write everytime you call printf. printf is a C library function. write is a system call and as we know system calls are expensive. On the other hand, printf uses a buffer which suits our needs better at that point

C Memory Model

Structs

Struct packing

Structs may require something called padding (tutorial). **We do not expect you to pack structs in this course, just know that it is there** This is because in the early days (and even now) when you have to an address from memory you have to do it in 32bit or 64bit blocks. This also meant that you could only request addresses that were multiples of that. Meaning that

```
struct picture{
    int height;
    pixel** data;
    int width;
    char* enconding;
}
// You think picture looks like this. One box is four bytes
| h   data    w   encod |
|___ ___ ___ ___ ___ ___|
|___|___ ___|___|___ ___|
```

Would conceptually look like this

```
struct picture{
    int height;
    char slop1[4];
    pixel** data;
    int width;
    char slop2[4];
    char* enconding;
}

| h   ?   data    w   ?   encod |
|___ ___ ___ ___ ___ ___ ___ ___|
|___|___|___ ___|___|___|___ ___|
```

This is on a 64-bit system. This is not always the case because sometimes your processor supports unaligned accesses. What does this mean? Well there are two options you can set an attribute

```
struct __attribute__((packed, aligned(4))) picture{
    int height;
    pixel** data;
    int width;
    char* enconding;
}
// Will look like this
| h   data    w   encod |
|___ ___ ___ ___ ___ ___|
|___|___ ___|___|___ ___|
```

But now, every time I want to access `data` or `encoding`, I have to do two memory accesses. The other thing you can do is reorder the struct, although this is not always possible

```
struct picture{
    int height;
    int width;
    pixel** data;
    char* enconding;
}
// You think picture looks like this
| h   w    data   encod |
|___ ___ ___ ___ ___ ___|
|___|___|___ ___|___ ___|
```

C Null-Terminated Strings

Strings in C are represented as characters in memory. The end of the string includes a NULL (0) byte [2]. So "ABC" requires four(4) bytes `[A,B,C,\0]`. The only way to find out the length of a C string is to keep reading memory until you find the NULL byte. C characters are always exactly one byte each.

When you write a string literal `"ABC"` in an expression the string literal evaluates to a char pointer (`char *`), which points to the first byte/char of the string. This means `ptr` in the example below will hold the memory address of the first character in the string.

String constants are constant

```
char array[] = "Hi!"; // array contains a mutable copy
strcpy(array, "OK");

char *ptr = "Can't change me"; // ptr points to some immutable memory
strcpy(ptr, "Will not work");
```

String literals are character arrays stored in the code segment of the program, which is immutable. Two string literals may share the same space in memory. An example follows:

```
char *str1 = "Brandon Chong is the best TA";
char *str2 = "Brandon Chong is the best TA";
```

The strings pointed to by `str1` and `str2` may actually reside in the same location in memory.

Char arrays, however, contain the literal value which has been copied from the code segment into either the stack or static memory. These following char arrays do not reside in the same place in memory.

```
char arr1[] = "Brandon Chong didn't write this";
char arr2[] = "Brandon Chong didn't write this";
```

```
char *ptr = "ABC"
```

Some common ways to initialize a string include:

```
char *str = "ABC";
char str[] = "ABC";
char str[]={'A','B','C','\0'};
```

```
char ary[] = "Hello";
char *ptr = "Hello";
```

Example

The array name points to the first byte of the array. Both `ary` and `ptr` can be printed out:

```
char ary[] = "Hello";
char *ptr = "Hello";
// Print out address and contents
printf("%p : %s\n", ary, ary);
printf("%p : %s\n", ptr, ptr);
```

The array is mutable, so we can change its contents. Be careful not to write bytes beyond the end of the array though. Fortunately, "World" is no longer than "Hello"

In this case, the char pointer `ptr` points to some read-only memory (where the statically allocated string literal is stored), so we cannot change those contents.

```
strcpy(ary, "World"); // OK
strcpy(ptr, "World"); // NOT OK - Segmentation fault (crashes)
```

We can, however, unlike the array, we change `ptr` to point to another piece of memory,

```
ptr = "World"; // OK!
ptr = ary; // OK!
ary = (..anything..) ; // WONT COMPILE
// ary is doomed to always refer to the original array.
printf("%p : %s\n", ptr, ptr);
strcpy(ptr, "World"); // OK because now ptr is pointing to mutable
    memory (the array)
```

What to take away from this is that pointers * can point to any type of memory while C arrays [] can only point to memory on the stack. In a more common case, pointers will point to heap memory in which case the memory referred to by the pointer CAN be modified.

Errors

Double Frees

A double free error is when you accidentally attempt to free the same allocation twice.

```
int *p = malloc(sizeof(int));
free(p);

*p = 123; // Oops! - Dangling pointer! Writing to memory we don't own
    anymore

free(p); // Oops! - Double free!
```

The fix is first to write correct programs! Secondly, it's good programming hygiene to reset pointers once the memory has been freed. This ensures the pointer can't be used incorrectly without the program crashing.

Fix:

```
p = NULL; // Now you can't use this pointer by mistake
```

Buffer Overflows

TODO

Famous example: Heart Bleed (performed a memcpy into a buffer that was of insufficient size). Simple example: implement a strcpy and forget to add one to strlen, when determining the size of the memory required.

Returning pointers to automatic variables

```
int *f() {
    int result = 42;
    static int imok;
    return &imok; // OK - static variables are not on the stack
    return &result; // Not OK
}
```

Automatic variables are bound to stack memory only for the lifetime of the function. After the function returns it is an error to continue to use the memory. ## Insufficient memory allocation

```
struct User {
   char name[100];
};
typedef struct User user_t;

user_t *user = (user_t *) malloc(sizeof(user));
```

In the above example, we needed to allocate enough bytes for the struct. Instead, we allocated enough bytes to hold a pointer. Once we start using the user pointer we will corrupt memory. The correct code is shown below.

```
struct User {
   char name[100];
};
typedef struct User user_t;

user_t * user = (user_t *) malloc(sizeof(user_t));
```

Pointers

Pointer Basics

Declaring a Pointer

A pointer refers to a memory address. The type of the pointer is useful - it tells the compiler how many bytes need to be read/written. You can declare a pointer as follows.

```
int *ptr1;
char *ptr2;
```

Due to C's grammar, an `int*` or any pointer is not actually its own type. You have to precede each pointer variable with an asterisk. As a common gotcha, the following

```
int* ptr3, ptr4;
```

Will only declare `*ptr3` as a pointer. `ptr4` will actually be a regular int variable. To fix this declaration, keep the `*` preceding to the pointer

```
int *ptr3, *ptr4;
```

Keep this in mind for structs as well. If one does not typedef them, then the pointer goes after the type.

```
struct person *ptr3;
```

Reading/Writing with pointers

Let's say that we declare a pointer `int *ptr`. For the sake of discussion, let's say that `ptr` points to memory address `0x1000`. If we want to write to a pointer, we can dereference and assign `*ptr`.

```
*ptr = 0; // Writes some memory.
```

What C will do is take the type of the pointer which is an `int` and writes `sizeof(int)` bytes from the start of the pointer, meaning that bytes `0x1000`, `0x1001`, `0x1002`, `0x1003` will all be zero. The number of bytes written depends on the pointer type. It is the same for all primitive types but structs are a little different.

Pointer Arithmetic

You can add an integer to a pointer. However, the pointer type is used to determine how much to increment the pointer. For char pointers this is trivial because characters are always one byte:

```
char *ptr = "Hello"; // ptr holds the memory location of 'H'
ptr += 2; //ptr now points to the first'l'
```

If an int is 4 bytes then ptr+1 points to 4 bytes after whatever ptr is pointing at.

```
char *ptr = "ABCDEFGH";
int *bna = (int *) ptr;
bna +=1; // Would cause iterate by one integer space (i.e 4 bytes on
    some systems)
ptr = (char *) bna;
printf("%s", ptr);
/* Notice how only 'EFGH' is printed. Why is that? Well as mentioned
    above, when performing 'bna+=1' we are increasing the **integer**
    pointer by 1, (translates to 4 bytes on most systems) which is
    equivalent to 4 characters (each character is only 1 byte)*/
return 0;
```

Because pointer arithmetic in C is always automatically scaled by the size of the type that is pointed to, you can't perform pointer arithmetic on void pointers.

You can think of pointer arithmetic in C as essentially doing the following

If I want to do

```
int *ptr1 = ...;
int *offset = ptr1 + 4;
```

Think

```
int *ptr1 = ...;
char *temp_ptr1 = (char*) ptr1;
int *offset = (int*)(temp_ptr1 + sizeof(int)*4);
```

To get the value. **Every time you do pointer arithmetic, take a deep breath and make sure that you are shifting over the number of bytes you think you are shifting over.**

What is a void pointer?

A pointer without a type (very similar to a void variable). Void pointers are used when either a datatype you're dealing with is unknown or when you're interfacing C code with other programming languages. You can think of this as a raw pointer, or just a memory address. You cannot directly read or write to it because the void type does not have a size. For Example

```
void *give_me_space = malloc(10);
char *string = give_me_space;
```

This does not require a cast because C automatically promotes `void*` to its appropriate type. **Note:**

`gcc` and `clang` are not total ISO-C compliant, meaning that they will let you do arithmetic on a void pointer. They will treat it as a `char *` pointer. Do not do this because it may not work with all compilers!

Shell

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.

Common Utilities

1. `cat`
2. `diff`
3. `grep`
4. `ls`
5. `cd`
6. `man`
7. `make`

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

Tips and tricks

Common Bugs

```
void mystrcpy(char*dest, char* src) {
  // void means no return value
  while( *src ) {dest = src; src ++; dest++; }
}
```

In the above code it simply changes the dest pointer to point to source string. Also the nuls bytes are not copied. Here's a better version -

```
  while( *src ) {*dest = *src; src ++; dest++; }
  *dest = *src;
```

Note it's also usual to see the following kind of implementation, which does everything inside the expression test, including copying the nul byte.

```
  while( (*dest++ = *src++ )) {};
```

Topics

- C Strings representation
- C Strings as pointers
- char p[]vs char* p
- Simple C string functions (strcmp, strcat, strcpy)
- sizeof char
- sizeof x vs x*

- Heap memory lifetime
- Calls to heap allocation
- Deferencing pointers
- Address-of operator
- Pointer arithmetic
- String duplication
- String truncation
- double-free error
- String literals
- Print formatting.
- memory out of bounds errors
- static memory
- fileio POSIX vs. C library
- C io fprintf and printf
- POSIX file IO (read, write, open)
- Buffering of stdout

Questions/Exercises

- What does the following print out?

```
int main(){
fprintf(stderr, "Hello ");
fprintf(stdout, "It's a small ");
fprintf(stderr, "World\n");
fprintf(stdout, "place\n");
return 0;
}
```

- What are the differences between the following two declarations? What does `sizeof` return for one of them?

```
char str1[] = "bhuvan";
char *str2 = "another one";
```

- What is a string in c?
- Code up a simple `my_strcmp`. How about `my_strcat`, `my_strcpy`, or `my_strdup`? Bonus: Code the functions while only going through the strings *once*.
- What should the following usually return?

```
int *ptr;
sizeof(ptr);
sizeof(*ptr);
```

- What is `malloc`? How is it different than `calloc`. Once memory is `malloc`ed how can I use `realloc`?
- What is the & operator? How about `*`?
- Pointer Arithmetic. Assume the following addresses. What are the following shifts?

```
char** ptr = malloc(10); //0x100
ptr[0] = malloc(20); //0x200
ptr[1] = malloc(20); //0x300
```

  - `ptr + 2`
  - `ptr + 4`
  - `ptr[0] + 4`
  - `ptr[1] + 2000`
  - `*((int)(ptr + 1)) + 3`
- How do we prevent double free errors?
- What is the printf specifier to print a string, `int`, or `char`?
- Is the following code valid? If so, why? Where is `output` located?

```
char *foo(int var){
static char output[20];
snprintf(output, 20, "%d", var);
return output;
}
```

- Write a function that accepts a string and opens that file prints out the file 40 bytes at a time but every other print reverses the string (try using POSIX API for this).

- What are some differences between the POSIX filedescriptor model and C's `FILE*` (ie what function calls are used and which is buffered)? Does POSIX use C's `FILE*` internally or vice versa?

Bibliography

[1] Apple Inc. Xnu kernel. `https://github.com/apple/darwin-xnu`, 2017.

[2] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988. ISBN 9780131103627. URL `https://books.google.com/books?id=161QAAAAMAAJ`.

[3] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988. ISBN 9780131103627. URL `https://books.google.com/books?id=161QAAAAMAAJ`.

[4] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. ISBN 0672329468, 9780672329463.

[5] Dennis M. Ritchie. The development of the c language. *SIGPLAN Not.*, 28(3):201–208, March 1993. ISSN 0362-1340. doi: 10.1145/155360.155580. URL `http://doi.acm.org/10.1145/155360.155580`.

Chapter 3

Deadlock

No, you can't always get what you want
You can't always get what you want
You can't always get what you want
But if you try sometime you find
You get what you need

The philosphers Jagger & Richards

Deadlock is defined as when the system cannot make and forward progress. In a lot of systems, Deadlock is just avoided by ignore the entire concept [5, P.237]. Have you heard about turn it on and off again? That is partly because of this. For products where the stakes are low when you deadlock (User Operating Systems, Phones), it may be more efficient not not keep track of all of the allocations in order to keep deadlock from happening. But in the cases where "failure is not an option" - Apollo 13, you need a system that tracks deadlock or better yet prevents it entirely. Take the Apollo 13 module. It may have not failed because of deadlock, but probably wouldn't be good to restart the system on liftoff.

Mission critical operating systems need this guarentee formally because playing the odds with people's lives isn't a good idea. Okay so how do we do this? We model the problem. Even though it is a common statistical phrase that all models are wrong, the more accurate the model is to the system that we are working with the better chance that it'll work better.

Resource Allocation Graphs

One such way is modeling the system with a resource allocationg graph. A resource allocation graph tracks which resource is held by which process and which process is waiting for a resource of a particular type. It is very powerful and simple tool to illustrate how interacting processes can deadlock. If a process is *using* a resource, an arrow is drawn from the resource node to the process node. If a process is *requesting* a resource, an arrow is drawn from the process node to the resource node.If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will deadlock. For example, if process 1 holds resource A, process 2 holds resource B and process 1 is waiting for B and process 2 is waiting for A, then process 1 and 2 process will be deadlocked.

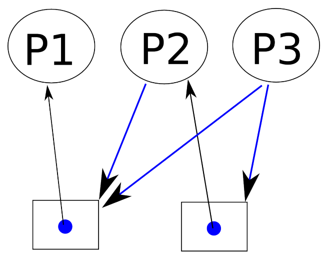

Figure 3.1: Resource allocation graph

Consider the following resource allocation graph. Assume that the processes ask for exclusive access to the file. If you have a bunch of processes running and they request resources and the operating system ends up in this state, you deadlock! You may not see this because the operating system may **preempt** some processes breaking the cycle but there is still a change that your three lonely processes could deadlock. You can also make these kind of graphs with `make` and rule dependencies with our parmake MP for example.

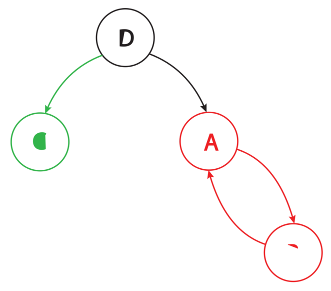

Figure 3.2: Colorful Deadlock

Coffman conditions

There are four *necessary* and *sufficient* conditions for deadlock – meaning if these conditions hold then there is a non-zero probability that the system will deadlock at any given iteration. These are known as the Coffman conditions [1].

- Mutual Exclusion: no two processes can obtain a resource at the same time.
- Circular Wait: there exists a cycle in the Resource Allocation Graph, or there exists a set of processes {P1,P2,...} such that P1 is waiting for resources held by P2, which is waiting for P3,..., which is waiting for P1.
- Hold and Wait: a process once obtaining a resource does not let go.
- No pre-emption: nothing can force the process to give up a resource.

Proof: Deadlock can happen if and only if the four coffman conditions are satisified.
$\rightarrow$ If the system is deadlocked, the four coffman conditions are apparent.

- For the purposes of contradiction, assume that there is no cicular wait. If not then that means the resource wait graph is acyclic, meaning that there is at least one one process that is not waiting on any resources or it can grab a resource. Since the system can move forward, the system is not deadlocked.
- For the purposes of contradiction, assume that there is no mutual exclusion. If not, that means that no process is waiting on any other process for a resource. This breaks circular wait and the previous argument proves correctness.
- For the purposes of contradiction, assume that processes don't hold and wait but our system still deadlocks. Since we have circular wait from the first condition at least one process must be waiting on another process. If that and processes don't hold and wait, that means one process must let go of a resource. Since the system has moved forward, it cannot be deadlocked.
- For the purposes of contradiction, assume that we have preemption, but the system cannot be un-deadlocked. Have one process, or create one process, that recognizes the circular wait that must be apprent from above and break on the of the links. By the first branch, we must not have deadlock.

$\leftarrow$ If the four conditions are apparent, the system is deadlocked. We will prove that if the system is not deadlocked, the four conditions are not apparent. Though this proof is not formal, let us build a system with the three requirements not including circular wait. Let assume that there is a set of processes $P = \{p_1, p_2, ..., p_n\}$ and there is a set of resources $R = \{r_1, r_2, ..., r_m\}$. For simplicity, a process can only request one resource at a time but the proof can be generalized to multiple. Let assume that the system is at different states at different times t. Let us assume that the state of the system is a tuple (h_t, w_t) where there are two function $h_t : R \to P \cup \{\text{unassigned}\}$ that maps resources to the processes that own them (this is a function, meaning that we have mutual exclusion) and or unassigned and $w_t : P \to R \cup \{\text{satisfied}\}$ that maps the requests that each process makes to a resource or if the process is satisfied. Let $L_t \subseteq P \times R$ be a set of list of requests that a process uses to release a resource at any given time. The evolution of the system is at each step at every time.

- Release all resources in L_t the a process requests to resource.
- Find a process that is requesting a resource
- If that resource is available give it to that process, generating a new (h_t, w_t) and exit the current iteration.
- Else find another process and try the above if.

If all processes have been surveyed and none updates the system, consider it deadlocked. More formally, this system is deadlocked means if $\exists t_0, \forall t \geq t_0, \forall p \in P, w_t(p) \neq \text{satisfied}$ and $\exists q, q \neq p \to h_t(w_t(p)) = q$ (which is what we need to prove).

Proof: These conditions imply deadlock. Deadlock for a system is defined as no work can be done now or later. Work can be done if a process is satisfied, or we can release a resource to a process. No process is satisfied by definition. Since the system can't preempt and release resources (by no preemption), the processes have to do it themselves. If the processes has a resource, it will not let it go until after it is satisfied by hold and wait. All resources requested by the processes are owned by other processes meaning that no process will let any resource go. Since we have shown no processes will give up a resource and no process is satisfied, the system is in deadlock. TODO: Formalize Subproof □

The last condition to address is circular wait. Circular wait means that there exists $\forall p \in P, w_t(p) \neq \text{satisfied}$ and $\exists q, q \neq p \to h_t(w_t(p)) = q$. Which is what we needed to show. □

If you break any of them, you cannot have deadlock! Consider the scenario where two students need to write both pen and paper and there is only one of each. Breaking mutual exclusion means that the students share the pen and paper. Breaking circular wait could be that the students agree to grab the pen then the paper. As a proof by contradiction, say that deadlock occurs under the rule and the conditions. Without loss of generality, that means a student would have to be waiting on a pen while holding the paper and the other waiting on a pen and holding the paper. We have contradicted ourselves because one student grabbed the paper without grabbing the pen, so deadlock must not be able to occur.

Breaking hold and wait could be that the students try to get the pen and then the paper and if a student fails to grab the paper then they release the pen. This introduces a new problem called *livelock* which will be discussed latter. Breaking preemption means that if the two students are in deadlock the teacher can come in and break up the deadlock by giving one of the students one the held on items or tell both students to put the items down.

Livelock relates to deadlock but it is not exactly deadlock. Consider the breaking hold and wait solution as above. Though deadlock is avoided, if we pick up the same device (a pen or the paper) again and again in the exact same pattern, neither of us will get any writing done. More generally, livelock happens when the process looks like it is executing but no meaningful work is done. Livelock is generally harder to detect because the processes generally look like they are working to the outside operating system wheras in deadlock the operating system generally knows when two processes are waiting on a system wide resource. Another problem is that there are nessisary conditions for livelock (i.e. deadlock does not occur) but not sufficient conditions – meaning there is no set of rules where livelock has to occur. You must formally prove in a system by what is known as an invariant. One has to enumerate each of the steps of a system and if each of the steps eventually (after some finite number of steps) leads to forward progress, the system is not livelocked. There are even better systems that prove bounded waits; a system can only be livelocked for at most n cycles which may be important for something like stock exchanges.

Approaches to solving deadlock

Ignoring deadlock is the most obvious approach that started the chapter out detailing. Quite humorously, the name for this approach is called the ostrich algorithm. Though there is no apparent source, the idea for the algorithm comes from the concept of an ostrich sticking its head in the sand. When the operating system detects deadlock, it does nothing out of the ordinary and hopes that the deadlock goes away. Now this is a slight misnomer because the operating system doesn't do anything *abnormal* – it is not like an operating system deadlocks every few minutes because it runs 100 processes all requesting shared libraries. An operating system still preempts processes when stopping them for context switches. The operating system has the ability to interrupt any system call, potentially breaking a deadlock scenario. The OS also makes some files read-only thus making the resource shareable. What the algorithm refers to is that if there is an adversary that specifically crafts a program – or equivalently a user who poorly writes a program – that deadlock could not be caught by the operating system. For everyday life, this tends to be fine. When it is not we can turn to the following method.

Deadlock detection allows the system to enter a deadlocked state. After entering, the system uses the information that it has to break deadlock. As an example, consider multiple processes accessing files. The operating system is able to keep track of all of the files/resources through file descriptors at some level either abstracted through an API or directly. If the operating system detects a directed cycle in the operating system file descriptor table it may break one process' hold through scheduling for example and let the system proceed. Why this is a popular choice in this realm is that there is no way of knowing which resources a program will select without running the program. This is an extension of Rice's theorem [4] that says that we cannot know any semantic feature without running the program (semantic meaning like what files it tries to open). So theoretically, it is sound. The problem then gets introduced that we could reach a livelock scenario if we preempt a set of resources again and again. The way around this is mostly probabilistic. The operating system chooses a random resource to break hold and wait. Now even though a user can craft a program where breaking hold and wait on each resource will result in a livelock, this doesn't happen as often on machines that run programs in practice or the livelock that does happen happens for a couple of cycles. These kind of systems are good for products that need to maintain a non-deadlocked state but can tolerate a small chance of livelock for a short period of time. The following proof **is not required for our 241 related puproses but is included for concreteness**.

Proof: That livelock terminates with high probability. This meaning that for any probability level l we can produce a number of iterations n that the probability that the system is not livelocked after that state is at least l.

Let $L = \{p_1, p_2, p_3, ...\}$ be an infinite set that has probabilities if we choose a resource that causes livelock in the ith iteration of the livelocked by breaking a random resource. Also let the set have the property that $\forall i > 0, \exists j > i, p_j < 1$ Meaning that for all elements after any given element that there is atleast one element int he future that has a probability of breaking deadlock. The probability that the system is not livelocked after n iterations is

$$P(n) = 1 - \prod_{i=1}^{n} p_i$$

Consider the new set $E = \{s_1, s_2, s_3, ...\}$ obtained by selecting all non-one elements. The set must be infinite by our sets property. It is easy to show that

$$\prod_{i=1}^{n} p_i = \prod_{i=1}^{n} s_i$$

And since

$$P(n) = 1 - \prod_{i=1}^{n} s_i$$

is a strictly monotonically decreasing function, it must pass the threshold for the probability level at some point. More rigorously

$$\begin{aligned}
Y &= \sup E \\
\prod_{i=1}^{n} s_i &< Y^n \\
1 - \prod_{i=1}^{n} s_i &> 1 - Y^n \\
P(n) &> 1 - Y^n \\
P(n) &> l \\
1 - Y^n &> l \\
\log_Y(1-l) &< n
\end{aligned}$$

We have found the number of steps in the set E and since our set has the property that gaps between non-one elements is finite, we can reconstruct the number of iterations by picking an n that satifies the E criteria and just adding the finite gaps together, which is what we needed to show. □

Deadlock prevention is making sure that deadlock cannot happen, meaning that you break a Coffman condition. This works the best inside a single program and the software engineer making the choice to break a certain coffman condition. Consider the Banker's Algorithm [3]. It is another algorithm for deadlock avoidance. The whole implementation is outside the scope of this class, just know that there are more generalized algorithms for operating systems.

Aside The banker algorithm is actually not too complicated. We can start out with the single resource solution. Let's say that I'm a banker. As a banker I have a finite amount of money. As having a finite amount of money, I want to make loans and eventually get my money back. Let's say that we have a set of n people where each of them have a set amount or a limit a_i (i being the ith process) that they need to obtain before they can do any work. I keep track in my book how much I've given to each person l_i, and I have some amount of principle p at any given time. For people to request money, they do the following. Consider the state of the system $(A = \{a_1, a_2, ...\}, L = \{l_1, l_2, ...\}, p)$. An assumption of this system is that we have $p > \inf a_i$, or we have enough money to satisfy one person. Also, each person will work for a finite period of time and give back our money.

- A person j requests m from me
  - if $m < p$, they are denied.
  - if $m + l_j > a_i$ they are denied
  - Pretend we are in a new state $(A = \{..., a_j, ...\}, L = \{.., l_j + m, ...\}, p - m)$ where the process is granted the resource.
- if now person j is either satisfied $(l_j == a_j)$ or $\exists i, a_j - l_j < p$. In other words we have enough money to satisfy one other person. If either, consider the transaction safe and give them the money.

Why does this work? Well at the start we are in a safe state – defined by we have enough money to satisfy at least one person. Each of these "loans" results in a safe state. If we have exhausted our reserve, one person is working and will give us money greater than or equal to our previous "loan", thus putting us in a safe state again. Since we always have the ability to make one additional move the system can never deadlock. Now, there is no guarentee that the system won't livelock. If the process we hope to request something never does, no work will be done – but not due to deadlock. This analogy expands to higher orders of magnitude but requires that either a process can do its work entirely or there exists a process whose combination of resources can be satisfied, which makes the algorithm a little more tricky (an additional for loop) but nothing too bad. There are a fair bit of downsides to this

- The program first needs to know how much of each resource a process needs. A lot of times that is impossible or the process requests the wrong amount because the programmer didn't forsee it.
- The system could livelock.
- We know in most systems that resources are generally not homogenous. Of course there are things like pipes and sockets but for the most part there is only 1 of a particular file. This could mean that the runtime of the algorithm could be slow for systems with millions of resources.

- Also, this can't keep track of resources that come and go. A process may delete a resource as a side effect or create a resource. The algorithm assumes a static allocation and that each process performs a non-destructive operation.

Dining Philosophers

The Dining Philosophers problem is a classic synchronization problem. Imagine I invite *n* (let's say 5) philosophers to a meal. We will sit them at a table with 5 chopsticks, one between each philosopher. A philosopher alternates between wanting to eat or think. To eat the philosopher must pick up the two chopsticks either side of their position. The original problem required each philosopher to have two forks, but one can eat with a single fork so we rule this out. However these chopsticks are shared with his neighbor.

Is it possible to design an efficient solution such that all philosophers get to eat? Or, will some philosophers starve, never obtaining a second chopstick? Or will all of them deadlock? For example, imagine each guest picks up the chopstick on their left and then waits for the chopstick on their right to be free. Oops - our philosophers have deadlocked! Each of the philosophers are essentially the same, meaning that each philosopher has the same instruction set based on the other philosopher ie you can't tell every even philosopher to do one thing and every odd philosopher to do another thing.

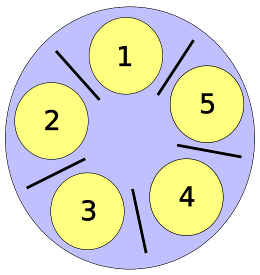

Figure 3.3: Dining Philosophers

Failed Solutions

```
void* philosopher(void* forks){
     info phil_info = forks;
     pthread_mutex_t* left_fork = phil_info->left_fork;
     pthread_mutex_t* right_fork = phil_info->right_fork;
     while(phil_info->simulation){
          pthread_mutex_lock(left_fork);
          pthread_mutex_lock(right_fork);
          eat(left_fork, right_fork);
          pthread_mutex_unlock(left_fork);
          pthread_mutex_unlock(right_fork);
     }
}
```

This looks good but. What if everyone picks up their left fork and is waiting on their right fork? We have deadlocked the program. It is important to note that deadlock doesn't happen all the time and the probability that this solution deadlocks goes down as the number of philosophers goes up. What is really important to note is that eventually that this solution will deadlock, letting threads starve which is bad. So now you are thinking about breaking one of the coffman conditions. Let's break Hold and Wait!

```
void* philosopher(void* forks){
     info phil_info = forks;
     pthread_mutex_t* left_fork = phil_info->left_fork;
     pthread_mutex_t* right_fork = phil_info->right_fork;
     while(phil_info->simulation){
          pthread_mutex_lock(left_fork);
          pthread_mutex_lock(right_fork);
          eat(left_fork, right_fork);
          pthread_mutex_unlock(left_fork);
          pthread_mutex_unlock(right_fork);
     }
}
```

Now our philosopher picks up the left fork and tries to grab the right. If it's available, they eat. If it's not available, they put the left fork down and try again. No deadlock! But, there is a problem. What if all the philosophers pick up their left at the same time, try to grab their right, put their left down, pick up their left, try to grab their right.... We have now livelocked our solution! Our poor philosopher are still starving, so let's give them some proper solutions.

Viable Solutions

The naive arbitrator solution is have one arbitrator (a mutex for example). Have each of the philosopher ask the arbitrator for permission to eat (i.e. trylock the mutex). This solution allows one philosopher to eat at a time. When they are done, another philosopher can ask for permission to eat. This prevents deadlock because there is no circular wait! No philosopher has to wait on any other philosopher. The advanced arbitrator solution is to implement a class that determines if the philosopher's forks are in the arbitrator's possession. If they are, they give them to the philosopher, let him eat, and take the forks back. This has the added bonus of being able to have multiple philosopher eat at the same time.

There are a lot of problems with these solutions. One is that they are slow and have a single point of failure or the arbitrator. Assuming that all the philosophers are good-willed, the arbitrator needs to be fair and be able to determine if a transaction would cause deadlock in the multi-arbitrator case. Further more in practical systems, the arbitrator tends to give forks to the same processes because of scheduling or pseudorandomness. Another important thing to note is that this prevents deadlock for the entire system. But in our model of dining philosophers, the philosopher has to release the lock themselves. Then you can consider the case of the malicious philosopher (let's say Decartes because of his Evil Demons) could hold on to the arbitrator forever. He would make forward progress and the system would make forward progress but there is no way of ensuring that each process makes forward progress without assuming something about the processes or having true preemption – meaning that a higher authority (let's say Steve Jobs) tells them to stop eating forcibly.

TODO: Prove arbitrator's doesn't deadlock

Leaving the Table (Stallings' Solution)

Why does the first solution deadlock? Well there are n philosophers and n chopsticks. What if there is only 1 philsopher at the table? Can we deadlock? No. How about 2 philsophers? 3? ... You can see where this is going. Stallings' [6, P. 280] solutions says to remove philosophers from the table until deadlock is not possible – think about what the magic number of philosophers at the table. The way to do this in actual system is through semaphores and letting a certain number of philosopher through. This has the benefit that multiple philosophers can be eating.

In the case that the philosophers aren't evil, this solution requires a lot of time-consuming context switching. There is also no reliable way to know the number of resources before hand. In the dining philosophers case, this is solved because everything is known but trying to specify and operating system where you don't know which file is going to get opened by what process leads you with a faulty solution. And again since semaphores are system constructs, they obey system timing clocks which means that the same processes tend to get added back into the queue again. Now if a philosopher becomes evil, then the problem becomes that there is no preemption. A philosopher can eat for as long as they want and the system will continue to function but that means the fairness of this solution can be low in the worst case. This works best with timeouts (or forced context switches) in order to ensure bounded wait times.

TODO: Prove stallings's doesn't deadlock

Partial Ordering (Dijkstra's Solution)

This is Dijkstra's solution [2, P. 20]. He was the one to propose this problem on an exam. Why does the first solution deadlock? Dijkstra thought that the last philosopher who picks up his left fork (causing the solution to deadlock) should pick up his right. He accomplishes it by number the forks $1..n$, and tells each of the philosopher to pick up his lower number fork. Let's run through the deadlock condition again. Everyone tries to pick up their lower number fork first. Philosopher 1 gets fork 1, Philosopher 2 gets fork 2, and so on until we get to Philosopher n. They have to choose between fork 1 and n. fork 1 is already held up by philosopher 1, so they can't pick up that fork, meaning he won't pick up fork n. We have broken circular wait! Meaning deadlock isn't possible.

The problems to this is that an entity either needs to know the finite set of resources or be able to produce a consistent partial order suck that circular wait cannot happen. This also implies that there needs to be some entity, either the operating system or another process, deciding on the number and all of the philosophers need to agree on the number as new resources come in. As we have also see with previous solutions, this relies on context switching so this prioritizes philosophers that have already eaten but can be made more fair by introducing random sleeps and waits.

TODO: Prove dijkstra's doesn't deadlock

Advanced Solutions

There are many more advanced solutions a non-exhaustive list includes

- Clean/Dirty Forks (Chandra/Misra Solution) TODO: Detail the Clean/Dirty Solution, and cite
- Actor Model (other Message passing models) TODO: Detail the Actor Model, and cite

Topics

- Coffman Conditions
- Resource Allocation Graphs
- Dining Philosophers
- Failed DP Solutions
- Livelocking DP Solutions
- Working DP Solutions: Benefits/Drawbacks

Questions

- What are the Coffman Conditions?
- What do each of the Coffman conditions mean? (e.g. can you provide a definition of each one)
- Give a real life example of breaking each Coffman condition in turn. A situation to consider: Painters, Paint, Paintbrushes etc. How would you assure that work would get done?
- Be able to identify when Dining Philosophers code causes a deadlock (or not). For example, if you saw the following code snippet which Coffman condition is not satisfied?

```
// Get both locks or none
pthread_mutex_lock(a);
if(pthread_mutex_trylock( b )) {/* failure */
  pthread_mutex_unlock( a );
}
```

- The following calls are made

```
// Thread 1
pthread_mutex_lock(m1) // success
pthread_mutex_lock(m2) // blocks

// Thread 2
pthread_mutex_lock(m2) // success
pthread_mutex_lock(m1) // blocks
```

  What happens and why? What happens if a third thread calls `pthread_mutex_lock(m1)` ?

- How many processes are blocked? As usual assume that a process is able to complete if it is able to acquire all of the resources listed below.
  - P1 acquires R1
  - P2 acquires R2
  - P1 acquires R3
  - P2 waits for R3
  - P3 acquires R5
  - P1 waits for R4
  - P3 waits for R1
  - P4 waits for R5
  - P5 waits for R1

(Draw out the resource graph!)

Bibliography

[1] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[2] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. published as [**?**]WD:EWD310pub, n.d. URL `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF`.

[3] Edsger Wybe Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[4] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. ISSN 00029947. URL `http://www.jstor.org/stable/1990888`.

[5] A. Silberschatz, P.B. Galvin, and G. Gagne. *OPERATING SYSTEM PRINCIPLES, 7TH ED*. Wiley student edition. Wiley India Pvt. Limited, 2006. ISBN 9788126509621. URL `https://books.google.com/books?id=WjvX0HmVTlMC`.

[6] William Stallings. *Operating Systems: Internals and Design Principles 7th Ed. by Stallings (International Economy Edition)*. PE, 2011. ISBN 9332518807. URL `https://www.amazon.com/Operating-Systems-Internals-Principles-International/dp/9332518807?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=9332518807`.

Chapter 4

Signals

What is that? It's ridiculous! [Jerry bobs agreeingly] You don't even know, what hotel she's staying at, you can't call her. That's a signal, Jerry, that's a signal! [snaps his fingers again] Signal!

George Costanza (Seinfeld)

TODO: Introduction

Exit statuses

To find the return value of `main()` or value included in `exit()`), Use the `Wait macros` - typically you will use `WIFEXITED` and `WEXITSTATUS` . See `wait/waitpid` man page for more information.

```
int status;
pid_t child = fork();
if (child == -1) return 1; //Failed
if (child > 0) {/* I am the parent - wait for the child to finish */
  pid_t pid = waitpid(child, &status, 0);
  if (pid != -1 && WIFEXITED(status)) {
     int low8bits = WEXITSTATUS(status);
     printf("Process %d returned %d" , pid, low8bits);
  }
} else {/* I am the child */
 // do something interesting
  execl("/bin/ls", "/bin/ls", ".", (char *) NULL); // "ls ."
}
```

A process can only have 256 return values, the rest of the bits are informational, this is done by bit shifting. But, The kernel has an internal way of keeping track of signaled, exited, or stopped. That API is abstracted so that that the kernel developers are free to change at will. Remember that these macros only make sense if the precondition is met. Meaning that a process' exit status won't be defined if the process is signaled. The macros will not do the checking for you, so it's up to the programmer to make sure the logic checks out. As an example above, you should use the `WIFSTOPPED` to check if a process was stopped and then the `WSTOPSIG` to find the signal that stopped it. As such there is no need to memorize the following, this is just a high level overview of how information is stored inside the status variables. From `sys/wait.h` of an old Berkeley kernel[1]:

```c
/* If WIFEXITED(STATUS), the low-order 8 bits of the status. */
#define _WSTATUS(x) (_W_INT(x) & 0177)
#define _WSTOPPED 0177 /* _WSTATUS if process is stopped */
#define WIFSTOPPED(x) (_WSTATUS(x) == _WSTOPPED)
#define WSTOPSIG(x) (_W_INT(x) >> 8)
#define WIFSIGNALED(x) (_WSTATUS(x) != _WSTOPPED && _WSTATUS(x) != 0)
#define WTERMSIG(x) (_WSTATUS(x))
#define WIFEXITED(x) (_WSTATUS(x) == 0)
```

There is an untold convention about exit codes. If the process exited normally and everything was successful, then a zero should be returned. Beyond that, there isn't too many conventions except the ones that you place on yourself. If you know how the program you spawn is going to interact, you may be able to make more sense of the 256 error codes. You could in fact write your program to return 1 if the program went to stage 1 (like writing to a file) 2 if it did something else etc... But none of the unix programs are designed to follow that for simplicity sake.

Signals

A signal is a construct provided to us by the kernel. It allows one process to asynchronously send an event (think a message) to another process. If that process wants to accept the signal, it can, and then, for most signals, can decide what to do with that signal. Here is a short list (non comprehensive) of signals. The overall process for how a kernel sends a signal as well as common use cases are below.

1. Before any signals are generated, the kernel sets up the default signal handlers for a process.

2. If still no signals have arrived, the process can install its own signal handlers. This is simple telling the kernel that when the process gets signal X it should jump to function Y.

3. Now is the fun part, time to deliver a signal! Signals can come from various places below. The signal is now in what we call the generated state.

4. As soon as the signal starts to get deliverd by the kernel, it is in the pending state.

5. The kernel then checks the signals `disposition`, which in layperson terms is whether the process is willing to accept that signal at this point. If not, then the signal is currently blocked and nothing happens.

6. If not, and there is no signal handler installed, the kernel executes the default action. Otherwise, the kernel delivers the signal by stopping *whatever* the process is doing at the current point, and jumping that process to the signal handler. If the program is multithreaded, then the process picks on thread with a signal disposition that can accept the signal and freezes the rest. The signal is now in the delivered phase.

7. Finally, we consider a signal caught if the process remains in tact after the signal was delivered.

Name	Default Action	Usual Use Case
SIGINT	Terminate Process (Can be caught)	Tell the process to stop nicely
SIGQUIT	Terminate Process (Can be caught)	Tells the process to stop harshly
SIGSTOP	Stop Process (Cannot be caught)	Stops the process to be continued
SIGCONT	Continues a Process	Continues to run the process
SIGKILL	Terminate Process (Cannot be caught)	You want your process gone

Sending Signals

Signals can be genrated multiple ways. The user can send a signal. For example, you are at the terminal, and you send `CTRL-C` this is rarely the case in operating systems but is included in user programs for convenience. Another way is when a system event happens. For example, if you access a page that you aren't supposed to, the hardware generates a segfault interrupt which gets intercepted by the kernel. The kernel finds the process that caused this and sends a software interrupt signal `SIGSEGV`. There are softer kernel events like a child being created or sometimes when the kernel wants to like when it is scheduling processes. Finally, another process can send a message when you execute `kill -9 PID`, it sends `SIGKILL` to the process. This could be used in low-stakes communication of events between process. If you are relying on signals to be the driver in your program, you should rethink your application design. There are many drawbacks to using signals for asynchronous communication that is avoided by having a dedicated thread and some form of proper Interprocess Communication.

You can temporarily pause a running process by sending it a SIGSTOP signal. If it succeeds it will freeze a process, the process will not be allocated any more CPU time. To allow a process to resume execution send it the SIGCONT signal. For example, Here's program that slowly prints a dot every second, up to 59 dots.

```c
#include <unistd.h>
#include <stdio.h>
int main() {
  printf("My pid is %d\n", getpid() );
  int i = 60;
  while(--i) {
    write(1, ".",1);
    sleep(1);
  }
  write(1, "Done!",5);
  return 0;
}
```

We will first start the process in the background (notice the & at the end). Then send it a signal from the shell process by using the kill command.

```
>./program &
My pid is 403
...
>kill -SIGSTOP 403
>kill -SIGCONT 403
```

In C, you can send a signal to the child using `kill` POSIX call,

```
kill(child, SIGUSR1); // Send a user-defined signal
kill(child, SIGSTOP); // Stop the child process (the child cannot
    prevent this)
kill(child, SIGTERM); // Terminate the child process (the child can
    prevent this)
kill(child, SIGINT); // Equivalent to CTRL-C (by default closes the
    process)
```

As we saw above there is also a `kill` command available in the shell. Another command `killall` works the exact same way but instead of looking up by PID, it tries to match the name of the process. `ps` is an important utility that can help you find the pid of a process.

```
# First let's use ps and grep to find the process we want to send a
    signal to
$ ps au | grep myprogram
angrave 4409   0.0  0.0  2434892   512 s004 R+    2:42PM  0:00.00
    myprogram 1 2 3

#Send SIGINT signal to process 4409 (equivalent of 'CTRL-C')
$ kill -SIGINT 4409

#Send SIGKILL (terminate the process)
$ kill -SIGKILL 4409
$ kill -9 4409
# Use kill all instead
$ killall -l firefox
```

In order to send a signal to the current, use `raise` or `kill` with `getpid()`

```
raise(int sig); // Send a signal to myself!
kill(getpid(), int sig); // Same as
```

For non-root processes, signals can only be sent to processes of the same user. You cant just SIGKILL my processes! `man -s2 kill` for more details.

Handling Signals

There are strict limitations on the executable code inside a signal handler. Most library and system calls are not `async-signal-safe` - they may not be used inside a signal handler because they are not re-entrant safe. Re-entrant safe means that imagine that your function can be frozen at any point and executed again, can you guarentee that your function wouldn't fail? Let's take the following

```
int func(const char *str) {
  static char buffer[200];
  strncpy(buffer, str, 199); # We finish this line and get recalled
  printf("%s\n", buffer)
}
```

1. We execute (func("Hello"))
2. The string gets copied over to the buffer completely (strcmp(buffer, "Hello") == 0)
3. A signal is deliverd and the function state freezes, we also stop accepting any new signals until after the handler (we do this for convenience)
4. We execute `func("World")`
5. Now (strcmp(buffer, "World") == 0) and the buffer is printed out "World".
6. We resume the interrupted function and now print out the buffer once again "World" instead of what the function call originally intended "Hello"

Guarenteeing that your functions are signal handler safe are not as simple as not having shared buffers. You must also think about multithreading and synchronization i.e. what happens when I double lock a mutex? You also have to make sure that each subfunction call is re-entrant safe. Suppose your original program was interrupted while executing the library code of `malloc` ; the memory structures used by malloc will not be in a consistent state. Calling `printf` (which uses `malloc`) as part of the signal handler is unsafe and will result in `undefined behavior` i.e. it is no longer a useful,predictable program. In practice your program might crash, compute or generate incorrect results or stop functioning (`deadlock`), depending on exactly what your program was executing when it was interrupted to execute the signal handler code. One common use of signal handlers is to set a boolean flag that is occasionally polled (read) as part of the normal running of the program. For example,

```
int pleaseStop ; // See notes on why "volatile sig_atomic_t" is better

void handle_sigint(int signal) {
  pleaseStop = 1;
}

int main() {
  signal(SIGINT, handle_sigint);
  pleaseStop = 0;
  while ( ! pleaseStop) {
     /* application logic here */
   }
  /* cleanup code here */
}
```

The above code might appear to be correct on paper. However, we need to provide a hint to the compiler and to the CPU core that will execute the `main()` loop. We need to prevent a compiler optimization: The expression `! pleaseStop` appears to be a loop invariant meaning it will be true forever, so can be simplified to `true`. Secondly, we need to ensure that the value of `pleaseStop` is not cached using a CPU register and instead always read from and written to main memory. The `sig_atomic_t` type implies that all the bits of the variable can be read or modified as an `atomic operation` - a single uninterruptable operation. It is impossible to read a value that is composed of some new bit values and old bit values.

By specifying `pleaseStop` with the correct type `volatile sig_atomic_t`, we can write portable code where the main loop will be exited after the signal handler returns. The `sig_atomic_t` type can be as large as an `int` on most modern platforms but on embedded systems can be as small as a `char` and only able to represent (-127 to 127) values.

```
volatile sig_atomic_t pleaseStop;
```

Two examples of this pattern can be found in `COMP` a terminal based 1Hz 4bit computer COMP. Two boolean flags are used. One to mark the delivery of `SIGINT` (CTRL-C), and gracefully shutdown the program, and the other to mark `SIGWINCH` signal to detect terminal resize and redraw the entire display.

You can also choose a handle pending signals asynchronously or synchronously. Install a signal handler to asynchronously handle signals use `sigaction` (or, for simple examples, `signal`). To synchronously catch a pending signal use `sigwait` which blocks until a signal is delivered or `signalfd` which also blocks and provides a file descriptor that can be `read()` to retrieve pending signals.

Sigaction

You should use `sigaction` instead of `signal` because it has better defined semantics. `signal` on different operating system does different things which is **bad** `sigaction` is more portable and is better defined for threads if need be. To change the `signal disposition` of a process - i.e. what happens when a signal is delivered to your process - use `sigaction` You can use system call `sigaction` to set the current handler for a signal or read the current signal handler for a particular signal.

```
int sigaction(int signum, const struct sigaction *act, struct
    sigaction *oldact);
```

The sigaction struct includes two callback functions (we will only look at the 'handler' version), a signal mask and a flags field -

```
struct sigaction {
             void     (*sa_handler)(int);
             void     (*sa_sigaction)(int, siginfo_t *, void *);
             sigset_t  sa_mask;
             int       sa_flags;
};
```

Suppose you installed a signal handler for the alarm signal,

```
signal(SIGALRM, myhandler);
```

The equivalent `sigaction` code is:

```
struct sigaction sa;
sa.sa_handler = myhandler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGALRM, &sa, NULL)
```

However, we typically may also set the mask and the flags field. The mask is a temporary signal mask used during the signal handler execution. The `SA_RESTART` flag will automatically restart some (but not all) system calls that otherwise would have returned early (with EINTR error). The latter means we can simplify the rest of code somewhat because a restart loop may no longer be required.

```
sigfillset(&sa.sa_mask);
sa.sa_flags = SA_RESTART; /* Restart functions if interrupted by
    handler */
```

Sigwait

Sigwait can be used to read one pending signal at a time. `sigwait` is used to synchronously wait for signals, rather than handle them in a callback. A typical use of sigwait in a multi-threaded program is shown below. Notice that the thread signal mask is set first (and will be inherited by new threads). This prevents signals from being *delivered* so they will remain in a pending state until sigwait is called. Also notice the same set sigset_t variable is used by sigwait - except rather than setting the set of blocked signals it is being used as the set of signals that sigwait can catch and return.

One advantage of writing a custom signal handling thread (such as the example below) rather than a callback function is that you can now use many more C library and system functions that otherwise could not be safely used in a signal handler because they are not async signal-safe.

Based on http://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_sigmask.htmlSigmask Code

```
static sigset_t signal_mask; /* signals to block      */

int main (int argc, char *argv[]) {
    pthread_t sig_thr_id;    /* signal handler thread ID */
    sigemptyset (&signal_mask);
    sigaddset (&signal_mask, SIGINT);
    sigaddset (&signal_mask, SIGTERM);
    pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);

    /* New threads will inherit this thread's mask */
    pthread_create (&sig_thr_id, NULL, signal_thread, NULL);

    /* APPLICATION CODE */
    ...
}

void *signal_thread (void *arg) {
    int      sig_caught;   /* signal caught     */

    /* Use same mask as the set of signals that we'd like to know
        about! */
    sigwait(&signal_mask, &sig_caught);
    switch (sig_caught)
    {
    case SIGINT:    /* process SIGINT */
        ...
        break;
    case SIGTERM:   /* process SIGTERM */
        ...
        break;
    default:        /* should normally not happen */
        fprintf (stderr, "\nUnexpected signal %d\n", sig_caught);
        break;
    }
}
```

Signal Disposition

For each process, each signal has a disposition which means what action will occur when a signal is delivered to the process. For example, the default disposition SIGINT is to terminate it. The signal disposition can be changed by calling signal() (which is simple but not portable as there are subtle variations in its implementation on different POSIX architectures and also not recommended for multi-threaded programs) or `sigaction` (discussed later). You can imagine the processes' disposition to all possible signals as a table of function pointers entries (one for each possible signal).

The default disposition for signals can be to ignore the signal, stop the process, continue a stopped process, terminate the process, or terminate the process and also dump a 'core' file. Note a core file is a representation of the processes' memory state that can be inspected using a debugger.

Multiple signals connot be queued. However it is possible to have signals that are in a pending state. If a signal is pending, it means it has not yet been delivered to the process. The most common reason for a signal to be pending is that the process (or thread) has currently blocked that particular signal. If a particular signal, e.g. SIGINT, is pending then it is not possible to queue up the same signal again. It *is* possible to have more than one signal of a different type in a pending state. For example SIGINT and SIGTERM signals may be pending (i.e. not yet delivered to the target process)

Signals can be blocked (meaning they will stay in the pending state) by setting the process signal mask or, when you are writing a multi-threaded program, the thread signal mask.

Disposition in Child Processes (No Threads)

After forking, The child process inherits a copy of the parent's signal dispositions and a copy of the parent's signal mask. In other words, if you have installed a SIGINT handler before forking, then the child process will also call the handler if a SIGINT is delivered to the child. Also if `SIGINT` is blocked in the parent, it will be blocked in the child too. Note pending signals for the child are *not* inherited during forking. But after `exec`, both the signal mask and the signal disposition carries over to the exec-ed program. https://www.gnu.org/software/libc/manual/html_node/Executing-a-File.html#Executing-a-File Pending signals are preserved as well. Signal handlers are reset, because the original handler code has disappeared along with the old process.

To block signals use `sigprocmask`! With sigprocmask you can set the new mask, add new signals to be blocked to the process mask, and unblock currently blocked signals. You can also determine the existing mask (and use it for later) by passing in a non-null value for oldset.

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);‘
```

From the Linux man page of sigprocmask,

```
SIG_BLOCK: The set of blocked signals is the union of the current set and the
  set argument.
SIG_UNBLOCK: The signals in set are removed from the current set of blocked
  signals. It is permissible to attempt to unblock a signal which is not blocked.
SIG_SETMASK: The set of blocked signals is set to the argument set.
```

The sigset type behaves as a bitmap, except functions are used rather than explicitly setting and unsetting bits using & and |. It is a common error to forget to initialize the signal set before modifying one bit. For example,

```
sigset_t set, oldset;
sigaddset(&set, SIGINT); // Ooops!
sigprocmask(SIG_SETMASK, &set, &oldset)
```

Correct code initializes the set to be all on or all off. For example,

```
sigfillset(&set); // all signals
sigprocmask(SIG_SETMASK, &set, NULL); // Block all the signals!
// (Actually SIGKILL or SIGSTOP cannot be blocked...)

sigemptyset(&set); // no signals
sigprocmask(SIG_SETMASK, &set, NULL); // set the mask to be empty
    again
```

Signals in a multithreaded program

The new thread inherits a copy of the calling thread's mask. On initialization the calling thread's mask is the exact same as the processes mask because threads are essentially processes. After a new thread is created though, the processes signal mask turns into a gray area. Instead, the kernel likes to threat the process as a collection of thread, each of which can institute a signal mask and receive signals. In order to start setting your mask you can use,

```
pthread_sigmask( ... ); // set my mask to block delivery of some
    signals
pthread_create( ... ); // new thread will start with a copy of the
    same mask
```

Blocking signals is similar in multi-threaded programs to single-threaded programs: * Use `pthread_sigmask` instead of `sigprocmask` * Block a signal in all threads to prevent its asynchronous delivery

The easiest method to ensure a signal is blocked in all threads is to set the signal mask in the main thread before new threads are created

```
sigemptyset(&set);
sigaddset(&set, SIGQUIT);
sigaddset(&set, SIGINT);
pthread_sigmask(SIG_BLOCK, &set, NULL);

// this thread and the new thread will block SIGQUIT and SIGINT
pthread_create(&thread_id, NULL, myfunc, funcparam);
```

Just as we saw with `sigprocmask`, `pthread_sigmask` includes a 'how' parameter that defines how the signal set is to be used:

```
pthread_sigmask(SIG_SETMASK, &set, NULL) - replace the thread's mask
    with given signal set
pthread_sigmask(SIG_BLOCK, &set, NULL) - add the signal set to the
    thread's mask
pthread_sigmask(SIG_UNBLOCK, &set, NULL) - remove the signal set from
    the thread's mask
```

A signal then can delivered to any signal thread that is not blocking that signal. If the two or more threads can receive the signal then which thread will be interrupted is arbitrary! A common practice is to have one thread that can receive all signals or if there is a certain signal that requires special logic, have multiple threads for multiple signals. Even though programs from the outside can't send signals to specific threads (unless a thread is assigned a signal), you can do that in your program with `pthread_kill(pthread_t thread, int sig)`. In the example below, the newly created thread executing `func` will be interrupted by `SIGINT`

```
pthread_create(&tid, NULL, func, args);
pthread_kill(tid, SIGINT);
pthread_kill(pthread_self(), SIGKILL); // send SIGKILL to myself
```

As a word of warning pthread$_{k}ill(threadid,SIGKILL)willkilltheentireprocess.Thoughindividualthreadscans$

The linux man pages discusses signal system calls in section 2. There is also a longer article in section 7 (though not in OSX/BSD):

```
man -s7 signal
```

Topics

- Signals
- Signal Handler Safe
- Signal Disposition
- Signal States
- Pending Signals when Forking/Exec
- Signal Disposition when Forking/Exec
- Raising Signals in C
- Raising Signals in a multithreaded program

Questions

- What is a signal?
- How are signals served under UNIX? (Bonus: How about Windows?)
- What does it mean that a function is signal handler safe
- What is a process Signal Disposition?
- How do I change the signal disposition in a single threaded program? How about multithreaded?
- Why sigaction vs signal?
- How do I asynchronously and synchronously catch a signal?
- What happens to pending signals after I fork? Exec?
- What happens to my signal disposition after I fork? Exec?

Bibliography

[1]

Chapter 5

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->siz 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.

Life in the terminal

TODO:

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

System Programming Jokes

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.