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 [6] and the XNU kernel Inc. [2] 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 [7]. 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 [5]. 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 but also it doesn't want to be completely compliant with a bunch of different standards because then it has to ensue increasing development costs to maintain compliance.

Fast forward however many years, and we are at the current C standard put forth by ISO: C11. Not all the code that we us in this class will be in this format. We will aim to using C99 as the standard that most computers recognize. We will talk about some off-hand features like `getline` because they are so widely used with the GNU-C library. We'll begin by providing a decently comprehensive overview of the language with pairing facilities.

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 [4]. 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. [4]
4. `return 0;`. `main` has to return an integer. By convention, `return 0` means success and anything else means failure [4].

```
$ 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 the following susbtitution is performed.

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

There are side effects to the preprocessor though. One problem is that the preprocessor needs to be able to tokenize properly, meaning trying to redefine the interals of the C language with a preprocessor may be impossible. Another problem is that they can't be nested infinitely – there is an unbounded depth where they need to stop. Macros are also just simple text substitutions.

```c
#define min(a,b) ((a)<(b) ? (a) : (b))
int x = 4;
if(min(x++, 5)) 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 is opaque what gets printed out but it will be 6. Also consider the edge case when operator precedence comes into play.

```c
#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`. You can have logical problems with the flexibility of certain parameters. One common source of confusion is with static arrays and the `sizeof` operator.

```c
#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 number of 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.

```c
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

```c
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` is to declare an enumeration. An enumeration is a type that can take on many, finite values. If you have an enum and don't specify any numerics, the c compiler when generate a unique number for that enum (within the context of the current enum) and use that for comparisons. To declare an instance of an enum, you must say `enum <type> varname`. The added benefit to this is that C can type check these expressions to make sure that you are only comparing alike types.

```
enum day{ monday, tuesday, wednesday,
  thursday, friday, saturday, sunday};

void process_day(enum day foo) {
  switch(day) {
    case monday:
      printf("Go home!\n"); break;
    // ...
  }
}
```

It is completely possible to assign enum values to either be different or the same. Just don't rely on the compiler for consistent numbering. If you are going to use this abstraction, try not to break it.

```
enum day{
  monday = 0,
  tuesday = 0,
  wednesday = 0,
  thursday = 1,
  friday = 10,
  saturday = 10,
  sunday = 0};

void process_day(enum day foo) {
  switch(day) {
    case monday:
      printf("Go home!\n"); break;
    // ...
  }
}
```

6. `extern` is a special keyword that tells the compiler that the variable may be defined in another object file or a library, so the compiler doesn't throw an error when either the variable is not defined or if the variable is defined twice because the first file will really be referencing the variable in the other file.

```c
// file1.c
extern int panic;

void foo() {
  if (panic) {
    printf("NONONONONO");
  } else {
    printf("This is fine");
  }
}

//file2.c

int panic = 1;
```

7. `for` is a keyword that allows you to iterate with an initialization condition, a loop invariant, and an update condition. This is meant to be a replacement for the while loop

```c
for (initialization; check; update) {
  //...
}

// Typically
int i;
for (i = 0; i < 10; i++) {
  //...
}
```

One thing to note is that as of the C89 standard, you cannot declare variables inside the `for` loop. This is because there was a disagreement in the standard for how the scoping rules of a variable defined in the loop would work. It has since been resolved with more recent standards, so people can use the for loop that they know and love today

```c
for(int i = 0; i < 10; ++i) {

}
```

The order of evaluation for a `for` loop is as follows

(a) Perform the initialization condition.

(b) Check the invariant. If false, terminate the loop and execute the next statement. If true, continue to the body of the loop.

(c) Perform the body of the loop.

(d) Perform the update condition.

(e) Jump to (2).

8. `goto` is a keyword that allows you to do conditional jumps. Do not use `goto` in your programs. The reason being is that it makes your code infinitely more hard to understand when strung together with multiple chains. It is fine to use in some contexts though. The keyword is usually used in kernel contexts when adding another stack frame for cleanup isn't a good idea. The canonical example of kernel cleanup is as below.

```
void setup(void) {
  Doe *deer;
  Ray *drop;
  Mi *myself;

  if (!setupdoe(deer)) {
    goto finish;
  }

  if (!setupray(drop)) {
    goto cleanupdoe;
  }

  if (!setupmi(myself)) {
    goto cleanupray;
  }

  perform_action(deer, drop, myself);

cleanupray:
  cleanup(drop);
cleanupdoe:
  cleanup(deer);
finish:
  return;
}
```

9. `if else else-if` are control flow keywords. There are a few ways to use these (1) A bare if (2) An if with an else (3) an if with an else-if (4) an if with an else if and else. The statements are always executed from the if to the else. If any of the intermediate conditions are true, the if block performs that action and goes to the end of that block.

```
// (1)

if (connect(...))
  return -1;

// (2)
if (connect(...)) {
  exit(-1);
} else {
  printf("Connected!");
}

// (3)
if (connect(...)) {
  exit(-1);
} else if (bind(..)) {
  exit(-2);
}

// (1)
if (connect(...)) {
  exit(-1);
} else if (bind(..)) {
  exit(-2);
} else {
  printf("Successfully bound!");
}
```

10. `inline` is a compiler keyword that tells the compiler it's okay not to create a new function in the assembly. Instead, the compile is hinted at substituting the function body directly into the calling function. This is not always recommended explicitly as the compiler is usually smart enough to know when to `inline` a function for you.

```
inline int max(int a, int b) {
  return a < b ? a : b;
}

int main() {
  printf("Max %d", max(a, b));
  // printf("Max %d", a < b ? a : b);
}
```

11. `restrict` is a keyword that tells the compiler that this particular memory region shouldn't overlap with all other memory regions. The use case for this is to tell users of the program that it is undefined behavior if the memory regions overlap.

```
memcpy(void * restrict dest, const void* restrict src, size_t
    bytes);

void add_array(int *a, int * restrict c) {
  *a += *c;
}
int *a = malloc(3*sizeof(*a));
*a = 1; *a = 2; *a = 3;
add_array(a + 1, a) // Well defined
add_array(a, a) // Undefined
```

12. `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.

```
void process() {
  if (connect(...)) {
    return -1;
  } else if (bind(...)) {
    return -2
  }
  return 0;
}
```

13. `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.

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

14. `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.

```
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

```c
// 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. Also, if the compiler knows the full length of an array object, it will use that in the expression instead of decaying it to a pointer.

```c
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!

15. `static` is a type specifier with three meanings.
16. When used with a global variable or function declaration it means that the scope of the variable or the function is only limited to the file.
17. When used with a function variable, that declares that the variable has static allocation – meaning that the variable is allocated once at program start up not every time the program is run.

```c
static int i = 0;

static int _perform_calculation(void) {
  // ...
}

char *print_time(void) {
  static char buffer[200]; // Shared every time a function is
      called
  // ...
}
```

18. `struct` is a keyword that allows you to pair multiple types together into a new structure. Structs are contiguous regions of memory that one can access specific elements of each memory as if they were separate variables.

```c
struct hostname {
  const char *port;
  const char *name;
  const char *resource;
}; // You need the semicolon at the end
// Assign each individually
struct hostname facebook;
facebook.port = "80";
facebook.name = "www.google.com";
facebook.resource = "/";

// You can use static initialization in later versions of c
struct hostname google = {"80", "www.google.com", "/"};
```

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.

```c
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` is a new type specifier. A union is one piece of memory that a bunch of variables occupy. It is used to maintain consistency while having the flexibility to switch between types without mainting functions to keep track of the bits. Consider an example where we have different pixel

values.

```
union pixel {
  struct values {
    char red;
    char blue;
    char green;
    char alpha;
  } values;
  uint32_t encoded;
}; // Ending semicolon needed
union pixel a;
// When modifying or reading
a.values.red;
a.values.blue = 0x0;

// When writing to a file
fprintf(picture, "%d", a.encoded);
```

22. `unsigned` is a type modifier that forces `unsigned` behavior in the variables they modify. Unsigned can only be on primitive int types (like `int` and `long`). There is a lot of behavior associated with unsigned arthmetic and whatnot, just know for the most part unless you need to do bit shifting you probably won't need it.

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.

    ```
    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.

    ```
    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.

```
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.

```
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 [3, P. 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 [1]. 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

Operators are language constructs in C that are defined as part of the grammar of the language.

1. `[]` is the subscript operator. `a[n] == (a + n)*` where `n` is a number type and `a` is a pointer type.

2. `->` is the structure dereference operator. If you have a pointer to a struct `*p`, you can use this to access one of its elements. `p->element`.

3. `.` is the structure reference operator. If you have an object on the stack `a` then you can access an element `a.element`.

4. `+/-a` is the unary plus and minus operator. They either keep or negate the sign, respectively, of the integer or float type underneath.

5. `*a` is the dereference operator. If you have a pointer `*p`, you can use this to access the element located at this memory address. If you are reading, the return value will be the size of the underlying type. If you are writing, the value will be written with an offset.

6. `&a` is the addressof operator. This takes the an element and returns its address.

7. `++` is the increment operator. You can either take it prefix or postfix, meaning that the variable that is being incremented can either be before or after the operator. `a = 0; ++a === 1` and `a = 1; a++ === 0`.

8. `-` is the decrement operator. Same semantics as the increment operator except with decreasing the value by one.

9. `sizeof` is the sizeof operator. This is also mentioned in the keywords section.

10. `a <mop> b` where `<mop>=+, -, *, %, /` are the mathematical binary operators. If the operands are both number types, then the operations are plus, minus, times, modulo, and division respectively. If the left operand is a pointer and the right operand is an integer type, then only plus or minux may be used and the rules for pointer arithmetic are invoked.

11. `»/«` are the bit shift operators. The operand on the right has to be an integer type whose signedness is ignored unless it is signed negative in which case the behavior is undefined. The operator on the left decides a lot of semantics. If we are left shifting, there will always be zeros introduced on the right. If we are right shifting there are a few different cases

    - If the operand on the left is signed, then the integer is sign extended. This means that if the number has the sign bit set, then any shift right will introduce ones on the left. If the number does not have the sign bit set, any shift right will introduce zeros on the left.
    - If the operand is unsigned, zeros will be introduced on the left either way.

```
unsigned short uns = -127; // 1111111110000001
short sig = 1; // 0000000000000001
uns << 2; // 1111111000000100
sig << 2; // 0000000000000100
uns >> 2; // 1111111111100000
sig >> 2; // 0000000000000000
```

12. `<=/>=` are the greater than equal to/less than equal to operators. They do as the name implies.

13. </> are the greater than/less than operators. They again do as the name implies.
14. ==/= are the equal/not equal to operators. They once again do as the name implies.
15. && is the logical and operator. If the first operand is zero, the second won't be evaluated and the expression will evaluate to 0. Otherwise, it yields a 1-0 value of the second operand.
16. || is the logical or operator. If the first operand is not zero, then second won't be evaluated and the expression will evaluate to 1. Otherwise, it yields a 1-0 value of the second operand.
17. ! is the logical not operator. If the operand is zero, then this will return 1. Otherwise, it will return 0.
18. & If a bit is set in both operands, it is set in the output. Otherwise, it is not.
19. | If a bit is set in either operand, it is set in the output. Otherwise, it is not.
20. ~ If a bit is set in the input, it will not be set in the output and vice versa.
21. ?: is the tertinary operator. You put a boolean condition before the and if it evaluates to non-zero the element before the colon is returned otherwise the element after is. `1 ? a : b === a` and `0 ? a : b === b`.
22. `a, b` is the comma operator. `a` is evaluated and then `b` is evaluated and `b` is returned.

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

```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
```

`printf` calls the system call `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

To 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`

```
puts("Current selection: ");
putchar('1');
```

To 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`. You can also use file descriptors in the printf family of functions! 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.

To print data into a C string, use `sprintf` or better `snprintf`. `snprintf` returns the number of characters written excluding the terminating byte. In the above example, this would be a maximum of 199. We would use `sprintf` in cases where we know that the size of the string will not be anything more than a certain fixed amount (think about printing an integer, it will never be more than 11 characters with the null byte)

```
// Fixed
char int_string[20];
sprintf(int_string, "%d", integer);

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

If I want to `printf` to call write without a newline `fflush( FILE* inp )`. The contents of the file will be written. If I wanted to write "Hello World" with no newline, I could write it like this.

```
int main(){
    fprintf(stdout, "Hello World");
    fflush(stdout);
    return 0;
}
```

In addition to the printf family, there is `gets`. `gets` is deprecated in C99 standard and has been removed from the latest C standard (C11). Programs should use `fgets` or `getline` instead.

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

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

// Example, the following will not read more than 9 chars
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. On the other hand, one of the advantages of `getline` is that will automatically (re-) allocate a buffer on the heap of sufficient size.

```c
// 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);
```

In addition to those functions, we have `perror` that has a two-fold meaning. Let's say that you have a function call that just failed because you checked the man page and it is a failing return code.`perror(const char* message)` will print the English version of the error to stderr.

```c
int main(){
    int ret = open("IDoNotExist.txt", O_RDONLY);
    if(ret < 0){
        perror("Opening IDoNotExist:");
    }
    //...
    return 0;
}
```

To have a library function parse input, use `scanf` (or `fscanf` or `sscanf`) to get input from the default input stream, an arbitrary file stream or a C string respectively. It's a good idea to check the

return value to see how many items were parsed. `scanf` functions require valid pointers. It's a common source of error to pass in an incorrect pointer value. For example,

```
int *data = (int *) malloc(sizeof(int));
char *line = "v 10";
char type;
// Good practice: Check scanf parsed the line and read two values:
int ok = 2 == sscanf(line, "%c %d", &type, &data); // pointer error
```

We wanted to write the character value into c and the integer value into the malloc'd memory. However, we passed the address of the data pointer, not what the pointer is pointing to! So `sscanf` will change the pointer itself. i.e. the pointer will now point to address 10 so this code will later fail e.g. when free(data) is called.

Now, scanf will just keep reading characters until the string ends. To stop scanf from causing a buffer overflow, use a format specifier. Make sure to pass one less than the size of the buffer.

```
char buffer[10];
scanf("%9s", buffer); // reads up to 9 charactes from input (leave
    room for the 10th byte to be the terminating byte)
```

string.h

More information about all of these functions. Any behavior not in the docs like passing `strlen(NULL)` is considered undefined behavior.

- `int strlen(const char *s)` returns the length of the string not including the null byte
- `int strcmp(const char *s1, const char *s2)` returns an integer determining the lexicographic order of the strings. If s1 where to come before s2 in a dictionary, then a -1 is returned. If the two strings are equal, then 0. Else, 1.
- `char *strcpy(char *dest, const char *src)` Copies the string at `src` to `dest`. **assumes dest has enough space for src**
- `char *strcat(char *dest, const char *src)` Concatenates the string at `src` to the end of destination. **This function assumes that there is enough space for `src` at the end of destination including the NULL byte**
- `char *strdup(const char *dest)` Returns a `malloc`'ed copy of the string.
- `char *strchr(const char *haystack, int needle)` Returns a pointer to the first occurrence of `needle` in the `haystack`. If none found, `NULL` is returned.
- `char *strstr(const char *haystack, const char *needle)` Same as above but this time a string!
- `char *strtock(const char *str, const char *delims)`

A dangerous but useful function strtok takes a string and tokenizes it. Meaning that it will transform the strings into separate strings. This function has a lot of specs so please read the man pages a contrived example is below.

```c
#include <stdio.h>
#include <string.h>

int main(){
    char* upped = strdup("strtok,is,tricky,!!");
    char* start = strtok(upped, ",");
    do{
        printf("%s\n", start);
    }while((start = strtok(NULL, ",")));
    return 0;
}
```

Output

```
strtok
is
tricky
!!
```

What happens when I change upped like this?

```c
char* upped = strdup("strtok,is,tricky,,,!!");
```

- For integer parsing use `long int strtol(const char *nptr, char **endptr, int base);` or `long long int strtoll(const char *nptr, char **endptr, int base);`.

  What these functions do is take the pointer to your string `*nptr` and a `base` (ie binary, octal, decimal, hexadecimal etc) and an optional pointer `endptr` and returns a parsed value.

```c
int main(){
    const char *nptr = "1A2436";
    char* endptr;
    long int result = strtol(nptr, &endptr, 16);
    return 0;
}
```

Be careful though! Error handling is tricky because the function won't return an error code. If you give it a string that is not a number it will return 0. This means you cant differentiate between a

valid “0” and an invalid string. See the man page for more details on strol behavior with invalid and out of bounds values. A safer alternative is use to `sscanf` (and check the return value).

```c
int main(){
    const char *input = "0"; // or "!##@" or ""
    char* endptr;
    long int parsed = strtol(input, &endptr, 10);
    if(parsed == 0){
        // Either the input string was not a valid base-10
            number or it really was zero!

    }
    return 0;
}
```

- `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 below.
- `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. `memcpy` and `memmove` both in `<string.h>`? Because strings are essentially raw memory with a null byte at the end of them!

Conventions/Errno

TODO: Conventions and errno, talk about what the unix conventions are for processes and what the errno conventions are

System Calls

TODO: System Calls, talk about what a system call is and an aside for the practicalities of system calls and what actually happens

C Memory Model

Structs

In low-level terms, a struct is just a piece of contiguous memory, nothing more. Just like an array, a struct has enough space to keep all of its members. But unlike an array, it can store different types. Consider the contact struct declared above

```
struct contact {
    char firstname[20];
    char lastname[20];
    unsigned int phone;
};

struct contact bhuvan;
```

```
/* a lot of times we will do the following typdef
 so we can just write contact contact1 */

typedef struct contact contact;
contact bhuvan;

/* You can also declare the struct like this to get
 it done in one statement */
typedef struct optional_name {
    ...
} contact;
```

If you compile the code without any optimizations and reordering, you can expect the addresses of each of the variables to look like this.

```
&bhuvan          // 0x100
&bhuvan.firstname // 0x100 = 0x100+0x00
&bhuvan.lastname // 0x114 = 0x100+0x14
&bhuvan.phone    // 0x128 = 0x100+0x28
```

Because all your compiler does is say 'hey reserve this much space, and I will go and calculate the offsets of whatever variables you want to write to'. The offsets are where the variable starts at. The phone variables starts at the 0x128th bytes and continues for sizeof(int) bytes, but not always. **Offsets don't determine where the variable ends though**. Consider the following hack that you see in a lot of kernel code.

```c
typedef struct {
    int length;
    char c_str[0];
} string;

const char* to_convert = "bhuvan";
int length = strlen(to_convert);

// Let's convert to a c string
string* bhuvan_name;
bhuvan_name = malloc(sizeof(string) + length+1);
/*
Currently, our memory looks like this with junk in those black spaces
                 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
 bhuvan_name = |___|___|___|___|___|___|___|___|___|___|___|

*/


bhuvan_name->length = length;
/*
This writes the following values to the first four bytes
The rest is still garbage
                 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
 bhuvan_name = | 0 | 0 | 0 | 6 |___|___|___|___|___|___|___|

*/


strcpy(bhuvan_name->c_str, to_convert);
/*
Now our string is filled in correctly at the end of the struct


                 ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ____
 bhuvan_name = | 0 | 0 | 0 | 6 | b | h | u | v | a | n | \0 |
                                                       âĂ¿
*/

strcmp(bhuvan_name->c_str, "bhuvan") == 0 //The strings are equal!
```

Aside

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 |
|___ ___ ___ ___ ___ ___|
|___|___|___ ___|___ ___|
```

Strings in C

Figure 2.1: String

In C we have Null Terminated strings rather than Length Prefixed for historical reasons. What that means for your average everyday programming is that you need to remember the null character! A string in C is defined as a bunch of bytes until you reach " or the Null Byte.

Two places for strings

Whenever you define a constant string (ie one in the form `char* str = "constant"`) That string is stored in the *data* or *code* segment that is **read-only** meaning that any attempt to modify the string will cause a segfault.

If one, however, `malloc`'s space, one can change that string to be whatever they want. Forgetting to NULL terminate a string is a big affect on the strings! Bounds checking is important. The heart bleed bug mentioned earlier in the wiki book is partially because of this.

Strings in C are represented as characters in memory. The end of the string includes a NULL (0) byte. 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.

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.

```c
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

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

Think

```c
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

```c
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

Look in the appendix for Life in the Terminal!

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++ )) {};
```

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
```

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));
```

Buffer overflow/ underflow

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.

```
#define N (10)
int i = N, array[N];
for( ; i >= 0; i--) array[i] = i;
```

C does not check that pointers are valid. The above example writes into `array[10]` which is outside the array bounds. This can cause memory corruption because that memory location is probably being used for something else. In practice, this can be harder to spot because the overflow/underflow may

occur in a library call e.g.

```
gets(array); // Let's hope the input is shorter than my array!
```

Strings require strlen(s)+1 bytes Every string must have a null byte after the last characters. To store the string "Hi" it takes 3 bytes: [H] [i] [].

```
  char *strdup(const char *input) {/* return a copy of 'input' */
    char *copy;
    copy = malloc(sizeof(char*));  /* nope! this allocates space for a
        pointer, not a string */
    copy = malloc(strlen(input));  /* Almost...but what about the null
        terminator? */
    copy = malloc(strlen(input) + 1); /* That's right. */
    strcpy(copy, input); /* strcpy will provide the null terminator */
    return copy;
}
```

Using uninitialized variables

```
int myfunction() {
  int x;
  int y = x + 2;
...
```

Automatic variables hold garbage (whatever bit pattern happened to be in memory). It is an error to assume that it will always be initialized to zero.

Assuming Uninitialized memory will be zeroed

```
void myfunct() {
   char array[10];
   char *p = malloc(10);
```

Automatic (temporary variables) are not automatically initialized to zero. Heap allocations using malloc are not automatically initialized to zero.

Logic and Program flow mistakes

Equal vs. equality

```
int answer = 3; // Will print out the answer.
if (answer = 42) {printf("I've solved the answer! It's %d", answer);}
```

Undeclared or incorrectly prototyped functions

```
time_t start = time();
```

The system function 'time' actually takes a parameter (a pointer to some memory that can receive the time_t structure). The compiler did not catch this error because the programmer did not provide a valid function prototype by including `time.h`

Extra Semicolons

```
for(int i = 0; i < 5; i++) ; printf("I'm printed once");
while(x < 10); x++ ; // X is never incremented
```

However, the following code is perfectly OK.

```
for(int i = 0; i < 5; i++){
    printf("%d\n", i);;;;;;;;;;;;;;
}
```

It is OK to have this kind of code, because the C language uses semicolons (;) to separate statements. If there is no statement in between semicolons, then there is nothing to do and the compiler moves on to the next statement

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] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008. doi: 10.1109/IEEESTD.2008.4610935.

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

[3] ISO 1124:2005. ISO C Standard. Standard, International Organization for Standardization, Geneva, CH, March 2005. URL `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[4] 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`.

[5] 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`.

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

[7] 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

Processes

Who needs process isolation?

Intel PR on Meltdown (Joke)

Aside

In the beginning, there is a kernel. The operating system kernel is a special piece of software. This is the piece of software that is loaded up before all of your other programs even consider getting booted up. What the kernel does is the following, abbreviated

1. The operating system executes ROM or read only code
2. The operating system then executes a `boot_loader` or `EFI` extensions nowadays
3. The boot_loader loads your kernels
4. Your kernel executes `init` to bootstrap itself from nothing
5. The kernel executes start up scripts
6. The kernel executes userland scripts, and you get to use your computer!

You don't need to know the specifics of the booting process, but there it is. When you are executing in user space the kernel provides some important operations that programs don't have to worry about.

- Scheduling Processes and threads
- Handling synchronization primitives
- Providing System Calls like `write` or `read`
- Manages virtual memory and low level binary devices like `usb` drivers
- Handles reading and understanding a filesystem
- Handles communicating over networks

- Handles communications with other processes
- Dynamically linking libraries

The kernel handles all of this stuff in kernel mode. Kernel mode gets you greater power, like executing extra CPU instructions but at the cost of one failure crashes your entire computer – ouch. That is what you are going to interacting with in this class. One of the things that you have already become familiar with is that the kernel gives you file descriptors when you open text files. Here is a zine from Julia Evans that details it a bit.

Figure 3.1: File Descriptors

As the little zine shows, the Kernel keeps track of the file descriptors and what they point to. We will see later that file descriptors need not point to actual files and the OS keeps track of them for you. Also, notice that between processes file descriptors may be reused but inside of a process they are unique. File descriptors also have a notion of position. You can read a file on disk completely because the OS keeps track of the position in the file, and that belongs to your process as well.

Processes

A process an instance of a computer program that may be running. Processes have a lot of things at their disposal. At the start of each program you get one process, but each program can make more processes. In fact, your operating system starts up with only one process and all other processes are forked off of that – all of that is done under the hood when booting up. A program consists of

- A binary format: This tells the operating system which set of bits in the binary are what – which part is executable, which parts are constants, which libraries to include etc.
- A set of machine instructions
- A number denoting which instruction to start from
- Constants
- Libraries to link and where to fill in the address of those libraries

When your operating system starts on a linux machine, there is a process called `init.d` that gets created. That process is a special one handling signals, interrupts, and a persistence module for certain kernel elements. Whenever you want to make a new process, you call `fork` and use `exec` to load another program.

Processes are very powerful but they are isolated! That means that by default, no process can communicate with another process. This is very important because if you have a large system (let's say EWS) then you want some processes to have higher privileges (monitoring, admin) than your average user, and one certainly doesn't want the average user to be able to bring down the entire system either on purpose or accidentally by modifying a process.

```
int secrets;
secrets++;
printf("%d\n", secrets);
```

On two different terminals, as you would guess they would both print out 1 not 2. Even if we changed the code to do something really hacky, there would be no way to change another process' state (okay maybe dirty cow or meltdown but that is getting a little too in depth).

Process Contents

Memory Layout

When a process starts, it gets its own address space. Meaning that each process gets :

- **A Stack**. The stack is the place where automatic variable and function call return addresses are stored. Every time a new variable is declared, the program moves the stack pointer down to reserve space for the variable. This segment of the stack is Writable but not executable. If the stack grows too far – meaning that it either grows beyond a preset boundary or intersects the heap – you will get a stackoverflow most likely resulting in a SEGFAULT or something similar. **The stack is statically allocated by default meaning that there is only a certain amount of space to which one can write**

- **A Heap**. The heap is an expanding region of memory. If you want to allocate a large object, it goes here. The heap starts at the top of the text segment and grows upward (meaning sometimes when you call `malloc` that it asks the operating system to push the heap boundary upward). This area is also Writable but not Executable. One can run out of heap memory if the system is constrained or if you run out of addresses (more common on a 32bit system).

- **A Data Segment** This contains all of your globals. This section starts at the end of the text segment and is static in size because the amount of globals is known at compile time. There are two areas to the data usually the **IBSS** and the **UBSS** which stand for the initialized basic service set and the uninitialized data segment respectively. This section is Writable but not Executable and there isn't anything else too fancy here.

- **A Text Segment**. This is, arguably, the most important section of the address. This is where all your code is stored. Since assembly compiles to 1's and 0's, this is where the 1's and 0's get stored. The program counter moves through this segment executing instructions and moving down the next instruction. It is important to note that this is the only Executable section of the code. If you try to change the code while it's running, most likely you will segfault (there are ways around it but just assume that it segfaults). * Why doesn't it start at zero? It is outside the scope of this class but it is for security.

Other Contents

To keep track of all these processes, your operating system gives each process a number and that process is called the PID, process ID. Processes also have a `ppid` which is short for parent process id. Every process has a parent, that parent could be `init.d`

Processes could also contain

- Running State - Whether a process is getting ready, running, stopped, terminated etc.

- File Descriptors - List of mappings from integers to real devices (files, usb sticks, sockets)

- Permissions - What `user` the file is running on and what `group` the process belongs to. The process can then only do this admissible to the `user` or `group` like opening a file that the `user` has made exclusives. There are tricks to make a program not be the user who started the program i.e. `sudo` takes a program that a `user` starts and executes it as `root`.

- Arguments - a list of strings that tell your program what parameters to run under * Environment List - a list of strings in the form `NAME=VALUE` that one can modify.

A word of warning

Process forking is a powerful and dangerous tool. If you mess up and cause a fork bomb, **you can bring down the entire system**. To reduce the chances of this, limit your maximum number of processes to a small number e.g 40 by typing `ulimit -u 40` into a command line. Note, this limit is only for the user, which means if you fork bomb, then you won't be able to kill all of the processes you just created since calling `killall` requires your shell to `fork()` ... ironic right? One solution is to spawn another shell instance as another user (for example root) before hand and kill processes from there. Another is to use the built in `exec` command to kill all the user processes (careful you only have one shot at this). Finally you could reboot the system, but you only have one shot at this with the exec function. When testing fork() code, ensure that you have either root and/or physical access to the machine involved. If you must work on fork () code remotely, remember that **kill -9 -1** will save you in the event of an emergency.

TL;DR: Fork can be **extremely** dangerous if you aren't prepared for it. **You have been warned.**

Intro to Fork

What does fork do?

The `fork` system call clones the current process to create a new process. It creates a new process (the child process) by duplicating the state of the existing process with a few minor differences. The child process does not start from main. Instead it executes the next line after the `fork()` just as the parent process does. Just as a side remark, in older UNIX systems, the entire address space of the parent process was directly copied (regardless of whether the resource was modified or not). These days, kernel performs copy-on-write, which saves a lot of resources, while being very time efficient. Here's a very simple example...

```
printf("I'm printed once!\n");
fork();
// Now there are two processes running
// and each process will print out the next line.
printf("You see this line twice!\n");
```

The following program may print out 42 twice - but the `fork()` is after the `printf`!? Why?

```
#include <unistd.h> /*fork declared here*/
#include <stdio.h> /* printf declared here*/
int main() {
   int answer = 84 >> 1;
   printf("Answer: %d", answer);
   fork();
   return 0;
}
```

The `printf` line *is* executed only once however notice that the printed contents is not flushed to standard out. There's no newline printed, we didn't call `fflush`, or change the buffering mode. The

output text is therefore still in process memory waiting to be sent. When `fork()` is executed the entire process memory is duplicated including the buffer. Thus the child process starts with a non-empty output buffer which will be flushed when the program exits.

To write code that is different for the parent and child process, check the return value of `fork()`. If fork() returns -1, that implies something went wrong in the process of creating a new child. One should check the value stored in *errno* to determine what kind of error occurred; commons one include EAGAIN and ENOMEM (check this page to get a description of the errors). Similarly, a return value of 0 indicates that we are in the child process, while a positive integer shows that we are in parent process. The positive value returned by fork() gives as the process id (*pid*) of the child.

One way to remember which is which is that the child process can find its parent - the original process that was duplicated - by calling `getppid()` - so does not need any additional return information from `fork()`. The parent process however can only find out the id of the new child process from the return value of `fork`:

```
pid_t id = fork();
if (id == -1) exit(1); // fork failed
if (id > 0)
{
// I'm the original parent and
// I just created a child process with id 'id'
// Use waitpid to wait for the child to finish
} else {// returned zero
// I must be the newly made child process
}
```

A slightly silly example is shown below. What will it print? Try it with multiple arguments to your program.

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv) {
  pid_t id;
  int status;
  while (--argc && (id=fork())) {
    waitpid(id,&status,0); /* Wait for child*/
  }
  printf("%d:%s\n", argc, argv[argc]);
  return 0;
}
```

The amazing parallel apparent-O(N) *sleepsort* is today's silly winner. First published on 4chan in 2011. A version of this awful but amusing sorting algorithm is shown below.

```c
int main(int c, char **v)
{
        while (--c > 1 && !fork());
        int val = atoi(v[c]);
        sleep(val);
        printf("%d\n", val);
        return 0;
}
```

Note: The algorithm isn't actually O(N) because of how the system scheduler works. Though there are parallel algorithms that run in O(log(N)) per process, this is sadly not one of them.

What is a fork bomb ?

A 'fork bomb' is when you attempt to create an infinite number of processes. This will often bring a system to a near-standstill as it attempts to allocate CPU time and memory to a very large number of processes that are ready to run. Comment: System administrators don't like fork-bombs and may set upper limits on the number of processes each user can have or may revoke login rights because it creates a disturbance in the force for other users' programs. You can also limit the number of child processes created by using `setrlimit()`. fork bombs are not necessarily malicious - they occasionally occur due to student coding errors. Angrave suggests that the Matrix trilogy, where the machine and man finally work together to defeat the multiplying Agent-Smith, was a cinematic plot based on an AI-driven fork-bomb.

```c
while (1) fork();
```

There may even be subtle forkbombs that occur when you are being careless while coding.

```c
#include <unistd.h>
#define HELLO_NUMBER 10

int main(){
    pid_t children[HELLO_NUMBER];
    int i;
    for(i = 0; i < HELLO_NUMBER; i++){
        pid_t child = fork();
        if(child == -1){
            break;
        }
        if(child == 0){ //I am the child
             execlp("ehco", "echo", "hello", NULL);
        }
        else{
            children[i] = child;
        }
    }

    int j;
    for(j = 0; j < i; j++){
        waitpid(children[j], NULL, 0);
    }
    return 0;
}
```

We misspelled `ehco`, so we can't `exec` it. What does this mean? Instead of creating 10 processes we just created 2**10 processes, fork bombing our machine. How could we prevent this? Put an exit right after exec so in case exec fails we won't end up fork bombing our machine.**

Waiting and Execing

If the parent process waits for the child to finish, `waitpid` (or `wait`).

```c
pid_t child_id = fork();
if (child_id == -1) {perror("fork"); exit(EXIT_FAILURE);}
if (child_id > 0) {
  // We have a child! Get their exit code
  int status;
  waitpid( child_id, &status, 0 );
  // code not shown to get exit status from child
} else {// In child ...
  // start calculation
  exit(123);
}
```

Zombies and Orphans

You don't always need to wait for your children! Your parent process can continue to execute code without having to wait for the child process. Note in practice background processes can also be disconnected

from the parent's input and output streams by calling `close` on the open file descriptors before calling exec. However child processes that finish before their parent finishes can become zombies.

If a parent dies without waiting on its children, a process can orphan its children. Once a parent process completes, any of its children will be assigned to "init" - the first process with pid of 1. Thus these children would see getppid() return a value of 1. These orphans will eventually finish and for a brief moment become a zombie. Fortunately, the init process automatically waits for all of its children, thus removing these zombies from the system.

But if the parent is a long running process, the child becomes a zombie. When a child finishes (or terminates) it still takes up a slot in the kernel process table. Furthermore, they still contain information about the process that got terminated, such as process id, exit status, etc. (i.e. a skeleton of the original process still remains). Only when the child has been 'waited on' will the slot be available and the remaining information can be accessed by the parent. A long running program could create many zombies by continually creating processes and never `wait`-ing for them. If you never wait eventually there would be insufficient space in the kernel process table to create a new processes. Thus `fork()` would fail and could make the system difficult / impossible to use - for example just logging in requires a new process! To prevent zombies, Wait on your child!

```
waitpid(child, &status, 0); // Clean up and wait for my child process
    to finish.
```

Note we assume that the only reason to get a SIGCHLD event is that a child has finished (this is not quite true - see man page for more details). A robust implementation would also check for interrupted status and include the above in a loop. Read on for a discussion of a more robust implementation.

Aside

How can I asynchronously wait for my child using SIGCHLD?

Warning: This section uses signals which we have not yet fully introduced. The parent gets the signal SIGCHLD when a child completes, so the signal handler can wait on the process. A slightly simplified version is shown below.

```
pid_t child;

void cleanup(int signal) {
  int status;
  waitpid(child, &status, 0);
  write(1,"cleanup!\n",9);
}
int main() {
   // Register signal handler BEFORE the child can finish
   signal(SIGCHLD, cleanup); // or better - sigaction
   child = fork();
   if (child == -1) {exit(EXIT_FAILURE);}

   if (child == 0) {/* I am the child!*/
     // Do background stuff e.g. call exec
   } else {/* I'm the parent! */
      sleep(4); // so we can see the cleanup
      puts("Parent is done");
   }
   return 0;
}
```

The above example however misses a couple of subtle points: * More than one child may have finished but the parent will only get one SIGCHLD signal (signals are not queued) * SIGCHLD signals can be sent for other reasons (e.g. a child process is temporarily stopped)

A more robust code to reap zombies is shown below.

```
void cleanup(int signal) {
  int status;
  while (waitpid((pid_t) (-1), 0, WNOHANG) > 0) {}
}
```

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]:

```
/* 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.

exec

To make the child process execute another program, use one of the `exec` functions after forking. The `exec` set of functions replaces the process image with the the process image of what is being called. This means that any lines of code after the `exec` call are replaced. Any other work you want the child process to do should be done before the `exec` call. The Wikipedia article does a great job helping you make sense of the names of the exec family. The naming schemes can be shortened mnemonically.

e – An array of pointers to environment variables is explicitly passed to the new process image.

l – Command-line arguments are passed individually (a list) to the function.

p – Uses the PATH environment variable to find the file named in the file argument to be executed.

v – Command-line arguments are passed to the function as an array (vector) of pointers.

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char**argv) {
  pid_t child = fork();
  if (child == -1) return EXIT_FAILURE;
  if (child) {/* I have a child! */
    int status;
    waitpid(child , &status ,0);
    return EXIT_SUCCESS;

  } else {/* I am the child */
    // Other versions of exec pass in arguments as arrays
    // Remember first arg is the program name
    // Last arg must be a char pointer to NULL

    execl("/bin/ls", "ls","-alh", (char *) NULL);

    // If we get to this line, something went wrong!
    perror("exec failed!");
  }
}
```

```c
#include <unistd.h>
#include <fcntl.h> // O_CREAT, O_APPEND etc. defined here

int main() {
   close(1); // close standard out
   open("log.txt", O_RDWR | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR);
   puts("Captain's log");
   chdir("/usr/include");
   // execl( executable, arguments for executable including program
       name and NULL at the end)

   execl("/bin/ls", /* Remaining items sent to ls*/ "/bin/ls", ".",
       (char *) NULL); // "ls ."
   perror("exec failed");
   return 0; // Not expected
}
```

There's no error checking in the above code (we assume close,open,chdir etc works as expected). * open: will use the lowest available file descriptor (i.e. 1) ; so standard out now goes to the log file. * chdir : Change the current directory to /usr/include * execl : Replace the program image with /bin/ls and call its main() method * perror : We don't expect to get here - if we did then exec failed.

`system` pre-packs the above code. Here is how to use it:

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

int main(int argc, char**argv) {
  system("ls"); // execl("/bin/sh", "/bin/sh", "-c", "\"ls\"")
  return 0;
}
```

The `system` call will fork, execute the command passed by parameter and the original parent process will wait for this to finish. This also means that `system` is a blocking call: The parent process can't continue until the process started by `system` exits. This may or may not be useful. Also, `system` actually creates a shell which is then given the string, which is more overhead than just using `exec` directly. The standard shell will use the `PATH` environment variable to search for a filename that matches the command. Using system will usually be sufficient for many simple run-this-command problems but can quickly become limiting for more complex or subtle problems, and it hides the mechanics of the fork-exec-wait pattern so we encourage you to learn and use `fork` `exec` and `waitpid` instead.

Differences/Similarities Between Child Processes

Key differences include:

- The process id returned by `getpid()`. The parent process id returned by `getppid()`.
- The parent is notified via a signal, SIGCHLD, when the child process finishes but not vice versa.
- The child does not inherit pending signals or timer alarms. For a complete list see the fork man page
- The child has its own set of environment variables

Key smiliraties include:

- Both processes use the same underlying kernel file descriptor. For example if one process rewinds the random access position back to the beginning of the file, then both processes are affected. Both child and parent should `close` (or `fclose`) their file descriptors or file handle respectively.
- Since we have copy on write, read-only memory addresses are shared between processes
- If you set up certain regions of memory, they are shared between processes.
- Signal handlers are inherited but can be changed
- Current working directory is inherited but can be changed
- Environment variables are inherited but can be changed

fork man page

The fork-exec-wait Pattern

A common programming pattern is to call `fork` followed by `exec` and `wait`. The original process calls fork, which creates a child process. The child process then uses exec to start execution of a new program. Meanwhile the parent uses `wait` (or `waitpid`) to wait for the child process to finish.

```c
#include <unistd.h>

int main() {
  pid_t pid = fork();
  if (pid < 0) {// fork failure
    exit(1);
  } else if (pid > 0) {// I am the parent
    int status;
    waitpid(pid, &status, 0);
  } else {// I am the child
    execl("/bin/ls", "/bin/ls", NULL);
    exit(1);
  }
}
```

So what are environment variables?

Environment variables are variables that the system keeps for all processes to use. Your system has these set up right now! In Bash, you can check some of these

```
$ echo $HOME
/home/bhuvy
$ echo $PATH
/usr/local/sbin:/usr/bin:...
```

How would you get these in C/C++? You can use the `getenv` and `setenv` function

```c
char* home = getenv("HOME"); // Will return /home/bhuvy
setenv("HOME", "/home/bhuvan", 1 /*set overwrite to true*/ );
```

Further Reading

Read the man pages!

- fork
- exec
- wait

Topics

- Correct use of fork, exec and waitpid
- Using exec with a path
- Understanding what fork and exec and waitpid do. E.g. how to use their return values.
- SIGKILL vs SIGSTOP vs SIGINT.
- What signal is sent when you press CTRL-C
- Using kill from the shell or the kill POSIX call.
- Process memory isolation.
- Process memory layout (where is the heap, stack etc; invalid memory addresses).
- What is a fork bomb, zombie and orphan? How to create/remove them.
- getpid vs getppid
- How to use the WAIT exit status macros WIFEXITED etc.

Questions/Exercises

- What is the difference between execs with a p and without a p? What does the operating system
- How do you pass in command line arguments to `execl*`? How about `execv*`? What should be the first command line argument by convention?
- How do you know if `exec` or `fork` failed?
- What is the `int *status` pointer passed into wait? When does wait fail?
- What are some differences between `SIGKILL`, `SIGSTOP`, `SIGCONT`, `SIGINT`? What are the default behaviors? Which ones can you set up a signal handler for?
- What signal is sent when you press `CTRL-C`?
- My terminal is anchored to PID = 1337 and has just become unresponsive. Write me the terminal command and the C code to send `SIGQUIT` to it.
- Can one process alter another processes memory through normal means? Why?
- Where is the heap, stack, data, and text segment? Which segments can you write to? What are invalid memory addresses?
- Code me up a fork bomb in C (please don't run it).
- What is an orphan? How does it become a zombie? How do I be a good parent?
- Don't you hate it when your parents tell you that you can't do something? Write me a program that sends `SIGSTOP` to your parent.
- Write a function that fork exec waits an executable, and using the wait macros tells me if the process exited normally or if it was signaled. If the process exited normally, then print that with the return value. If not, then print the signal number that caused the process to terminate.

Bibliography

[1] Source to sys/wait.h. URL http://unix.superglobalmegacorp.com/Net2/newsrc/sys/wait.h.html.

Chapter 4

Memory Allocators

TODO: Epigraph

TODO: Introduction

Memory allocation is very important! Allocating and de-allocating heap memory is a common operation in most applications.

C Memory Allocation Functions

- `malloc(size_t bytes)` is a C library call and is used to reserve a contiguous block of memory. Unlike stack memory, the memory remains allocated until `free` is called with the same pointer. If `malloc` fails to reserve any more memory then it returns `NULL`. Robust programs should check the return value. If your code assumes `malloc` succeeds and it does not, then your program will likely crash (segfault) when it tries to write to address 0. Also, malloc may not zero out memory because of performance – check your code to make sure that you are not using unitialized values.

- `realloc(void *space, size_t bytes)` allows you to resize an existing memory allocation that was previously allocated on the heap (via malloc, calloc, or realloc). The most common use of realloc is to resize memory used to hold an array of values. There are two gotchas with realloc. One, a new pointer may be returned. Two, it can fail. A naive but readable version of realloc is suggested below.

```
void * realloc(void * ptr, size_t newsize) {
  // Simple implementation always reserves more memory
  // and has no error checking
  void *result = malloc(newsize);
  size_t oldsize = ... //(depends on allocator's internal data
      structure)
  if (ptr) memcpy(result, ptr, newsize < oldsize ? newsize :
      oldsize);
  free(ptr);
  return result;
}
```

```
// 1
int *array = malloc(sizeof(int) * 2);
array[0] = 10; array[1] = 20;
// Ooops need a bigger array - so use realloc..
realloc (array, 3); // ERRORS!
array[2] = 30;

array = realloc(array, 3 * sizeof(int));
// ...
```

- `calloc(size_t nmemb, size_t size)` initializes memory contents to zero and also takes two arguments: the number of items and the size in bytes of each item. An advanced discussion of these limitations is here. Programmers often use `calloc` rather than explicitly calling `memset` after `malloc`, to set the memory contents to zero. Note `calloc(x,y)` is identical to `calloc(y,x)`, but you should follow the conventions of the manual. A naive implementation of calloc is below.

```
void *calloc(size_t n, size_t size)
{
    size_t total = n * size; // Does not check for overflow!
    void *result = malloc(total);

    if (!result) return NULL;

// If we're using new memory pages
// just allocated from the system by calling sbrk
// then they will be zero so zero-ing out is unnecessary,

    memset(result, 0, total);
    return result;
}
```

- `free` takes a pointer to the start of a piece of memory and makes it available for use in the subsequent calls to the other allocation functions. This is important because we don't want every process in our address space to take an enormous amount of memory. Once we are done using memory, we stop using it with free. A simple usage is below.

```
int *ptr = malloc(sizeof(*ptr));
do_something(ptr);
free(ptr);
```

Heaps and sbrk

The heap is part of the process memory and it does not have a fixed size. Heap memory allocation is performed by the C library when you call `malloc` (`calloc`, `realloc`) and `free`. By calling `sbrk` the

C library can increase the size of the heap as your program demands more heap memory. As the heap and stack (one for each thread) need to grow, we put them at opposite ends of the address space. So for typical architectures the heap will grow upwards and the stack grows downwards.

Truthiness: Modern operating system memory allocators no longer need `sbrk` - instead they can request independent regions of virtual memory and maintain multiple memory regions. For example gigabyte requests may be placed in a different memory region than small allocation requests. However this detail is an unwanted complexity: The problems of fragmentation and allocating memory efficiently still apply, so we will ignore this implementation nicety here and will write as if the heap is a single region. If we write a multi-threaded program (more about that later) we will need multiple stacks (one per thread) but there's only ever one heap. On typical architectures, the heap is part of the `Data segment` and starts just above the code and global variables.

Programs don't need to call brk or sbrk typically (though calling `sbrk(0)` can be interesting because it tells you where your heap currently ends). Instead programs use `malloc,calloc,realloc` and `free` which are part of the C library. The internal implementation of these functions will call `sbrk` when additional heap memory is required.

```
void *top_of_heap = sbrk(0);
malloc(16384);
void *top_of_heap2 = sbrk(0);
printf("The top of heap went from %p to %p \n", top_of_heap,
    top_of_heap2);
// Example output: The top of heap went from 0x4000 to 0xa000
```

If the operating system did not zero out contents of physical RAM it might be possible for one process to learn about the memory of another process that had previously used the memory. This would be a security leak. Unfortunately this means that for `malloc` requests before any memory has been freed and simple programs (which end up using newly reserved memory from the system) the memory is *often* zero. Then programmers mistaken write C programs that assume malloc'd memory will *always* be zero.

```
char* ptr = malloc(300);
// contents is probably zero because we get brand new memory
// so beginner programs appear to work!
// strcpy(ptr, "Some data"); // work with the data
free(ptr);
// later
char *ptr2 = malloc(308); // Contents might now contain existing data
    and is probably not zero
```

Intro to Allocating

```
void* malloc(size_t size)
// Ask the system for more bytes by extending the heap space.
// sbrk Returns -1 on failure
   void *p = sbrk(size);
   if(p == (void *) -1) return NULL; // No space left
   return p;
}
void free() {/* Do nothing */}
```

Above is the simpliest implementation of malloc, there are a few drawbacks though.

- System calls are slow (compared to library calls). We should reserve a large amount of memory and only occasionally ask for more from the system.
- No reuse of freed memory. Our program never re-uses heap memory - it just keeps asking for a bigger heap.

If this allocator was used in a typical program, the process would quickly exhaust all available memory. Instead we need an allocator that can efficiently use heap space and only ask for more memory when necessary.

Placement Strategies

During program execution, memory is allocated and de-allocated (freed), so there will be gaps (holes) in the heap memory that can be re-used for future memory requests. The memory allocator needs to keep track of which parts of the heap are currently allocated and which are parts are available. Suppose our current heap size is 64K, though not all of it is in use because some earlier malloc'd memory has already been freed by the program:

16KB free	10KB allocated	1KB free	1KB allocated	30KB free	4KB allocated	2KB free

If a new malloc request for 2KB is executed (`malloc(2048)`), where should `malloc` reserve the memory? It could use the last 2KB hole (which happens to be the perfect size!) or it could split one of the other two free holes. These choices represent different placement strategies. Whichever hole is chosen, the allocator will need to split the hole into two: The newly allocated space (which will be returned to the program) and a smaller hole (if there is spare space left over). A perfect-fit strategy finds the smallest hole that is of sufficient size (at least 2KB):

16KB free	10KB allocated	1KB free	1KB allocated	30KB free	4KB allocated	2KB `HERE!`

A worst-fit strategy finds the largest hole that is of sufficient size (so break the 30KB hole into two):

16KB free	10KB allocated	1KB free	1KB allocated	2KB `HERE!`	28KB free	4KB allocated	2KB free

A first-fit strategy finds the first available hole that is of sufficient size (break the 16KB hole into two):

2KB HERE!	14KB free	10KB allocated	1KB free	1KB allocated	30KB free	4KB allocated	2KB free

External fragmentation is that even though we have enough memory in the heap, it may be divided up in a way that wear are not able to give the full amount. In the example below, of the 64KB of heap memory, 17KB is allocated, and 47KB is free. However the largest available block is only 30KB because our available unallocated heap memory is fragmented into smaller pieces.

16KB free	10KB allocated	1KB free	1KB allocated	30KB free	4KB allocated	2KB free

What effect do placement strategies have on external fragmentation and performance?

Different strategies affect the fragmentation of heap memory in non-obvious ways, which only are discovered by mathematical analysis or careful simulations under real-world conditions (for example simulating the memory allocation requests of a database or webserver). For example, best-fit at first glance appears to be an excellent strategy however, if we can not find a perfectly-sized hole then this placement creates many tiny unusable holes, leading to high fragmentation. It also requires a scan of all possible holes.

First fit has the advantage that it will not evaluate all possible placements and therefore be faster.

Since Worst-fit targets the largest unallocated space, it is a poor choice if large allocations are required.

In practice first-fit and next-fit (which is not discussed here) are often common placement strategy. Hybrid approaches and many other alternatives exist (see implementing a memory allocator page).

The challenges of writing a heap allocator are

- Need to minimize fragmentation (i.e. maximize memory utilization)
- Need high performance
- Fiddly implementation (lots of pointer manipulation using linked lists and pointer arithmetic)
- Both fragmentation and performance depend on the application allocation profile, which can be evaluated but not predicted and in practice, under-specific usage conditions, a special-purpose allocator can often out-perform a general purpose implementation.
- The allocator doesn't know the program's memory allocation requests in advance. Even if we did, this is the Knapsack problem which is known to be NP hard!

Memory Allocator Tutorial

A memory allocator needs to keep track of which bytes are currently allocated and which are available for use. This page introduces the implementation and conceptual details of building an allocator, i.e. the actual code that implements `malloc` and `free`.

Conceptually we are thinking about creating linked lists and lists of blocks! We are writing integers and pointers into memory that we already controlm so we can later consistently hop from one address to the next. This internal information represents some overhead. Even if we had requested 1024 KB of contiguous memory from the system, we will not be able to provide all of it to the running program.

We can think of our heap memory as a list of blocks where each block is either allocated or unallocated. Rather than storing an explicit list of pointers we store information about the block's size *as part of the block*. Thus there is conceptually a list of free blocks, but it is implicit, i.e. in the form of block size information that we store as part of each block.

We could navigate from one block to the next block just by adding the block's size. For example if you have a pointer `p` that points to the start of a block, then `next_block` with be at `((char *)p) + *(size_t *) p`, if you are storing the size of the blocks in bytes. The cast to `char *` ensures that pointer arithmetic is calculated in bytes. The cast to `size_t *` ensures the memory at `p` is read as a size value and would be necessarily if `p` was a `void *` or `char *` type.

The calling program never sees these values; they are internal to the implementation of the memory allocator. As an example, suppose your allocator is asked to reserve 80 bytes (`malloc(80)`) and requires 8 bytes of internal header data. The allocator would need to find an unallocated space of at least 88 bytes. After updating the heap data it would return a pointer to the block. However, the returned pointer does not point to the start of the block because that's where the internal size data is stored! Instead we would return the start of the block + 8 bytes. In the implementation, remember that pointer arithmetic depends on type. For example, `p += 8` adds `8 * sizeof(p)`, not necessarily 8 bytes!

Implementing malloc

The simplest implementation uses first fit: Start at the first block, assuming it exists, and iterate until a block that represents unallocated space of sufficient size is found, or we've checked all the blocks.

If no suitable block is found, it's time to call `sbrk()` again to sufficiently extend the size of the heap. A fast implementation might extend it a significant amount so that we will not need to request more heap memory in the near future.

When a free block is found, it may be larger than the space we need. If so, we will create two entries in our implicit list. The first entry is the allocated block, the second entry is the remaining space. There are two simple ways to note if a block is in use or available. The first is to store it as a byte in the header information along with the size and the second to encode it as the lowest bit in the size! Thus block size information would be limited to only even values:

```
// Assumes p is a reasonable pointer type, e.g. 'size_t *'.
isallocated = (*p) & 1;
realsize = (*p) & ~1; // mask out the lowest bit
```

Alignment and rounding up considerations

Many architectures expect multi-byte primitives to be aligned to some multiple of 2^n. For example, it's common to require 4-byte types to be aligned to 4-byte boundaries (and 8-byte types on 8-byte boundaries). If multi-byte primitives are not stored on a reasonable boundary (for example starting at an odd address) then the performance can be significantly impacted because it may require two memory read requests instead of one. On some architectures the penalty is even greater - the program will crash with a bus error.

As `malloc` does not know how the user will use the allocated memory (array of doubles? array of chars?), the pointer returned to the program needs to be aligned for the worst case, which is architecture dependent.

From glibc documentation, the glibc `malloc` uses the following heuristic: " The block that malloc gives you is guaranteed to be aligned so that it can hold any type of data. On GNU systems, the address

is always a multiple of eight on most systems, and a multiple of 16 on 64-bit systems." For example, if you need to calculate how many 16 byte units are required, don't forget to round up.

```
int s = (requested_bytes + tag_overhead_bytes + 15) / 16
```

The additional constant ensures incomplete units are rounded up. Note, real code is more likely to symbol sizes e.g. `sizeof(x) - 1`, rather than coding numerical constant 15.

Here's a great article on memory alignment, if you are further interested ## A note about internal fragmentation

Internal fragmentation happens when the block you give them is larger than their allocation size. Let's say that we have a free block of size 16B (not including metadata). If they allocate 7 bytes, you may want to round up to 16B and just return the entire block. This gets very sinister when you implementing coalescing and splitting (next section). If you don't implement either, then you may end up returning a block of size 64B for a 7B allocation! There is a *lot* of overhead for that allocation which is what we are trying to avoid.

Implementing free

When `free` is called we need to re-apply the offset to get back to the 'real' start of the block (remember we didn't give the user a pointer to the actual start of the block?), i.e. to where we stored the size information.

A naive implementation would simply mark the block as unused. If we are storing the block allocation status in the lowest size bit, then we just need to clear the bit:

```
*p = (*p) & ~1; // Clear lowest bit
```

However, we have a bit more work to do: If the current block and the next block (if it exists) are both free we need to coalesce these blocks into a single block. Similarly, we also need to check the previous block, too. If that exists and represents an unallocated memory, then we need to coalesce the blocks into a single large block.

To be able to coalesce a free block with a previous free block we will also need to find the previous block, so we store the block's size at the end of the block, too. These are called "boundary tags" (ref Knuth73). As the blocks are contiguous, the end of one blocks sits right next to the start of the next block. So the current block (apart from the first one) can look a few bytes further back to lookup the size of the previous block. With this information you can now jump backwards!

Performance

With the above description it's possible to build a memory allocator. It's main advantage is simplicity - at least simple compared to other allocators! Allocating memory is a worst-case linear time operation (search linked lists for a sufficiently large free block) and de-allocation is constant time (no more than 3 blocks will need to be coalesced into a single block). Using this allocator it is possible to experiment with different placement strategies. For example, you could start searching from where you last free'd a block, or where you last allocated from. If you do store pointers to blocks, you need to be very careful that they always remain valid (e.g. when coalescing blocks or other malloc or free calls that change the heap structure)

Explicit Free Lists Allocators

Better performance can be achieved by implementing an explicit doubly-linked list of free nodes. In that case, we can immediately traverse to the next free block and the previous free block. This can halve the search time, because the linked list only includes unallocated blocks. A second advantage is that we now have some control over the ordering of the linked list. For example, when a block is free'd, we could choose to insert it into the beginning of the linked list rather than always between its neighbors. This is discussed below.

Where do we store the pointers of our linked list? A simple trick is to realize that the block itself is not being used and store the next and previous pointers as part of the block (though now you have to ensure that the free blocks are always sufficiently large to hold two pointers). We still need to implement Boundary Tags (i.e. an implicit list using sizes), so that we can correctly free blocks and coalesce them with their two neighbors. Consequently, explicit free lists require more code and complexity. With explicit linked lists a fast and simple 'Find-First' algorithm is used to find the first sufficiently large link. However, since the link order can be modified, this corresponds to different placement strategies. For example if the links are maintained from largest to smallest, then this produces a 'Worst-Fit' placement strategy.

Explicit linked list insertion policy

The newly free'd block can be inserted easily into two possible positions: at the beginning or in address order (by using the boundary tags to first find the neighbors).

Inserting at the beginning creates a LIFO (last-in, first-out) policy: The most recently free'd spaces will be reused. Studies suggest fragmentation is worse than using address order.

Inserting in address order ("Address ordered policy") inserts free'd blocks so that the blocks are visited in increasing address order. This policy required more time to free a block because the boundary tags (size data) must be used to find the next and previous unallocated blocks. However, there is less fragmentation.

Case study: Buddy Allocator (an example of a segregated list)

A segregated allocator is one that divides the heap into different areas that are handled by different sub-allocators dependent on the size of the allocation request. Sizes are grouped into classes (e.g. powers of two) and each size is handled by a different sub-allocator and each size maintains its own free list.

A well known allocator of this type is the buddy allocator [1, P. 85]. We'll discuss the binary buddy allocator which splits allocation into blocks of size 2ˆn (n = 1, 2, 3, ...) times some base unit number of bytes, but others also exist (e.g. Fibonacci split - can you see why it's named?). The basic concept is simple: If there are no free blocks of size 2ˆn, go to the next level and steal that block and split it into two. If two neighboring blocks of the same size become unallocated, they can be coalesced back together into a single large block of twice the size.

Buddy allocators are fast because the neighboring blocks to coalesce with can be calculated from the free'd block's address, rather than traversing the size tags. Ultimate performance often requires a small amount of assembler code to use a specialized CPU instruction to find the lowest non-zero bit.

The main disadvantage of the Buddy allocator is that they suffer from *internal fragmentation*, because allocations are rounded up to the nearest block size. For example, a 68-byte allocation will require a 128-byte block.

Further Reading

There are many other allocation schemes. One of three allocators used internally by the Linux Kernel. See the man page!

- SLUB (wikipedia)
- See Foundations of Software Technology and Theoretical Computer Science 1999 proceedings (Google books,page 85)
- Wikipedia's buddy memory allocation page

Topics

- Best Fit
- Worst Fit
- First Fit
- Buddy Allocator
- Internal Fragmentation
- External Fragmentation
- sbrk
- Natural Alignment
- Boundary Tag
- Coalescing
- Splitting
- Slab Allocation/Memory Pool

Questions/Exercises

- What is Internal Fragmentation? When does it become an issue?
- What is External Fragmentation? When does it become an issue?
- What is a Best Fit placement strategy? How is it with External Fragmentation? Time Complexity?
- What is a Worst Fit placement strategy? Is it any better with External Fragmentation? Time Complexity?
- What is the First Fit Placement strategy? It's a little bit better with Fragmentation, right? Expected Time Complexity?
- Let's say that we are using a buddy allocator with a new slab of 64kb. How does it go about allocating 1.5kb?
- When does the 5 line `sbrk` implementation of malloc have a use?
- What is natural alignment?
- What is Coalescing/Splitting? How do they increase/decrease fragmentation? When can you coalesce or split?
- How do boundary tags work? How can they be used to coalesce or split?

Bibliography

[1] C.P. Rangan, V. Raman, and R. Ramanujam. *Foundations of Software Technology and Theoretical Computer Science: 19th Conference, Chennai, India, December 13-15, 1999 Proceedings*. FOUNDATIONS OF SOFTWARE TECHNOLOGY AND THEORETICAL COMPUTER SCIENCE. Springer, 1999. ISBN 9783540668367. URL https://books.google.com/books?id=0uHME7EfjQEC.

Chapter 5

Interprocess Communication

TODO: Epigraph

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

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

MMU and Translating Addresses

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

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

4. Otherwise,
    (a) If the page exists in memory, check if the process has permissions to perform the operation on the page meaning the process has access to the page, and it is reading from the page/writing to a page that is not marked as read only.
        i. If so then provide the address, cache the results in the TLB
        ii. Otherwise trigger a hardware interrupt. The kernel will most likely send a SIGSEGV or a Segmentation Violation.
    (b) If the page doesn't exist in memory, generate an Interrupt.
        i. The kernel could realize that this page could either be not allocated or on disk. If it fits the mapping, allocate the page and try the operation again.
        ii. Otherwise, this is an invalid access and the kernel will most likely send a SIGSEGV to the process.

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

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

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

Terminology

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

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

A simple page table could be imagined as an array.

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

For a 32 bit machine with 4KB pages, each entry needs to hold a frame number - i.e. 20 bits because we calculated there are 2^{20} frames. That's 2.5 bytes per entry! In practice, we'll round that up to 4 bytes per entry and find a use for those spare bits. With 4 bytes per entry x 2^{20} entries = 4 MB of physical memory are required to hold the page table For a 64 bit machine with 4KB pages, each entry needs 52 bits. Let's round up to 64 bits (8 bytes) per entry. With 2^{52} entries thats 2^{55} bytes (roughly 40 peta bytes...) Oops our page table is too large In 64 bit architectures memory addresses are sparse, so we need a mechanism to reduce the page table size, given that most of the entries will never be used.

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

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

Multi-level page tables

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

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

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

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

Page Table Disadvantages

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

To overcome this overhead, the MMU includes an associative cache of recently-used virtual-page-to-frame lookups. This cache is called the TLB ("translation lookaside buffer"). Everytime a virtual address needs to be translated into a physical memory location, the TLB is queried in parallel to the page table. For most memory accesses of most programs, there is a significant chance that the TLB has cached the

results. However if a program does not have good cache coherence (for example is reading from random memory locations of many different pages) then the TLB will not have the result cache and now the MMU must use the much slower page table to determine the physical frame.

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

Advanced Frames and Page Protections

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

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

Read-only bit

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

Dirty bit

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

Execution bit

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

Page Faults

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

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

2. **Major** If the mapping to the page is not in memory but on disk. What this will do is swap the page into memory and swap another page out. If this happens frequently enough, your program is said to *thrash* the MMU.

3. **Invalid** When you try to write to a non-writable memory address or read to a non-readable memory address. The MMU generates an invalid fault and the OS will usually generate a `SIGSEGV` meaning segmentation violation meaning that you wrote outside the segment that you could write to.

Pipes

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

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

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

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

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

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

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

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

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

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

    return 0;
}
```

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

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

Pipe Gotchas

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Why is my pipe hanging?

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

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

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

Race condition with named pipes

What is wrong with the following program?

```
//Program 1

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

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

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

1. Process 1: open(O_RDWR) & write()
2. Process 2: open(O_RDONLY) & read()
3. Process 1: close() & exit()
4. Process 2: print() & exit()

1. Process 1: open(O_RDWR) & write()
2. Process 1: close() & exit()
3. Process 2: open(O_RDONLY) (Blocks indefinitely)

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

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

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

Are pipes process safe?

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

The lifetime of pipes

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

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

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

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

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

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

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

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

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

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

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

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

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

When do I need two pipes?

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

Named Pipes

How do I create named pipes?

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

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

You give it the path name and the operation mode, it will be ready to go! Named pipes take up no

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

Two types of files

On linux, there are two abstractions with files. The first is the linux `fd` level abstraction.

- `open` takes a path to a file and creates a file descriptor entry in the process table. If the file is not available to you, it errors out.
- `read` takes a number of bytes that the kernel has received and reads them into a user space buffer. If the file is not open in read mode, this will break.
- `write` outputs a number of bytes to a file descriptor. If the file is not open in write mode, this will break. This may be buffered internally.
- `close` removes a file descriptor from a process' file descriptors. This always succeeds on a valid file descriptor.
- `lseek` takes a file descriptor and moves it to a certain position. Can fail if the seek is out of bounds.
- `fcntl` is the catch all function for file descriptors. You can do everything with this function. Set file locks, read, write, edit permissions, etc ...
- ...

And so on. The linux interface is very powerful and expressive, but sometimes we need portability (for example if we are writing for a mac or windows). This is where C's abstraction comes into play. On different operating systems, C uses the low level functions to create a wrapper around files you can use everywhere, meaning that C on linux uses the above calls.

- `fopen` opens a file and returns an object. `null` is returned if you don't have permission for the file.
- `fread` reads a certain number of bytes from a file. An error is returned if already at the end of file when which you must call `feof()` in order to check.
- `fgetc/fgets`
- `fscanf`
- `fwrite`
- `fprintf`
- `fclose`
- `fflush`

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

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

How do I tell how large a file is?

For files less than the size of a long, using fseek and ftell is a simple way to accomplish this:
Move to the end of the file and find out the current position.

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

This tells us the current position in the file in bytes - i.e. the length of the file!
`fseek` can also be used to set the absolute position.

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

All future reads and writes in the parent or child processes will honor this position. Note writing or reading from the file will change the current position.
See the man pages for fseek and ftell for more information.

But try not to do this

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

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

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

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

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

How about mmap for files?

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

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

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

For every mmap

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

Bibliography

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

Chapter 6

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)

Signals have been a unix construct since the beginning. They are a convenient way to deliver low-priority information and for users to interact with their programs when no other form of interaction is available like using standard input. Signals allow a program to cleanup or perform an action in the case of an event. Some time, a program can choose to ignore events and that is completely fine and even supported by the standard. Crafting a program that uses signals well is tricky due ot all the rules with inheritance. As such, signals are usually kept as cleanup or termination measures.

This chapter will go over how to first read information from a process that has either exited or been signaled and then it will deep dive into what are signals, how does the kernel deal with a signal, and the various ways processes can handle signals both in a single and multithreaded way.

The Deep Dive of 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,

```c
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 [3]. 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 Sigmask Code[2]

```c
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 [1]. 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_kill(threadid, SIGKILL)` will kill the entire process. Though individual threads can set a signal mask, the signal disposition (the table of handlers/action performed for each signal) is *per-process* not *per-thread*. This means `sigaction` can be called from any thread because you will be setting a signal handler for all threads in the process.

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] Executing a file. URL https://www.gnu.org/software/libc/manual/html_node/Executing-a-File.html#Executing-a-File.

[2] pthread$_sigmask.URL$.

Jure Åăorn. gto76/comp-cpp, Jun 2015. URL https://github.com/gto76/comp-cpp/blob/1bf9a77eaf8f57f7358a316e5bbada97f2dc8987/src/output.c.

Chapter 7

Appendix

The Hitchhiker's Guide to Debugging C Programs

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

In-Code Debugging

Clean code

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

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

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

}
```

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

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

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

Asserts!

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

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

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

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

printfs

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

Valgrind

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

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

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

```c
#include <stdlib.h>

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

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

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

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

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

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

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

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

TSAN

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

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

int global;

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

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

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

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

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

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

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

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

GDB

Introduction to gdb

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

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

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

Checking memory content Memory Content

For example,

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

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

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

(gdb)
```

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

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

Shell

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

```
(Stuff) $
```

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

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

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

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

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

Shell tricks and tips

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

Alright then what's a terminal?

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

Common Utilities

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

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

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

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

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

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

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

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

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

Syntactic

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

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

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

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

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

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

TODO: Shell Tricks

Tips and tricks

TODO: Shell Tricks

What are environment variables?

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

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

Stack Smashing

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

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

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

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

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

    return 0;
}
```

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

System Programming Jokes

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

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

Light bulb jokes

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

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

A. None they prefer an empty socket.

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

Groaners

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

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

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

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

System Programmer (Definition)

A system programmer is...

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

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

Someone who believes zombies are real.

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

A system program ...

Evolves until it can send email.

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

Glossary

C A multipurpose programming language. 7

Linux Kernel A widely used operating system kernel. 7

Portable Works on multiple operating systems or machines. 7

POSIX Portable Operating System Interface, a set of standard defined by IEEE for an operating system. 7