CS 241 Wikibook Project

University of Illinois at Urbana-Champaign

Contents

1 Introduction

Hey Everyone! Thanks for checking out the wikibook-project. This project is very much still in beta but has been released to everyone, so we can iterate on it this semester and fill in any gaps for the coming semesters. A few things to note

1. You'll be responsible for learning everything in the original wikibook. This is meant as a supplement for that to understand concepts and read more as a book
2. This book strives for being more readable than compatible with the lectures, meaning that one lecture can go over many different sections of the book at various levels of resolution. The book is meant to be read through and make sense on a read through.
3. This is still a wikibook, if you have thoughts, comments, suggestions, and fixes: please let us know!

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.

```
#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.

```
#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.

```
#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.

```
#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.

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

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

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

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

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

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

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

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

```c
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.

```c
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.

```c
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.

```c
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

```
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++));
```

```
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.

```
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.

```
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

```c
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.

```c
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.

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

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

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

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

```
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.

```
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)

```c
// 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.

```c
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.

```
// 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.

```
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.

```c
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?

```
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.

```
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).

```
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.

```
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 |
                                                       âǍ¿
*/

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.

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

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

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

If I want to do

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

Think

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

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

What is a void pointer?

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

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

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

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

Shell

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`.

3 Processes

Who needs process isolation?

Intel Marketing on Meltdown and Spectre

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.

```c
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.

```
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.

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

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

```
#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`).

```
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.

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;
}
```

```c
// 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.

```c
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.

```c
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 2^n (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 2^n, 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`.

5 Threads

TODO: Epigraph

A thread is short for 'thread-of-execution'. It represents the sequence of instructions that the CPU has (and will) execute. To remember how to return from function calls, and to store the values of automatic variables and parameters a thread uses a stack. A thread is a process (meaning that creating a thread is similar to `fork`) except there is **no copying** meaning no copy on write. What this allows is for a process to share the same address space, variables, heap, file descriptors and etc. The actual system call to create a thread is similar to `fork`; it's `clone`. We won't go into the specifics but you can read the man pages keeping in mind that it is outside the direct scope of this course. LWP or threads are preferred to forking for a lot of scenarios because there is a lot less overhead creating them. But in some cases (notably python uses this) multiprocessing is the way to make your code faster.

Processes vs threads

Creating separate processes is useful when

- When more security is desired. For example, Chrome browser uses different processes for different tabs.
- When running an existing and complete program then a new process is required (e.g. starting 'gcc')
- When you are running into synchronization primitives and each process is operating on something in the system.
- When you have too many threads – the kernel tries to schedule all the threads near each other which could cause more harm than good.
- When you don't want to worry about race conditions
- If one thread blocks in a task (say IO) then all threads block. Processes don't have that same restriction.
- When the amount of communication is minimal enough that simple IPC needs to be used.

On the other hand, creating threads is more useful when

- You want to leverage the power of a multi-core system to do one task
- When you can't deal with the overhead of processes
- When you want communication between the processes simplified
- When you want to threads to be part of the same process

Thread Internals

Your main function (and other functions you might call) has automatic variables. We will store them in memory using a stack and keep track of how large the stack is by using a simple pointer (the "stack pointer"). If the thread calls another function, we move our stack pointer down, so that we have more space for parameters and automatic variables. Once it returns from a function, we can move the stack pointer back up to its previous value. We keep a copy of the old stack pointer value - on the stack! This is why returning from a function is very quick - it's easy to 'free' the memory used by automatic variables - we just need to change the stack pointer.

In a multi threaded program, there are multiple stack but only one address space. The pthread library allocates some stack space (either in the heap or using a part of the main program's stack) and uses the `clone` function call to start the thread at that stack address.

Figure 5.1: Thread address space

How many threads can my process have?

You can have more than one thread running inside a process. You get the first thread for free! It runs the code you write inside 'main'. If you need more threads you can call `pthread_create` to create a new thread using the pthread library. You'll need to pass a pointer to a function so that the thread knows where to start.

The threads you create all live inside the same virtual memory because they are part of the same process. Thus they can all see the heap, the global variables and the program code etc. Thus you can have two (or more) CPUs working on your program at the same time and inside the same process. It's up to the operating system to assign the threads to CPUs. If you have more active threads than CPUs then the kernel will assign the thread to a CPU for a short duration (or until it runs out of things to do) and then will automatically switch the CPU to work on another thread. For example, one CPU might be processing the game AI while another thread is computing the graphics output.

Simple Usage

To use pthreads you will need to include `pthread.h` and compile with `-pthread` (or `-lpthread`) compiler option. This option tells the compiler that your program requires threading support. To create a thread use the function `pthread_create`. This function takes four arguments:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

- The first is a pointer to a variable that will hold the id of the newly created thread.
- The second is a pointer to attributes that we can use to tweak and tune some of the advanced features of pthreads.
- The third is a pointer to a function that we want to run
- Fourth is a pointer that will be given to our function

The argument `void *(*start_routine) (void *)` is difficult to read! It means a pointer that takes a `void *` pointer and returns a `void *` pointer. It looks like a function declaration except that the name of the function is wrapped with `(* .... )`

```c
#include <stdio.h>
#include <pthread.h>
// remember to set compilation option -pthread

void *busy(void *ptr) {
// ptr will point to "Hi"
    puts("Hello World");
    return NULL;
}
int main() {
    pthread_t id;
    pthread_create(&id, NULL, busy, "Hi");
#if loop_forever // Loop forever
    while (1) {}
#else // Join Threads
    void *result;
    pthread_join(id, &result);
#endif
}
```

In the above example, `result` will be `null` because the busy function returned `null`. We need to pass the address-of result because `pthread_join` will be writing into the contents of our pointer.

Pthread Functions

- `pthread_create`. Creates a new thread. Every thread that gets created gets a new stack. For example if you call `pthread_create` twice, Your process will contain three stacks - one for each thread. The first thread is created when the process starts, and you created two more. Actually there can be more stacks than this, but let's keep it simple. The important idea is that each thread requires a stack because the stack contains automatic variables and the old CPU PC register, so that it can back to executing the calling function after the function is finished.

- `pthread_cancel` stops a thread. Note the thread may not actually be stopped immediately. For example it can be terminated when the thread makes an operating system call (e.g. `write`). In practice, `pthread_cancel` is rarely used because it does not give a thread an opportunity to clean up after itself (for example, it may have opened some files). An alternative implementation is to use a boolean (int) variable whose value is used to inform other threads that they should finish and clean up.

- `pthread_exit(void *)` stops the calling thread i.e. the thread never returns after calling `pthread_exit`. The pthread library will automatically finish the process if there are no other threads running. `pthread_exit(...)` is equivalent to returning from the thread's function; both finish the thread and also set the return value (void *pointer) for the thread. Calling `pthread_exit` in the the `main` thread is a common way for simple programs to ensure that all threads finish. For example, in the following program, the `myfunc` threads will probably not have time to get started. On the other hand `exit()` exits the entire process and sets the processes exit value. This is equivalent to `return ();` in the main method. All threads inside the process are stopped. Note the `pthread_exit` version creates thread zombies, however this is not a long-running processes, so we don't care.

```c
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, myfunc, "Jabberwocky");
  pthread_create(&tid2, NULL, myfunc, "Vorpel");
  if (keep_threads_going) {
    pthread_exit(NULL);
  } else {
    exit(42); //or return 42;
  }

  // No code is run after exit
}
```

- `pthread_join()` waits for a thread to finish and records its return value. Finished threads will continue to consume resources. Eventually, if enough threads are created, `pthread_create` will fail. In practice, this is only an issue for long-running processes but is not an issue for simple, short-lived processes as all thread resources are automatically freed when the process exits. This is equivalent to turning your children into zombies, so keep this in mind for long running processes. In the exit example, we could also wait on all the threads.

```c
// ...
  void* result;
  pthread_join(tid1, &result);
  pthread_join(tid2, &result);
  return 42;
// ...
```

- You've heard about this already, but how can a thread be terminated?
  - Returning from the thread function
  - Calling `pthread_exit`
  - Cancelling the thread with `pthread_cancel`
  - Terminating the process (e.g. SIGTERM); exit(); returning from `main`

Race Conditions

Race conditions are whenever the outcome of a program is determined by its sequence of events. Meaning that the same program can run multiple times and depending on how the kernel schedules the threads could produce inaccurate results. Take for example this race condition with one thread. We create a stack variable and pass it to our pthread function.

```c
pthread_t start_threads() {
  int start = 42;
  pthread_t tid;
  pthread_create(&tid, 0, myfunc, &start); // ERROR!
  return tid;
}
```

The above code is invalid because the function `start_threads` will likely return before `myfunc` even starts. The function passes the address-of `start`, however by the time `myfunc` is executes, `start` is no longer in scope and its address will re-used for another variable. This ia race condition because there is a situation where the thread that called pthread_create could be suspended indefinitely, and the code actually works. One way we can fix this is keep the function from returning before the thread finishes.

```
void start_threads() {
  int start = 42;
  void *result;
  pthread_t tid;
  pthread_create(&tid, 0, myfunc, &start); // OK - start will be valid!
  pthread_join(tid, &result);
}
```

Here is another small race condition. The following code is supposed to start ten threads with values 0,1,2,3,...9 However, when run prints out `1 7 8 8 8 8 8 8 8 10`! Can you see why?

```
#include <pthread.h>
void* myfunc(void* ptr) {
    int i = *((int *) ptr);
    printf("%d ", i);
    return NULL;
}

int main() {
    // Each thread gets a different value of i to process
    int i;
    pthread_t tid;
    for(i =0; i < 10; i++) {
        pthread_create(&tid, NULL, myfunc, &i); // ERROR
    }
    pthread_exit(NULL);
}
```

The above code suffers from a `race condition` - the value of i is changing. The new threads start later (in the example output the last thread starts after the loop has finished). To overcome this race-condition, we will give each thread a pointer to it's own data area. For example, for each thread we may want to store the id, a starting value and an output value. We will instead treat i as a pointer and cast it by value.

```c
void* myfunc(void* ptr) {
    int data = ((int) ptr);
    printf("%d ", data);
    return NULL;
}

int main() {
    // Each thread gets a different value of i to process
    int i;
    pthread_t tid;
    for(i =0; i < 10; i++) {
        pthread_create(&tid, NULL, myfunc, (void *)i);
    }
    pthread_exit(NULL);
}
```

Some functions e.g. asctime, getenv, strtok, strerror not thread-safe. Let's look at a simple function that is also not 'thread-safe' The result buffer could be stored in global memory. This is good - we wouldn't want to return a pointer to an invalid address on the stack, but there's only one result buffer in the entire memory. If two threads were to use it at the same time then one would corrupt the other:

```c
char *to_message(int num) {
    char static result [256];
    if (num < 10) sprintf(result, "%d : blah blah" , num);
    else strcpy(result, "Unknown");
    return result;
}
```

There are ways around this like using synchronization locks. These are synchronization locks that are used to prevent race conditions and ensure proper synchronization between threads running in the same program. In addition, these locks are conceptually identical to the primitives used inside the kernel.

In case you were wondering, you can fork inside a process with multiple threads! However, the child process only has a single thread, which is a clone of the thread that called `fork`. We can see this as a simple example, where the background threads never print out a second message in the child process.

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

static pid_t child = -2;

void *sleepnprint(void *arg) {
  printf("%d:%s starting up...\n", getpid(), (char *) arg);

  while (child == -2) {sleep(1);} /* Later we will use condition variables */

  printf("%d:%s finishing...\n",getpid(), (char*)arg);

  return NULL;
}
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1,NULL, sleepnprint, "New Thread One");
  pthread_create(&tid2,NULL, sleepnprint, "New Thread Two");

  child = fork();
  printf("%d:%s\n",getpid(), "fork()ing complete");
  sleep(3);

  printf("%d:%s\n",getpid(), "Main thread finished");

  pthread_exit(NULL);
  return 0; /* Never executes */
}
```

```
8970:New Thread One starting up...
8970:fork()ing complete
8973:fork()ing complete
8970:New Thread Two starting up...
8970:New Thread Two finishing...
8970:New Thread One finishing...
8970:Main thread finished
8973:Main thread finished
```

In practice, creating threads before forking can lead to unexpected errors because (as demonstrated above) the other threads are immediately terminated when forking. Another thread might have just lock a mutex (e.g. by calling malloc) and never unlock it again. Advanced users may find `pthread_atfork` useful however we suggest you usually try to avoid creating threads before forking unless you fully understand the limitations and difficulties of this approach.

How can I find out more?

- man page
- pthread reference guide
- Concise third party sample code explaining create, join and exit

Aside

Embarrassingly Parallel Problems

The study of parallel algorithms has exploded over the past few years. An embarrassingly parallel problem is any problem that needs little effort to turn parallel. A lot of them have some synchronization concepts with them but not always. You already know a parallelizable algorithm, Merge Sort!

```
void merge_sort(int *arr, size_t len){
   if(len > 1){
   //Mergesort the left half
   //Mergesort the right half
   //Merge the two halves
   }
```

With your new understanding of threads, all you need to do is create a thread for the left half, and one for the right half. Given that your CPU has multiple real cores, you will see a speedup in accordance with Amdahl's Law. The time complexity analysis gets interesting here as well. The parallel algorithm runs in $O(\log^3(n))$ running time (because we fancy analysis assuming that we have a lot of cores.

In practice though, we typically do two changes. One, once the array gets small enough, we ditch the parallel mergesort algorithm and do a quicksort or other algorithm that works fast on small arrays (something something cache coherency). The other thing that we know is that CPUs don't have infinite cores. To get around that, we typically keep a worker pool. You won't see the speedup right away because of things like cache coherency and scheduling extra threads.

Another problem, Parallel Map

Say we want to apply a function to an entire array, one element at a time.

```
int *map(int (*func)(int), int *arr, size_t len){
  int *ret = malloc(len*sizeof(*arr));
  for(size_t i = 0; i < len; ++i)
      ret[i] = func(arr[i]);
  return ret;
}
```

Since none of the elements depend on any other element, how would you go about parallelizing this? What do you think would be the best way to split up the work between threads.

Scheduling

There are a few ways to split up the work.

- `static scheduling` break up the problems into fixed size chunks (predetermined) and have each thread work on each of the chunks. This works well when each of the subproblems take roughly the same time because there is no additional overhead. All you need to do is write a loop and give the map function to each subarray.
- `dynamic scheduling` as a new problem becomes available have a thread serve it. This is useful when you don't know how long the scheduling will take
- `guided scheduling` This is a mix of the above with a mix of the benefits and the tradeoffs. You start with a static scheduling and move slowly to dynamic if needed

- `runtime scheduling` You have absolutely no idea how long the problems are going to take. Instead of deciding it yourself, let the program decide what to do!

source, but no need to memorize.

Other Problems

From Wikipedia

- Serving static files on a webserver to multiple users at once.
- The Mandelbrot set, Perlin noise and similar images, where each point is calculated independently.
- Rendering of computer graphics. In computer animation, each frame may be rendered independently (see parallel rendering).
- Brute-force searches in cryptography.
- Notable real-world examples include distributed.net and proof-of-work systems used in cryptocurrency.
- BLAST searches in bioinformatics for multiple queries (but not for individual large queries)
- Large scale facial recognition systems that compare thousands of arbitrary acquired faces (e.g., a security or surveillance video via closed-circuit television) with similarly large number of previously stored faces (e.g., a rogues gallery or similar watch list).
- Computer simulations comparing many independent scenarios, such as climate models.
- Evolutionary computation metaheuristics such as genetic algorithms.
- Ensemble calculations of numerical weather prediction.
- Event simulation and reconstruction in particle physics.
- The marching squares algorithm
- Sieving step of the quadratic sieve and the number field sieve.
- Tree growth step of the random forest machine learning technique.
- Discrete Fourier Transform where each harmonic is independently calculated.

Topics

- pthread lifecycle
- Each thread has a stack
- Capturing return values from a thread
- Using `pthread_join`
- Using `pthread_create`
- Using `pthread_exit`
- Under what conditions will a process exit

Questions

- What happens when a pthread gets created? (you don't need to go into super specifics)
- Where is each thread's stack?
- How do you get a return value given a `pthread_t`? What are the ways a thread can set that return value? What happens if you discard the return value?
- Why is `pthread_join` important (think stack space, registers, return values)?
- What does `pthread_exit` do under normal circumstances (ie you are not the last thread)? What other functions are called when you call pthread_exit?
- Give me three conditions under which a multithreaded process will exit. Can you think of any more?
- What is an embarrassingly parallel problem?

Bibliography

6 Synchronization

When multithreading gets interesting

Bhuvy

Synchronization are a series of mechanisms to control what threads are allowed to perform what operation at a time. Most of the time, the threads can progress without having to communicate, but every so often two or more threads may want to access a critical section. A critical section is a section of code that can only be executed by one thread at a time, if the program is to function correctly. If two threads (or processes) were to execute code inside the critical section at the same time, it is possible that program may no longer have correct behavior.

Something as simple as incrementing a variable could be a critical section. Incrementing a variable (`i++`) is performed in three individual steps: Copy the memory contents to the CPU register. Increment the value in the CPU. Store the new value in memory. If the memory location is only accessible by one thread (e.g. automatic variable `i` below) then there is no possibility of a race condition and no Critical Section associated with `i`. However the `sum` variable is a global variable and accessed by two threads. It is possible that two threads may attempt to increment the variable at the same time.

```c
#include <stdio.h>
#include <pthread.h>
// Compile with -pthread

int sum = 0; //shared

void *countgold(void *param) {
    int i; //local to each thread
    for (i = 0; i < 10000000; i++) {
        sum += 1;
    }
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, countgold, NULL);
    pthread_create(&tid2, NULL, countgold, NULL);

    //Wait for both threads to finish:
    pthread_join(tid1, NULL);
```

```
    pthread_join(tid2, NULL);

    printf("ARRRRG sum is %d\n", sum);
    return 0;
}
```

Typical output of the above code is `ARGGGH sum is <some number less than expected>` because there is a race condition. The code does not stop two threads from reading-writing `sum` at the same time. For example, both threads copy the current value of sum into CPU that runs each thread (let's pick 123). Both threads increment one to their own copy. Both threads write back the value (124). If the threads had accessed the sum at different times then the count would have been 125. A few of the possible different orderings are below.

Permittable Pattern

Thread 1	Thread 2
Load Addr, Add 1 (i=1 locally)	...
Store (i=1 globally)	...
...	Load Addr, Add 1 (i=2 locally)
...	Store (i=2 globally)

Overlap Partial

Thread 1	Thread 2
Load Addr, Add 1 (i=1 locally)	...
Store (i=1 globally)	Load Addr, Add 1 (i=1 locally)
...	Store (i=1 globally)

Full Overlap

Thread 1	Thread 2
Load Addr, Add 1 (i=1 locally)	Load Addr, Add 1 (i=1 locally)
Store (i=1 globally)	Store (i=1 globally)

We would like the first pattern of the code being mutually exclusive. Which leads us to our first synchronization primitive, a Mutex.

Mutex

To ensure only one thread at a time can access a global variable, use a mutex (short for Mutual Exclusion). If one thread is currently inside a critical section we would like another thread to wait until the first thread is complete.

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER; // global variable
pthread_mutex_lock(&m); // start of Critical Section
// Critical section
pthread_mutex_unlock(&m); //end of Critical Section
```

Lifetime

There are a few ways of initializing a mutex. You can use the macro `PTHREAD_MUTEX_INITIALIZER` only for global ('static') variables. `m = PTHREAD_MUTEX_INITIALIZER` is equivalent to the more general purpose `pthread_mutex_init(m,NULL)`. The init version includes options to trade performance for additional error-checking and advanced sharing options. You can also call the init function inside of a program for a mutex located on the heap.

```
pthread_mutex_t *lock = malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL);
```

```
//later
pthread_mutex_destroy(lock);
free(lock);
```

Once we are finished with the mutex we should also call `pthread_mutex_destroy(m)` too. Note, you can only destroy an unlocked mutex. Calling destroy on a destroyed lock, initializing an initialized lock, locking an already locked lock, unlocking an unlocked lock etc are undefined behavior.

Things to keep in mind about `init` and `destroy`

1. Multiple threads init/destroy has undefined behavior
2. Destroying a locked mutex has undefined behavior
3. Basically try to keep to the pattern of one thread initializing a mutex and one and only one thread initializing a mutex.
4. Copying the bytes of the mutex to a new memory location and then using the copy is *not* supported. To reference a mutex, you have to have a pointer to that memory address.

Mutex Gotchas

Mutexes do not lock variables. A mutex is not that smart - it works with code, not data. If I lock a mutex, the other threads will continue. It's only when a thread attempts to lock a mutex that is already locked, will the thread have to wait. As soon as the original thread unlocks the mutex, the second (waiting) thread will acquire the lock and be able to continue. The following code creates a mutex that does effectively nothing.

```
int a;
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER,
                m2 = = PTHREAD_MUTEX_INITIALIZER;
// later
// Thread 1
pthread_mutex_lock(&m1);
a++;
pthread_mutex_unlock(&m1);

// Thread 2
pthread_mutex_lock(&m2);
a++;
pthread_mutex_unlock(&m2);
```

You can create mutex before fork-ing - however the child and parent process will not share virtual memory and each one will have a mutex independent of the other. Advanced note: There are advanced options using shared memory that allow a child and parent to share a mutex if it's created with the correct options and uses a shared memory segment. See stackoverflow example

As some other notes, the thread that locks a mutex is the only thread that can unlock it. Each program can have multiple mutex locks, usually one lock per data structure. If you only have one lock, then they may be significant contention for the lock between two threads that was unnecessary. For example if two threads were updating two different counters, it might not be necessary to use the same lock. However, simply creating many locks is insufficient: It's important to be able to reason about critical sections e.g. it's important that one thread can't read two data structures while they are being updated and temporarily in an inconsistent state. There is a small amount of overhead of calling `pthread_mutex_lock` and `pthread_mutex_unlock`; however, this is the price you pay for correctly functioning programs!

Simplest complete example

Here is a complete example

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

// Compile with -pthread
// Create a mutex this ready to be locked!
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int sum = 0;

void *countgold(void *param) {
    int i;

    //Same thread that locks the mutex must unlock it
    //Critical section is just 'sum += 1'
    //However locking and unlocking a million times
    //has significant overhead

    pthread_mutex_lock(&m);

    // Other threads that call lock will have to wait until we call
       unlock

    for (i = 0; i < 10000000; i++) {
        sum += 1;
    }
    pthread_mutex_unlock(&m);
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, countgold, NULL);
    pthread_create(&tid2, NULL, countgold, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("ARRRRG sum is %d\n", sum);
    return 0;
}
```

In the code above, the thread gets the lock to the counting house before entering. The critical section is only the `sum+=1` so the following version is also correct.

```c
    for (i = 0; i < 10000000; i++) {
        pthread_mutex_lock(&m);
        sum += 1;
```

```
        pthread_mutex_unlock(&m);
    }
    return NULL;
}
```

This process runs slower because we lock and unlock the mutex a million times, which is expensive - at least compared with incrementing a variable. In this simple example, we didn't really need threads - we could have added up twice! A faster multi-thread example would be to add one million using an automatic(local) variable and only then adding it to a shared total after the calculation loop has finished:

```
    int local = 0;
    for (i = 0; i < 10000000; i++) {
       local += 1;
    }

    pthread_mutex_lock(&m);
    sum += local;
    pthread_mutex_unlock(&m);

    return NULL;
}
```

If I forget to unlock, you get deadlock! We will talk about deadlock a little bit later but what is the problem with this loop if called by multiple threads.

```
while(not_stop){
    //stdin may not be thread safe
    pthread_mutex_lock(&m);
    char *line = getline(...);
    if(rand() % 2) {/* randomly skip lines */
        continue;
    }
    pthread_mutex_unlock(&m);

    process_line(line);
}
```

Other possible problems with synchronization.

- Not unlocking a mutex due to an early return during an error condition
- Resource leak (not calling `pthread_mutex_destroy`)
- Using an unitialized mutex or using a mutex that has already been destroyed
- Locking a mutex twice on a thread without unlocking first
- Deadlock

Mutex Implementation

An incorrect implementation is shown below. The `unlock` function simply unlocks the mutex and returns. The lock function first checks to see if the lock is already locked. If it is currently locked, it will keep checking again until another thread has unlocked the mutex.

```
// Version 1 (Incorrect!)

void lock(mutex_t *m) {
  while(m->locked) {/*Locked? Nevermind - just loop and check
      again!*/ }

  m->locked = 1;
}
void unlock(mutex_t *m) {
  m->locked = 0;
}
```

Version 1 uses 'busy-waiting' (unnecessarily wasting CPU resources) however there is a more serious problem: We have a race-condition! If two threads both called `lock` concurrently it is possible that both threads would read `m_locked` as zero. Thus both threads would believe they have exclusive access to the lock and both threads will continue. Ooops!

We might attempt to reduce the CPU overhead a little by calling `pthread_yield()` inside the loop - pthread_yield suggests to the operating system that the thread does not use the CPU for a short while, so the CPU may be assigned to threads that are waiting to run. But does not fix the race-condition. We need a better implementation - can you work how to prevent the race-condition?

Aside

Implementing a Mutex with hardware

We can use C11 Atomics to do that perfectly! A complete solution is detailed here. This is a spinlock mutex, futex implementations can be found online.

```
typedef struct mutex_{
    atomic_int_least8_t lock;
    pthread_t owner;
} mutex;

#define UNLOCKED 0
#define LOCKED 1
#define UNASSIGNED_OWNER 0

int mutex_init(mutex* mtx){
    if(!mtx){
        return 0;
    }
    atomic_init(&mtx->lock, UNLOCKED); // Not thread safe the
        user has to take care of this
```

```
    mtx->owner = UNASSIGNED_OWNER;
    return 1;
}
```

This is the initialization code, nothing fancy here. We set the state of the mutex to unlocked and set the owner to locked.

```
int mutex_lock(mutex* mtx){
    int_least8_t zero = UNLOCKED;
    while(!atomic_compare_exchange_weak_explicit
            (&mtx->lock,
             &zero,
             LOCKED,
             memory_order_relaxed,
             memory_order_relaxed)){
        zero = UNLOCKED;
        sched_yield(); // Use system calls for scheduling speed
    }
    // We have the lock now
    mtx->owner = pthread_self();
    return 1;
}
```

Yikes! What does this code do? Well to start it it initializes a variable that we will keep as the unlocked state. Atomic Compare and Exchange is an instruction supported by most modern architectures (on x86 it's `lock cmpxchg`). The pseudocode for this operation looks like this

```
int atomic_compare_exchange_pseudo(int* addr1, int* addr2, int
    val){
    if(*addr1 == *addr2){
        *addr1 = val;
        return 1;
    }else{
        *addr2 = *addr1;
        return 0;
    }
}
```

Except it is all done *atomically* meaning in one uninterruptible operation. What does the *weak* part mean? Well atomic instructions are prone to **spurious failures** meaning that there are two versions to these atomic functions a *strong* and a *weak* part, strong guarantees the success or failure while weak may fail. We are using weak because weak is faster, and we are in a loop! That means we are okay if it fails a little bit more often because we will just keep spinning around anyway.

Inside the while loop, we have failed to grab the lock! We reset zero to unlocked and sleep for a little while. When we wake up we try to grab the lock again. Once we successfully swap, we are in the critical

section! We set the mutex's owner to the current thread for the unlock method and return successful.

How does this guarantee mutual exclusion, when working with atomics we are not entirely sure! But in this simple example we can because the thread that is able to successfully expect the lock to be UNLOCKED (0) and swap it to a LOCKED (1) state is considered the winner. How do we implement unlock?

What is this memory order business? We were talking about memory fences earlier, here it is! We won't go into detail because it is outside the scope of this course but not the scope of this article.

```
int mutex_unlock(mutex* mtx){
    if(unlikely(pthread_self() != mtx->owner)){
        return 0; //You can't unlock a mutex if you aren't the
            owner
    }
    int_least8_t one = 1;
    //Critical section ends after this atomic
    mtx->owner = UNASSIGNED_OWNER;
    if(!atomic_compare_exchange_strong_explicit(
                &mtx->lock,
                &one,
                UNLOCKED,
                memory_order_relaxed,
                memory_order_relaxed)){
        //The mutex was never locked in the first place
        return 0;
    }
    return 1;
}
```

To satisfy the api, you can't unlock the mutex unless you are the one who owns it. Then we unassign the mutex owner, because critical section is over after the atomic. We want a strong exchange because we don't want to block (pthread_mutex_unlock doesn't block). We expect the mutex to be locked, and we swap it to unlock. If the swap was successful, we unlocked the mutex. If the swap wasn't, that means that the mutex was UNLOCKED and we tried to switch it from UNLOCKED to UNLOCKED, preserving the non blocking of unlock.

Semaphores

A counting semaphore is another type of synchronization primitive. It contains a value and supports two operations "wait" and "post". Post increments the semaphore and immediately returns. "wait" will wait if the count is zero. If the count is non-zero, the semaphore decrements the count and immediately returns. An analogy is a count of the cookies in a cookie jar. Before taking a cookie, call 'wait'. If there are no cookies left then `wait` will not return: It will `wait` until another thread increments the semaphore by calling post. In short, `post` increments and immediately returns whereas `wait` will wait if the count is zero. Before returning, it will decrement count.

Using a semaphore is as easy as creating a mutex. First decide if the initial value should be zero or some other value (e.g. the number of remaining spaces in an array). Unlike pthread mutex there are not shortcuts to creating a semaphore - use `sem_init`

```
#include <semaphore.h>
```

```c
sem_t s;
int main() {
  sem_init(&s, 0, 10); // returns -1 (=FAILED) on OS X
  sem_wait(&s); // Could do this 10 times without blocking
  sem_post(&s); // Announce that we've finished (and one more
      resource item is available; increment count)
  sem_destroy(&s); // release resources of the semaphore
}
```

When using a semaphore, wait and post can be called from different threads! Unlike a mutex, the increment and decrement can be from different threads. This becomes especially useful if you want to use a semaphore to implement a mutex. A mutex is a semaphore that always `waits` before it `posts`. Some textbooks will refer to a mutex as a binary semaphore. You do have to be careful to never add more than one to a semaphore or otherwise your mutex abstraction breaks. That is usually why a mutex is used to implement a semaphore and vice versa.

- Initialize the semaphore with a count of one.
- Replace `pthread_mutex_lock` with `sem_wait`
- Replace `pthread_mutex_unlock` with `sem_post`

```c
sem_t s;
sem_init(&s, 0, 1);

sem_wait(&s);
// Critical Section
sem_post(&s);
```

Also, keywordsem_post is one of a handful of functions that can be correctly used inside a signal handler (`pthread_mutex_unlock` is not). This means we can release a waiting thread which can now make all of the calls that we were not allowed to call inside the signal handler itself e.g. `printf`.

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

sem_t s;

void handler(int signal) {
    sem_post(&s); /* Release the Kraken! */
}

void *singsong(void *param) {
    sem_wait(&s);
    printf("I had to wait until your signal released me!\n");
}
```

```c
int main() {
    int ok = sem_init(&s, 0, 0 /* Initial value of zero*/);
    if (ok == -1) {
       perror("Could not create unnamed semaphore");
       return 1;
    }
    signal(SIGINT, handler); // Too simple! See Signals chapter

    pthread_t tid;
    pthread_create(&tid, NULL, singsong, NULL);
    pthread_exit(NULL); /* Process will exit when there are no more
        threads */
}
```

Condition Variables

Condition variables allow a set of threads to sleep until woken up! You can wake up one thread or all threads that are sleeping. If you only wake one thread then the operating system will decide which thread to wake up. You don't wake threads directly instead you 'signal' the condition variable, which then will wake up one (or all) threads that are sleeping inside the condition variable.

Condition variables are also generally used with a mutex and with a loop, so when woken up they have to check a condition in a critical section. If you just need to be woken up not in a critical section, there are other ways to do this in POSIX. Threads sleeping inside a condition variable are woken up by calling `pthread_cond_broadcast` (wake up all) or `pthread_cond_signal` (wake up one). Note despite the function name, this has nothing to do with POSIX `signals`!

Occasionally a waiting thread may appear to wake up for no reason (this is called a *spurious wake*)! This is not an issue because you always use `wait` inside a loop that tests a condition that must be true to continue.

Why do spurious wakeups happen? For performance. On multi-CPU systems it is possible that a race-condition could cause a wake-up (signal) request to be unnoticed. The kernel may not detect this lost wake-up call but can detect when it might occur. To avoid the potential lost signal the thread is woken up so that the program code can test the condition again.

Example

The call `pthread_cond_wait` performs three actions:

1. Unlock the mutex
2. Sleeps until `pthread_cond_signal` is called on the same condition variable)
3. Before returning, locks the mutex

Condition variables are *always* used with a mutex lock. Before calling *wait*, the mutex lock must be locked and *wait* must be wrapped with a loop.

```c
pthread_cond_t cv;
pthread_mutex_t m;
int count;

// Initialize
pthread_cond_init(&cv, NULL);
pthread_mutex_init(&m, NULL);
count = 0;
```

```
pthread_mutex_lock(&m);
while (count < 10) {
   pthread_cond_wait(&cv, &m);
/* Remember that cond_wait unlocks the mutex before blocking
    (waiting)! */
/* After unlocking, other threads can claim the mutex. */
/* When this thread is later woken it will */
/* re-lock the mutex before returning */
}
pthread_mutex_unlock(&m);

//later clean up with pthread_cond_destroy(&cv); and mutex_destroy


// In another thread increment count:
while (1) {
  pthread_mutex_lock(&m);
  count++;
  pthread_cond_signal(&cv);
  /* Even though the other thread is woken up it cannot not return
     */
  /* from pthread_cond_wait until we have unlocked the mutex. This
     is */
  /* a good thing! In fact, it is usually the best practice to call
     */
  /* cond_signal or cond_broadcast before unlocking the mutex */
  pthread_mutex_unlock(&m);
}
```

Thread Safe Data Structures

To paraphrase Wikipedia,

> An operation (or set of operations) is atomic or uninterruptible if it appears to the rest of the system to occur instantaneously. Without locks, only simple CPU instructions ("read this byte from memory") are atomic (indivisible). On a single CPU system or inside kernels, one could temporarily disable interrupts (so a sequence of operations cannot be interrupted) but in practice atomicity is achieved by using synchronization primitives, typically a mutex lock.

Incrementing a variable (`i++`) is *not* atomic because it requires three distinct steps: Copying the bit pattern from memory into the CPU; performing a calculation using the CPU's registers; copying the bit pattern back to memory. During this increment sequence, another thread or process can still read the old value and other writes to the same memory would also be over-written when the increment sequence completes.

As such, we can use the tools in the previous section in order to make our data structures thread safe. For the most part, we will be using mutexes because they carry more semantic meaning than a binary semaphore. Note, this is just an introduction - writing high-performance thread-safe data structures requires its own book!

```
// A simple fixed-sized stack (version 1)
#define STACK_SIZE 20
int count;
double values[STACK_SIZE];
```

```
void push(double v) {
    values[count++] = v;
}

double pop() {
    return values[--count];
}

int is_empty() {
    return count == 0;
}
```

Version 1 of the stack is not thread-safe because if two threads call push or pop at the same time then the results or the stack can be inconsistent. For example, imagine if two threads call pop at the same time then both threads may read the same value, both may read the original count value.

To turn this into a thread-safe data structure we need to identify the *critical sections* of our code i.e. which section(s) of the code must only have one thread at a time. In the above example the `push`,`pop` and `is_empty` functions access the same variables (i.e. memory) and all critical sections for the stack. While `push` (and `pop`) is executing, the datastructure is an inconsistent state (for example the count may not have been written to, so may still contain the original value). By wrapping these methods with a mutex we can ensure that only one thread at a time can update (or read) the stack. A candidate 'solution' is shown below. Is it correct? If not, how will it fail?

```
// An attempt at a thread-safe stack (version 2)
#define STACK_SIZE 20
int count;
double values[STACK_SIZE];

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;

void push(double v) {
    pthread_mutex_lock(&m1);
    values[count++] = v;
    pthread_mutex_unlock(&m1);
}

double pop() {
    pthread_mutex_lock(&m2);
    double v = values[--count];
    pthread_mutex_unlock(&m2);

    return v;
}

int is_empty() {
    pthread_mutex_lock(&m1);
    return count == 0;
    pthread_mutex_unlock(&m1);
}
```

Version 2 contains at least one error. Take a moment to see if you can the error(s) and work out the consequence(s).

If three threads called `push()` at the same time the lock `m1` ensures that only one thread at time manipulates the stack (two threads will need to wait until the first thread completes (calls unlock), then a second thread will be allowed to continue into the critical section and finally the third thread will be allowed to continue once the second thread has finished).

A similar argument applies to concurrent calls (calls at the same time) to `pop`. However version 2 does not prevent push and pop from running at the same time because `push` and `pop` use two different mutex locks. The fix is simple in this case - use the same mutex lock for both the push and pop functions.

The code has a second error; `is_empty` returns after the comparison and will not unlock the mutex. However the error would not be spotted immediately. For example, suppose one thread calls `is_empty` and a second thread later calls `push`. This thread would mysteriously stop. Using debugger you can discover that the thread is stuck at the lock() method inside the `push` method because the lock was never unlocked by the earlier `is_empty` call. Thus an oversight in one thread led to problems much later in time in an arbitrary other thread.

```
// An attempt at a thread-safe stack (version 3)
int count;
double values[count];
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void push(double v) {
  pthread_mutex_lock(&m);
  values[count++] = v;
  pthread_mutex_unlock(&m);
}
double pop() {
  pthread_mutex_lock(&m);
  double v = values[--count];
  pthread_mutex_unlock(&m);
  return v;
}
int is_empty() {
  pthread_mutex_lock(&m);
  int result= count == 0;
  pthread_mutex_unlock(&m);
  return result;
}
```

Version 3 is thread-safe. We have ensured mutual exclusion for all of the critical sections.

- `is_empty` is thread-safe but its result may already be out-of date i.e. the stack may no longer be empty by the time the thread gets the result!
- There is no protection against underflow (popping on an empty stack) or overflow (pushing onto an already-full stack)

The latter point can be fixed using counting semaphores. The implementation assumes a single stack. A more general purpose version might include the mutex as part of the memory struct and use pthread_mutex_init to initialize the mutex. For example,

```
// Support for multiple stacks (each one has a mutex)
typedef struct stack {
```

```
    int count;
    pthread_mutex_t m;
    double *values;
} stack_t;

stack_t* stack_create(int capacity) {
    stack_t *result = malloc(sizeof(stack_t));
    result->count = 0;
    result->values = malloc(sizeof(double) * capacity);
    pthread_mutex_init(&result->m, NULL);
    return result;
}
void stack_destroy(stack_t *s) {
    free(s->values);
    pthread_mutex_destroy(&s->m);
    free(s);
}

// Warning no underflow or overflow checks!

void push(stack_t *s, double v) {
    pthread_mutex_lock(&s->m);
    s->values[(s->count)++] = v;
    pthread_mutex_unlock(&s->m);
}

double pop(stack_t *s) {
    pthread_mutex_lock(&s->m);
    double v = s->values[--(s->count)];
    pthread_mutex_unlock(&s->m);
    return v;
}

int is_empty(stack_t *s) {
    pthread_mutex_lock(&s->m);
    int result = s->count == 0;
    pthread_mutex_unlock(&s->m);
    return result;
}

int main() {
    stack_t *s1 = stack_create(10 /* Max capacity*/);
    stack_t *s2 = stack_create(10);
    push(s1, 3.141);
    push(s2, pop(s1));
    stack_destroy(s2);
    stack_destroy(s1);
}
```

Using semaphores

We can also use a counting semaphore to keep track of how many spaces remain and another semaphore to keep to track the number of items in the stack. We will call these two semaphores ‘sremain’ and ‘sitems’. Remember

`sem_wait` will wait if the semaphore's count has been decremented to zero (by another thread calling sem_post).

```
// Sketch #1

sem_t sitems;
sem_t sremain;
void stack_init(){
  sem_init(&sitems, 0, 0);
  sem_init(&sremain, 0, 10);
}


double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);
  ...

void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);
  ...
```

Sketch #2 has implemented the `post` too early. Another thread waiting in push can erroneously attempt to write into a full stack (and similarly a thread waiting in the pop() is allowed to continue too early).

```
// Sketch #2 (Error!)
double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);
  sem_post(&sremain); // error! wakes up pushing() thread too early
  return values[--count];
}
void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);
  sem_post(&sitems); // error! wakes up a popping() thread too early
  values[count++] = v;
}
```

Sketch 3 implements the correct semaphore logic but can you spot the error?

```
// Sketch #3 (Error!)
double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);
  double v= values[--count];
  sem_post(&sremain);
  return v;
```

```
}

void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);
  values[count++] = v;
  sem_post(&sitems);
}
```

Sketch 3 correctly enforces buffer full and buffer empty conditions using semaphores. However there is no *mutual exclusion*: Two threads can be in the *critical section* at the same time, which would corrupt the data structure (or least lead to data loss). The fix is to wrap a mutex around the critical section:

```
// Simple single stack - see above example on how to convert this
    into a multiple stacks.
// Also a robust POSIX implementation would check for EINTR and
    error codes of sem_wait.

// PTHREAD_MUTEX_INITIALIZER for statics (use pthread_mutex_init()
    for stack/heap memory)

pthread_mutex_t m= PTHREAD_MUTEX_INITIALIZER;
int count = 0;
double values[10];
sem_t sitems, sremain;

void init() {
  sem_init(&sitems, 0, 0);
  sem_init(&sremains, 0, 10); // 10 spaces
}

double pop() {
  // Wait until there's at least one item
  sem_wait(&sitems);

  pthread_mutex_lock(&m); // CRITICAL SECTION
  double v= values[--count];
  pthread_mutex_unlock(&m);

  sem_post(&sremain); // Hey world, there's at least one space
  return v;
}

void push(double v) {
  // Wait until there's at least one space
  sem_wait(&sremain);

  pthread_mutex_lock(&m); // CRITICAL SECTION
  values[count++] = v;
  pthread_mutex_unlock(&m);
```

```
  sem_post(&sitems); // Hey world, there's at least one item
}
// Note a robust solution will need to check sem_wait's result for
    EINTR (more about this later)
```

Candidate Solutions

As already discussed, there are critical parts of our code that can only be executed by one thread at a time. We describe this requirement as 'mutual exclusion'; only one thread (or process) may have access to the shared resource. In multi-threaded programs, we can wrap a critical section with mutex lock and unlock calls:

```
pthread_mutex_lock() - one thread allowed at a time! (others will
    have to wait here)
... Do Critical Section stuff here!
pthread_mutex_unlock() - let other waiting threads continue
```

How would we implement these lock and unlock calls? Can we create an algorithm that assures mutual exclusion?

```
pthread_mutex_lock(p_mutex_t *m) { while(m->lock) {}; m->lock = 1;}
pthread_mutex_unlock(p_mutex_t *m) {m->lock = 0; }
```

At first glance, the code appears to work; if one thread attempts to locks the mutex, a later thread must wait until the lock is cleared. However this implementation *does not satisfy Mutual Exclusion*. Let's take a close look at this 'implementation' from the point of view of two threads running around the same time.

TODO: Thread ascii art or x86

To simplify the discussion we consider only two threads. Note, these arguments work for threads and processes and the classic CS literature discusses these problem in terms of two processes that need exclusive access (i.e. mutual exclusion) to a critical section or shared resource. Raising a flag represents a thread/process's intention to enter the critical section.

Remember that the psuedo-code outlined below is part of a larger program; the thread or process will typically need to enter the critical section many times during the lifetime of the process. So imagine each example as wrapped inside a loop where for a random amount of time the thread or process is working on something else.

Is there anything wrong with candidate solution described below?

```
// Candidate #1
wait until your flag is lowered
raise my flag
// Do Critical Section stuff
lower my flag
```

Answer: Candidate solution #1 also suffers a race condition i.e. it does not satisfy Mutual Exclusion because both threads/processes could read each other's flag value (=lowered) and continue.

This suggests we should raise the flag *before* checking the other thread's flag - which is candidate solution #2

below.

```
// Candidate #2
raise my flag
wait until your flag is lowered
// Do Critical Section stuff
lower my flag
```

Candidate #2 satisfies mutual exclusion - it is impossible for two threads to be inside the critical section at the same time. However this code suffers from deadlock! Suppose two threads wish to enter the critical section at the same time:

Time | Thread 1 | Thread 2 —-|———-|——— 1 | raise flag 2 | | raise flag 3 | wait … | wait … Ooops both threads / processes are now waiting for the other one to lower their flags. Neither one will enter the critical section as both are now stuck forever!

This suggests we should use a turn-based variable to try to resolve who should proceed.

Turn-based solutions

The following candidate solution #3 uses a turn-based variable to politely allow one thread and then the other to continue

```
// Candidate #3
wait until my turn is myid
// Do Critical Section stuff
turn = yourid
```

Candidate #3 satisfies mutual exclusion (each thread or process gets exclusive access to the Critical Section), however both threads/processes must take a strict turn-based approach to using the critical section; i.e. they are forced into an alternating critical section access pattern. For example, if thread 1 wishes to read a hashtable every millisecond but another thread writes to a hashtable every second, then the reading thread would have to wait another 999ms before being able to read from the hashtable again. This 'solution' is not effective, because our threads should be able to make progress and enter the critical section if no other thread is currently in the critical section.

Desired properties for solutions to the Critical Section Problem?

There are three main desirable properties that we desire in a solution the critical section problem * Mutual Exclusion - the thread/process gets exclusive access; others must wait until it exits the critical section. * Bounded Wait - if the thread/process has to wait, then it should only have to wait for a finite, amount of time (infinite waiting times are not allowed!). The exact definition of bounded wait is that there is an upper (non-infinite) bound on the number of times any other process can enter its critical section before the given process enters. * Progress - if no thread/process is inside the critical section, then the thread/process should be able to proceed (make progress) without having to wait.

With these ideas in mind let's examine another candidate solution that uses a turn-based flag only if two threads both required access at the same time.

Turn and Flag solutions

Is the following a correct solution to CSP?

```
\\ Candidate #4
```

```
raise my flag
if your flag is raised, wait until my turn
// Do Critical Section stuff
turn = yourid
lower my flag
```

One instructor and another CS faculty member initially thought so! However, analyzing these solutions is tricky. Even peer-reviewed papers on this specific subject contain incorrect solutions! At first glance it appears to satisfy Mutual Exclusion, Bounded Wait and Progress: The turn-based flag is only used in the event of a tie (so Progress and Bounded Wait is allowed) and mutual exclusion appears to be satisfied. However.... Perhaps you can find a counter-example?

Candidate #4 fails because a thread does not wait until the other thread lowers their flag. After some thought (or inspiration) the following scenario can be created to demonstrate how Mutual Exclusion is not satisfied.

Imagine the first thread runs this code twice (so the the turn flag now points to the second thread). While the first thread is still inside the Critical Section, the second thread arrives. The second thread can immediately continue into the Critical Section!

Time	Turn	Thread #1	Thread #2
1	2	raise my flag	
2	2	if your flag is raised, wait until my turn	raise my flag
3	2	// Do Critical Section stuff	if your flag is raised, wait until my turn(TRUE!)
4	2	// Do Critical Section stuff	// Do Critical Section stuff - OOPS

Working Solutions

What is Peterson's solution?

Peterson published his novel and surprisingly simple solution in a 2 page paper in 1981. A version of his algorithm is shown below that uses a shared variable `turn`:

```
\\ Candidate #5
raise my flag
turn = your_id
wait until your flag is lowered and turn is yourid
// Do Critical Section stuff
lower my flag
```

This solution satisfies Mutual Exclusion, Bounded Wait and Progress. If thread #2 has set turn to 2 and is currently inside the critical section. Thread #1 arrives, *sets the turn back to 1* and now waits until thread 2 lowers

the flag.

Link to Peterson's original article pdf: G. L. Peterson: "Myths About the Mutual Exclusion Problem", Information Processing Letters 12(3) 1981, 115–116

Was Peterson's solution the first solution?

No, Dekkers Algorithm (1962) was the first provably correct solution. A version of the algorithm is below.

```
raise my flag
while-loop(your flag is raised) :
   if it's your turn to win :
     lower my flag
     wait while your turn
     raise my flag
// Do Critical Section stuff
set your turn to win
lower my flag
```

Notice how the process's flag is always raised during the critical section no matter if the loop is iterated zero, once or more times. Further the flag can be interpreted as an immediate intent to enter the critical section. Only if the other process has also raised the flag will one process defer, lower their intent flag and wait.

Aside

Can I just implement Peterson's (or Dekkers) algorithm in C or assembler?

Yes - and with a bit searching it is possible even today to find it in production for specific simple mobile processors: Peterson's algorithm is used to implement low-level Linux Kernel locks for the Tegra mobile processor (a system-on-chip ARM process and GPU core by Nvidia) https://android.googlesource.com/kernel/tegra.git/+/android-tegra-3.10/arch/arm/mach-tegra/sleep.S#58

However in general, CPUs and C compilers can re-order CPU instructions or use CPU-core-specific local cache values that are stale if another core updates the shared variables. Thus a simple pseudo-code to C implementation is too naive for most platforms. You can stop reading now.

Oh… you decided to keep reading. Well, here be dragons! Don't say we didn't warn you. Consider this advanced and gnarly topic but (spoiler alert) a happy ending.

Consider the following code,

```
while(flag2 ) {/* busy loop - go around again */
```

An efficient compiler would infer that `flag2` variable is never changed inside the loop, so that test can be optimized to `while(true)` Using `volatile` goes someway to prevent compiler optimizations of this kind.

Independent instructions can be re-ordered by an optimizing compiler or at runtime by an out-of-order execution optimization by the CPU. These sophisticated optimizations if the code requires variables to be modified and checked and a precise order.

A related challenge is that CPU cores include a data cache to store recently read or modified main memory values. Modified values may not be written back to main memory or re-read from memory immediately.

Thus data changes, such as the state of a flag and turn variable in the above examples, may not be shared between two CPU codes.

But there is happy ending. Fortunately, modern hardware addresses these issues using 'memory fences' (also known as memory barrier) CPU instructions to ensure that main memory and the CPUs' cache is in a reasonable and coherent state. Higher level synchronization primitives, such as `pthread_mutex_lock` are will call these CPU instructions as part of their implementation. Thus, in practice, surrounding critical section with a mutex lock and unlock calls is sufficient to ignore these lower-level problems.

Further reading: we suggest the following web post that discusses implementing Peterson's algorithm on an x86 process and the linux documentation on memory barriers.

1. http://bartoszmilewski.com/2008/11/05/who-ordered-memory-fences-on-an-x86/
2. http://lxr.free-electrons.com/source/Documentation/memory-barriers.txt

Implementing Counting Semaphore

- We can implement a counting semaphore using condition variables.
- Each semaphore needs a count, a condition variable and a mutex

```
typedef struct sem_t {
  int count;
  pthread_mutex_t m;
  pthread_condition_t cv;
} sem_t;
```

Implement `sem_init` to initialize the mutex and condition variable

```
int sem_init(sem_t *s, int pshared, int value) {
    if (pshared) {
        errno = ENOSYS /* 'Not implemented'*/;
        return -1;
    }

    s->count = value;
    pthread_mutex_init(&s->m, NULL);
    pthread_cond_init(&s->cv, NULL);
    return 0;
}
```

Our implementation of `sem_post` needs to increment the count. We will also wake up any threads sleeping inside the condition variable. Notice we lock and unlock the mutex so only one thread can be inside the critical section at a time.

```
sem_post(sem_t *s) {
```

```
  pthread_mutex_lock(&s->m);
  s->count++;
  pthread_cond_signal(&s->cv); /* See note */
  /* A woken thread must acquire the lock, so it will also have to
      wait until we call unlock*/

  pthread_mutex_unlock(&s->m);
}
```

Our implementation of `sem_wait` may need to sleep if the semaphore's count is zero. Just like `sem_post` we wrap the critical section using the lock (so only one thread can be executing our code at a time). Notice if the thread does need to wait then the mutex will be unlocked, allowing another thread to enter `sem_post` and waken us from our sleep!

Notice that even if a thread is woken up, before it returns from `pthread_cond_wait` it must re-acquire the lock, so it will have to wait a little bit more (e.g. until sem_post finishes).

```
sem_wait(sem_t *s) {
  pthread_mutex_lock(&s->m);
  while (s->count == 0) {
      pthread_cond_wait(&s->cv, &s->m); /*unlock mutex, wait,
          relock mutex*/
  }
  s->count--;
  pthread_mutex_unlock(&s->m);
}
```

Wait `sem_post` keeps calling `pthread_cond_signal` won't that break sem_wait? Answer: No! We can't get past the loop until the count is non-zero. In practice this means `sem_post` would unnecessary call `pthread_cond_signal` even if there are no waiting threads. A more efficient implementation would only call `pthread_cond_signal` when necessary i.e.

```
  /* Did we increment from zero to one- time to signal a thread
      sleeping inside sem_post */
  if (s->count == 1) /* Wake up one waiting thread!*/
     pthread_cond_signal(&s->cv);
```

Other semaphore considerations

- Real semaphores implementation include a queue and scheduling concerns to ensure fairness and priority e.g. wake up the highest-priority longest sleeping thread.
- Also, an advanced use of `sem_init` allows semaphores to be shared across processes. Our implementation only works for threads inside the same process.

How do I wait for N threads to reach a certain point before continuing onto the next step?

Suppose we wanted to perform a multi-threaded calculation that has two stages, but we don't want to advance to the second stage until the first stage is completed.

We could use a synchronization method called a **barrier**. When a thread reaches a barrier, it will wait at the barrier until all the threads reach the barrier, and then they'll all proceed together.

Think of it like being out for a hike with some friends. You agree to wait for each other at the top of each hill (and you make a mental note how many are in your group). Say you're the first one to reach the top of the first hill. You'll wait there at the top for your friends. One by one, they'll arrive at the top, but nobody will continue until the last person in your group arrives. Once they do, you'll all proceed.

Pthreads has a function `pthread_barrier_wait()` that implements this. You'll need to declare a `pthread_barrier_t` variable and initialize it with `pthread_barrier_init()`. `pthread_barrier_init()` takes the number of threads that will be participating in the barrier as an argument. Here's an example.

Now let's implement our own barrier and use it to keep all the threads in sync in a large calculation.

```
double data[256][8192]

1 Threads do first calculation (use and change values in data)

2 Barrier! Wait for all threads to finish first calculation before
    continuing

3 Threads do second calculation (use and change values in data)
```

The thread function has four main parts-

```
void *calc(void *arg) {
  /* Do my part of the first calculation */
  /* Am I the last thread to finish? If so wake up all the other
     threads! */
  /* Otherwise wait until the other threads has finished part one */
  /* Do my part of the second calculation */
}
```

Our main thread will create the 16 threads and we will divide each calculation into 16 separate pieces. Each thread will be given a unique value (0,1,2,..15), so it can work on its own block. Since a (void*) type can hold small integers, we will pass the value of `i` by casting it to a void pointer.

```
#define N (16)
double data[256][8192] ;
int main() {
    pthread_t ids[N];
    for(int i = 0; i < N; i++)
        pthread_create(&ids[i], NULL, calc, (void *) i);
```

Note, we will never dereference this pointer value as an actual memory location - we will just cast it straight back to an integer:

```
void *calc(void *ptr) {
```

```
// Thread 0 will work on rows 0..15, thread 1 on rows 16..31
  int x, y, start = N * (int) ptr;
  int end = start + N;
  for(x = start; x < end; x++) for (y = 0; y < 8192; y++) {/* do
      calc #1 */ }
```

After calculation 1 completes, we need to wait for the slower threads (unless we are the last thread!). So, keep track of the number of threads that have arrived at our barrier aka 'checkpoint':

```
// Global:
int remain = N;


// After calc #1 code:
remain--; // We finished
if (remain ==0) {/*I'm last! - Time for everyone to wake up! */ }
else {
  while (remain != 0) {/* spin spin spin*/ }
}
```

However the above code has a race condition (two threads might try to decrement `remain`) and the loop is a busy loop. We can do better! Let's use a condition variable and then we will use a broadcast/signal functions to wake up the sleeping threads.

A reminder, that a condition variable is similar to a house! Threads go there to sleep (`pthread_cond_wait`). You can choose to wake up one thread (`pthread_cond_signal`) or all of them (`pthread_cond_broadcast`). If there are no threads currently waiting then these two calls have no effect.

A condition variable version is usually very similar to a busy loop incorrect solution - as we will show next. First, let's add a mutex and condition global variables and don't forget to initialize them in `main` ...

```
//global variables
pthread_mutex_t m;
pthread_cond_t cv;

main() {
  pthread_mutex_init(&m, NULL);
  pthread_cond_init(&cv, NULL);
```

We will use the mutex to ensure that only one thread modifies `remain` at a time. The last arriving thread needs to wake up *all* sleeping threads - so we will use `pthread_cond_broadcast(cv)` not `pthread_cond_signal`

```
pthread_mutex_lock(&m);
remain--;
if (remain ==0) {pthread_cond_broadcast(&cv); }
else {
  while(remain != 0) {pthread_cond_wait(&cv, &m); }
}
```

```
pthread_mutex_unlock(&m);
```

When a thread enters `pthread_cond_wait`, it releases the mutex and sleeps. At some point in the future, it will be awoken. Once we bring a thread back from its sleep, before returning it must wait until it can lock the mutex. Notice that even if a sleeping thread wakes up early, it will check the while loop condition and re-enter wait if necessary.

The above barrier is not reusable Meaning that if we stick it into any old calculation loop there is a good chance that the code will encounter a condition where the barrier either deadlocks or a thread races ahead one iteration faster. Think about how you can make the above barrier reusable, meaning that if mutliple threads call `barrier_wait` in a loop then one can guarantee that they are on the same iteration.

What is the Reader Writer Problem?

Imagine you had a key-value map data structure which is used by many threads. Multiple threads should be able to look up (read) values at the same time provided the data structure is not being written to. The writers are not so gregarious - to avoid data corruption, only one thread at a time may modify (`write`) the data structure (and no readers may be reading at that time).

This is an example of the *Reader Writer Problem*. Namely how can we efficiently synchronize multiple readers and writers such that multiple readers can read together but a writer gets exclusive access?

An incorrect attempt is shown below ("lock" is a shorthand for `pthread_mutex_lock`):

Attempt #1

At least our first attempt does not suffer from data corruption (readers must wait while a writer is writing and vice versa)! However readers must also wait for other readers. So let's try another implementation..

Attempt #2:

Our second attempt suffers from a race condition - imagine if two threads both called `read` and `write` (or both called write) at the same time. Both threads would be able to proceed! Secondly, we can have multiple readers and multiple writers, so lets keep track of the total number of readers or writers. Which brings us to attempt #3,

Attempt #3

Remember that `pthread_cond_wait` performs *Three* actions. Firstly it atomically unlocks the mutex and then sleeps (until it is woken by `pthread_cond_signal` or `pthread_cond_broadcast`). Thirdly the awoken thread must re-acquire the mutex lock before returning. Thus only one thread can actually be running inside the critical section defined by the lock and unlock() methods.

Implementation #3 below ensures that a reader will enter the cond_wait if there are any writers writing.

```
read() {
    lock(&m)
    while (writing)
        cond_wait(&cv, &m)
    reading++;

/* Read here! */

    reading--
    cond_signal(&cv)
    unlock(&m)
}
```

However only one reader a time can read because candidate #3 did not unlock the mutex. A better version unlocks before reading :

```
read() {
    lock(&m);
    while (writing)
        cond_wait(&cv, &m)
    reading++;
    unlock(&m)
/* Read here! */
    lock(&m)
    reading--
    cond_signal(&cv)
    unlock(&m)
}
```

Does this mean that a writer and read could read and write at the same time? No! First of all, remember cond_wait requires the thread re-acquire the mutex lock before returning. Thus only one thread can be executing code inside the critical section (marked with **) at a time!

```
read() {
    lock(&m);
**  while (writing)
**      cond_wait(&cv, &m)
**  reading++;
    unlock(&m)
/* Read here! */
    lock(&m)
**  reading--
**  cond_signal(&cv)
    unlock(&m)
}
```

Writers must wait for everyone. Mutual exclusion is assured by the lock.

```
write() {
    lock(&m);
**  while (reading || writing)
**      cond_wait(&cv, &m);
**  writing++;
**
** /* Write here! */
**  writing--;
**  cond_signal(&cv);
    unlock(&m);
}
```

Candidate #3 above also uses `pthread_cond_signal` ; this will only wake up one thread. For example, if many readers are waiting for the writer to complete then only one sleeping reader will be awoken from their slumber. The reader and writer should use `cond_broadcast` so that all threads should wake up and check their while-loop condition.

Starving writers

Candidate #3 above suffers from starvation. If readers are constantly arriving then a writer will never be able to proceed (the 'reading' count never reduces to zero). This is known as *starvation* and would be discovered under heavy loads. Our fix is to implement a bounded-wait for the writer. If a writer arrives they will still need to wait for existing readers however future readers must be placed in a "holding pen" and wait for the writer to finish. The "holding pen" can be implemented using a variable and a condition variable (so that we can wake up the threads once the writer has finished).

Our plan is that when a writer arrives, and before waiting for current readers to finish, register our intent to write (by incrementing a counter 'writer'). Sketched below -

```
write() {
    lock()
    writer++

    while (reading || writing)
    cond_wait
    unlock()
  ...
}
```

And incoming readers will not be allowed to continue while writer is nonzero. Notice 'writer' indicates a writer has arrived, while 'reading' and 'writing' counters indicate there is an *active* reader or writer.

```
read() {
    lock()
    // readers that arrive *after* the writer arrived will have to
       wait here!
    while(writer)
    cond_wait(&cv,&m)

    // readers that arrive while there is an active writer
    // will also wait.
    while (writing)
        cond_wait(&cv,&m)
    reading++
    unlock
  ...
}
```

Attempt #4

Below is our first working solution to the Reader-Writer problem. Note if you continue to read about the "Reader Writer problem" then you will discover that we solved the "Second Reader Writer problem" by giving writers preferential access to the lock. This solution is not optimal. However it satisfies our original problem (N active

readers, single active writer, avoids starvation of the writer if there is a constant stream of readers).

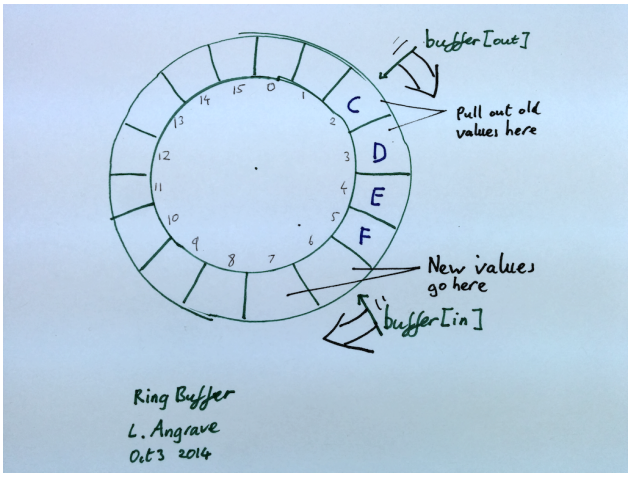
Can you identify any improvements? For example, how would you improve the code so that we only woke up readers or one writer?

```
int writers; // Number writer threads that want to enter the
    critical section (some or all of these may be blocked)
int writing; // Number of threads that are actually writing inside
    the C.S. (can only be zero or one)
int reading; // Number of threads that are actually reading inside
    the C.S.
// if writing !=0 then reading must be zero (and vice versa)

reader() {
    lock(&m)
    while (writers)
        cond_wait(&turn, &m)
    // No need to wait while(writing here) because we can only exit
        the above loop
    // when writing is zero
    reading++
    unlock(&m)

  // perform reading here

    lock(&m)
    reading--
    cond_broadcast(&turn)
    unlock(&m)
}

writer() {
    lock(&m)
    writers++
    while (reading || writing)
        cond_wait(&turn, &m)
    writing++
    unlock(&m)
    // perform writing here
    lock(&m)
    writing--
    writers--
    cond_broadcast(&turn)
    unlock(&m)
}
```

Ring Buffer

A ring buffer is a simple, usually fixed-sized, storage mechanism where contiguous memory is treated as if it is circular, and two index counters keep track of the current beginning and end of the queue. As array indexing is not circular, the index counters must wrap around to zero when moved past the end of the array. As data is added

(enqueued) to the front of the queue or removed (dequeued) from tail of the queue, the current items in the buffer form a train that appears to circle the track



A simple (single-threaded) implementation is shown below. Note enqueue and dequeue do not guard against underflow or overflow - it's possible to add an item when when the queue is full and possible to remove an item when the queue is empty. For example if we added 20 integers (1,2,3...) to the queue and did not dequeue any items then values `17,18,19,20` would overwrite the `1,2,3,4`. We won't fix this problem right now, instead when we create the multi-threaded version we will ensure enqueue-ing and dequeue-ing threads are blocked while the ring buffer is full or empty respectively.

```
void *buffer[16];
int in = 0, out = 0;

void enqueue(void *value) {/* Add one item to the front of the
    queue*/
  buffer[in] = value;
  in++; /* Advance the index for next time */
  if (in == 16) in = 0; /* Wrap around! */
}

void *dequeue() {/* Remove one item to the end of the queue.*/
  void *result = buffer[out];
  out++;
  if (out == 16) out = 0;
  return result;
}
```

What are gotchas of implementing a Ring Buffer?

It's very tempting to write the enqueue or dequeue method in the following compact form (N is the capacity of the buffer e.g. 16):

```
void enqueue(void *value)
  b[ (in++) % N ] = value;
}
```

This method would appear to work (pass simple tests etc) but contains a subtle bug. With enough enqueue operations (a bit more than two billion) the int value of `in` will overflow and become negative! The modulo (or 'remainder') operator `%` preserves the sign. Thus you might end up writing into `b[-14]` for example!

A compact form is correct uses bit masking provided N is 2^x (16,32,64,...)

```
b[ (in++) & (N-1) ] = value;
```

This buffer does not yet prevent buffer underflow or overflow. For that, we'll turn to our multi-threaded attempt that will block a thread until there is space or there is at least one item to remove.

Multithreaded Correctness

The following code is an incorrect implementation. What will happen? Will `enqueue` and/or `dequeue` block? Is mutual exclusion satisfied? Can the buffer underflow? Can the buffer overflow? For clarity `pthread_mutex` is shortened to `p_m` and we assume sem_wait cannot be interrupted.

```
#define N 16
void *b[N]
int in = 0, out = 0
p_m_t lock
sem_t s1,s2
void init() {
    p_m_init(&lock, NULL)
    sem_init(&s1, 0, 16)
    sem_init(&s2, 0, 0)
}

enqueue(void *value) {
    p_m_lock(&lock)

    // Hint: Wait while zero. Decrement and return
    sem_wait( &s1 )

    b[ (in++) & (N-1) ] = value

    // Hint: Increment. Will wake up a waiting thread
    sem_post(&s1)
    p_m_unlock(&lock)
}
void *dequeue(){
    p_m_lock(&lock)
    sem_wait(&s2)
    void *result = b[(out++) & (N-1) ]
    sem_post(&s2)
    p_m_unlock(&lock)
    return result
}
```

Analysis

Before reading on, see how many mistakes you can find. Then determine what would happen if threads called the enqueue and dequeue methods.

- The enqueue method waits and posts on the same semaphore (s1) and similarly with equeue and (s2) i.e. we decrement the value and then immediately increment the value, so by the end of the function the semaphore value is unchanged!
- The initial value of s1 is 16, so the semaphore will never be reduced to zero - enqueue will not block if the ring buffer is full - so overflow is possible.
- The initial value of s2 is zero, so calls to dequeue will always block and never return!
- The order of mutex lock and sem_wait will need to be swapped (however this example is so broken that this bug has no effect!) ## Checking a multi-threaded implementation for correctness (Example 1)

The following code is an incorrect implementation. What will happen? Will `enqueue` and/or `dequeue` block? Is mutual exclusion satisfied? Can the buffer underflow? Can the buffer overflow? For clarity `pthread_mutex` is shortened to `p_m` and we assume sem_wait cannot be interrupted.

```
void *b[16]
int in = 0, out = 0
p_m_t lock
sem_t s1, s2
void init() {
    sem_init(&s1,0,16)
    sem_init(&s2,0,0)
}

enqueue(void *value){

 sem_wait(&s2)
 p_m_lock(&lock)

 b[ (in++) & (N-1) ] = value

 p_m_unlock(&lock)
 sem_post(&s1)
}

void *dequeue(){
  sem_wait(&s1)
  p_m_lock(&lock)
  void *result = b[(out++) & 15]
  p_m_unlock(&lock)
  sem_post(&s2)

  return result;
}
```

Analysis

- The initial value of s2 is 0. Thus enqueue will block on the first call to sem_wait even though the buffer is empty!
- The initial value of s1 is 16. Thus dequeue will not block on the first call to sem_wait even though the buffer is empty - oops Underflow! The dequeue method will return invalid data.
- The code does not satisfy Mutual Exclusion; two threads can modify `in` or `out` at the same time! The code appears to use mutex lock. Unfortunately the lock was never initialized with `pthread_mutex_init()` or `PTHREAD_MUTEX_INITIALIZER` - so the lock may not work (`pthread_mutex_lock` may simply do nothing)

Correct implementation of a ring buffer

The pseudo-code (`pthread_mutex` shortened to `p_m` etc) is shown below.

As the mutex lock is stored in global (static) memory it can be initialized with `PTHREAD_MUTEX_INITIALIZER`.If we had allocated space for the mutex on the heap, then we would have used `pthread_mutex_init(ptr, NULL)`

```
#include <pthread.h>
#include <semaphore.h>
// N must be 2^i
#define N (16)

void *b[N]
int in = 0, out = 0
p_m_t lock = PTHREAD_MUTEX_INITIALIZER
sem_t countsem, spacesem

void init() {
  sem_init(&countsem, 0, 0)
  sem_init(&spacesem, 0, 16)
}
```

The enqueue method is shown below. Notice: * The lock is only held during the critical section (access to the data structure). * A complete implementation would need to guard against early returns from `sem_wait` due to POSIX signals.

```
enqueue(void *value){
 // wait if there is no space left:
 sem_wait( &spacesem )

 p_m_lock(&lock)
 b[ (in++) & (N-1) ] = value
 p_m_unlock(&lock)

 // increment the count of the number of items
 sem_post(&countsem)
}
```

The dequeue implementation is shown below. Notice the symmetry of the synchronization calls to enqueue. In both cases the functions first wait if the count of spaces or count of items is zero.

```
void *dequeue(){
  // Wait if there are no items in the buffer
  sem_wait(&countsem)

  p_m_lock(&lock)
  void *result = b[(out++) & (N-1)]
  p_m_unlock(&lock)

  // Increment the count of the number of spaces
  sem_post(&spacesem)

  return result
}
```

Food for thought

- What would happen if the order of `pthread_mutex_unlock` and `sem_post` calls were swapped?
- What would happen if the order of `sem_wait` and `pthread_mutex_lock` calls were swapped?

Aside

Process Synchronization

You thought that you were using different processes, so you don't have to synchronize? Think again! You may not have race conditions within a process but what if your process needs to interact with the system around it? Let's consider a motivating example

```
void write_string(const char *data) {
    int fd = open("my_file.txt", O_WRONLY);
    write(fd, data, strlen(data));
    close(fd);
}

int main() {
    if(!fork()) {
        write_string("key1: value1");
        wait(NULL);
    } else {
        write_string("key2: value2");
    }
    return 0;
}
```

If none of the system calls fail then we should get something that looks like this (given the file was empty to begin with).

```
key1: value1
key2: value2
```

or

```
key2: value2
key1: value1
```

Interruption

But, there is a hidden nuance. Most system calls can be `interrupted` meaning that the operating system can stop an ongoing system call because it needs to stop the process. So barring `fork` `wait` `open` and `close` from failing – they typically go to completion – what happens if `write` fails? If write fails and no bytes are written, we can get something like `key1:  value1` or `key2:  value2`. This is data loss which is incorrect but won't corrupt the file. What happens if write gets interrupted after a partial write? We get all sorts of madness. For example,

```
key2: key1: value1
```

Solution

So what should we do? We should use a shared mutex! Consider the following code.

```
pthread_mutex_t * mutex = NULL;
pthread_mutexattr_t attr;

void write_string(const char *data) {
    pthread_mutex_lock(mutex);
    int fd = open("my_file.txt", O_WRONLY);
    int bytes_to_write = strlen(data), written = 0;
    while(written < bytes_to_write) {
        written += write(fd, data + written, bytes_to_write -
            written);
    }
    close(fd);
    pthread_mutex_unlock(mutex);
}

int main() {
    pthread_mutexattr_init(&attr);
```

```
    pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
    pmutex = mmap (NULL, sizeof(pthread_mutex_t),
                PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANON, -1,
                    0);
    pthread_mutex_init(pmutex, &attrmutex);
    if(!fork()) {
        write_string("key1: value1");
        wait(NULL);
        pthread_mutex_destroy(pmutex);
        pthread_mutexattr_destroy(&attrmutex);
        munmap((void *)pmutex, sizeof(*pmutex));
    } else {
        write_string("key2: value2");
    }
    return 0;
}
```

What the code does in main is initialize a process shared mutex using a piece of `shared` memory. You will find out what this call to `mmap` does later – just assume for the time being that it create memory that is shared between processes. We can initialize a `pthread_mutex_t` in that special piece of memory and use it as normal. To counter `write` failing, we have put the `write` call inside a while loop that keeps writing so long as there are bytes left to write. Now if all the other system calls function, there should be more more race conditions.

Most programs try to avoid this problem entirely by writing to separate files, but it is good to know that there are mutexes across processes, and they are useful. You can use all of the primitives that you were taught previously! Barriers, semaphores, and condition variables can all be initialized on a shared piece of memory and used in similar ways to their multithreading counterparts. You don't need to know the implementation, just need to know that mutexes and other synchronization primitives can be shared across processes and some of the benefits.

- You don't have to worry about arbitrary memory addresses becoming race condition candidates. This means that only areas that you specifically `mmap` or outside system resources like files are ever in danger.
- You get the nice isolation of a processes so if one process fails the system can maintain intact
- When you have a lot of threads, creating a process might ease the system load

External Resources

- pthread_mutex_lock man page
- pthread_mutex_unlock man page
- pthread_mutex_init man page
- pthread_mutex_destroy man page
- sem_init
- sem_wait
- sem_post

- sem_destroy

Topics

- Atomic operations
- Critical Section
- Producer Consumer Problem
- Using Condition Variables
- Using Counting Semaphore
- Implementing a barrier
- Implementing a ring buffer
- Using pthread_mutex
- Implementing producer consumer
- Analyzing multi-threaded coded

Questions

- What is atomic operation?
- Why will the following not work in parallel code

```
//In the global section
size_t a;
//In pthread function
for(int i = 0; i < 100000000; i++) a++;
```

  And this will?

```
//In the global section
atomic_size_t a;
//In pthread function
for(int i = 0; i < 100000000; i++) atomic_fetch_add(a, 1);
```

- What are some downsides to atomic operations? What would be faster: keeping a local variable or many atomic operations?
- What is the critical section?
- Once you have identified a critical section, what is one way of assuring that only one thread will be in the section at a time?
- Identify the critical section here

```
struct linked_list;
struct node;
void add_linked_list(linked_list *ll, void* elem){
    node* packaged = new_node(elem);
    if(ll->head){
        ll->head =
    }else{
        packaged->next = ll->head;
        ll->head = packaged;
        ll->size++;
    }

}

void* pop_elem(linked_list *ll, size_t index){
    if(index >= ll->size) return NULL;

    node *i, *prev;
    for(i = ll->head; i && index; i = i->next, index--){
        prev = i;
    }

    //i points to the element we need to pop, prev before
    if(prev->next) prev->next = prev->next->next;
    ll->size--;
    void* elem = i->elem;
    destroy_node(i);
    return elem;
}
```

- How tight can you make the critical section?
- What is a producer consumer problem? How might the above be a producer consumer problem be used in the above section? How is a producer consumer problem related to a reader writer problem?
- What is a condition variable? Why is there an advantage to using one over a `while` loop?
- Why is this code dangerous?

```
if(not_ready){
     pthread_cond_wait(&cv, &mtx);
}
```

- What is a counting semaphore? Give me an analogy to a cookie jar/pizza box/limited food item.
- What is a thread barrier?
- Use a counting semaphore to implement a barrier.
- Write up a Producer/Consumer queue, How about a producer consumer stack?

- Give me an implementation of a reader-writer lock with condition variables, make a struct with whatever you need, it just needs to be able to support the following functions

```
void reader_lock(rw_lock_t* lck);
void writer_lock(rw_lock_t* lck);
void reader_unlock(rw_lock_t* lck);
void writer_unlock(rw_lock_t* lck);
```

  The only specification is that in between `reader_lock` and `reader_unlock`, no writers can write. In between the writer locks, only one writer may be writing at a time.

- Write code to implement a producer consumer using ONLY three counting semaphores. Assume there can be more than one thread calling enqueue and dequeue. Determine the initial value of each semaphore.
- Write code to implement a producer consumer using condition variables and a mutex. Assume there can be more than one thread calling enqueue and dequeue.
- Use CVs to implement add(unsigned int) and subtract(unsigned int) blocking functions that never allow the global value to be greater than 100.
- Use CVs to implement a barrier for 15 threads.
- How many of the following statements are true?
- There can be multiple active readers
- There can be multiple active writers
- When there is an active writer the number of active readers must be zero
- If there is an active reader the number of active writers must be zero
- A writer must wait until the current active readers have finished
- Todo: Analyzing mulithreaded code snippets

Bibliography

7 Deadlock

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

The philosphers Jagger & Richards

Deadlock is defined as when a system cannot make and forward progress. We define a system for the rest of the chapter as a set of rules by which a set of processes can move from one state to another, where a state is either working or waiting for a particular resource. Forward progress is defined as if there is at least one process wokring or we can award a process waiting for a resource that resource. In a lot of systems, Deadlock is just avoided by ignore the entire concept [5, P.237]. Have you heard about turn it on and off again? For products where the stakes are low (User Operating Systems, Phones), it may be more efficient not prevent deadlock. But in the cases where "failure is not an option" - Apollo 13, you need a system that tracks deadlock and breaks it or better yet prevents it entirely. Apollo 13 may have not failed because of deadlock, but probably wouldn't be good to restart the system on liftoff.

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

Resource Allocation Graphs

One such way is modeling the system with a resource allocation graph (RAG). A resource allocation graph tracks which resource is held by which process and which process is waiting for a resource of a particular type. It is very powerful and simple tool to illustrate how interacting processes can deadlock. If a process is *using* a resource, an arrow is drawn from the resource node to the process node. If a process is *requesting* a resource, an arrow is drawn from the process node to the resource node. If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will deadlock. For example, if process 1 holds resource A, process 2 holds resource B and process 1 is waiting for B and process 2 is waiting for A, then process 1 and 2 process will be deadlocked 7.1.

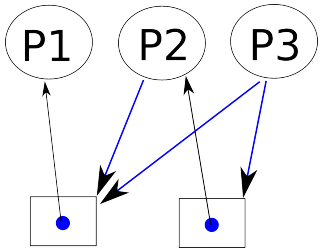


Figure 7.1: Resource allocation graph

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

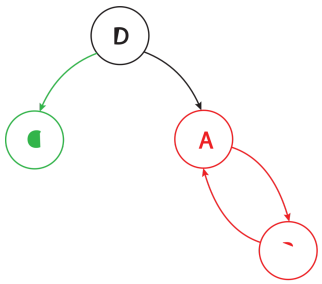


Figure 7.2: Graph based Deadlock

Coffman conditions

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

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

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

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

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

of list of requests that a process uses to release a resource at any given time. The evolution of the system is at each step at every time.

- Release all resources in L_t.
- Find a process that is requesting a resource
- If that resource is available give it to that process, generating a new (h_{t+1}, w_{t+1}) and exit the current iteration.
- Else find another process and try the same resource allocation procedure in the previous step.

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

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

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

If you break any of them, you cannot have deadlock! Consider the scenario where two students need to write both pen and paper and there is only one of each. Breaking mutual exclusion means that the students share the pen and paper. Breaking circular wait could be that the students agree to grab the pen then the paper. As a proof by contradiction, say that deadlock occurs under the rule and the conditions. Without loss of generality, that means a student would have to be waiting on a pen while holding the paper and the other waiting on a pen and holding the paper. We have contradicted ourselves because one student grabbed the paper without grabbing the pen, so deadlock must not be able to occur. Breaking hold and wait could be that the students try to get the pen and then the paper and if a student fails to grab the paper then they release the pen. This introduces a new problem called *livelock* which will be discussed latter. Breaking preemption means that if the two students are in deadlock the teacher can come in and break up the deadlock by giving one of the students one the held on items or tell both students to put the items down.

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

Approaches to solving deadlock

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

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

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

Aside

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

- A person j requests m from me
  - if $m < p$, they are denied.
  - if $m + l_j > a_i$ they are denied
  - Pretend we are in a new state $(A = \{..., a_j, ...\}, L = \{.., l_j + m, ...\}, p - m)$ where the process is granted the resource.

- if now person j is either satisfied ($l_j == a_j$) or $\exists i, a_j - l_j < p$. In other words we have enough money to satisfy one other person. If either, consider the transaction safe and give them the money.

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

- The program first needs to know how much of each resource a process needs. A lot of times that is impossible or the process requests the wrong amount because the programmer didn't forsee it.
- The system could livelock.
- We know in most systems that resources are generally not homogenous. Of course there are things like pipes and sockets but for the most part there is only 1 of a particular file. This could mean that the runtime of the algorithm could be slow for systems with millions of resources.
- Also, this can't keep track of resources that come and go. A process may delete a resource as a side effect or create a resource. The algorithm assumes a static allocation and that each process performs a non-destructive operation.

Dining Philosophers

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

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

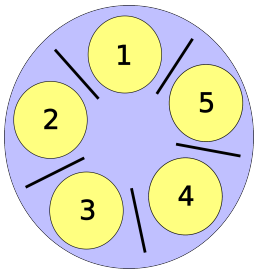


Figure 7.3: Dining Philosophers

Failed Solutions

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

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

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

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

Viable Solutions

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

There are a lot of problems with these solutions. One is that they are slow and have a single point of failure or the arbitrator. Assuming that all the philosophers are good-willed, the arbitrator needs to be fair and be able to determine if a transaction would cause deadlock in the multi-arbitrator case. Further more in practical systems, the arbitrator tends to give forks to the same processes because of scheduling or pseudorandomness. Another

important thing to note is that this prevents deadlock for the entire system. But in our model of dining philosophers, the philosopher has to release the lock themselves. Then you can consider the case of the malicious philosopher (let's say Decartes because of his Evil Demons) could hold on to the arbitrator forever. He would make forward progress and the system would make forward progress but there is no way of ensuring that each process makes forward progress without assuming something about the processes or having true preemption – meaning that a higher authority (let's say Steve Jobs) tells them to stop eating forcibly.

TODO: Prove arbitrator's doesn't deadlock

Leaving the Table (Stallings' Solution)

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

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

Proof: Stallings' Solution Doesn't Deadlock.

Let's number the philosophers $\{p_0, p_1, .., p_{n-1}\}$ and the resources $\{r_0, r_1, .., r_{n-1}\}$. A philosopher p_i needs resource $r_{i-1 \mod n}$ and $r_{i+1 \mod n}$. Without loss of generality, let us take p_i out of the picture. Each resource had exactly two philosophers that could use it. Now resources $r_{i-1 \mod n}$ and $r_{i+1 \mod n}$ only have on philosopher waiting on it. Even if hold and wait, no preemption, and mutual exclusion or present, the resources can never enter a state where one philosopher requests them and they are held by another philosopher because only one philosopher can request them. Since there is no way to generate a cycle otherwise, circular wait cannot hold. Since circular wait cannot hold, deadlock cannot happen.

□

Partial Ordering (Dijkstra's Solution)

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

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

TODO: Prove dijkstra's doesn't deadlock

Aside

Advanced Solutions: Clean/Dirty Forks (Chandra/Misra Solution)

There are many more advanced solutions.

TODO: Detail the Clean/Dirty Solution, and cite

Advanced Solutions: Actor Model (other Message passing models)

TODO: Detail the Actor Model, and cite

Topics

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

Questions

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

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

- The following calls are made

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

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

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

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

(Draw out the resource graph!)

Bibliography

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

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

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

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

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

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

8

Scheduling

I wish that I could fly
There's danger if I dare to stop and here's the reason why
You see I'm overdue
I'm in a rabbit stew
Can't even say "Good-bye", hello
I'm late, I'm late, I'm late
No, no, no, no, no, no, no!

Alice in Wonderland

CPU Scheduling is the problem of efficiently selecting which process to run on a system's CPU cores. In a busy system, there will be more ready-to-run processes than there are CPU cores, so the system kernel must evaluate which processes should be scheduled to run and which processes should be executed later. The system must also decide whether or not it should take a particular process and pause its execution – along with any associated threads. The balance comes from stopping processes often enough where you have a responsive computer but infrequently enough where the programs themselves are not spending a lot of cycles context switching. It is a hard balance it get right.

The additional complexity of multi-threaded and multiple CPU cores are considered a distraction to this initial exposition so are ignored here. Another gotcha for non-native speakers is the dual meanings of "Time": The word "Time" can be used in both clock and elapsed duration context. For example "The arrival time of the first process was 9:00am." and, "The running time of the algorithm is 3 seconds".

One clarification that we will make is that our scheduling will mainly deal with short term or cpu scheduling. That means we will assume that the processes are in memory and ready to go. The other types of scheduling are long and medium term. Long term schedulers act as gatekeepers to the processing world. When a process requests another process to be executed, it can either tell tell the process yes, no, or wait. The medium term scheduler deals with the caveats of moving a process from the paused state in memory to the paused state on disk when there are too many processes or some process are known not to be used for a significant amount of CPU cycles (think about a process that only checks something once an hour).

Measurements

Scheduling effects the performance of the system, specifically the *latency* and *throughput* of the system. The throughput might be measured by a system value, for example the I/O throughput - the number of bits written per second, or number of small processes that can complete per unit time. The latency might be measured by the response time – elapse time before a process can start to send a response – or wait time or turnaround time –the elapsed time to complete a task. Different schedulers offer different optimization trade-offs that may or may not be appropriate to desired use. There is no optimal scheduler for all possible environments and goals. For example 'shortest-job-first' will minimize total wait time across all jobs but in interactive (UI) environments it would be preferable to minimize response time at the expense of some throughput, while FCFS seems intuitively fair and

easy to implement but suffers from the Convoy Effect. Arrival time is the time at which a process first arrives at the ready queue, and is ready to start executing. If a CPU is idle, the arrival time would also be the starting time of execution.

What is preemption?

Without preemption processes will run until they are unable to utilize the CPU any further. For example the following conditions would remove a process from the CPU and the CPU would be available to be scheduled for other processes: The process terminates due to a signal, is blocked waiting for concurrency primitive, or exits normally. Thus once a process is scheduled it will continue even if another process with a high priority (e.g. shorter job) appears on the ready queue.

With preemption, the existing processes may be removed immediately if a more preferable process is added to the ready queue. For example, suppose at t=0 with a Shortest Job First scheduler there are two processes (P1 P2) with 10 and 20 ms execution times. P1 is scheduled. P1 immediately creates a new process P3, with execution time of 5 ms, which is added to the ready queue. Without preemption, P3 will run 10ms later (after P1 has completed). With preemption, P1 will be immediately evicted from the CPU and instead placed back in the ready queue, and P3 will be executed instead by the CPU.

Which schedulers suffer from starvation?

Any scheduler that uses a form of prioritization can result in starvation because earlier processes may never be scheduled to run (assigned a CPU). For example with SJF, longer jobs may never be scheduled if the system continues to have many short jobs to schedule. It all depends on the type of scheduler.

Why might a process (or thread) be placed on the ready queue?

A process is placed on the ready queue when it is able to use a CPU. Some examples include:

- A process was blocked waiting for a `read` from storage or socket to complete and data is now available.
- A new process has been created and is ready to start.
- A process thread was blocked on a synchronization primitive (condition variable, semaphore, mutex lock) but is now able to continue.
- A process is blocked waiting for a system call to complete but a signal has been delivered and the signal handler needs to run.

Similar examples can be generated when considering threads.

Measures of Efficiency

First some definitions

1. `start_time` is the wall-clock start time of the process (CPU starts working on it)
2. `end_time` is the end wall-clock of the process (CPU finishes the process)
3. `run_time` is the total amount of CPU time required
4. `arrival_time` is the time the process enters the scheduler (CPU may not start working on it)

Here are measures of efficiency

1. `Turnaround Time` is the total time from when you the process arrives to when it ends. `end_time - arrival_time`
2. `Response Time` is the total latency (time) that it takes from when the process arrives to when the CPU actually starts working on it. `start_time - arrival_time`

3. `Wait Time` is the *total* wait time i.e. the total time that a process is on the ready queue. A common mistake is to believe it is only the initial waiting time in the ready queue. If a CPU intensive process with no I/O takes 7 minutes of CPU time to complete but required 9 minutes of wall-clock time to complete we can conclude that it was placed on the ready-queue for 2 minutes. For those 2 minutes the process was ready to run but had no CPU assigned. It does not matter when the job was waiting, the wait time is 2 minutes. `end_time - arrival_time - run_time`

What is the Convoy Effect?

"The Convoy Effect is where I/O intensive processes are continually backed up, waiting for CPU-intensive processes that hog the CPU. This results in poor I/O performance, even for processes that have tiny CPU needs."

Suppose the CPU is currently assigned to a CPU intensive task and there is a set of I/O intensive processes that are in the ready queue. These processes require just a tiny amount of CPU time but they are unable to proceed because they are waiting for the CPU-intensive task to be removed from the processor. These processes are starved until the the CPU bound process releases the CPU. But the CPU will rarely be released (for example in the case of a FCFS scheduler, we must wait until the processes is blocked due to an I/O request). The I/O intensive processes can now finally satisfy their CPU needs, which they can do quickly because their CPU needs are small and the CPU is assigned back to the CPU-intensive process again. Thus the I/O performance of the whole system suffers through an indirect effect of starvation of CPU needs of all processes.

This effect is usually discussed in the context of FCFS scheduler, however a round robin scheduler can also exhibit the Convoy effect for long time-quanta.

Aside

Linux Scheduling

As of February 2016, Linux by default uses the *Completely Fair Scheduler* for CPU scheduling and the Budget Fair Scheduling "BFQ" for I/O scheduling. Appropriate scheduling can have a significant impact on throughput and latency. Latency is particularly important for interactive and soft-real time applications such as audio and video streaming. See the discussion and comparative benchmarks here for more information.

Here is how the CFS schedules

- The CPU creates a Red-Black tree with the processes virtual runtime (runtime / nice_value) and sleeper fairness flag (if the process is waiting on something give it the CPU when it is done waiting).
- (Nice values are the kernel's way of giving priority to certain processes, the lower nice value the higher priority)
- The kernel chooses the lowest one based on this metric and schedules that process to run next, taking it off the queue. Since the red-black tree is self balancing this operation is guaranteed $O(log(n))$ (selecting the min process is the same runtime)

Although it is called the Fair Scheduler there are a fair bit of problems.

- Groups of processes that are scheduled may have imbalanced loads so the scheduler roughly distributes the load. When another CPU gets free it can only look at the average load of a group schedule not the individual cores. So the free CPU may not take the work from a CPU that is burning so long as the average is fine.
- If a group of processes is running on non-adjacent cores then there is a bug. If the two cores are more than a hop away, the load balancing algorithm won't even consider that core. Meaning if a CPU is free and a CPU that is doing more work is more than a hop away, it won't take the work (may have been patched).

- After a thread goes to sleep on a subset of cores, when it wakes up it can only be scheduled on the cores that it was sleeping on. If those cores are now busy, the thread will have to wait on them, wasting opportunities to use other idle cores.
- To read more on the problems of the Fair Scheduler, read here.

Scheduling Algorithms

For all the examples,
Process 1: Runtime 1000ms
Process 2: Runtime 2000ms
Process 3: Runtime 3000ms
Process 4: Runtime 4000ms
Process 5: Runtime 5000ms

Shortest Job First (SJF)

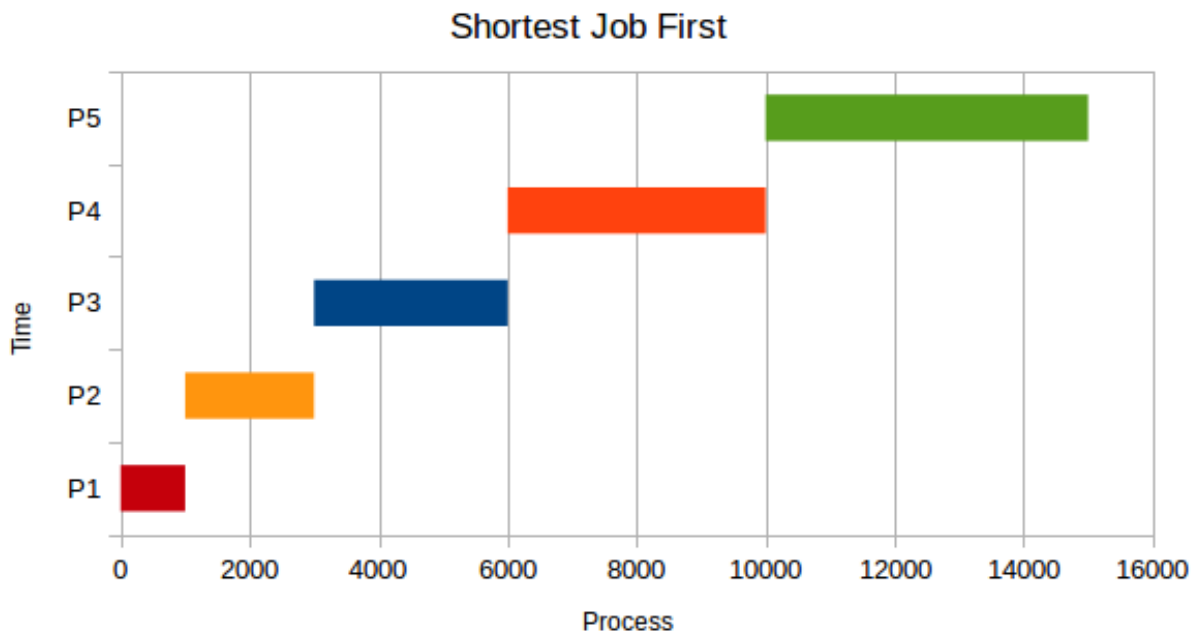


Figure 8.1:

- P1 Arrival: 0ms
- P2 Arrival: 0ms
- P3 Arrival: 0ms
- P4 Arrival: 0ms
- P5 Arrival: 0ms

The processes all arrive at the start and the scheduler schedules the job with the shortest total CPU time. The glaring problem is that this scheduler needs to know how long this program will run over time before it ran the program.

Technical Note: A realistic SJF implementation would not use the total execution time of the process but the burst time (the total CPU time including future computational execution before the process will no longer be ready to run). The expected burst time can be estimated by using an exponentially decaying weighted rolling average based on the previous burst time but for this exposition we will simplify this discussion to use the total running time of the process as a proxy for the burst time.

Advantages

1. Shorter jobs tend to get run first

Disadvantages

1. Needs algorithm to be omniscient

Preemptive Shortest Job First (PSJF)

Preemptive shortest job first is like shortest job first but if a new job comes in with a shorter runtime than the total runtime of the current job, it is run instead. (If it is equal like our example our algorithm can choose). The scheduler uses the *total* runtime of the process. If you want the shortest *remaining* time left, that is a variant of PSJF called Shortest Remaining Time First (SRTF).

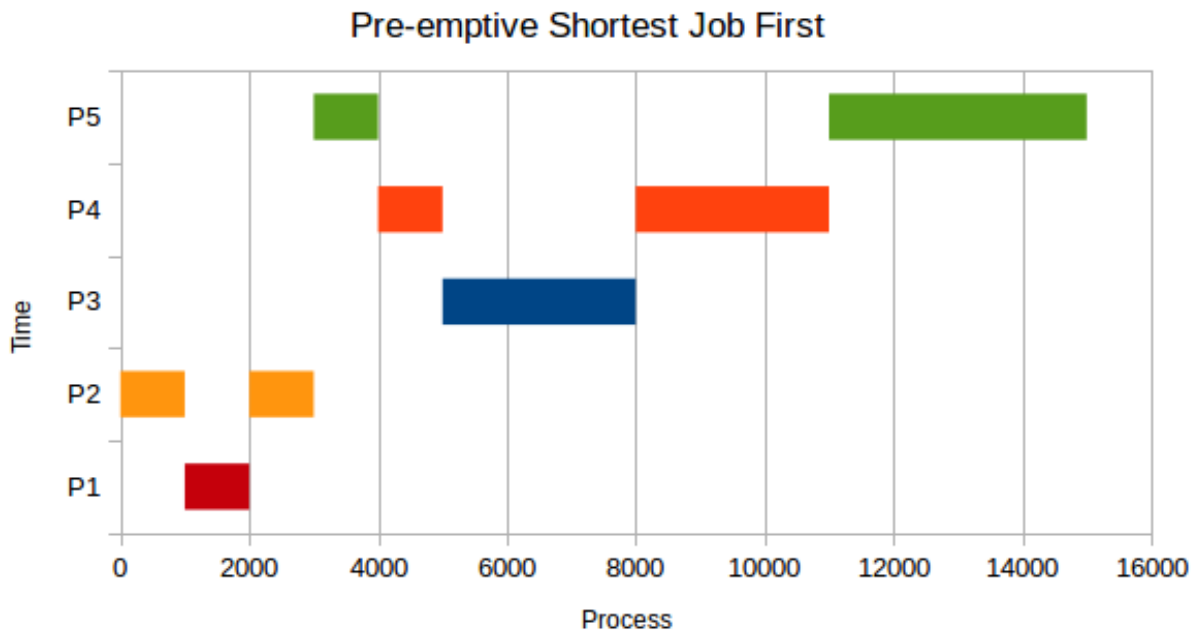


Figure 8.2:

- P2 at 0ms
- P1 at 1000ms
- P5 at 3000ms
- P4 at 4000ms
- P3 at 5000ms

Here's what our algorithm does. It runs P2 because it is the only thing to run. Then P1 comes in at 1000ms, P2 runs for 2000ms, so our scheduler preemptively stops P2, and let's P1 run all the way through (this is completely up to the algorithm because the times are equal). Then, P5 Comes in – since there are no processes running, the scheduler will run process 5. P4 comes in, and since the runtimes are equal P5, the scheduler stops P5 and runs P4. Finally P3 comes in, preempts P4, and runs to completion. Then P4 runs, then P5 runs.

Advantages

1. Ensures shorter jobs get run first

Disadvantages

1. Need to know the runtime again

First Come First Served (FCFS)

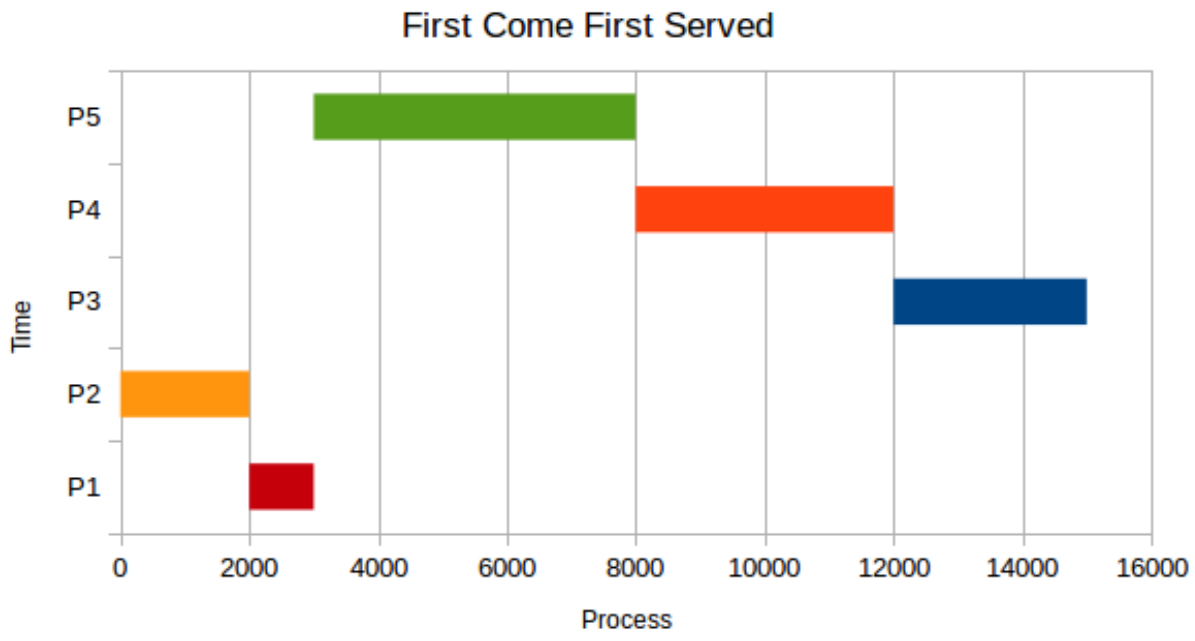


Figure 8.3:

- P2 at 0ms
- P1 at 1000ms
- P5 at 3000ms
- P4 at 4000ms
- P3 at 5000ms

Processes are scheduled in the order of arrival. One advantage of FCFS is that scheduling algorithm is simple: the ready queue is a just a FIFO (first in first out) queue. FCFS suffers from the Convoy effect. Here P2 Arrives, then P1 arrives, then P5, then P4, then P3. You can see the convoy effect for P5.

Advantages

- Simple algorithm and implementation

- Context switches infrequent when there are long running processes
- No starvation if all processes are guarenteed to terminate

Disadvantages

- Simple algorithm and implementation
- Context switches infrequent when there are long running processes

Round Robin (RR)

Processes are scheduled in order of their arrival in the ready queue. However after a small time step a running process will be forcibly removed from the running state and placed back on the ready queue. This ensures that a long-running process can not starve all other processes from running. The maximum amount of time that a process can execute before being returned to the ready queue is called the time quanta. In the limit of large time quanta (where the time quanta is longer than the running time of all processes) round robin will be equivalent to FCFS.

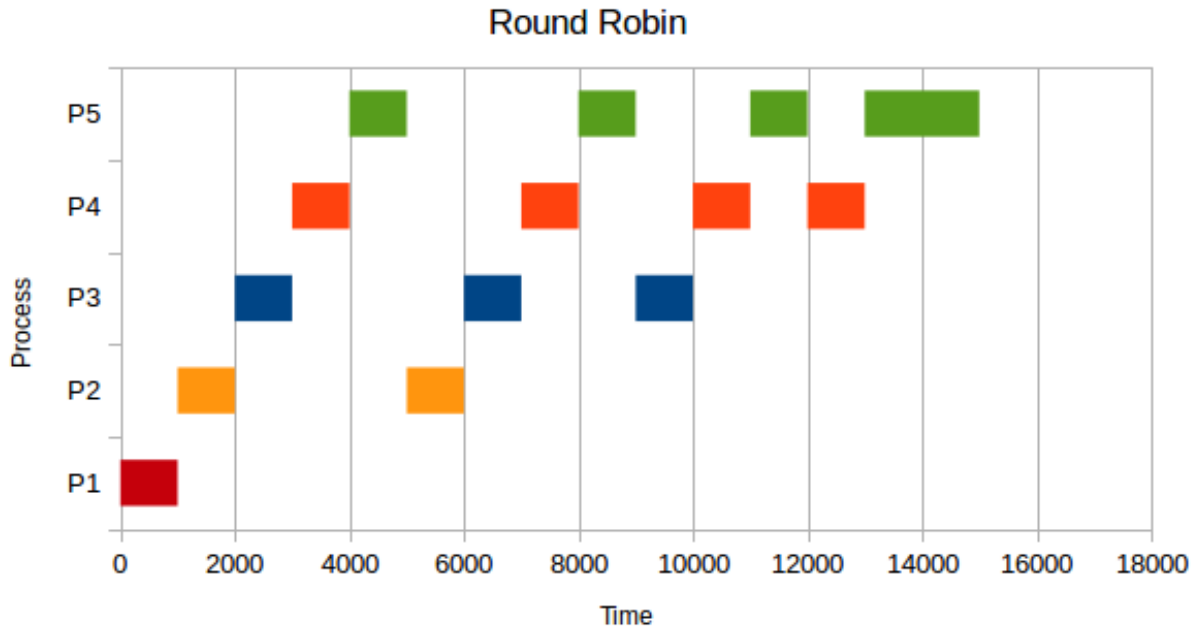


Figure 8.4:

- P1 Arrival: 0ms
- P2 Arrival: 0ms
- P3 Arrival: 0ms
- P4 Arrival: 0ms
- P5 Arrival: 0ms

Quantum = 1000ms

Here all processes arrive at the same time. P1 is run for 1 quantum and is finished. P2 for one quantum; then, it is stopped for P3. After all other processes run for a quantum we cycle back to P2 until all the processes are finished.

Advantages

1. Ensures some notion of fairness

Disadvantages

1. Large number of processes = Lots of switching

Priority

Processes are scheduled in the order of priority value. For example, a navigation process might be more important to execute than a logging process.

Topics

- Scheduling Algorithms
- Measures of Efficiency

Questions

- What is scheduling?
- What is turnaround time? Response Time? Wait time?
- What is the convoy effect?
- Which algorithms have the best turnaround/response/wait time on average

Bibliography

9 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?

```c
//Program 1

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

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

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

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] = 'l';
```

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`.

10 Networking

> The Web as I envisaged it, we have not seen it yet. The future is still so much bigger than the past
>
> Tim Berners-Lee

Networking has become arguably the most important use of computers in the past 10-20 years. Most of us nowadays can't stand a place without wifi or any connectivity, so it is crucial as programmers that you have an understanding of networking and how to program to communicate across networks. Although it may sound complicated, POSIX has defined nice standards that make connecting to the outside world easy. POSIX also lets you peer underneath the hood and optimize all the little parts of each connection to write high performant

The OSI Model

The Open Source Interconnection 7 layer model (OSI Model) is a sequence of segments that define standards for both infrastructure and protocols for forms of radio communication, in our case the internet. The 7 layer model is as follows

1. Layer 1: The physical layer. These are the actual waves that carry the bauds across the wire. As an aside, bits don't cross the wire because in most mediums you can alter two characterstics of a wave – the amplitude and the frequency – and get more bits per clock cycle.
2. Layer 2: The link layer. This is how each of the agents react to certain events (error detection, noisy channels, etc). This is where Ethernet and WiFi live.
3. Layer 3: The network layer. This is the heart of the internet. The bottom two protocols deal with communication between two different computers that are directly connected. This layer deals with routing packets from one endpoint to another.
4. Layer 4: The transport layer. This layer specifies how the slices of data are received. The bottom three layers make no guarantee about the order that packets are received and what happens when a packet is dropped. Using different protocols, this layer can.
5. Layer 5: The session layer. This layer makes sure that if a connection in the previous layers is dropped, a new connection in the lower layers can be established, and it looks like a nothing happened to the end user.
6. Layer 6: The presentation layer. This layer deals with encryption, compression, and data translation. For example, portability between different operating systems like translating newlines to windows newlines.
7. Layer 7: The application layer. The application layer is where many different protocols live. HTTP and FTP are both defined at this level. This is typically where we define protocols across the internet. As programmers, we only go lower when we think we can create algorithms that are more suited to our needs than all of the below.

Just to be clear this is not a networking class. We won't go over most of these layers in depth. We will focus on some aspects of layers 3, 4, and 7 because they are essential to know if you are going to be doing something with the internet, which at some point in your career you will be. As for another definition, a protocol is a set of specifications put forward by the Internet Engineering Task Force that govern how implementers of protocol have their program or circuit behave under specific circumastnces.

Layer 3: The Internet Protocol

The following is the "30 second" introduction to internet protocol (IP) - which is the primary way to send packets ("datagrams") of information from one machine to another. "IP4", or more precisely, "IPv4" is version 4 of the Internet Protocol that describes how to send packets of information across a network from one machine to another. Roughly 95% of all packets on the Internet today are IPv4 packets. A significant limitation of IPv4 is that source and destination addresses are limited to 32 bits. IPv4 was designed at a time when the idea of 4 billion devices connected to the same network was unthinkable - or at least not worth making the packet size larger. IPv4 address are written typically in a sequence of four octets delimited by periods "255.255.255.0" for example.

Each IPv4 packet includes a very small header - typically 20 bytes (more precisely, "octets"), that includes a source and destination address. Conceptually the source and destination addresses can be split into two: a network number (the upper bits) and the lower bits represent a particular host number on that network.

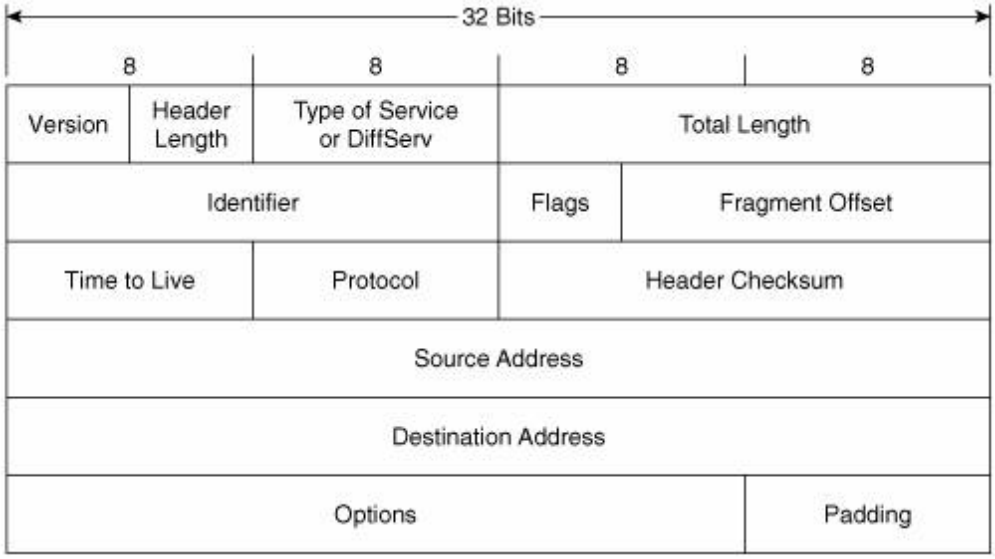
A newer packet protocol "IPv6" solves many of the limitations of IPv4 like making routing tables simpler and 128 bit addresses. However, less than 5% of web traffic is IPv6 based. We write IPv6 addresses in a sequence of eight, four hexadecimal delimiters like "1F45:0000:0000:0000:0000:0000:0000:0000". Since that can get unruly, we can omit the zeros "1F45::".

A machine can have an IPv6 address and an IPv4 address.

A special IPv4 address is `127.0.0.1`, IPv6 as `0:0:0:0:0:0:0:1` or `::1` also known as localhost. Packets sent to 127.0.0.1 will never leave the machine; the address is specified to be the same machine.

In-depth IPv4 Specification

The internet protocol deals with routing, fragmentation, and reassembly of datagram fragments. Datagrams are formatted as such



1. The first octet is the version number, either 4 or 6
2. The next octet is how long the header is. Although it may seem that the header is constant size, you can include optional parameters to augment the path taken or other instructions
3. The next two octets specify the total length of the datagram. This means this is the header, the data, footer, and padding. This is given in multiple of octets, meaning that a value of 20 means 20 octets.

4. The next two are Identification number. IP handles taking packets that are too big to be sent over the phsyical wire and chunks them up. As such, this number identifies what datagram this originally belonged to.
5. The next octet is various bit flags that can be set.
6. The next octet and half is fragment number. If this packet was fragmented, this is the number this fragment represents
7. The next octet is time to live. So this is the number of "hops" (travels over a wire) a packet is allowed to go. This is set because different routing protocols could cuase packets to go in circles, the packets must be dropped at some point.
8. The next octet is the protocol number. Although protocols between different layers of the OCI model are supposed to be black boxes, this is included, so that hardware can peer into the underlying protocol efficiently. Take for example IP over IP (yes you can do that!). Your ISP wraps IPv4 packets sent from your computer to the ISP in another IP layer and sends the packet off to be delivered to the website. On the reverse trip the packet is "unwrapped" and the original IP datagram is sent to your computer. This was done because we ran out of IP addresses, and this adds additional overhead but it is a necessary fix. Other common protocols are TCP, UDP, etc.
9. The next two octets is an internet checksum. This is a CRC that is calculated to make sure that a wide variety of bit errors are detected.
10. Source address is what people generally refer to as the IP address. There is no verification of this, so one host can pretend to be any IP address possible
11. Destination address is where you want the packet to be sent to. This is crucial in the routing process as you need that to route.
12. After: Your data! All layer of higher order protocols are put in there
13. Additional options: Hosts of additional options
14. Footer: A bit of padding to make sure your data is a multiple of 8

Aside

Routing

The internet protocol routing is an amazing intersection of theory and application. We can imagine the entire internet as a set of graphs. Most peers are connected to what we call "peering points" these are the WIFI routers and the ethernet ports that one finds in their house, work, or public. These peering points are then connected to a wired network of routers, switches, and servers that all route themselves. At a top level there are two types of routing

1. Internal Routing Protocols. Internal protocols is routing designed for within an ISP's network. These protocols are meant to be fast and more trusting because all computers, switches, and routers are part of an ISP. communication between two routers.
2. External Routing Protocols. These typically happen to be ISP to ISP protocol. Certain routers are designated as border routers. These routers talk to routers from ISPs have have different policies from accepting or receiving packets. If an evil ISP is trying to dump all network traffic onto your ISP, these routers would deal with that. These protocols also deal with gathering information about the outside world to each router. In most routing protocols using link state or OSPF, a router must necessarily calculate the shortest path to the destination. This means it needs information about the "foreign" routers which is disseminated according to these protocols.

These two protocols have to interplay with each other nicely in order to make sure that packets are mostly delivered. In addition, ISPs need to be nice to each other because theoretically an ISP can handle lower load by forwarding all packets to another ISP. If everyone does that then, no packets get delivered at all which won't make customers happy at all. So these two protocols need to be fair so the end result works

If you want to read more about this, look at the wikipedia page for routing here Routing.

Fragmentation/Reassembly

Lower layers like WiFi and Ethernet have maximum transmission sizes. The reason being is

1. One host shouldn't crowd the medium for too long
2. If an error occurs, we want some sort of "progress bar" on how far the communication has gone instead of retransmitting the stream
3. There are physical limitations as well, keeping a laser beam in optics working continuously may cause bit errors.

As such if the internet protocol receives a packet that is too big for the maximum size, it must chunk it up. TCP calculates how many datagrams it needs to construct a packet and ensures that they are all transmitted and reconstructed at the end receiver. The reason that we barely use this feature is that if any fragment is lost, the entire packet is lost. Meaning that, assuming the probability of receiving a packet assuming each fragment is lost with an independent percentage, the probability of successfully sending a packet drops off exponentially as packet size increases.

As such, TCP slices its packets so that it fits inside on IP datagram. The only time that this applies is when sending UDP packets that are too big, but most people who are using UDP optimize and set the same packet size as well.

IP Multicast

A little known feature is that using the IP protocol one can send a datagram to all devices connected to a router in what is called a multicast. Multicasts can also be configured with groups, so one can efficiently slice up all connected routers and send a piece of information to all of them efficiently. To access this in a higher protocol, you need to use UDP and specify a few more options. Note that this will cause undo stress on the network, so a series of multicasts could flood the network fast.

What's the deal with IPv6?

One of the big features of IPv6 is the address space. The world ran out of IP addresses a while ago and has been using hacks to get around that. With IPv6 there are enough internal and external addresses, so that unless we discover alien civilizations, we probably won't run out. The other benefit is that these addresses are leased not bought, meaning that if something drastic happens in let's say the internet of things and there needs to be a change in the block addressing scheme, it can be done.

Another big feature is security through IPsec. IPv4 was designed with little to no security in mind. As such, now there is a key exchange similar to TLS in higher layers that allows you to encrypt communication.

Another feature is simplified processing. In order to make the internet fast, IPv4 and IPv6 headers alike are actually implemented in hardware. That means that all header options are processed in circuits as they come in. The problem is that as the IPv4 spec grew to include a copious amount of headers, the hardware had to become more and more advanced to support those headers. IPv6 reorders the headers so that packets can be dropped and routed with less hardware cycles. In the case of the internet, every cycle matters when trying to route the world's traffic.

getnameinfo Example: What's my IP address?

To obtain a linked list of IP addresses of the current machine use `getifaddrs` which will return a linked list of IPv4 and IPv6 IP addresses (and potentially other interfaces too). We can examine each entry and use `getnameinfo` to print the host's IP address. The ifaddrs struct includes the family but does not include the sizeof the struct.

Therefore we need to manually determine the struct sized based on the family (IPv4 v IPv6)

```
(family == AF_INET) ? sizeof(struct sockaddr_in) : sizeof(struct
    sockaddr_in6)
```

The complete code is shown below.

```
    int required_family = AF_INET; // Change to AF_INET6 for IPv6
    struct ifaddrs *myaddrs, *ifa;
    getifaddrs(&myaddrs);
    char host[256], port[256];
    for (ifa = myaddrs; ifa != NULL; ifa = ifa->ifa_next) {
        int family = ifa->ifa_addr->sa_family;
        if (family == required_family && ifa->ifa_addr) {
            if (0 == getnameinfo(ifa->ifa_addr,
                                 (family == AF_INET) ? sizeof(struct
                                     sockaddr_in) :
                                 sizeof(struct sockaddr_in6),
                                 host, sizeof(host), port, sizeof(port)
                                  , NI_NUMERICHOST | NI_NUMERICSERV ))
                puts(host);
            }
        }
```

IP Address, Shell Version

Answer: use `ifconfig` (or Windows's ipconfig) However this command generates a lot of output for each interface, so we can filter the output using grep

```
ifconfig | grep inet

Example output:
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
    inet6 fe80::7256:81ff:fe9a:9141%en1 prefixlen 64 scopeid 0x5
    inet 192.168.1.100 netmask 0xffffff00 broadcast 192.168.1.255
```

How do I use `getaddrinfo` to convert the hostname into an IP address?

The function `getaddrinfo` can convert a human readable domain name (e.g. `www.illinois.edu`) into an IPv4 and IPv6 address. In fact it will return a linked-list of addrinfo structs:

```
struct addrinfo {
    int             ai_flags;
```

```
    int             ai_family;
    int             ai_socktype;
    int             ai_protocol;
    socklen_t       ai_addrlen;
    struct sockaddr *ai_addr;
    char           *ai_canonname;
    struct addrinfo *ai_next;
};
```

It's very easy to use. For example, suppose you wanted to find out the numeric IPv4 address of a webserver at www.bbc.com. We do this in two stages. First use getaddrinfo to build a linked-list of possible connections. Secondly use `getnameinfo` to convert the binary address into a readable form.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

struct addrinfo hints, *infoptr; // So no need to use memset global
    variables

int main() {
  hints.ai_family = AF_INET; // AF_INET means IPv4 only addresses

  int result = getaddrinfo("www.bbc.com", NULL, &hints, &infoptr);
  if (result) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
    exit(1);
  }

  struct addrinfo *p;
  char host[256];

  for(p = infoptr; p != NULL; p = p->ai_next) {

    getnameinfo(p->ai_addr, p->ai_addrlen, host, sizeof(host),
        NULL, 0, NI_NUMERICHOST);
    puts(host);
  }

  freeaddrinfo(infoptr);
  return 0;
}
```

Typical output:

```
212.58.244.70
212.58.244.71
```

If you are wondering how the the computer maps hostnames to addresses, we will talk about that in Layer 7. Spoiler: It is a service called DNS

What is a port?

A port is actually not a part of the IP protocol. IP packets are sent to addresses, but we frequently see IP addresses linked with ports. Ports help higher order protocols multiplexes multiple clients sending packets to the same IP address.

A process can listen for incoming packets on a particular port. However only processes with super-user (root) access can listen on ports < 1024. Any process can listen on ports 1024 or higher. An often used port is port 80: Port 80 is used for unencrypted http requests (i.e. web pages). For example, if a web browser connects to http://www.bbc.com/, then it will be connecting to port 80.

Layer 4: TCP and Client

What is TCP? When is it used?

TCP is a connection-based protocol that is built on top of IPv4 and IPv6 (and therefore can be described as "TCP/IP" or "TCP over IP"). TCP creates a *pipe* between two machines and abstracts away the low level packet-nature of the Internet: Thus, under most conditions, bytes sent from one machine will eventually arrive at the other end without duplication or data loss.

TCP will automatically manage resending packets, ignoring duplicate packets, re-arranging out-of-order packets and changing the rate at which packets are sent.

TCP's three way handshake is known as SYN, SYN-ACK, and ACK. The diagram on this page helps with understanding the TCP handshake. TCP Handshake

Most services on the Internet today (e.g. a web service) use TCP because it hides the complexity of lower, packet-level nature of the Internet.

Aside

TCP In Depth

TCP has a number of features that set it apart from the other transport protocol UDP.

1. Ports. With IP, you are only allowed to send packets to a machine. If you want one machine to handle multiple flows of data, you have to do it manually with IP. TCP abstracts that an gives the programmer a set of virtual sockets. Clients specify the socket that you want the packet sent to and the TCP protocol makes sure that applications that are waiting for packets on that port receive that.
2. Retransmission. Packets can get dropped due to network errors or congestion. As such, they need to be retransmitted but at the same time the retransmission shouldn't cause packets more packets to be dropped. This needs to balance the tradeoff between flooding the network and speed.
3. Out of order packets. Packets may get routed more favorably due to various reasons in IP. If a later packet arrives before another packet, the protocol should detect and reorder them.
4. Duplicate packets. Packets can arrive twice. Packets can arrive twice. As such, a protocol need to be able to differentiate between two packets given a sequence number subject to overflow.
5. Error correction. There is a TCP checksum that handles bit errors. This is rarely used though.
6. Flow Control. Flow control is performed on the receiver side. This may be done so that a slow receiver doesn't get overwhelmed with packets. Servers especially that may handle 10000 or 10

million concurrent connection may need to tell receivers to slow down, but not disconnect due to load. There are also other prorblem of making sure the local network is not overwhelmed

7. Congestion control. Congestion control is performed on the sender side. Congestion control is to avoid a sender from flooding the network with too many packets. This is really important to make sure that each TCP connection is treated fairly. Meaning that two connections leaving a computer to google and youtube receive the same bandwidth and ping as each other. One can easily define a protocol that takes all the bandwidth and leaves other protocols in the dust, but this tends to be malicious because more often than not limiting a computer to a single TCP connection will yield the same result.
8. Connection oriented/lifecycle oriented. You can really imagine a TCP connection as a series of bytes sent through a pipe. TCP handles setting up the connection through SYN SYN-ACK ACK. Then every so often if a packet is not sent, TCP will trade zero length packets to make sure the connection is still alive. At any point, the client and server can say that I am done writing on this connection, I will only consume and same for reading. Finally TCP has ways of detecting a connection dies and whether a connection comes back to life in the specified time frame.

There are a list of things that TCP doesn't provide though

1. Security. This means that if you connect to an IP address that says that it is a certain website, TCP does not verify that this website is in fact that IP address. You could be sending packets to a malicious computer.
2. Encryption. Anybody can listen in on plain TCP. The packets in transport are in plain text meaning that important things like your passwords could easily be skimmed by servers and regularly are.
3. Session Reconnection. This is handled by a higher protocols, but if a TCP connection dies then a whole new one hast to be created and the transmission has to be started over again.
4. Delimiting Requests. TCP is naturally connection oriented. Applications that are communicating over TCP need to find a unique way of telling each other that this request or response is over. HTTP uses either a length field or one keeps listening until the connection closes

Note on network orders

Integers can be represented in least significant byte first or most-significant byte first. Either approach is reasonable as long as the machine itself is internally consistent. For network communications we need to standardize on agreed format.

`htons(xyz)` returns the 16 bit unsigned integer 'short' value xyz in network byte order. `htonl(xyz)` returns the 32 bit unsigned integer 'long' value xyz in network byte order.

These functions are read as 'host to network'; the inverse functions (ntohs, ntohl) convert network ordered byte values to host-ordered ordering. So, is host-ordering little-endian or big-endian? The answer is - it depends on your machine! It depends on the actual architecture of the host running the code. If the architecture happens to be the same as network ordering then the result of these functions is just the argument. For x86 machines, the host and network ordering *is* different.

Summary: Whenever you read or write the low level C network structures (e.g. port and address information), remember to use the above functions to ensure correct conversion to/from a machine format. Otherwise the displayed or specified value may be incorrect.

How do I connect to a TCP server?

There are three basic system calls you need to connect to a remote machine:

1. `getaddrinfo` determines the remote addresses of a remote host
2. `socket` creates a socket

3. `connect` connects to the remote host using the socket and address information

The `getaddrinfo` call if successful, creates a linked-list of `addrinfo` structs and sets the given pointer to point to the first one.

The socket call creates an outgoing socket and returns a descriptor (sometimes called a 'file descriptor') that can be used with `read` and `write` etc. In this sense it is the network analog of `open` that opens a file stream - except that we haven't connected the socket to anything yet!

Finally the connect call attempts the connection to the remote machine. We pass the original socket descriptor and also the socket address information which is stored inside the addrinfo structure. There are different kinds of socket address structures (e.g. IPv4 vs IPv6) which can require more memory. So in addition to passing the pointer, the size of the structure is also passed:

```
// Pull out the socket address info from the addrinfo struct:
connect(sockfd, p->ai_addr, p->ai_addrlen)
```

To clean up code call `freeaddrinfo` on the top-most `addrinfo` struct:

```
void freeaddrinfo(struct addrinfo *ai);
```

Error handling with `getaddrinfo` is a little different: The return value *is* the error code (i.e. don't use `errno`) * Use `gai_strerror` to get the equivalent short English error text.

```
int result = getaddrinfo(...);
if(result) {
   const char *mesg = gai_strerror(result);
   ...
}
```

Can I request only IPv4 or IPv6 connection? TCP only?

Yes! Use the addrinfo structure that is passed into `getaddrinfo` to define the kind of connection you'd like.

For example, to specify stream-based protocols over IPv6:

```
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));

hints.ai_family = AF_INET6; // Only want IPv6 (use AF_INET for IPv4)
hints.ai_socktype = SOCK_STREAM; // Only want stream-based
    connection
```

What about code examples that use gethostbyname?

The old function `gethostbyname` is deprecated; it's the old way convert a host name into an IP address. The port address still needs to be manually set using htons function. It's much easier to write code to support IPv4 AND

IPv6 using the newer `getaddrinfo`

Is it that easy!?

Yes and no. It's easy to create a simple TCP client - however network communications offers many different levels of abstraction and several attributes and options that can be set at each level of abstraction (for example we haven't talked about `setsockopt` which can manipulate options for the socket). For more information see this guide.

`socket`

```
int socket(int domain, int socket_type, int protocol);
```

Socket creates a socket with domain (e.g. AF_INET for IPv4 or AF_INET6 for IPv6), `socket_type` is whether to use UDP or TCP or other socket type, `protocol` is an optional choice of protocol configuration (for our examples this we can just leave this as 0 for default). This call creates a socket object in the kernel with which one can communicate with the outside world/network. You can use the result of `getaddressinfo` to fill in the `socket` parameters, or provide them manually.

The socket call returns an integer - a file descriptor - and, for TCP clients, you can use it like a regular file descriptor i.e. you can use `read` and `write` to receive or send packets.

TCP sockets are similar to `pipes` except that they allow full duplex communication i.e. you can send and receive data in both directions independently.

getaddressinfo

We saw this in the last section! You're experts at this.

`connect`

```
int connectok = connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Pass `connect` the socket file descriptor, the address you want to connect to and the length in bytes of the address structure. To help identify errors and mistakes it is good practice to check the return value of all networking calls, including `connect`

`read/write`

Once we have a successful connection we can read or write like any old file descriptor. Keep in mind if you are connected to a website, you want to conform to the HTTP protocol specification in order to get any sort of meaningful results back. There are libraries to do this, usually you don't connect at the socket level because there are other libraries or packages around it.

The number of bytes read or written may be smaller than expected. Thus it is important to check the return value of read and write.

Layer 4: TCP Server

The four system calls required to create a TCP server are: `socket`, `bind` `listen` and `accept`. Each has a specific purpose and should be called in the above order

The port information (used by bind) can be set manually (many older IPv4-only C code examples do this), or be created using `getaddrinfo`

We also see examples of setsockopt later too.

`int socket(int domain, int socket_type, int protocol)`

To create a endpoint for networking communication. A new socket by itself is not particularly useful. Though we've specified either a packet or stream-based connections, it is not bound to a particular network interface or port. Instead socket returns a network descriptor that can be used with later calls to bind, listen and accept.

`int bind(int sockfd, const struct sockaddr *addr, socklen`$_t$ *addrlen*);

The `bind` call associates an abstract socket with an actual network interface and port. It is possible to call bind on a TCP client.

`int listen(int sockfd, int backlog);`

The `listen` call specifies the queue size for the number of incoming, unhandled connections i.e. that have not yet been assigned a network descriptor by `accept` Typical values for a high performance server are 128 or more.

`int accept(int sockfd, struct sockaddr *addr,` $\texttt{socklen}_t * addrlen$);

Once the server socket has been initialized the server calls `accept` to wait for new connections. Unlike `socket bind` and `listen`, this call will block. i.e. if there are no new connections, this call will block and only return when a new client connects. The returned TCP socket is associated with a particular tuple `(client IP, client port, server IP, server port)` and will be used for all future incoming and outgoing TCP packets that match this tuple.

Note the `accept` call returns a new file descriptor. This file descriptor is specific to a particular client. It is common programming mistake to use the original server socket descriptor for server I/O and then wonder why networking code has failed.

Why are server sockets passive?

Server sockets do not actively try to connect to another host; instead they wait for incoming connections. Additionally, server sockets are not closed when the peer disconnects. Instead the client communicates with a separate active socket on the server that is specific to that connection.

Unique TCP connections are identified by the tuple `(source ip, source port, destination ip, destination port)` It is possible to have multiple connections from a web browser to the same server port (e.g. port 80) because the the source port on each arriving packet is unique. i.e. For a particular server port (e.g. port 80) there can be one passive server socket but multiple active sockets (one for each currently open connection) and the server's operating system maintains a lookup table that associates a unique tuple with active sockets, so that incoming packets can be correctly routed to the correct socket.

What are the gotchas of creating a TCP-server?

- Using the socket descriptor of the passive server socket (described above)
- Not specifying SOCK_STREAM requirement for getaddrinfo
- Not being able to re-use an existing port.
- Not initializing the unused struct entries
- The `bind` call will fail if the port is currently in use

Note, ports are per machine- not per process or per user. In other words, you cannot use port 1234 while another process is using that port. Worse, ports are by default 'tied up' after a process has finished.

Server code example

A working simple server example is shown below. Note this example is incomplete - for example it does not close either socket descriptor, or free up memory created by `getaddrinfo`

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
```

```
    int s;
    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

    struct addrinfo hints, *result;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    s = getaddrinfo(NULL, "1234", &hints, &result);
    if (s != 0) {
            fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
            exit(1);
    }

    if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
        perror("bind()");
        exit(1);
    }

    if (listen(sock_fd, 10) != 0) {
        perror("listen()");
        exit(1);
    }

    struct sockaddr_in *result_addr = (struct sockaddr_in *)
        result->ai_addr;
    printf("Listening on file descriptor %d, port %d\n", sock_fd,
        ntohs(result_addr->sin_port));

    printf("Waiting for connection...\n");
    int client_fd = accept(sock_fd, NULL, NULL);
    printf("Connection made: client_fd=%d\n", client_fd);

    char buffer[1000];
    int len = read(client_fd, buffer, sizeof(buffer) - 1);
    buffer[len] = '\0';

    printf("Read %d chars\n", len);
    printf("===\n");
    printf("%s\n", buffer);

    return 0;
}
```

Why can't my server re-use the port?

By default a port is not immediately released when the server socket is closed. Instead, the port enters a "TIMED-WAIT" state. This can lead to significant confusion during development because the timeout can make valid networking code appear to fail.

To be able to immediately re-use a port, specify `SO_REUSEPORT` before binding to the port.

```
int optval = 1;
setsockopt(sfd, SOL_SOCKET, SO_REUSEPORT, &optval, sizeof(optval));

bind(....
```

Here's an extended stackoverflow introductory discussion of `SO_REUSEPORT`.

What is the difference shutdown and close?

Use the `shutdown` call when you no longer need to read any more data from the socket, write more data, or have finished doing both. When you shutdown a socket for further writing (or reading) that information is also sent to the other end of the connection. For example if you shutdown the socket for further writing at the server end, then a moment later, a blocked `read` call could return 0 to indicate that no more bytes are expected.

Use `close` when your process no longer needs the socket file descriptor.

If you `fork`-ed after creating a socket file descriptor, all processes need to close the socket before the socket resources can be re-used. If you shutdown a socket for further read then all process are be affected because you've changed the socket, not just the file descriptor.

Well written code will `shutdown` a socket before calling `close` it.

Who connected to my server?

The `accept` system call can optionally provide information about the remote client, by passing in a sockaddr struct. Different protocols have differently variants of the `struct sockaddr`, which are different sizes. The simplest struct to use is the `sockaddr_storage` which is sufficiently large to represent all possible types of sockaddr. Notice that C does not have any model of inheritance. Therefore we need to explicitly cast our struct to the 'base type' struct sockaddr.

```
struct sockaddr_storage clientaddr;
socklen_t clientaddrsize = sizeof(clientaddr);
int client_id = accept(passive_socket,
       (struct sockaddr *) &clientaddr,
        &clientaddrsize);
```

We've already seen `getaddrinfo` that can build a linked list of addrinfo entries (and each one of these can include socket configuration data). What if we wanted to turn socket data into IP and port addresses? Enter `getnameinfo` that can be used to convert a local or remote socket information into a domain name or numeric IP. Similarly the port number can be represented as a service name (e.g. "http" for port 80). In the example below we request numeric versions for the client IP address and client port number.

```
socklen_t clientaddrsize = sizeof(clientaddr);
int client_id = accept(sock_id, (struct sockaddr *) &clientaddr,
    &clientaddrsize);
char host[256], port[256];
getnameinfo((struct sockaddr *) &clientaddr,
      clientaddrsize, host, sizeof(host), port, sizeof(port),
      NI_NUMERICHOST | NI_NUMERICSERV);
```

TODO: Discuss NI_MAXHOST and NI_MAXSERV, and NI_NUMERICHOST

Layer 4: UDP

What is UDP? When is it used?

UDP is a connectionless protocol that is built on top of IPv4 and IPv6. It's very simple to use: Decide the destination address and port and send your data packet! However the network makes no guarantee about whether the packets will arrive. Packets (aka Datagrams) may be dropped if the network is congested. Packets may be duplicated or arrive out of order.

Between two distant data-centers it's typical to see 3% packet loss. A typical use case for UDP is when receiving up to date data is more important than receiving all of the data. For example, a game may send continuous updates of player positions. A streaming video signal may send picture updates using UDP

UDP Server

There are a variety of function calls available to send UDP sockets. We will use the newer getaddrinfo to help set up a socket structure. Remember that UDP is a simple packet-based ('data-gram') protocol ; there is no connection to set up between the two hosts.

First, initialize the hints addrinfo struct to request an IPv6, passive datagram socket.

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET6; // use AF_INET instead for IPv4
hints.ai_socktype = SOCK_DGRAM;
hints.ai_flags = AI_PASSIVE;
```

Next, use getaddrinfo to specify the port number (we don't need to specify a host as we are creating a server socket, not sending a packet to a remote host).

```
getaddrinfo(NULL, "300", &hints, &res);

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
bind(sockfd, res->ai_addr, res->ai_addrlen);
```

The port number is less than 1024, so the program will need `root` privileges. We could have also specified a service name instead of a numeric port value.

So far the calls have been similar to a TCP server. For a stream-based service we would call `listen` and accept. For our UDP-serve we can just start waiting for the arrival of a packet on the socket-

```
struct sockaddr_storage addr;
int addrlen = sizeof(addr);

// ssize_t recvfrom(int socket, void* buffer, size_t buflen, int
    flags, struct sockaddr *addr, socklen_t * address_len);

byte_count = recvfrom(sockfd, buf, sizeof(buf), 0, &addr, &addrlen);
```

The addr struct will hold sender (source) information about the arriving packet. Note the `sockaddr_storage` type is a sufficiently large enough to hold all possible types of socket addresses (e.g. IPv4, IPv6 and other socket types).

Full Code

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <arpa/inet.h>

int main(int argc, char **argv)
{
    int s;

    struct addrinfo hints, *res;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET6; // INET for IPv4
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE;

    getaddrinfo(NULL, "300", &hints, &res);

    int sockfd = socket(res->ai_family, res->ai_socktype,
        res->ai_protocol);

    if (bind(sockfd, res->ai_addr, res->ai_addrlen) != 0) {
        perror("bind()");
        exit(1);
    }
    struct sockaddr_storage addr;
    int addrlen = sizeof(addr);

    while(1){
        char buf[1024];
        ssize_t byte_count = recvfrom(sockfd, buf, sizeof(buf), 0,
            &addr, &addrlen);
        buf[byte_count] = '\0';

        printf("Read %d chars\n", byte_count);
        printf("===\n");
        printf("%s\n", buf);
    }

    return 0;
}
```

Layer 7: HTTP

Layer 7 of the OSI layer deals with application level interfaces. Meaning that you can ignore everything below this layer and treat an internet as a way of communicating with another computer than can be secure and the session may reconnect. Common layer 7 protocols are the following

1. HTTP(S) - Hyper Text Transfer Protocol. Sends arbitrary data and executes remote actions on a web server.
2. FTP - File Transfer Protocol. Transfers a file from one computer to another
3. TFTP - Trivial File Transfer Protocol. Same as above but using UDP.
4. DNS - Domain Name Service. Translates hostnames to IP addresses
5. SMTP - Simple Mail Transfer Protocol. Allows one to send plain text emails to an email server
6. SSH - Secure SHell. Allows one computer to connect to another computer and execute commands remotely.
7. Bitcoin - Decentralized crypto currency
8. BitTorrent - Peer to peer file sharing protocol
9. NTP - Network Time Protocol. This protocol helps keep your computer's clock synced with the outside world

Complete Simple TCP Client Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int s;
    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

    struct addrinfo hints, *result;
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_INET; /* IPv4 only */
    hints.ai_socktype = SOCK_STREAM; /* TCP */

    s = getaddrinfo("www.illinois.edu", "80", &hints, &result);
    if (s != 0) {
           fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
           exit(1);
    }

    if(connect(sock_fd, result->ai_addr, result->ai_addrlen) == -1){
              perror("connect");
              exit(2);
       }

    char *buffer = "GET / HTTP/1.0\r\n\r\n";
```

```
    printf("SENDING: %s", buffer);
    printf("===\n");

        // For this trivial demo just assume write() sends all bytes
            in one go and is not interrupted

    write(sock_fd, buffer, strlen(buffer));


    char resp[1000];
    int len = read(sock_fd, resp, 999);
    resp[len] = '\0';
    printf("%s\n", resp);

    return 0;
}
```

Example output:

```
SENDING: GET / HTTP/1.0

===
HTTP/1.1 200 OK
Date: Mon, 27 Oct 2014 19:19:05 GMT
Server: Apache/2.2.15 (Red Hat) mod_ssl/2.2.15 OpenSSL/1.0.1e-fips
    mod_jk/1.2.32
Last-Modified: Fri, 03 Feb 2012 16:51:10 GMT
ETag: "401b0-49-4b8121ea69b80"
Accept-Ranges: bytes
Content-Length: 73
Connection: close
Content-Type: text/html

Provided by Web Services at Public Affairs at the University of
    Illinois
```

Comment on HTTP request and response

The example above demonstrates a request to the server using Hypertext Transfer Protocol. A web page (or other resources) are requested using the following request:

```
GET / HTTP/1.0
```

There are four parts (the method e.g. GET,POST,...); the resource (e.g. / /index.html /image.png); the proctocol "HTTP/1.0" and two new lines (˚˚)

The server's first response line describes the HTTP version used and whether the request is successful using a 3 digit response code:

```
HTTP/1.1 200 OK
```

If the client had requested a non existing file, e.g. `GET /nosuchfile.html HTTP/1.0` Then the first line includes the response code is the well-known 404 response code:

```
HTTP/1.1 404 Not Found
```

How is a website converted into an IP address?

A system called "DNS" (Domain Name Service) is used. If a machine does not hold the answer locally then it sends a UDP packet to a local DNS server. This server in turn may query other upstream DNS servers.

DNS by itself is fast but not secure. DNS requests are not encrypted and susceptible to 'man-in-the-middle' attacks. For example, a coffee shop internet connection could easily subvert your DNS requests and send back different IP addresses for a particular domain. The way this is usually subverted is that after the IP address is obtained then a connection is usually made over HTTPS. HTTPS uses what is called the TLS (formerly known as SSL) to secure transmissions and verify the IP address is who they say they are.

Nonblocking IO

Normally, when you call `read()`, if the data is not available yet it will wait until the data is ready before the function returns. When you're reading data from a disk, that delay may not be long, but when you're reading from a slow network connection it may take a long time for that data to arrive, if it ever arrives.

POSIX lets you set a flag on a file descriptor such that any call to `read()` on that file descriptor will return immediately, whether it has finished or not. With your file descriptor in this mode, your call to `read()` will start the read operation, and while it's working you can do other useful work. This is called "nonblocking" mode, since the call to `read()` doesn't block.

To set a file descriptor to be nonblocking:

```
// fd is my file descriptor
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

For a socket, you can create it in nonblocking mode by adding `SOCK_NONBLOCK` to the second argument to `socket()`:

```
fd = socket(AF_INET, SOCK_STREAM | SOCK_NONBLOCK, 0);
```

When a file is in nonblocking mode and you call `read()`, it will return immediately with whatever bytes are available. Say 100 bytes have arrived from the server at the other end of your socket and you call `read(fd, buf, 150)`. Read will return immediately with a value of 100, meaning it read 100 of the 150 bytes you asked for. Say you tried to read the remaining data with a call to `read(fd, buf+100, 50)`, but the last 50 bytes still hadn't arrived yet. `read()` would return -1 and set the global error variable **errno** to either EAGAIN or EWOULDBLOCK. That's the system's way of telling you the data isn't ready yet.

`write()` also works in nonblocking mode. Say you want to send 40,000 bytes to a remote server using a

socket. The system can only send so many bytes at a time. Common systems can send about 23,000 bytes at a time. In nonblocking mode, `write(fd, buf, 40000)` would return the number of bytes it was able to send immediately, or about 23,000. If you called `write()` right away again, it would return -1 and set errno to EAGAIN or EWOULDBLOCK. That's the system's way of telling you it's still busy sending the last chunk of data, and isn't ready to send more yet.

How do I check when the I/O has finished?

There are a few ways. Let's see how to do it using *select* and *epoll*.

```
int select(int nfds,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);
```

Given three sets of file descriptors, `select()` will wait for any of those file descriptors to become 'ready'.

1. `readfds` - a file descriptor in `readfds` is ready when there is data that can be read or EOF has been reached.
2. `writefds` - a file descriptor in `writefds` is ready when a call to write() will succeed.
3. `exceptfds` - system-specific, not well-defined. Just pass NULL for this.

`select()` returns the total number of file descriptors that are ready. If none of them become ready during the time defined by *timeout*, it will return 0. After `select()` returns, the caller will need to loop through the file descriptors in readfds and/or writefds to see which ones are ready. As readfds and writefds act as both input and output parameters, when `select()` indicates that there are file descriptors which are ready, it would have overwritten them to reflect only the file descriptors which are ready. Unless it is the caller's intention to call `select()` only once, it would be a good idea to save a copy of readfds and writefds before calling it.

```
fd_set readfds, writefds;
FD_ZERO(&readfds);
FD_ZERO(&writefds);
for (int i=0; i < read_fd_count; i++)
  FD_SET(my_read_fds[i], &readfds);
for (int i=0; i < write_fd_count; i++)
  FD_SET(my_write_fds[i], &writefds);

struct timeval timeout;
timeout.tv_sec = 3;
timeout.tv_usec = 0;

int num_ready = select(FD_SETSIZE, &readfds, &writefds, NULL,
    &timeout);

if (num_ready < 0) {
  perror("error in select()");
} else if (num_ready == 0) {
  printf("timeout\n");
} else {
  for (int i=0; i < read_fd_count; i++)
    if (FD_ISSET(my_read_fds[i], &readfds))
```

```
      printf("fd %d is ready for reading\n", my_read_fds[i]);
  for (int i=0; i < write_fd_count; i++)
    if (FD_ISSET(my_write_fds[i], &writefds))
      printf("fd %d is ready for writing\n", my_write_fds[i]);
}
```

For more information on select()

epoll

epoll is not part of POSIX, but it is supported by Linux. It is a more efficient way to wait for many file descriptors. It will tell you exactly which descriptors are ready. It even gives you a way to store a small amount of data with each descriptor, like an array index or a pointer, making it easier to access your data associated with that descriptor.

To use epoll, first you must create a special file descriptor with epoll_create(). You won't read or write to this file descriptor; you'll just pass it to the other epoll_xxx functions and call close() on it at the end.

```
    epfd = epoll_create(1);
```

For each file descriptor you want to monitor with epoll, you'll need to add it to the epoll data structures using epoll_ctl() with the `EPOLL_CTL_ADD` option. You can add any number of file descriptors to it.

```
struct epoll_event event;
event.events = EPOLLOUT; // EPOLLIN==read, EPOLLOUT==write
event.data.ptr = mypointer;
epoll_ctl(epfd, EPOLL_CTL_ADD, mypointer->fd, &event)
```

To wait for some of the file descriptors to become ready, use epoll_wait(). The epoll_event struct that it fills out will contain the data you provided in event.data when you added this file descriptor. This makes it easy for you to look up your own data associated with this file descriptor.

```
int num_ready = epoll_wait(epfd, &event, 1, timeout_milliseconds);
if (num_ready > 0) {
  MyData *mypointer = (MyData*) event.data.ptr;
  printf("ready to write on %d\n", mypointer->fd);
}
```

Say you were waiting to write data to a file descriptor, but now you want to wait to read data from it. Just use `epoll_ctl()` with the `EPOLL_CTL_MOD` option to change the type of operation you're monitoring.

```
event.events = EPOLLOUT;
event.data.ptr = mypointer;
epoll_ctl(epfd, EPOLL_CTL_MOD, mypointer->fd, &event);
```

To unsubscribe one file descriptor from epoll while leaving others active, use `epoll_ctl()` with the `EPOLL_CTL_DEL` option.

```
epoll_ctl(epfd, EPOLL_CTL_DEL, mypointer->fd, NULL);
```

To shut down an epoll instance, close its file descriptor.

```
close(epfd);
```

In addition to nonblocking `read()` and `write()`, any calls to `connect()` on a nonblocking socket will also be nonblocking. To wait for the connection to complete, use `select()` or epoll to wait for the socket to be writable. There are definitely reasons to use epoll over select but due to to interface, there are fundamental problems with doing so.

Blogpost about select being broken

Remote Procedure Calls

Remote Procedure Call. RPC is the idea that we can execute a procedure (function) on a different machine. In practice the procedure may execute on the same machine, however it may be in a different context - for example under a different user with different permissions and different lifecycle.

What is Privilege Separation?

The remote code will execute under a different user and with different privileges from the caller. In practice the remote call may execute with more or fewer privileges than the caller. This in principle can be used to improve the security of a system (by ensuring components operate with least privilege). Unfortunately, security concerns need to be carefully assessed to ensure that RPC mechanisms cannot be subverted to perform unwanted actions. For example, an RPC implementation may implicitly trust any connected client to perform any action, rather than a subset of actions on a subset of the data.

What is stub code? What is marshalling?

The stub code is the necessary code to hide the complexity of performing a remote procedure call. One of the roles of the stub code is to *marshall* the necessary data into a format that can be sent as a byte stream to a remote server.

```
// On the outside 'getHiscore' looks like a normal function call
// On the inside the stub code performs all of the work to send and
   receive the data to and from the remote machine.

int getHighScore(char* game) {
  // Marshall the request into a sequence of bytes:
  char* buffer;
  asprintf(&buffer,"getHiscore(%s)!", name);

  // Send down the wire (we do not send the zero byte; the '!'
     signifies the end of the message)
  write(fd, buffer, strlen(buffer) );

  // Wait for the server to send a response
  ssize_t bytesread = read(fd, buffer, sizeof(buffer));
```

```
  // Example: unmarshal the bytes received back from text into an
     int
  buffer[bytesread] = 0; // Turn the result into a C string

  int score= atoi(buffer);
  free(buffer);
  return score;
}
```

What is server stub code? What is unmarshalling?

The server stub code will receive the request, unmarshall the request into a valid in-memory data call the underlying implementation and send the result back to the caller.

How do you send an int? float? a struct? A linked list? A graph?

To implement RPC you need to decide (and document) which conventions you will use to serialize the data into a byte sequence. Even a simple integer has several common choices:

1. Signed or unsigned?
2. ASCII, Unicode Text Format 8, some other encoding?
3. Fixed number of bytes or variable depending on magnitude
4. Little or Big endian binary format?

To marshall a struct, decide which fields need to be serialized. It may not be necessary to send all data items (for example, some items may be irrelevant to the specific RPC or can be re-computed by the server from the other data items present).

To marshall a linked list it is unnecessary to send the link pointers- just stream the values. As part of unmarshalling the server can recreate a linked list structure from the byte sequence.

By starting at the head node/vertex, a simple tree can be recursively visited to create a serialized version of the data. A cyclic graph will usually require additional memory to ensure that each edge and vertex is processed exactly once.

What is an Interface Description Language (IDL)?

Writing stub code by hand is painful, tedious, error prone, difficult to maintain and difficult to reverse engineer the wire protocol from the implemented code. A better approach is specify the data objects, messages and services and automatically generate the client and server code.

A modern example of an Interface Description Language is Google's Protocol Buffer .proto files.

Complexity and challenges of RPC vs local calls?

Remote Procedure Calls are significantly slower (10x to 100x) and more complex than local calls. An RPC must marshall data into a wire-compatible format. This may require multiple passes through the data structure, temporary memory allocation and transformation of the data representation.

Robust RPC stub code must intelligently handle network failures and versioning. For example, a server may have to process requests from clients that are still running an early version of the stub code.

A secure RPC will need to implement additional security checks (including authentication and authorization), validate data and encrypt communication between the client and host.

Transferring large amounts of structured data

Let's examine three methods of transferring data using 3 different formats - JSON, XML and Google Protocol Buffers. JSON and XML are text-based protocols. Examples of JSON and XML messages are below.

```
<ticket><price
    currency='dollar'>10</price><vendor>travelocity</vendor></ticket>
```

```
{ 'currency':'dollar' , 'vendor':'travelocity', 'price':'10' }
```

Google Protocol Buffers is an open-source efficient binary protocol that places a strong emphasis on high throughput with low CPU overhead and minimal memory copying. Implementations exist for multiple languages including Go, Python, C++ and C. This means client and server stub code in multiple languages can be generated from the .proto specification file to marshall data to and from a binary stream.

Google Protocol Buffers reduces the versioning problem by ignoring unknown fields that are present in a message. See the introduction to Protocol Buffers for more information.

Topics

- IPv4 vs IPv6
- TCP vs UDP
- Packet Loss/Connection Based
- Get address info
- DNS
- TCP client calls
- TCP server calls
- shutdown
- recvfrom
- epoll vs select
- RPC

Questions

- What is IPv4? IPv6? What are the differences between them?
- What is TCP? UDP? Give me advantages and disadvantages of both of them. When would I use one and not the other?
- Which protocol is connection less and which one is connection based?
- What is DNS? What is the route that DNS takes?
- What does socket do?

- What are the calls to set up a TCP client?
- What are the calls to set up a TCP server?
- What is the difference between a socket shutdown and closing?
- When can you use `read` and `write`? How about `recvfrom` and `sendto`?
- What are some advantages to `epoll` over `select`? How about `select` over `epoll`?
- What is a remote procedure call? When should I use it?
- What is marshalling/unmarshalling? Why is HTTP *not* an RPC?

Bibliography

11

Filesystems

TODO

What is a filesystem?

You may have encountered the old unix adage, "everything is a file". In most UNIX systems, files operations provide an interface to abstract many different operations. Network sockets, hardware devices and even data on disk are all represented by a file-like object. A file-like object must follow certain conventions:

1. It must present it self to the filesystem
2. It must support common filesystem operations, such as `open`, `read`, `write`

A filesystem is an implementation of the file interface. In this chapter, we will be exploring the various callbacks a filesystem provides and some typical functionality or implementation details associated with this. In this class, we will mostly talk about filesystems that serve to allow users to access data on disk. These filesystems are integral to modern computers. Filesystems not only deal with storing local files, they handle special devices that allow for safe communication between the kernel and user space. Filesystems also deal with failures, scalability, indexing, encryption, compression and performance. Filesystems handle the abstraction between a file which contains data and how exactly that data is stored on disk, partitioned, and protected.

Although filesystems are usually thought of as a kind of tree, most filesystems are usually a directed graph, a model we will explore in depth later in this chapter. Before we dive into the details of a filesystem, let's take a look at some examples.

1. `ext4` Usually mounted at /, this is the filesystem that usually provides disk access as you're used to.
2. `procfs` Usually mounted at /proc, provides information and control over processes.
3. `sysfs` Usually mounted at /sys, a more mordern version of /proc that also allows control over various other hardware such as network sockets.
4. `tmpfs` Mounted at /tmp in some systems, an in-memory filesystem to hold temporary files.
5. `sshfs` A filesystem that syncs files across the ssh protocol.

To clarify, a mount point is simply a mapping of a directory to a filesystem represented in the kernel. What this means is that to resolve which filesystem a structure a call must resolve to. Meaning that `/root` is resolved by the `ext4` filesystem in our case, but `/proc/2` is resolved by the `procfs` system even though it contains `/` as a subsystem.

As you may have noticed, some of these filesystems provide an interface to things that aren't a "file" as you might colloquially refer to them. Filesystems such as `procfs` are usually refered to as *virtual* filesystems, since they don't provide data access in the same sense as a traditional filesystem would. Technically, all filesystems in the kernel are represented by virtual filesystem, but in our class we will differentiate *virtual* filesystems as filesystems that actually don't store anything on a hard disk.

The File API

A filesystem must provide callback functions to a variety of actions. Some of them as listed below:

- `open` Opens a file for IO
- `read` Read contents of a file
- `write` Write to a file
- `close` Close a file and free associated resources
- `chmod` Modify permissions of a file
- `ioctl` Interact with device parameters of character devices such as terminals

Not every filesystem supports all the possible callback functions. For example many filesystems do not implement `ioctl` or `link`. In this chapter, we will not be examining each filesystem callback. If you would like to learn more about this interface, try looking at the documentation for FUSE, the source code for glibc or the linux manpages.

Storing data on disk

In order to understand how a filesystem interacts with data on disk, there are three key terms we will be using.

1. `disk block` A disk block is a portion of the disk that is reserved for storing the contents of a file or a directory.
2. `inode` An inode *is* a file or directory. This means that an inode contains metadata about the file as well as pointers to disk blocks so that the file can actually be written to or read from.
3. `superblock` A superblock contains metadata about the inodes and disk blocks. An example superblock can store how full each disk block is, which inodes are being used etc. Modern filesystems may actually contain multiple superblocks and a sort-of super-super block that keeps track of which sectors are governed by which superblocks. This tends to help with fragmentation.

These structures all are presented in the diagram below.

Figure 11.1:

It may seem overwhelming, but by the end of this chapter, we will be able to make sense of every part of the filesystem.

In order to reason about data on some form of storage (spinning disks, solid state drives, magnetic tape, etc.), it is common practice to first consider the medium of storage as a collection of *blocks*. A block can be thought of as a continguous region on disk, and while it's size is sometimes determined by some property of the underlying hardware, it is more frequently determined based off of the size of a page of memory for a given system, so that data from the disk can be cached in memory for faster access - a very important feature of many filesystems.

Usually, a filesystem has a special block denoted as a *superblock* which stores metadata about the filesystem such as a journal (which logs changes to the filesystem), a table of inodes and where the first inode is stored on disk, etc. The important thing about a superblock is that it is in a known location on disk.

The inode is the most important structure for our filesystem as it represents a file. Before we explore it in depth, let's list out the key information we need to have a usable file:

- Name

- File size
- Time created, last modified, last accessed
- Permissions
- Filepath
- Checksum
- File data

File Contents

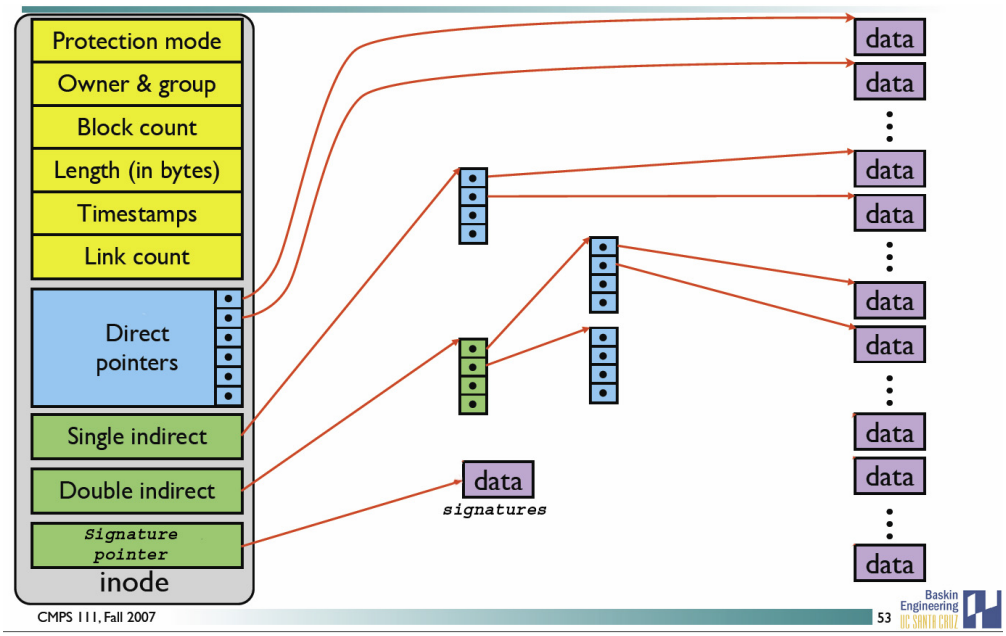


Figure 11.2:

From Wikipedia:

> *In a Unix-style file system, an index node, informally referred to as an inode, is a data structure used to represent a filesystem object, which can be one of various things including a file or a directory. Each inode stores the attributes and disk block location(s) of the filesystem object's data. Filesystem object attributes may include manipulation metadata (e.g. change, access, modify time), as well as owner and permission data (e.g. group-id, user-id, permissions).*

Typically, the superblock stores an array of inodes, each of which stores direct, and potentially several kinds of indirect pointers to disk blocks. Since inodes are stored in the superblock, most filesystems have a limit on how many inodes can exist. Since each inode corresponds to a file, this is also a limit on how many files that filesystem can have. Trying to overcome this problem by storing inodes in some other location greatly increases the complexity of the filesystem. Trying to reallocate space for the inode table is also infeasible since every byte following the end of the inode array would have to be shifted, a highly expensive operation. This isn't to say there aren't any solutions at all, although typically there is no need to increase the number of inodes since the number of inodes is usually sufficiently high.

Big idea: Forget names of files: The 'inode' is the file.

It is common to think of the file name as the 'actual' file. It's not! Instead consider the inode as the file. The inode holds the meta-information (last accessed, ownership, size) and points to the disk blocks used to hold the file

contents. However, the inode does not usually store a filename. Filenames are usually only stored in directories. (see below)

For example, to read the first few bytes of the file, follow the first direct block pointer to the first direct block and read the first few bytes. Writing follows the same process. If you want to read the entire file, keep reading direct blocks until you've read a number of bytes equal to the size of the file. If the total size of the file is less than that of the number of direct blocks multiplied by the size of a block, then unsued block pointers will be undefined. Similarly, if the size of a file is not a multiple of the size of a block, data past the end of the last byte in the last block will be garbage.

What if a file is bigger than the maximum space addressable by it's direct blocks?

> "All problems in computer science can be solved by another level of indirection." - David Wheeler

Except the problem of too many layers of indirection.

To solve this problem, we introduce the indirect blocks. An indirect block is a block that store pointers to more data blocks. Similarly a double indirect block stores pointers to indirect blocks and the concept can be generalized to arbitrary levels of indirection. This is a very important concept, since as inodes are stored in the superblock, or some other strucutre in a well known location with a constant amount of space, indirection allows exponential increases in the amount of space an inode can keep track of.

As a worked example, suppose we divide the disk into 4KB blocks and we want to address up to 2^32 blocks. The maximum disk size is 4KB *2^32 = 16TB (remember 2^10 = 1024). A disk block can store 4KB / 4B (each pointer needs to be 32 bits) = 1024 pointers. Each pointer refers to a 4KB disk block - so you can refer up to 1024*4KB = 4MB of data. For the same disk configuration, a double indirect block stores 1024 pointers to 1024 indirection tables. Thus a double-indirect block can refer up to 1024 * 4MB = 4GB of data. Similarly, a triple indirect block can refer up to 4TB of data. Naturally though, this is three times as slow.

So... How do we implement a directory?

A directory is just a mapping of names to inode numbers. It's typically just a normal file, but with some special bits set in its inode and a very specific structure for its contents. POSIX provides a small set of functions to read the filename and inode number for each entry, which we will talk about in depth later in this chapter.

Let's think about what it looks like in the actual file system. Theoretically, directories are just like actual files. The disk blocks will contain *directory entries* or *dirents*. What that means is that our disk block can look like this

```
| inode_num | name | | ------------ | ------ |
| 2043567  | hi.txt | | ... |
```

Each directory entry could either be a fixed size, or a variable c-string. It depends on how the particular filesystem implements it at the lower level. To see a mapping of filenames to inode numbers on a POSIX system, from a shell, use `ls` with the `-i` option

```
# ls -i
12983989 dirlist.c    12984068 sandwich.c
```

Unix Directory Conventions

In standard unix file systems the following entries are specially added on requests to read a directory.

1. `.` represents the current directory
2. `..` represents the parent directory
3. `~` is the name of the home directory usually

Note that `...` is NOT a valid representation of any directory (this not the grandparent directory). It *could* however be the name of a file on disk. Though confusingly, `zsh` provides this as a handy shortcut to the grandparent directory should it exist.

Additional facts about name-related conventions:

1. Files that start with '.' on disk are conventionally considered 'hidden' and will not be listed by programs like `ls` without additional flags (`-a`). This is not a feature of the filesystem and programs may choose to ignore this.

2. Some files may also start with a null byte. These are usually *abstract unix sockets* and are used to prevent cluttering up the filesystem since they will be effectively hidden by any program not expecting them. They will, however, be listed by tools that detail information about sockets, so this is not a feature providing security.

How do I list the contents of a directory?

While interacting with a file in C is done by using `open` to open the file and then `read` or `write` to interact with the file before calling `close` to release resources, directories have special calls such as, `opendir`, `closedir` and `readdir`. There is no function `writedir` since typically that implies creating a file or link.

To explore these functions, let's write a program to search the contents of a directory for a particular file. (The code below has a bug, try to spot it!)

```
int exists(char *directory, char *name) {
    struct dirent *dp;
    DIR *dirp = opendir(directory);
    while ((dp = readdir(dirp)) != NULL) {
        puts(dp->d_name);
        if (!strcmp(dp->d_name, name)) {
        return 1; /* Found */
        }
    }
    closedir(dirp);
    return 0; /* Not Found */
}
```

The above code has a subtle bug: It leaks resources! If a matching filename is found then 'closedir' is never called as part of the early return. Any file descriptors opened, and any memory allocated, by opendir are never released. This means eventually the process will run out of resources and an `open` or `opendir` call will fail.

The fix is to ensure we free up resources in every possible code-path. In the above code this means calling `closedir` before `return 1`. Forgetting to release resources is a common C programming bug because there is no support in the C lanaguage to ensure resources are always released with all codepaths.

Note: after a call to fork(), either (XOR) the parent or the child can use readdir(), rewinddir() or seekdir(). If both the parent and the child use the above, behavior is undefined.

There are two main gotchas and one consideration: The `readdir` function returns "." (current directory) and ".." (parent directory). If you are looking for sub-directories, you need to explicitly exclude these directories.

For many applications it's reasonable to check the current directory first before recursively searching sub-directories. This can be achieved by storing the results in a linked list, or resetting the directory struct to restart from the beginning.

The following code attempts to list all files in a directory recursively. As an exercise, try to identify the bugs it introduces.

```
void dirlist(char *path) {
```

```
  struct dirent *dp;
  DIR *dirp = opendir(path);
  while ((dp = readdir(dirp)) != NULL) {
     char newpath[strlen(path) + strlen(dp->d_name) + 1];
     sprintf(newpath,"%s/%s", newpath, dp->d_name);
     printf("%s\n", dp->d_name);
     dirlist(newpath);
  }
}

int main(int argc, char **argv) {dirlist(argv[1]); return 0; }
```

Did you find all 5 bugs?

```
// Check opendir result (perhaps user gave us a path that can not
    be opened as a directory
if (!dirp) {perror("Could not open directory"); return; }
// +2 as we need space for the / and the terminating 0
char newpath[strlen(path) + strlen(dp->d_name) + 2];
// Correct parameter
sprintf(newpath,"%s/%s", path, dp->d_name);
// Perform stat test (and verify) before recursing
if (0 == stat(newpath,&s) && S_ISDIR(s.st_mode)) dirlist(newpath)
// Resource leak: the directory file handle is not closed after the
    while loop
closedir(dirp);
```

One final note of caution: `readdir` is not thread-safe! For multi-threaded searches use `readdir_r` which requires the caller to pass in the address of an existing dirent struct.

See the man page of readdir for more details.

Linking

Links are what force us to model a filesystem as a tree rather than a graph. While modelling the filesystem as a tree would imply that every inode has a unique parent directory, links allow inodes to present themselves as files in multiple places, potentially with different names, thus leading to an inode having multiple parents directories.

The first kind of link is a hard link. A hard link is simply an entry in a directory assigning some name to an inode number that already has a different name and mapping in either the same directory or a different one. If we already have a file on a file system we can create another link to the same inode using the 'ln' command:

```
$ ln file1.txt blip.txt
```

However blip.txt *is* the same file; if I edit blip I'm editing the same file as 'file1.txt!' We can prove this by showing that both file names refer to the same inode:

```
$ ls -i file1.txt blip.txt
```

```
134235 file1.txt
134235 blip.txt
```

The equivalent C call is `link`

```c
// Function Prototype
int link(const char *path1, const char *path2);

link("file1.txt", "blip.txt");
```

For simplicity the above examples made hard links inside the same directory. Hard links can be created anywhere inside the same filesystem.

The second kind of link is a soft link - or a symbolic link (or a symlink if you don't have the time). A symbolic link is different because it does not deal with inode numbers directly. Instead a symbolic link is a regular file with a special bit set and stores a path to another file. Quite simply, without the special bit, it is nothing more than a text file with a file path inside. Note that when people generally talk about a link without specifying hard or soft, they are referring to a hard link.

To create a symbolic link in the shell use `ln -s`. To read the contents of the link as just a file use `readlink`. These are both demonstrated below:

```
$ ln -s file1.txt file2.txt
$ ls -i file1.txt blip.txt
134235 file1.txt
134236 file2.txt
134235 blip.txt
$ cat file1.txt
file1!
$ cat file2.txt
file1!
$ cat blip.txt
file1!
$ echo edited file2 >> file2.txt # >> is bash syntax for append to
    file
$ cat file1.txt
file1!
edited file2
$ cat file2.txt
I'm file1!
edited file2
$ cat blip.txt
file1!
edited file2
$ readlink myfile.txt
file2.txt
```

Note that `file2.txt` and `file1.txt` have different inode numbers, unlike the hard link, `blip.txt`.

There is a C library call to create symlinks which is similar to link:

```
symlink(const char *target, const char *symlink);
```

Advantages of symbolic links

- Can refer to files that don't exist yet
- Unlike hard links, can refer to directories as well as regular files
- Can refer to files (and directories) that exist outside of the current file system

However, symlinks have a key disadvantage, they as slower than regular files and directories. When the links contents are read, they must be interpreted as a new path to the target file, resulting in an additional call to open and read since the real file must be opened and read.

What happens when I `rm` (remove) a file?

What happens when I rm (remove) a file?

When you remove a file (using `rm` or `unlink`) you are removing an inode reference from a directory. However the inode may still be referenced from other directories. In order to determine if the contents of the file are still required, each inode keeps a reference count that is updated whenever a new link is created or destroyed. This count only tracks hard links, symlinks are allowed to refer to a non-existant file and thus, do not matter.

Case study: Back up software that minimizes file duplication

An example use of hard-links is to efficiently create multiple archives of a file system at different points in time. Once the archive area has a copy of a particular file, then future archives can re-use these archive files rather than creating a duplicate file. Apple's "Time Machine" software does this.

Can I create hard links to directories as well as regular files?

While this is dependant on the underlying filesystem, the POSIX standard says no you may not! The `ln` command will only allow root to do this and only if you provide the `-d` option. However even root may not be able to perform this because most filesystems prevent it!

The integrity of the file system assumes the directory structure (excluding softlinks which we will talk about later) is a non-cyclic tree that is reachable from the root directory. It becomes expensive to enforce or verify this constraint if directory linking is allowed. Breaking these assumptions can cause file integrity tools to not be able to repair the file system. Recursive searches potentially never terminate and directories can have more than one parent but ".." can only refer to a single parent. All in all, a bad idea.

Pathing

Now that we have definitions talk about directories, we come across the concept of a path. A path is a sequence of directories that provide one with a "path" in the filesystem graph. However, there are some nuances. It is possible to have a path called `a/b/../c/./`. Since `..` and `.` are special entries in directories, this is a valid path that actually refers to `a/c`. Most filesystem functions will allow uncompressed paths to be passed in. The C library provides a function `realpath` to compress the path or get the realpath. To simplify by hand remember that `..` means 'parent folder' and that `.` means 'current folder'. Below is an example that illustrates the simplification of the `a/b/../c/.` by using `cd` in a shell to navigate a filesystem.

1. `cd a` (in a)
2. `cd b` (in a/b)
3. `cd ..` (in a, because .. represents 'parent folder')
4. `cd c` (in a/c)
5. `cd .` (in a/c, because . represents 'current folder')

Thus, this path can be simplified to `a/c`.

How do I find out meta-information about a file (or directory)?

How can we distinguish between a regular file and a directory? For that matter there's many other attributes that files also might contain. You may know that on most UNIX systems, unlike windows systems, a file's type is not determined by its extension. How does the system know what type the file is?

All of this information is stored within an inode. To access it, use the stat calls. For example, to find out when my 'notes.txt' file was last accessed -

```
struct stat s;
stat("notes.txt", &s);
printf("Last accessed %s", ctime(&s.st_atime));
```

There are actually three versions of `stat`;

```
int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

For example you can use `fstat` to find out the meta-information about a file if you already have an file descriptor associated with that file

```
FILE *file = fopen("notes.txt", "r");
int fd = fileno(file); /* Just for fun - extract the file
    descriptor from a C FILE struct */
struct stat s;
fstat(fd, & s);
printf("Last accessed %s", ctime(&s.st_atime));
```

`lstat` is almost the same as `stat` but handles symbolic links differently. From the `stat` man page:

> lstat() is identical to stat(), except that if pathname is a symbolic link, then it returns information about the link itself, not the file that it refers to.

The stat functions make use of `struct stat`. From the `stat` man page:

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* Inode number */
    mode_t    st_mode;    /* File type and mode */
    nlink_t   st_nlink;   /* Number of hard links */
    uid_t     st_uid;     /* User ID of owner */
    gid_t     st_gid;     /* Group ID of owner */
    dev_t     st_rdev;    /* Device ID (if special file) */
    off_t     st_size;    /* Total size, in bytes */
    blksize_t st_blksize; /* Block size for filesystem I/O */
    blkcnt_t st_blocks;   /* Number of 512B blocks allocated */
    struct timespec st_atim; /* Time of last access */
```

```
    struct timespec st_mtim; /* Time of last modification */
    struct timespec st_ctim; /* Time of last status change */
};
```

The `st_mode` field can be used to distinguish between regular files and directories. To accomplish this, you will also need the macros, `S_ISDIR` and `S_ISREG`.

```
struct stat s;
if (0 == stat(name, &s)) {
   printf("%s ", name);
   if (S_ISDIR( s.st_mode)) puts("is a directory");
   if (S_ISREG( s.st_mode)) puts("is a regular file");
} else {
   perror("stat failed - are you sure I can read this file's
       meta data?");
}
```

Permissions

Permissions are a key part of the way UNIX systems provide security in a filesystem. You may have noticed that the `st_mode` field in `struct stat` contains more than just the file type. It also contains the mode, a description detailing what a user can and can't do with a given file. There are usually three sets of permissions for any file. Permissions for the *user*, the *group* and *other*. For each of the three catagories we need to keep track of whether or not the user is allowed to read the file, write to the file, and execute the file. Since there are three categories and three permissions, permissions are usually represented as a 3-digit octal number. For each digit the least significant byte corresponds to read privilages, the middle one to write privilages and the final byte to execute privilages. They are always presented as *User*, *Group*, *Other* (*UGO*). Below are some common examples:

1. 755: `rwx r-x r-x`

   user: `rwx`, group: `r-x`, others: `r-x`

   User can read, write and execute. Group and others can only read and execute.

2. 644: `rw- r- r-`

   user: `rw-`, group: `r-`, others: `r-`

   User can read and write. Group and others can only read.

It is worth noting that the `rwx` bits for a directory have slightly different meaning. Write-access to a directory will allow you to create or delete new files or directories inside (you can think about this as just having write access to the dirent mappings). Read-access to a directory will allow you to list a directory's contents (this is just read access to the dirent mapping). Execute will allow you to enter the directory and access it. Without the execute bit it is not possible to create or remove files or directories since you cannot access them. You can, however, list the contents of the directory.

There are several command line utilities for interacting with a file's mode. `mknod` changes the type of the file. `chmod` takes a number and a file and changes the permission bits. However, before we can dicuss chmod in detail, we must also understand the user id (`uid`) and group id (`gid`) as well.

User id/Group id

Every user in a UNIX system has a user id. This is a unique number that can identify a user. Similarly, users can be added to collections called groups, and every group also has a uniquely identifying number. Groups have a variety

of uses on UNIX systems. They can be assigned capabilities - a way of describing the level of control a user has over a system. For example, a group you may have run into is the `sudoers` group, a set of trusted users who are allowed to use the command `sudo` to temporarily gain higher privilages. (We'll talk more about how `sudo` works in this chapter). Every file, upon creation, an owner, the creator of the file. This owner's user id (`uid`) can be found inside the `st_mode` file of a `struct stat` with a call to `stat`. Similarly the group id (`gid`) is set as well.

Every process can determine it's `uid` and `gid` with `getuid` and `getgid`. When a processes tries to open a file with a specific mode, it's `uid` and `gid` are compared with the `uid` and `gid` of the file. If the `uid`s match, then the process's request to open the file will be compared with the bits on the user field of the file's permissions. Similarly, if the `gid`s match, then the process's request will be compared with the group field of the permissions. Finally, if none of the ids match, then the other field will apply.

Reading/Changing file permissions

Before we discuss how to change permission bits, we should be able to read them. In C, the `stat` family of library calls can be used. To read permission bits from the command line, use 'ls -l'. Note that the permissions will outputed in the format 'drwxrwxrwx'. The first character indicates the type of file type. Possible values for the first character:

1. (-) regular file
2. (d) directory
3. (c) character device file
4. (l) symbolic link
5. (p) pipe
6. (b) block device
7. (s) socket

Alternatively, use the program `stat` which presents all the information that one could retrieve from the `stat` library call.

To change the permission bits, there is a system call, `int chmod(const char *path, mode_t mode);`. In order to simplify our examples, we will be using the command line utility of the same name `chmod` (short of "change mode"). There are two common ways to use `chmod` ; either with an octal value or with a symbolic string:

```
$ chmod 644 file1
$ chmod 755 file2
$ chmod 700 file3
$ chmod ugo-w file4
$ chmod o-rx file4
```

The base-8 ('octal') digits describe the permissions for each role: The user who owns the file, the group and everyone else. The octal number is the sum of three values given to the three types of permission: read(4), write(2), execute(1)

Example: chmod 755 myfile

1. r + w + x = digit * user has 4+2+1, full permission
2. group has 4+0+1, read and execute permission
3. all users have 4+0+1, read and execute permission

Understanding the 'umask'

The umask *subtracts* (reduces) permission bits from 777 and is used when new files and new directories are created by open, mkdir etc. By default the umask is set to 022 (octal), which means that group and other privileges will not include the writable bit . Each process (including the shell) has a current umask value. When forking, the child inherits the parent's umask value.

For example, by setting the umask to 077 in the shell, ensures that future file and directory creation will only be accessible to the current user,

```
$ umask 077
$ mkdir secretdir
```

As a code example, suppose a new file is created with `open()` and mode bits 666 (write and read bits for user,group and other):

```
open("myfile", O_CREAT, S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP |
    S_IROTH | S_IWOTH);
```

If umask is octal 022, then the permissions of the created file will be 0666 & ~022 ie.

```
S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH
```

A bit about bits

The 'setuid' bit

You may have noticed an additional bit that files with execute permission may have set. This bit is the `setuid` bit. It indicated that when run, the program with set the uid of the user to that of the owner of the file. Similar, there is a `setgid` bit which sets the gid of the executor to the gid of the owner. The canonical example of a program with `setuid` set is `sudo`.

`sudo` is usually a program that is owned by the root user - a user that has all capabilities. By using `sudo` an otherwise unprivilaged user can gain access to most parts of the system. This is useful for running programs that may require elevated privilages, such as using `chown` to change ownership of a file, or to use `mount` to mount or unmount filesystems (an action we will discuss later in this chapter). Here are some examples:

```
$ sudo mount /dev/sda2 /stuff/mydisk
$ sudo adduser fred
$ ls -l /usr/bin/sudo
-r-s--x--x 1 root wheel 327920 Oct 24 09:04 /usr/bin/sudo
```

Effective uid/gid

When executing a process with the setuid bit, it is still possible to determine a user's original uid with `getuid`. The real action of the `setuid` bit is to set the effective userid (`euid`) which can be determined with `geteuid`.

The actions of `getuid` and `geteuid` are described below.

- `getuid` returns the real user id (zero if logged in as root)
- `geteuid` returns the effective userid (zero if acting as root, e.g. due to the setuid flag set on a program)

These functions can allow one to write a program that can only be run by a privilaged user by checking `geteuid` or go a step further and ensure that the only user who can run the code is root by using `getuid`.

The 'sticky' bit

Sticky bits as we use them today do not serve the same purpose as their intial introduction. Sticky bits were a bit that could be set on an executable file that would allow a program's text segment to remain in swap even after the end of the program's execution. This made subsequent executions of the same program faster. Today, this behavior is no longer supported and the sticky bit only holds meaning when set on a directory,

When a directory's sticky bit is set only the file's owner, the directory's owner, and the root user can rename or delete the file. This is useful when multiple users have write access to a common directory. A common use of the sticky bit is for the shared and writable `/tmp` directory.

To set the sticky bit, use `chmod +t`.

```
aneesh$ mkdir sticky
aneesh$ chmod +t sticky
aneesh$ ls -l
drwxr-xr-x 7 aneesh aneesh  4096 Nov 1 14:19 .
drwxr-xr-x 53 aneesh aneesh 4096 Nov 1 14:19 ..
drwxr-xr-t 2 aneesh aneesh  4096 Nov 1 14:19 sticky
aneesh$ su newuser
newuser$ rm -rf sticky
rm: cannot remove 'sticky': Permission denied
newuser$ exit
aneesh$ rm -rf sticky
aneesh$ ls -l
drwxr-xr-x 7 aneesh aneesh  4096 Nov 1 14:19 .
drwxr-xr-x 53 aneesh aneesh 4096 Nov 1 14:19 ..
```

Note that in the example above, the username is prepended to the prompt, and the command `su` is used to switch users.

Virtual filesystems and other filesystems

POSIX systems, such as Linux and Mac OSX (which is based on BSD) include several virtual filesystems that are mounted (available) as part of the file-system. Files inside these virtual filesystems do not exist on the disk; they are generated dynamically by the kernel when a process requests a directory listing. Linux provides 3 main virtual filesystems

```
/dev  - A list of physical and virtual devices (for example network
    card, cdrom, random number generator)
/proc - A list of resources used by each process and (by tradition)
    set of system information
/sys - An organized list of internal kernel entities
```

For example if I want a continuous stream of 0s, I can `cat /dev/zero`.

Another example is the file `/dev/null` - a great place to store bits that you never need to read! Bytes sent to `/dev/null/` are never stored - they are simply discarded. A common use of `/dev/null` is to discard standard output. For example,

```
$ ls . >/dev/null
```

Managing files and filesystems

Given the multitude of operations that are availible to you from the filesystem, let's explore some tools and techniques that can be used to manage files and filesystems.

One example is creating a secure directory. Suppose you created your own directory in /tmp and then set the permissions so that only you can use the directory (see below). Is this secure?

```
$ mkdir /tmp/mystuff
$ chmod 700 /tmp/mystuff
```

There is a window of opportunity between when the directory is created and when it's permissions are changed. This leads to several vulnerabilities that are based on a race condition.

Another user replaces `mystuff` with a hardlink to an existing file or directory owned by the second user, then they would be able to read and control the contents of the `mystuff` directory. Oh no - our secrets are no longer secret!

However in this specific example the `/tmp` directory has the sticky bit set, so other users may not delete the `mystuff` directory, and the simple attack scenario described above is impossible. This does not mean that creating the directory and then later making the directory private is secure! A better version is to atomically create the directory with the correct permissions from its inception -

```
$ mkdir -m 700 /tmp/mystuff
```

Obtaining random data

`/dev/random` is a file which contains number generator where the entropy is determined from environmental noise. Random will block/wait until enough entropy is collected from the environment.

`/dev/urandom` is like random, but differs in the fact that it allows for repetition (lower entropy threshold), thus wont block.

How can I copy bytes from one file to another?

Use the versatile `dd` command. For example, the following command copies 1 MB of data from the file `/dev/urandom` to the file `/dev/null`. The data is copied as 1024 blocks of blocksize 1024 bytes.

```
$ dd if=/dev/urandom of=/dev/null bs=1k count=1024
```

Both the input and output files in the example above are virtual - they don't exist on a disk. This means the speed of the transfer is unaffected by hardware power.

`dd` is also commonly used to make a copy of a disk or an entire filesystem to create images that can either be burned on to other disks or to distribute data to other users.

What happens when I touch a file?

The `touch` executable creates file if it does not exist and also updates the file's last modified time to be the current time. For example, we can make a new private file with the current time:

```
$ umask 077     # all future new files will maskout all r,w,x bits
    for group and other access
$ touch file123 # create a file if it does not exist, and update
    its modified time
$ stat file123
  File: ‘file123’
  Size: 0         Blocks: 0        IO Block: 65536 regular empty
      file
Device: 21h/33d Inode: 226148  Links: 1
Access: (0600/-rw-------) Uid: (395606/ angrave) Gid: (61019/ ews)
Access: 2014-11-12 13:42:06.000000000 -0600
Modify: 2014-11-12 13:42:06.001787000 -0600
Change: 2014-11-12 13:42:06.001787000 -0600
```

An example use of touch is to force make to recompile a file that is unchanged after modifying the compiler options inside the makefile. Remeber that make is 'lazy' - it will compare the modified time of the source file with the corresponding output file to see if the file needs to be recompiled

```
$ touch myprogram.c # force my source file to be recompiled
$ make
```

Managing Filesystems

To manage filesystems on your machine, use `mount`. Using mount without any options generates a list (one filesystem per line) of mounted filesystems including networked, virtual and local (spinning disk / SSD-based) filesystems. Here is a typical output of mount

```
$ mount
/dev/mapper/cs241--server_sys-root on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
tmpfs on /dev/shm type tmpfs
    (rw,rootcontext="system_u:object_r:tmpfs_t:s0")
/dev/sda1 on /boot type ext3 (rw)
/dev/mapper/cs241--server_sys-srv on /srv type ext4 (rw)
/dev/mapper/cs241--server_sys-tmp on /tmp type ext4 (rw)
/dev/mapper/cs241--server_sys-var on /var type ext4 (rw)rw,bind)
/srv/software/Mathematica-8.0 on /software/Mathematica-8.0 type
    none (rw,bind)
engr-ews-homes.engr.illinois.edu:/fs1-homes/angrave/linux on
```

```
    /home/angrave type nfs
    (rw,soft,intr,tcp,noacl,acregmin=30,vers=3,sec=sys,sloppy,addr=128.174.252.102)
```

Notice that each line includes the filesystem type source of the filesystem and mount point. To reduce this output we can pipe it into `grep` and only see lines that match a regular expression.

```
>mount | grep proc # only see lines that contain 'proc'
proc on /proc type proc (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
```

How do I mount a disk image?

Suppose you had downloaded a bootable linux disk image from the URL

```
wget $URL
```

Before putting the filesystem on a CD, we can mount the file as a filesystem and explore its contents. Note, mount requires root access, so let's run it using sudo

```
$ mkdir arch
$ sudo mount -o loop archlinux-2015.04.01-dual.iso ./arch
$ cd arch
```

Before the mount command, the arch directory is new and obviously empty. After mounting, the contents of `arch/` will be drawn from the files and directories stored in the filesystem stored inside the `archlinux-2014.11.01-dual.iso` file. The `loop` option is required because we want to mount a regular file not a block device such as a physical disk.

The loop option wraps the original file as a block device - in this example we will find out below that the file system is provided under `/dev/loop0` : We can check the filesystem type and mount options by running the mount command without any parameters. We will pipe the output into `grep` so that we only see the relevant output line(s) that contain 'arch'

```
$ mount | grep arch
/home/demo/archlinux-2014.11.01-dual.iso on /home/demo/arch type
    iso9660 (rw,loop=/dev/loop0)
```

The iso9660 filesystem is a read-only filesystem originally designed for optical storage media (i.e. CDRoms). Attempting to change the contents of the filesystem will fail

```
$ touch arch/nocando
touch: cannot touch '/home/demo/arch/nocando': Read-only file system
```

Memory Mapped IO

While we traditionally think of reading and writing from a file as operation that happen by using the `read` and `write` calls, there is an alternative, mapping a file into memory using `mmap`. `mmap` can also be used for IPC, and you can see more about `mmap` as a system call that enables shared memory in the IPC chapter. In this chapter, we'll briefly explore `mmap` as a filesystem operation.

`mmap` takes a file and maps its contents into memory. This allows a user to treat the entire file as a buffer in memory for easier semantics while programming, and to avoid having to read a file as discreete chunks explicitly.

Not all filesystems support using `mmap` for IO, but amongst those that do, not all have the same behavior. Some will simply implement `mmap` as a wrapper around `read` and `write`. Others will add additional optimizations by taking advantage of the kernel's page cache. Of course, such optimization can be used in the implementation of `read` and `write` as well, so often using `mmap` does not impact performance.

`mmap` is used to perform some operations such as loading libraries and processes into memory. If many programs only need read-access to the same file (e.g. `/bin/bash`, the C library) then the same physical memory can be shared between multiple processes.

The process to map a file into memory is simple:

1. `mmap` requires a filedescriptor, so we need to `open` the file first
2. We seek to our desired size and write one byte to ensure that the file is sufficient length
3. When finished call munmap to unmap the file from memory.

Here is a quick example.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>


int fail(char *filename, int linenumber) {
  fprintf(stderr, "%s:%d %s\n", filename, linenumber,
      strerror(errno));
  exit(1);
  return 0; /*Make compiler happy */
}
#define QUIT fail(__FILE__, __LINE__ )

int main() {
  // We want a file big enough to hold 10 integers
  int size = sizeof(int) * 10;

  int fd = open("data", O_RDWR | O_CREAT | O_TRUNC, 0600); //6 =
      read+write for me!

  lseek(fd, size, SEEK_SET);
  write(fd, "A", 1);
```

```
  void *addr = mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, 0);
  printf("Mapped at %p\n", addr);
  if (addr == (void*) -1 ) QUIT;

  int *array = addr;
  array[0] = 0x12345678;
  array[1] = 0xdeadc0de;

  munmap(addr,size);
  return 0;

}
```

The careful reader may notice that our integers were written in least-significant-byte format (because that is the endianess of the CPU) and that we allocated a file that is one byte too many! The `PROT_READ | PROT_WRITE` options specify the virtual memory protection. The option `PROT_EXEC` (not used here) can be set to allow CPU execution of instructions in memory (e.g. this would be useful if you mapped an executable or library).

Reliable Single Disk Filesystems

Most filesystems cache significant amounts of disk data in physical memory. Linux, in this respect, is particularly extreme: All unused memory is used as a giant disk cache. The disk cache can have significant impact on overall system performance because disk I/O is slow. This is especially true for random access requests on spinning disks where the disk read-write latency is dominated by the seek time required to move the read-write disk head to the correct position.

For efficiency, the kernel caches recently used disk blocks. For writing, we have to choose a trade-off between performance and reliability: Disk writes can also be cached ("Write-back cache") where modified disk blocks are stored in memory until evicted. Alternatively a 'write-through cache' policy can be employed where disk writes are sent immediately to the disk. The latter is safer (as filesystem modifications are quickly stored to persistent media) but slower than a write-back cache; If writes are cached then they can be delayed and efficiently scheduled based on the physical position of each disk block. Note this is a simplified description because solid state drives (SSDs) can be used as a secondary write-back cache.

Both solid state disks (SSD) and spinning disks have improved performance when reading or writing sequential data. Thus operating system can often use a read-ahead strategy to amortize the read-request costs (e.g. time cost for a spinning disk) and request several contiguous disk blocks per request. By issuing an I/O request for the next disk block before the user application requires the next disk block, the apparent disk I/O latency can be reduced.

If your data is important and needs to be force written to disk, call `sync` to request that a filesystem changes be written (flushed) to disk. However, not all operating systems honor this request and even if the data is evicted from the kernel buffers the disk firmware use an internal on-disk cache or may not yet have finished changing the physical media. Note you can also request that all changes associated with a particular file descriptor are flushed to disk using `fsync(int fd)`

If your operating system fails in the middle of an operation, most modern file systems do something called **journalling** that work around this. What the file system does is before it completes a potentially expensive operation, is that it writes what it is going to do down in a journal. In the case of a crash or failure, one can step through the journal and see which files are corrupt and fix them. This is a way to salvage hard disks in cases there is critical data and there is no apparent backup.

Even though it is unlikely for your computer, programming for datacenters means that disks fail every few seconds. Disk failures are measured using "Mean-Time-Failure". For large arrays, the mean failure time can be surprisingly short. For example if the MTTF(single disk) = 30,000 hours, then the MTTF(1000 disks)= 30000/1000=30 hours or about a day and a half!

RAID

One way to protect against this is to store the data twice! This is the main principle of a "RAID-1" disk array. RAID is short for redundant array of inexpensive disks. By duplicating the writes to a disk with writes to another backup disk, there are exactly two copies of the data. If one disk fails, the other disk serves as the only copy until it can be re-cloned. Reading data is faster since data can be requested from either disk, but writes are potentially twice as slow because now two write commands need to be issued for every disk block write. Compared to using a single disk, the cost of storage per byte has doubled.

Another common RAID scheme is RAID-0, meaning that a file could be split up amoung two disks, but if any one of the disks fail then the files are irrecoverable. This has the benefit of halving write times because one part of the file could be writing to hard disk one and another part to hard disk two.

It is also common to combine these systems. If you have a lot of hard disks, consider RAID-10. This is where you have two systems of RAID-1, but the systems are hooked up in RAID-0 to each other. This means you would get roughly the same speed from the slowdowns but now any one disk can fail and you can recover that disk. If two disks from opposing raid partitions fail there is a chance that recover can happen though we don't could on it most of the time.

Higher Level Raids

RAID-3 uses parity codes instead of mirroring the data. For each N-bits written we will write one extra bit, the 'Parity bit' that ensures the total number of 1s written is even. The parity bit is written to an additional disk. If any one disk (including the parity disk) is lost, then its contents can still be computed using the contents of the other disks.

Figure 11.3:

One disadvantage of RAID-3 is that whenever a disk block is written, the parity block will always be written too. This means that there is effectively a bottleneck in a separate disk. In practice, this is more likely to cause a failure because one disk is being used 100% of the time and once that disk fails then the other disks are more prone to failure.

A single disk failure will not result in data loss (because there is sufficient data to rebuild the array from the remaining disks). Data-loss will occur when a two disks are unusable because there is no longer sufficient data to rebuild the array. We can calculate the probability of a two disk failure based on the repair time which includes not just the time to insert a new disk but the time required to rebuild the entire contents of the array.

```
MTTF = mean time to failure
MTTR = mean time to repair
N = number of original disks

p = MTTR / (MTTF-one-disk / (N-1))
```

Using typical numbers (MTTR=1day, MTTF=1000days, N-1 = 9, p=0.009)

There is a 1% chance that another drive will fail during the rebuild process (at that point you had better hope you still have an accessible backup of your original data. In practice the probability of a second failure during the repair process is likely higher because rebuilding the array is I/O-intensive (and on top of normal I/O request activity). This higher I/O load will also stress the disk array

RAID-5 is similar to RAID-3 except that the check block (parity information) is assigned to different disks for different blocks. The check-block is 'rotated' through the disk array. RAID-5 provides better read and write performance than RAID-3 because there is no longer the bottleneck of the single parity disk. The one drawback is that you need more disks to have this setup and there are more complicated algorithms need to be used

Failure is the common case. Google reports 2-10% of disks fail per year Now multiply that by 60,000+ disks in a single warehouse... Must survive failure of not just a disk, but a rack of servers or a whole data center

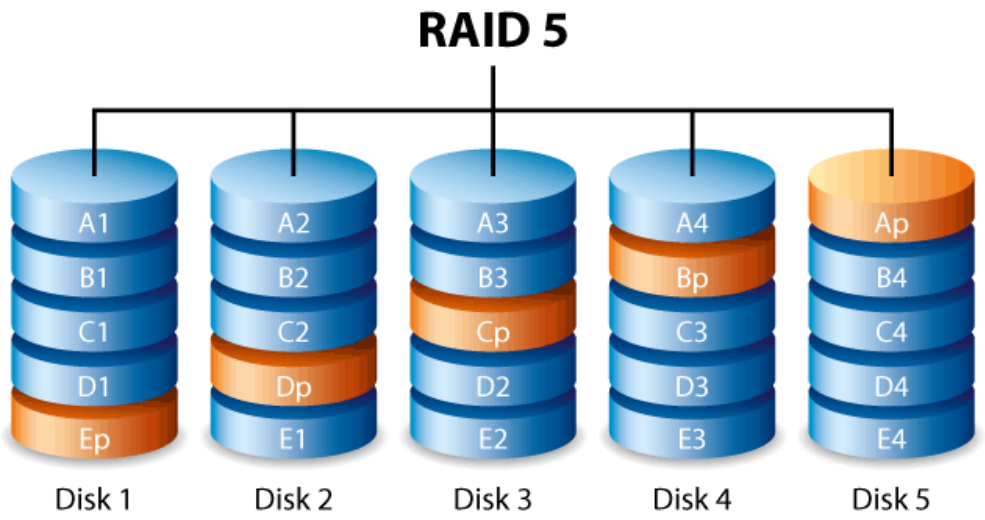


Figure 11.4:

Solutions Simple redundancy (2 or 3 copies of each file) e.g., Google GFS (2001) More efficient redundancy (analogous to RAID 3++) e.g., Google Colossus filesystem (~2010): customizable replication including Reed-Solomon codes with 1.5x redundancy

Under construction - Simple Filesystem Model

We'll build a virtual disk and then write some C code to access its contents. Our filesystem will divide the bytes available into space for inodes and a much larger space for disk blocks. Each disk block will be 4096 bytes.

```
// Disk size:
#define MAX_INODE (1024)
#define MAX_BLOCK (1024*1024)

// Each block is 4096 bytes:
typedef char[4096] block_t;

// A disk is an array of inodes and an array of disk blocks:
struct inode[MAX_INODE] inodes;
block[MAX_BLOCK] blocks;
```

Note for clarity we will not use 'unsigned' in this code example. Our fixed-sized inodes will contain the file's size in bytes, permission, user, group information, time meta-data. What is most relevant to the problem-at hand is that it will also include ten pointers to disk blocks that we will use to refer to the actual file's contents!

```
struct inode {
 int[10] directblocks; // indices for the block array i.e. where to
    the find the file's content
 long size;
 // ... standard inode meta-data e.g.
 int mode, userid,groupid;
```

```
 time_t ctime,atime,mtime;
}
```

Now we can work out how to read a byte at offset `position` of our file:

```
char readbyte(inode*inode,long position) {
  if(position <0 || position >= inode->size) return -1; // invalid
      offset

  int  block_count = position / 4096,offset = position % 4096;

  // block count better be 0..9 !
  int physical_idx = lookup_physical_block_index(inode, block_count
      );

  // sanity check that the disk block index is reasonable...
  assert(physical_idx >=0 && physical_idx < MAX_BLOCK);


  // read the disk block from our virtual disk 'blocks' and return
      the specific byte
  return blocks[physical_idx][offset];
}
```

Our initial version of lookup_physical_block is simple - we can use our table of 10 direct blocks!

```
int lookup_physical_block_index(inode*inode, int block_count) {
  assert(block_count>=0 && block_count < 10);

  return inode->directblocks[ block_count ]; // returns an index
      value between [0,MAX_BLOCK)
}
```

This simple representation is reasonable provided we can represent all possible files with just ten blocks i.e. up to 40KB. What about larger files? We need the inode struct to always be the same size so just increasing the existing direct block array to 20 would roughly double the size of our inodes. If most of our files require less than 10 blocks, then our inode storage is now wasteful. To solve this problem we will use a disk block called the *indirect block* to extend the array of pointers at our disposal. We will only need this for files > 40KB

```
struct inode {
 int[10] directblocks; // if size<4KB then only the first one is
     valid
 int indirectblock; // valid value when size >= 40KB
 int size;
 ...
}
```

The indirect block is just a regular disk block of 4096 bytes, but we will use it to hold pointers to disk blocks. Our pointers in this case are just integers, so we need to cast the pointer to an integer pointer:

```
int lookup_physical_block_index(inode*inode, int block_count) {
  assert(sizeof(int)==4); // Warning this code assumes an index is
      4 bytes!
  assert(block_count>=0 && block_count < 1024 + 10); // 0 <=
      block_count< 1034

  if( block_count < 10)
     return inode->directblocks[ block_count ];

  // read the indirect block from disk:
  block_t* oneblock = & blocks[ inode->indirectblock ];

  // Treat the 4KB as an array of 1024 pointers to other disk blocks
  int* table = (int*) oneblock;

 // Look up the correct entry in the table
 // Offset by 10 because the first 10 blocks of data are already
 // accounted for
  return table[ block_count - 10 ];
}
```

For a typical filesystem, our index values are 32 bits i.e. 4bytes. Thus in 4096 bytes we can store 4096 / 4 = 1024 entries. This means our indirect block can refer to 1024 * 4KB = 4MB of data. With the first ten direct blocks, we can therefore accommodate files up to 40KB + 1024 * 4KB= 4136KB . Some of the later table entries can be invalid for files that are smaller than this.

For even larger files, we could use two indirect blocks. However there's a better alternative, that will allow us to efficiently scale up to huge files. We will include a double-indirect pointer and if that's not enough a triple indirect pointer. The double indirect pointer means we have a table of 1024 entries to disk blocks that are used as 1024 entries. This means we can refer to 1024*1024 disk blocks of data. Here is a link

```
int lookup_physical_block_index(inode*inode, int block_count) {
  if( block_count < 10)
     return inode->directblocks[ block_count ];

  // Use indirect block for the next 1024 blocks:
  // Assumes 1024 ints can fit inside each block!
  if( block_count < 1024 + 10) {
      int* table = (int*) & blocks[ inode->indirectblock ];
      return table[ block_count - 10 ];
  }
  // For huge files we will use a table of tables
  int i = (block_count - 1034) / 1024 , j = (block_count - 1034) %
      1024;
  assert(i<1024); // triple-indirect is not implemented here!

  int* table1 = (int*) & blocks[ inode->doubleindirectblock ];
   // The first table tells us where to read the second table ...
```

```
  int* table2 = (int*) & blocks[ table1[i] ];
  return table2[j];

   // For gigantic files we will need to implement triple-indirect
      (table of tables of tables)
}
```

Notice that reading a byte using double indirect requires 3 disk block reads (two tables and the actual data block).

Topics

- Superblock
- Data Block
- Inode
- Relative Path
- File Metadata
- Hard and Soft Links
- Permission Bits
- Mode bits
- Working with Directories
- Virtual File System
- Reliable File Systems
- RAID

Questions

- How big can files be on a file system with 15 Direct blocks, 2 double, 3 triple indirect, 4kb blocks and 4byte entries? (Assume enough infinite blocks)
- What is a superblock? Inode? Datablock?
- How do I simplify `/./proc/../dev/./random/`
- In ext2, what is stored in an inode, and what is stored in a directory entry?
- What are /sys, /proc, /dev/random, and /dev/urandom?
- What are permission bits?
- How do you use chmod to set user/group/owner read/write/execute permissions?
- What does the "dd" command do?
- What is the difference between a hard link and a symbolic link? Does the file need to exist?
- "ls -l" shows the size of each file in a directory. Is the size stored in the directory or in the file's inode?

Bibliography

12 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,

```
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]

```
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$_s$$_i$$_g$$_m$$_a$$_s$$_k$.*URL*.

Jure Åăorn. gto76/comp-cpp, Jun 2015. URL https://github.com/gto76/comp-cpp/blob/1bf9a77eaf8f57f7358a316e
src/output.c.

13 Review

A non-comprehensive list of topics is below.

CSP (critical section problems)
HTTP
SIGINT
TCP
TLB
Virtual Memory
arrays
barrier
c strings
chmod
client/server
coffman conditions
condition variables
context switch
deadlock
dining philosophers
epoll
exit
file I/O
file system representation
fork/exec/wait
fprintf
free
heap allocator
heap/stack
inode vs name
malloc
mkfifo
mmap
mutexes
network ports
open/close
operating system terms
page fault
page tables
pipes
pointer arithmetic
pointers
printing (printf)
producer/consumer
progress/mutex
race conditions
read/write
reader/writer
resource allocation graphs
ring buffer
scanf
buffering
scheduling
select
semaphores
signals
sizeof
stat
stderr/stdout
symlinks
thread control (_create, _join, _exit)
variable initializers
variable scope
vm thrashing
wait macros
write/read with errno, EINTR and partial data

C

Memory and Strings

Q1.1

In the example below, which variables are guaranteed to print the value of zero?

```
int a;
static int b;

void func() {
   static int c;
   int d;
   printf("%d %d %d %d\n",a,b,c,d);
}
```

Q 1.2

In the example below, which variables are guaranteed to print the value of zero?

```
void func() {
   int* ptr1 = malloc( sizeof(int) );
   int* ptr2 = realloc(NULL, sizeof(int) );
   int* ptr3 = calloc( 1, sizeof(int) );
   int* ptr4 = calloc( sizeof(int) , 1);

   printf("%d %d %d %d\n",*ptr1,*ptr2,*ptr3,*ptr4);
}
```

Q 1.3

Explain the error in the following attempt to copy a string.

```
char* copy(char*src) {
 char*result = malloc( strlen(src) );
 strcpy(result, src);
 return result;
}
```

Q 1.4

Why does the following attempt to copy a string sometimes work and sometimes fail?

```
char* copy(char*src) {
 char*result = malloc( strlen(src) +1 );
 strcat(result, src);
 return result;
}
```

Q 1.4

Explain the two errors in the following code that attempts to copy a string.

```
char* copy(char*src) {
 char result[sizeof(src)];
 strcpy(result, src);
 return result;
}
```

Q 1.5

Which of the following is legal?

```
char a[] = "Hello"; strcpy(a, "World");
char b[] = "Hello"; strcpy(b, "World12345", b);
char* c = "Hello"; strcpy(c, "World");
```

Q 1.6

Complete the function pointer typedef to declare a pointer to a function that takes a void* argument and returns a void*. Name your type 'pthread_callback'

```
typedef ______________________;
```

Q 1.7

In addition to the function arguments what else is stored on a thread's stack?

Q 1.8

Implement a version of `char* strcat(char*dest, const char*src)` using only `strcpy strlen` and pointer arithmetic

```
char* mystrcat(char*dest, const char*src) {

  ? Use strcpy strlen here

  return dest;
}
```

Q 1.9

Implement version of size_t strlen(const char*) using a loop and no function calls.

```
size_t mystrlen(const char*s) {

}
```

Q 1.10

Identify the three bugs in the following implementation of `strcpy`.

```
char* strcpy(const char* dest, const char* src) {
  while(*src) {*dest++ = *src++; }
  return dest;
}
```

Printing

Q 2.1

Spot the two errors!

```
fprintf("You scored 100%");
```

Formatting and Printing to a file

Q 3.1

Complete the following code to print to a file. Print the name, a comma and the score to the file 'result.txt'

```
char* name = .....;
int score = ......
FILE *f = fopen("result.txt",_____);
if(f) {
    -----
}
fclose(f);
```

Printing to a string

Q 4.1

How would you print the values of variables a,mesg,val and ptr to a string? Print a as an integer, mesg as C string, val as a double val and ptr as a hexadecimal pointer. You may assume the mesg points to a short C string(<50 characters). Bonus: How would you make this code more robust or able to cope with?

```
char* toString(int a, char*mesg, double val, void* ptr) {
   char* result = malloc( strlen(mesg) + 50);
    -----
   return result;
}
```

Input parsing

Q 5.1

Why should you check the return value of sscanf and scanf? ## Q 5.2 Why is 'gets' dangerous?

Q 5.3

Write a complete program that uses `getline`. Ensure your program has no memory leaks.

Heap memory

When would you use calloc not malloc? When would realloc be useful?

(Todo - move this question to another page) What mistake did the programmer make in the following code? Is it possible to fix it i) using heap memory? ii) using global (static) memory?

```
static int id;

char* next_ticket() {
  id ++;
  char result[20];
  sprintf(result,"%d",id);
  return result;
}
```

Threading

Q1

Is the following code thread-safe? Redesign the following code to be thread-safe. Hint: A mutex is unnecessary if the message memory is unique to each call.

```
static char message[20];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *format(int v) {
  pthread_mutex_lock(&mutex);
  sprintf(message, ":%d:" ,v);
  pthread_mutex_unlock(&mutex);
  return message;
}
```

Q2

Which one of the following does not cause a process to exit? * Returning from the pthread's starting function in the last running thread. * The original thread returning from main. * Any thread causing a segmentation fault. *

Any thread calling `exit`. * Calling `pthread_exit` in the main thread with other threads still running.

Q3

Write a mathematical expression for the number of “W” characters that will be printed by the following program. Assume a,b,c,d are small positive integers. Your answer may use a ‘min’ function that returns its lowest valued argument.

```
unsigned int a=...,b=...,c=...,d=...;

void* func(void* ptr) {
  char m = * (char*)ptr;
  if(m == 'P') sem_post(s);
  if(m == 'W') sem_wait(s);
  putchar(m);
  return NULL;
}

int main(int argv, char** argc) {
  sem_init(s,0, a);
  while(b--) pthread_create(&tid, NULL, func, "W");
  while(c--) pthread_create(&tid, NULL, func, "P");
  while(d--) pthread_create(&tid, NULL, func, "W");
  pthread_exit(NULL);
  /*Process will finish when all threads have exited */
}
```

Q4

Complete the following code. The following code is supposed to print alternating `A` and `B`. It represents two threads that take turns to execute. Add condition variable calls to `func` so that the waiting thread does not need to continually check the `turn` variable. Q: Is pthread_cond_broadcast necessary or is pthread_cond_signal sufficient?

```
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void* turn;

void* func(void* mesg) {
  while(1) {
// Add mutex lock and condition variable calls ...

    while(turn == mesg) {
        /* poll again ... Change me - This busy loop burns CPU time! */
    }

    /* Do stuff on this thread */
    puts( (char*) mesg);
    turn = mesg;

  }
  return 0;
}

int main(int argc, char** argv){
  pthread_t tid1;
  pthread_create(&tid1, NULL, func, "A");
  func("B"); // no need to create another thread - just use the main thread
  return 0;
}
```

Q5

Identify the critical sections in the given code. Add mutex locking to make the code thread safe. Add condition variable calls so that `total` never becomes negative or above 1000. Instead the call should block until it is safe to proceed. Explain why `pthread_cond_broadcast` is necessary.

```
int total;
void add(int value) {
 if(value < 1) return;
 total += value;
}
void sub(int value) {
 if(value < 1) return;
 total -= value;
}
```

Q6

A non-threadsafe data structure has `size()` `enq` and `deq` methods. Use condition variable and mutex lock to complete the thread-safe, blocking versions.

```
void enqueue(void* data) {
  // should block if the size() would become greater than 256
  enq(data);
}
void* dequeue() {
  // should block if size() is 0
  return deq();
}
```

Q7

Your startup offers path planning using latest traffic information. Your overpaid intern has created a non-threadsafe data structure with two functions: `shortest` (which uses but does not modify the graph) and `set_edge` (which modifies the graph).

```
graph_t* create_graph(char* filename); // called once

// returns a new heap object that is the shortest path from vertex i to j
path_t* shortest(graph_t* graph, int i, int j);

// updates edge from vertex i to j
void set_edge(graph_t* graph, int i, int j, double time);
```

For performance, multiple threads must be able to call `shortest` at the same time but the graph can only be modified by one thread when no threads other are executing inside `shortest` or `set_edge`.

Use mutex lock and condition variables to implement a reader-writer solution. An incomplete attempt is shown below. Though this attempt is threadsafe (thus sufficient for demo day!), it does not allow multiple threads to calculate `shortest` path at the same time and will not have sufficient throughput.

```
path_t* shortest_safe(graph_t* graph, int i, int j) {
  pthread_mutex_lock(&m);
  path_t* path = shortest(graph, i, j);
  pthread_mutex_unlock(&m);
  return path;
}
void set_edge_safe(graph_t* graph, int i, int j, double dist) {
  pthread_mutex_lock(&m);
  set_edge(graph, i, j, dist);
  pthread_mutex_unlock(&m);
}
```

> Note thread-programming synchronization problems are on a separate wiki page. This page focuses on conceptual topics. Question numbers subject to change

Deadlock

Q1

What do each of the Coffman conditions mean? (e.g. can you provide a definition of each one) * Hold and wait * Circular wait * No pre-emption * Mutual exclusion

Q2

Give a real life example of breaking each Coffman condition in turn. A situation to consider: Painters, paint and paint brushes. Hold and wait Circular wait No pre-emption Mutual exclusion

Q3

Identify when Dining Philosophers code causes a deadlock (or not). For example, if you saw the following code snippet which Coffman condition is not satisfied?

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

Q4

How many processes are blocked?

- P1 acquires R1
- P2 acquires R2
- P1 acquires R3
- P2 waits for R3
- P3 acquires R5
- P1 acquires R4
- P3 waits for R1
- P4 waits for R5
- P5 waits for R1

Q5

How many of the following statements are true for the reader-writer problem?

- There can be multiple active readers
- There can be multiple active writers
- When there is an active writer the number of active readers must be zero
- If there is an active reader the number of active writers must be zero
- A writer must wait until the current active readers have finished

IPC

Q1

What are the following and what is their purpose? * Translation Lookaside Buffer * Physical Address * Memory Management Unit * The dirty bit

Q2

How do you determine how many bits are used in the page offset?

Q3

20 ms after a context switch the TLB contains all logical addresses used by your numerical code which performs main memory access 100% of the time. What is the overhead (slowdown) of a two-level page table compared to a single-level page table?

Q4

Explain why the TLB must be flushed when a context switch occurs (i.e. the CPU is assigned to work on a different process).

Question numbers subject to change

Processes

Q1

Fill in the blanks to make the following program print 123456789. If `cat` is given no arguments it simply prints its input until EOF. Bonus: Explain why the `close` call below is necessary.

```c
int main() {
  int i = 0;
  while(++i < 10) {
    pid_t pid = fork();
    if(pid == 0) {/* child */
      char buffer[16];
      sprintf(buffer, ______,i);
      int fds[ ______];
      pipe( fds);
      write( fds[1], ______,______ ); // Write the buffer into the pipe
      close( ______ );
      dup2( fds[0], ______);
      execlp( "cat", "cat", ______ );
      perror("exec"); exit(1);
    }
    waitpid(pid, NULL, 0);
  }
  return 0;
}
```

Q2

Use POSIX calls `fork` `pipe` `dup2` and `close` to implement an autograding program. Capture the standard output of a child process into a pipe. The child process should `exec` the program `./test` with no additional arguments (other than the process name). In the parent process read from the pipe: Exit the parent process as soon as the captured output contains the `!` character. Before exiting the parent process send SIGKILL to the child process. Exit 0 if the output contained a !. Otherwise if the child process exits causing the pipe write end to be closed, then exit with a value of 1. Be sure to close the unused ends of the pipe in the parent and child process

Q3 (Advanced)

This advanced challenge uses pipes to get an "AI player" to play itself until the game is complete. The program `tictactoe` accepts a line of input - the sequence of turns made so far, prints the same sequence followed by another turn, and then exits. A turn is specified using two characters. For example "A1" and "C3" are two opposite corner positions. The string `B2A1A3` is a game of 3 turns/plys. A valid response is `B2A1A3C1` (the C1 response blocks the diagonal B2 A3 threat). The output line may also include a suffix `-I win` `-You win` `-invalid` or

`-draw` Use pipes to control the input and output of each child process created. When the output contains a `-`, print the final output line (the entire game sequence and the result) and exit.

Mapped Memory

Q1

Write a function that uses fseek and ftell to replace the middle character of a file with an 'X'

```
void xout(char* filename) {
  FILE *f = fopen(filename, ____ );


}
```

Q2

In an `ext2` filesystem how many inodes are read from disk to access the first byte of the file `/dir1/subdirA/notes.txt` ? Assume the directory names and inode numbers in the root directory (but not the inodes themselves) are already in memory.

Q3

In an `ext2` filesystem what is the minimum number of disk blocks that must be read from disk to access the first byte of the file `/dir1/subdirA/notes.txt` ? Assume the directory names and inode numbers in the root directory and all inodes are already in memory.

Q4

In an `ext2` filesystem with 32 bit addresses and 4KB disk blocks, an inodes that can store 10 direct disk block numbers. What is the minimum file size required to require an single indirection table? ii) a double direction table?

Q5

Fix the shell command `chmod` below to set the permission of a file `secret.txt` so that the owner can read,write,and execute permissions the group can read and everyone else has no access.

```
chmod 000 secret.txt
```

- Wiki w/Interactive MC Questions
- See Coding questions
- See Short answer questions

Networking

Q1

What is a socket?

Q2

What is special about listening on port 1000 vs port 2000?

- Port 2000 is twice as slow as port 1000
- Port 2000 is twice as fast as port 1000
- Port 1000 requires root privileges
- Nothing

Q3

Describe one significant difference between IPv4 and IPv6

Q4

When and why would you use ntohs?

Q5

If a host address is 32 bits which IP scheme am I most likely using? 128 bits?

Q6

Which common network protocol is packet based and may not successfully deliver the data?

Q7

Which common protocol is stream-based and will resend data if packets are lost?

Q8

What is the SYN ACK ACK-SYN handshake?

Q9

Which one of the following is NOT a feature of TCP? - Packet re-ordering - Flow control - Packet re-tranmission - Simple error detection - Encryption

Q10

What protocol uses sequence numbers? What is their initial value? And why?

Q11

What are the minimum network calls are required to build a TCP server? What is their correct order?

Q12

What are the minimum network calls are required to build a TCP client? What is their correct order?

Q13

When would you call bind on a TCP client?

Q14

What is the purpose of socket bind listen accept ?

Q15

Which of the above calls can block, waiting for a new client to connect?

Q16

What is DNS? What does it do for you? Which of the CS241 network calls will use it for you?

Q17

For getaddrinfo, how do you specify a server socket?

Q18

Why may getaddrinfo generate network packets?

Q19

Which network call specifies the size of the allowed backlog?

Q20

Which network call returns a new file descriptor?

Q21

When are passive sockets used?

Q22

When is epoll a better choice than select? When is select a better choice than epoll?

Q23

Will `write(fd, data, 5000)` always send 5000 bytes of data? When can it fail?

Q24

How does Network Address Translation (NAT) work?

Q25

@MCQ Assuming a network has a 20ms Transmit Time between Client and Server, how much time would it take to establish a TCP Connection? 20 ms 40 ms 100 ms 60 ms @ANS 3 Way Handshake @EXP @END

Q26

What are some of the differences between HTTP 1.0 and HTTP 1.1? How many ms will it take to transmit 3 files from server to client if the network has a 20ms transmit time? How does the time taken differ between HTTP 1.0 and HTTP 1.1?

Coding questions

Q 2.1

Writing to a network socket may not send all of the bytes and may be interrupted due to a signal. Check the return value of `write` to implement `write_all` that will repeatedly call `write` with any remaining data. If `write` returns -1 then immediately return -1 unless the `errno` is `EINTR` - in which case repeat the last `write` attempt. You will need to use pointer arithmetic.

```c
// Returns -1 if write fails (unless EINTR in which case it recalls write
// Repeated calls write until all of the buffer is written.
ssize_t write_all(int fd, const char *buf, size_t nbyte) {
  ssize_t nb = write(fd, buf, nbyte);
  return nb;
}
```

Q 2.2

Implement a multithreaded TCP server that listens on port 2000. Each thread should read 128 bytes from the client file descriptor and echo it back to the client, before closing the connection and ending the thread.

Q 2.3

Implement a UDP server that listens on port 2000. Reserve a buffer of 200 bytes. Listen for an arriving packet. Valid packets are 200 bytes or less and start with four bytes 0x65 0x66 0x67 0x68. Ignore invalid packets. For valid packets add the value of the fifth byte as an unsigned value to a running total and print the total so far. If the running total is greater than 255 then exit.

Signals

Give the names of two signals that are normally generated by the kernel

Give the name of a signal that can not be caught by a signal

Why is it unsafe to call any function (something that it is not signal handler safe) in a signal handler?

Coding Questions

Write brief code that uses SIGACTION and a SIGNALSET to create a SIGALRM handler.

14 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:

```
#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.

```c
#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.

Assorted Man Pages

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. 9

Circular Wait When a set of processes are waiting for resources held by other processes. 114

Coffman Conditions Four necissary and sufficient conditions for deadlock. 114

Deadlock When a system cannot progress. 113

Hold and Wait Once a process obtains a resource, the process cannot give that resource up. 114

Linux Kernel A widely used operating system kernel. 9

Livelock TODO. 115

Mutual Exclusion When two processes cannot have the same resource at the same time. 114

Ostrich Algorithm For a system not to care about deadlock, instead stick its head in the sand. 116

Portable Works on multiple operating systems or machines. 9

POSIX Portable Operating System Interface, a set of standard defined by IEEE for an operating system. 9

Pre-emption When a process is forced to give up its resource by another process. 114

RAG A tool for helping identify deadlock if a cycle is apprent. 113