



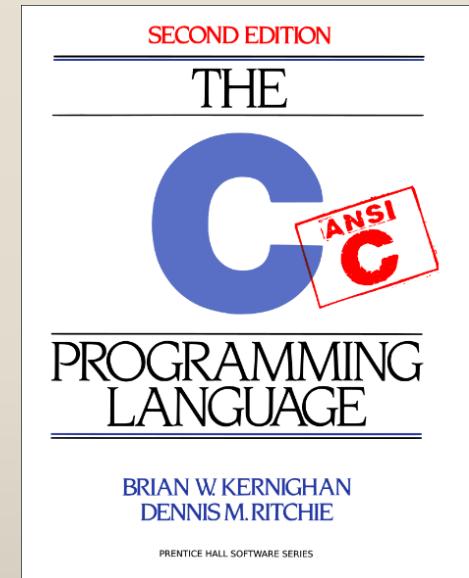
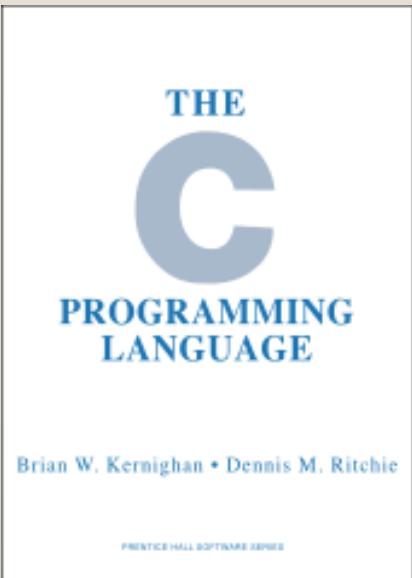
Intro to Operating Systems

The C Programming Language

The Mother Tongue

Some basic C capabilities:

- Structures and `typedef`
- Scope and extent
- Pointers
- Strings
- `cpp` – the C Preprocessor
 - conditional compilation
 - macros
- `stdio`, `printf`, `fgets`, and others.



Linux Programming in C

- Most programming on Linux is done in C (not C++, but plain C, **The Mother Tongue**).
- Most of the large programming assignments will be written in C (not C++, not Java, not Python).
 - You won't be able to use `cin` or `cout`. You will use `printf/fprintf` and `scanf/fscanf`.
 - You won't be able to use `new` or `delete`. You will use `malloc` and `free`.
- **You will also need to create a Makefile to build your C code for the labs.**



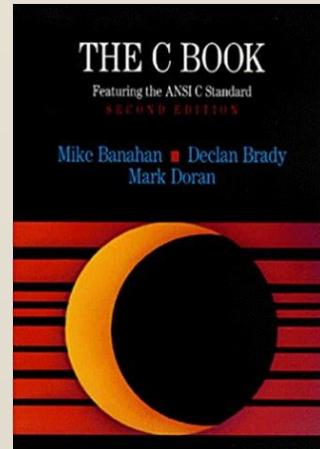
V I T A M I N

Linux Programming in C

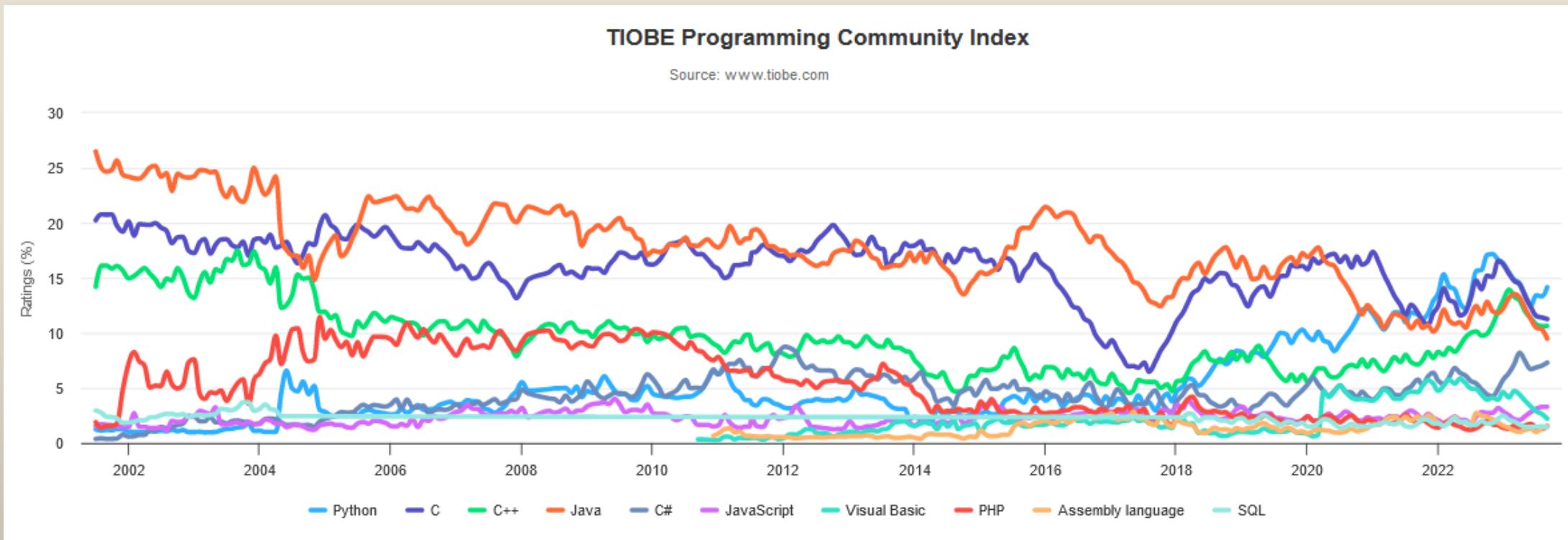
- C is a **systems programming** language, written to be able to manage system resources.
- By design, **C provides constructs that map efficiently to typical machine instructions**, and therefore it has found lasting use in applications that had formerly been coded in assembly language, **including operating systems**.
- It was **written by programmers for programmers**.
- C has been standardized by the American National Standards Institute (ANSI) since 1989.

Resources for C

- I've placed links to a number of resources about using C and using C on Linux in the Canvas site under Class Resources/Resources for C .
- **I highly recommend you give them a look.**
- One specific resource is the completely free online book **The C Book** (http://publications.gbdirect.co.uk/c_book/).
 - It is a little dated, but it contains most of everything you'll need for C programming in this class.
 - Of particular value is the page about the various formatting options for `printf()` and `scanf()` calls http://publications.gbdirect.co.uk/c_book/chapter9/formatted_io.html



This chart gives you an idea of how well the popularity of programming in C has held out over the years. C is the second most commonly used programming language.



As of September 25, 2023

<http://www.tiobe.com/tiobe-index/>

Mathematical Expressions

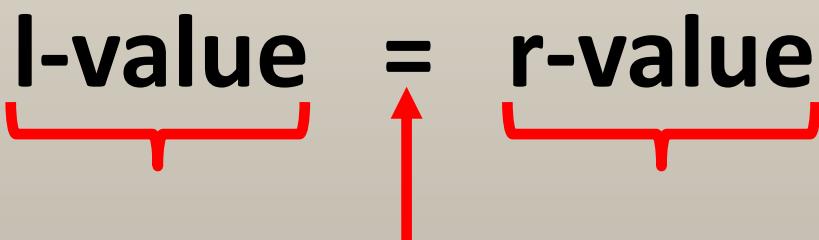
- ***l-value*** - refers to what can be placed on the **left of the assignment operator**

Constants and literals **cannot** be an l-value

- ***r-value*** - refers to what can be placed on the **right of the assignment operator**

Things like expressions and literals

l-value = **r-value**



assignment operator

```
int x = 0;  
x = ( 5 * 3 ) + 7;
```

C Language Operators

Operator	Type	Description	Example
+	Binary	Addition	a = b + 5;
-	Binary	Subtraction	a = b - 5;
-	Unary	Negation (changes sign of value)	a = -b;
*	Binary	Multiplication	a = b * 5;
/	Binary	Division	a = b / 5;
%	Binary	Modulus (integer remainder of dividing left operand by right operand)	a = b % 5;

The Modulus Operator

- Both operands of the **modulus** operator (`%`) **must** be integers.
- Modulus operator often causes confusion in beginning programmers.
- The integer remainder of dividing left operand by right operand.

Following results in value of 2

```
int var_a;  
var_a = 5 % 3;
```

The integer remainder of $5 \div 3$



The integer remainder of the left operator divided by the right operator.

```
int var_a;  
  
var_a = 5 % 3; // var_a equals 2  
  
var_a = 5 % 5; // var_a equals 0  
  
var_a = 5 % 2; // var_a equals 1  
  
var_a = 10 % 2; // var_a equals 0  
  
var_a = 10 % 3; // var_a equals 1  
  
var_a = 5 % 2; // var_a equals 1  
  
var_a = 8 % 2; // var_a equals 0
```

This is a common use of the modulus operator.
Even or odd?

Integer Division and Truncation

- When performing division, pay attention to the data types of the operands

- Examples:

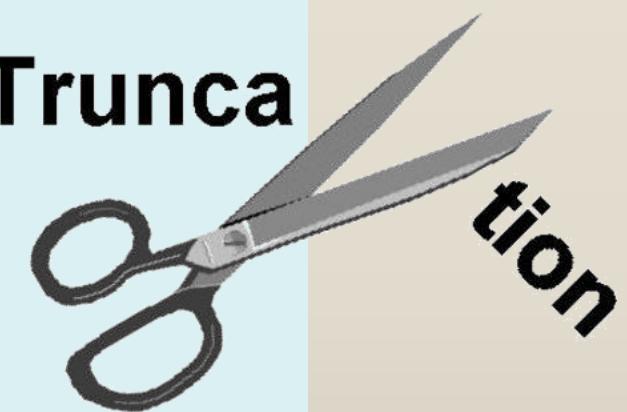
```
x = 5 / 9;
```

```
x = 5.0 / 9.0;
```

```
x = 5.0F / 9.0F;
```



Trunca



- First example - performs integer division resulting in a zero being stored in **x**.



PLEASE
NOTE

- ***Order of precedence*** - established order that must be adhered to when evaluating expressions with multiple operations.
- The table lists common operators starting with those that will be evaluated first.



Performed first

Highest
precedence

postfix ++, postfix --

prefix ++, prefix --, unary -

*, /, %

+, -

=, *=, /=, %=, +=, -=

Performed last

Lowest
precedence

When in doubt...

Parentheses are not endangered, use them.

((((((((((())))))))))

Relational Operator	Description
==	Equality (<i>be sure to use two equal signs</i>)
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

- Single equal sign (=) is an assignment.
- Double equal sign (==) tests for equality.

This is a very common mistake,
even for experienced developers.

Truth table - displays Boolean results produced when the operator is applied to specified operands.

Logical && and truth table	C1	C2	C1 && C2	C1 C2
true	true	true	true	true
true	false	false	false	true
false	true	false	false	true
false	false	false	false	false

Logical NOT truth table	Condition	! (Result)
true	false	
false	true	

Short-circuit Evaluation

- **Short-circuit evaluation** - once the outcome of condition can be determined, evaluation ends.
 - If you can determine the left operand to an `&&` operator is false, then short-circuit to false.
 - If you can determine that the left operand to a `||` operator is true, the short-circuit to true.

```
if ((b != 0) && ((a / b) > 100)) {  
    printf("Happy denominator\n");  
}  
else {  
    printf("Sad denominator\n");  
}
```

If `b` is equal to 0, the second half of the expression would cause an exception. Short-circuit evaluation prevents this.

The Ternary Operator

- The ***Conditional operator*** - considered a **ternary operator**, meaning it has three operands.

- Syntax:

```
<condition> ? <true expression> : <false expression>
```

Notice the question mark and the colon.

The ternary operator is basically an inline if-then-else statement.

The Ternary Operator

This if/then/else statement can be replaced with the single line expression below.

```
if(Expression1) {  
    variable = Expression2;  
}  
else {  
    variable = Expression3;  
}
```

```
variable = Expression1 ? Expression2 : Expression3;  
max_value = (n1 > n2) ? n1 : n2;  
printf("The maximum value is %d\n", (n1 > n2) ? n1 : n2);
```

Bitwise Operators

Operator	Description
&	Bitwise AND Operator copies a bit to the result if it exists in both operands. Note how this differs from the logical && operator.
	Bitwise OR Operator copies a bit if it exists in either operand. Note how this differs from the logical operator.
^	Bitwise XOR Operator copies the bit if it is set in one operand but not both.
~	Bitwise One's Complement Operator is unary and has the effect of flipping bits.
<<	Bitwise Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand.
>>	Bitwise Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.

Basic Data Types

Integral types

Real types

Pointer type

We will talk a LOT
more about this one
in another lecture.

Data Type	Num Bytes (on ada)	Num Bits
char	1	8
short	2	16
int	4	32
long	8	64
long long	8	64
float	4	32
double	8	64
long double	16	128
void *	8	64

The integral types can also be used with the **unsigned** qualifier:

unsigned short

unsigned int

...

The actual number of bytes (and therefore number of bits) for the types is not specified in the standard.

These are the sizes on the ada server.

Fixed-width Integer Types

Because the size of the integral data types was not part of the standard, an additional set of types was developed that are of known fixed size. These are defined in the include file stdint.h.

Data Type	Num Bytes	Num Bits
int8_t	1	8
int16_t	2	16
int32_t	4	32
int64_t	8	64
uint8_t	1	8
uint16_t	2	16
uint32_t	4	32
uint64_t	8	64

unsigned



C 2-Dimensional Arrays

In C, a 2 dimensional array is actually implemented as an **array of arrays**.

```
int a[2][4];
```

A 2 dimensional array of integers, with 2 rows,
each of which has 4 columns.

```
char c[12][20];
```

A 2 dimensional array of characters, with 12 rows,
each of which has 20 columns.

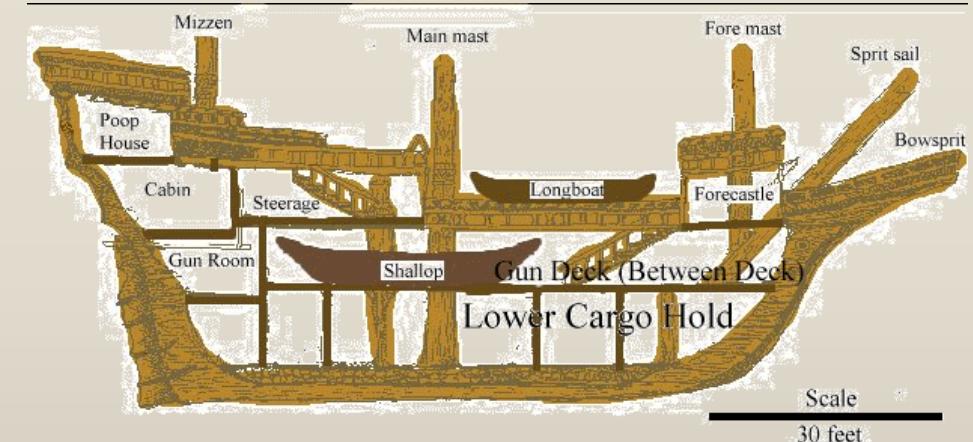
```
float f[6][7];
```

A 2 dimensional array of floating point values, with
6 rows, each of which has 7 columns.

C Structures

This declares a structure called account_s. The account_s structure contains 4 members: account_number, first_name, last_name, and balance. In this example, the type of each member is a primitive type (int, char, or float).

```
struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
};
```



If you want to make use of the account_s structure in your code, you must declare a variable like this:

```
struct account_s personal_account;
```

The struct keyword must be used when declaring a variable from a structure.

C Structures

```
struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
};  
  
struct account_s personal_account;
```



If you want to access a member of the `account_s` structure, you need to prefix the member name with the structure variable name and use a dot before the member name.

```
personal_account.account_number = 309026;
```

The name of the variable we declared in of the `struct`.

The dot before the member name.

C Structures and `typedef`

```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;
```



If you want to skip the need to prefix the `struct` keyword when you declare a variable, you can `typedef` the structure to give it a new name. Now, you can declare a variable to be of type `account_t` as:

```
account_t personal_account;
```

```
personal_account.account_number = 309026;
```

You access the variable members the same way.

As my own standard, I tend to use a `_s` as an ending on a structure and an ending of `_t` on a `typedef`.

C Structures are NOT Classes

C structures are not classes.

- They **do not have** constructors.
- They **do not have** ante-constructors (aka destructors).
- They **do not have** methods.
- They **do not have** friends.
- They **do have** data members, all of which are public.
- They **can be declared** to contain function pointers, but the function will not have a `this` pointer.



C Structures Memory Layout

```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;
```

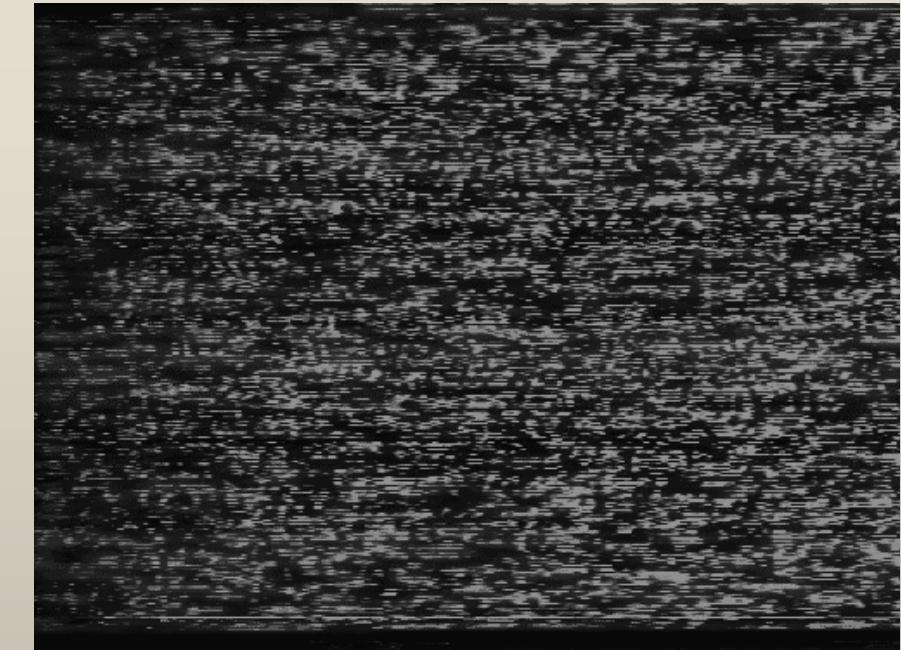
The memory for a C struct/typedef **will be laid out in the order that the members were specified in the struct definition.**

For the account_s structure, the memory for the account_number member will always be before the memory for the first_name member.

And, the memory for the first_name member will always be before the memory for the last_name member.

Some static Please

- In C, the reserved word ‘static’ has two, quite separate uses.
- The differences between the uses is often not well understood.
- In describing the `static` keyword, we first review the concept of scope.
- There are the concepts of **scope** and **extent**.
 - Scope refers to the ***visibility*** of variable or function.
 - Extent refers to the ***lifetime*** of a variable.



Some static Please

- The variable **num_occurrences** is defined outside the scope of any function (outside any braces).
- Its scope (visibility) is global to the entire module (compilation unit or file).
- If there are other modules that are part of the program, they cannot “see” the **num_occurrences** variable. It has file/module scope.
- The lifetime (extent) of **num_occurrences** is the life of the program.
- It will be initialized once, at the start of the program.
- As a global variable, the value in the **num_occurrences** will persist for the lifetime of the program.
- The **num_occurrences** will be stored in the data segment.

```
static int num_occurrences = 0;

int main()
{
    int value = 0;
    return 0;
}
```

- Contrast that to the variable **value**, which is local only to **main**.
- The **value** variable will be stored on the stack and will be deallocated from the stack when the function returns.

Some static Please

- The variable **num_occurrences** is defined locally within the function.
- Its scope (visibility) is strictly within the function.
- Its lifetime (extent) is the life of the program.
- It does not get recreated, on the stack, each time the function is called.
- It will be initialized once, at the start of the program.
- The value in the **num_occurrences** will persist beyond the lifetime of an individual invocation of the function.
- The **num_occurrences** will be stored in the data segment.

```
int count_occurrences(int num_to_add)
{
    static int num_occurrences = 0;
    int value = 0;

    num_occurrences += num_to_add;

    return num_occurrences;
}
```

- Contrast that to the variable **value**, which will be created and initialized each time the function is called.
- The **value** variable will be stored on the stack and will be deallocated from the stack when the function returns.

Some static Please

- The function `bla1` has **external linkage**. Its name is visible to functions both within and outside the module (compilation unit or file).
- The function `bla2` has **internal linkage**. Its name is only visible from within the module where it is defined.
- The scope of the `bla1` function is global to the program.
- The scope of the `bla2` function is local to the module.
 - In C++ terms, we can say that the `bla2` function is private to the file.

The extent (lifetime) of both functions is the lifetime of the program.

```
int bla1(int num_to_add)
{
    static int value = 0;
    value += num_to_add;
    return value;
}
static int bla2(int num_to_add)
{
    static int value = 0;
    value += num_to_add;
    return value;
}
```

C Pointers



C Pointers



```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;
```

Now we'll declare 2 variables:

```
account_t personal_account;  
account_t *personal_account_ptr = &personal_account;
```

Declare a **pointer** to something of type `account_t`.

Let's start off with the previous simple structure `typedef`.

This is how you get the address of a variable. The address of the variable is then a pointer back to that variable.
You will do this in this class.



Initialize it to be the address of the other variable.

C Pointers

```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;
```



Now assign a value to the `account_number` member, we can use the pointer syntax to access the member.

```
personal_account_ptr->account_number = 309026;
```

The pointer variable we declared in the previous slide.

You could use the other, **more cumbersome**, syntax of

```
*personal_account_ptr.account = 309026;
```

But, why? Which has a higher precedence, the `.` or the `*`

Strings in C

Things you may **think** you want to do:

```
char *str1;  
char str2[25] = "hello ";  
char str3[25] = "world";
```

```
str1 = "this is silly";  
str3 = str2;  
str2 = str2 + str3;
```

```
if ( str2 == str3 ) {  
    ...  
}
```

What will the result of
this comparison be?

NoNo Bad dog!

Why are these wrong?



In C++ terms, these are cStrings,
not the C++ string class.

Strings in C

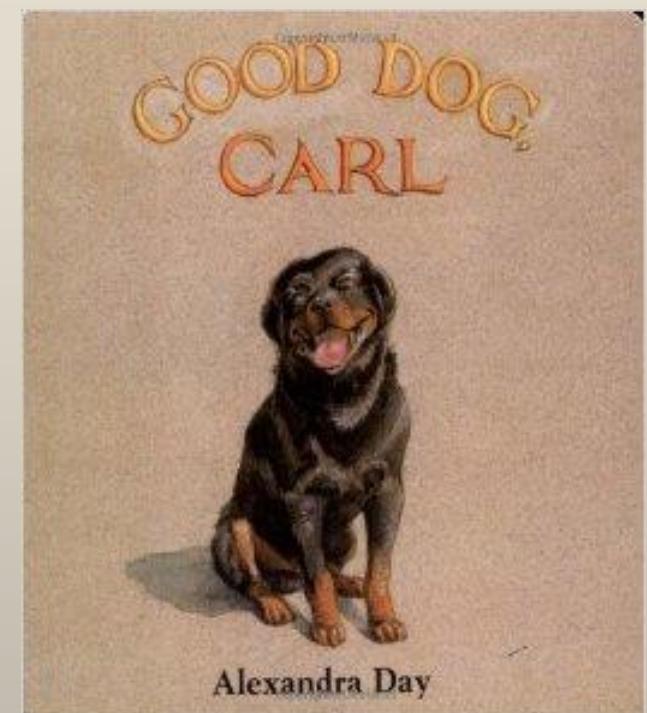
Things you **will** do:

```
char *str1;
char str2[25] = "hello ";
char str3[] = "world";
str1 = strup("this is silly");
strcpy(str3, str2);
strcat(str2, str3);

if (strcmp(str2, str3)) {
    ...
}
```

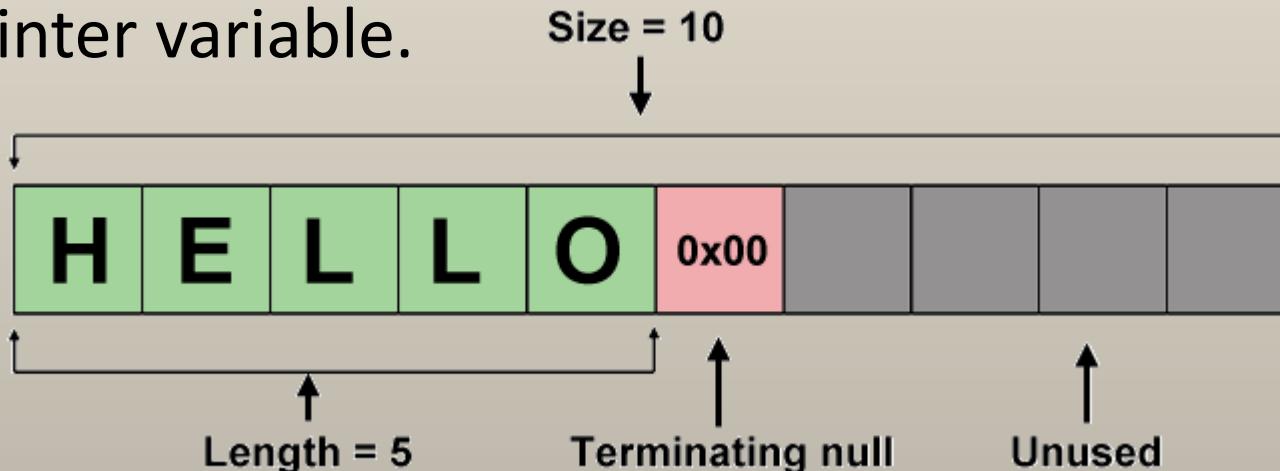
We'll cover more about
`strup()` in another lecture.

What are the return values
for `strcmp()`?



Strings in C

- In C, a string is a null-terminated array of bytes of type `char`, including the terminating null byte.
- String-valued variables are often declared to be **pointers** of type `char *`.
- Such pointer variables do not include space for the text of a string; that has to be stored somewhere else.
- It is up to you to store the address of the chosen memory space into the pointer variable.



Strings in C

- When creating a **string literal** in C, it must be within double quote marks.

"this is a string literal"

"A"

""

These are valid C string literals.

- If you want to create a **single character literal**, you can use single quotes, but it is not a string. It is a single `char` literal.

'A'

// This is a character literal, not a string literal.

Single quotes means single character.

Double quotes can be multiple characters.

The C String Functions

A partial list of the C String Functions:

- `strcat ()`
- `strnat ()`
- `strcmp ()`
- `strncmp ()`
- `strcpy ()`
- `strncpy ()`
- `strlen ()`

Notice that several of these have the variant call with the n.



https://en.wikibooks.org/wiki/C_Programming/Strings

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

```
char *strcat(char *dest, const char *src);
```



Append from src into dest.



```
char *strncat(char *dest, const char *src, size_t n);
```

- The `strcat()` function **appends** the `src` string to the `dest` string, overwriting the terminating null byte ('\0') at the **end of dest**, and then adds a terminating null byte.
- If `dest` is not large enough, program behavior is unpredictable.
- The `strnccat()` function is similar, except that it will use at most `n` bytes from `src`.

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

These do not return
true or false.

```
int strcmp(const char *s1, const char *s2);
```

```
int strncmp(const char *s1, const char *s2, size_t n);
```

- The `strcmp()` function compares the two strings `s1` and `s2`.
- It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`.
- The `strncmp()` function is similar, except it compares only the first (at most) `n` bytes of `s1` and `s2`.

int strcmp(s1, s2);	Return value	
s1 < s2	< 0	Notice that this is not equal to -1, rather less than 0.
s1 == s2	0	
s1 > s2	> 0	Notice that this is not equal to 1, rather greater than 0.

The `strcmp` function compares the string contents, not the pointer values.



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

```
char *strcpy(char *dest, const char *src);
```

```
char *strncpy(char *dest, const char *src, size_t n);
```

Copy from `src` into `dest`.
Think of the comma as an equal sign.



- The `strcpy()` function copies the string pointed to by `src`, including the terminating null byte ('\0'), to the buffer pointed to by `dest`.
- The destination string `dest` must be large enough to receive the copy.
- The `strncpy()` function is similar, except that at most `n` bytes of `src` are copied.
 - If there is no null byte among the first `n` bytes of `src`, the string placed in `dest` will not be null-terminated.

```
#include <string.h>  
  
size_t strlen(const char *s);
```



The `strlen()` function calculates the length of the string pointed to by `s`, excluding the terminating null byte ('\0').

The `strlen()` function returns the number of characters in the string pointed to by `s`.

If you call `strlen()` on something that is not NULL terminated, you will get unpredictable results.

Here is a test question you will see:
Does `strlen()` include the NULL character in the length of a string?

Strings in C

A closing note about chars and strings in C.

Just because something is an array of `char` (or a pointer to `char`), it does not mean that it is a string.

If the `char` array is not NULL terminated, it is not a string, it is just an array of characters.

Not all character data are strings.

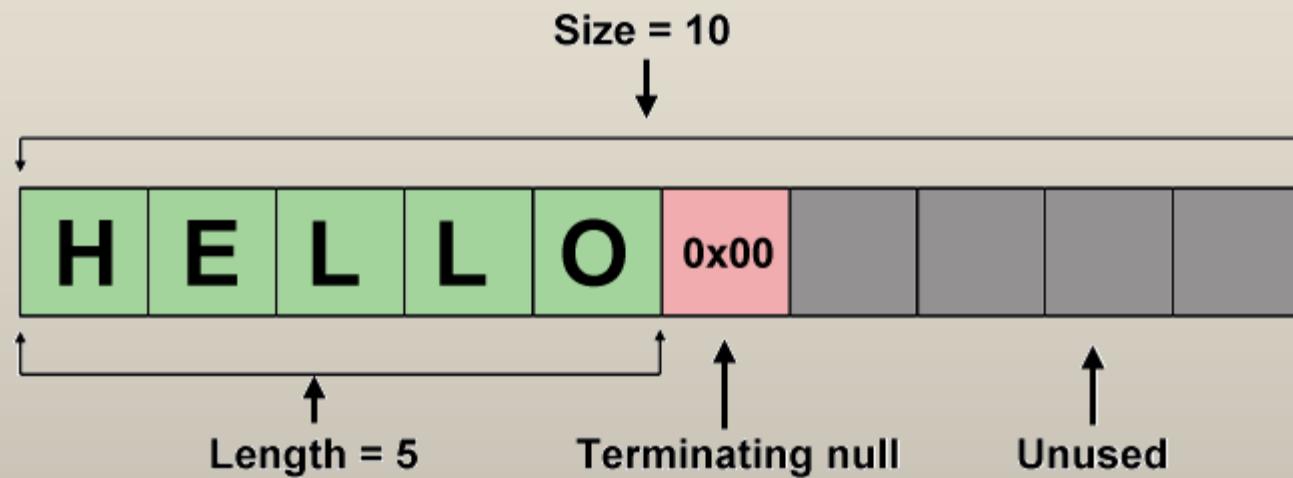
Don't get tied up in knots over this.



Strings in C

Here is a test question you will see:

In C, what is the difference between an array of `char` and a string?



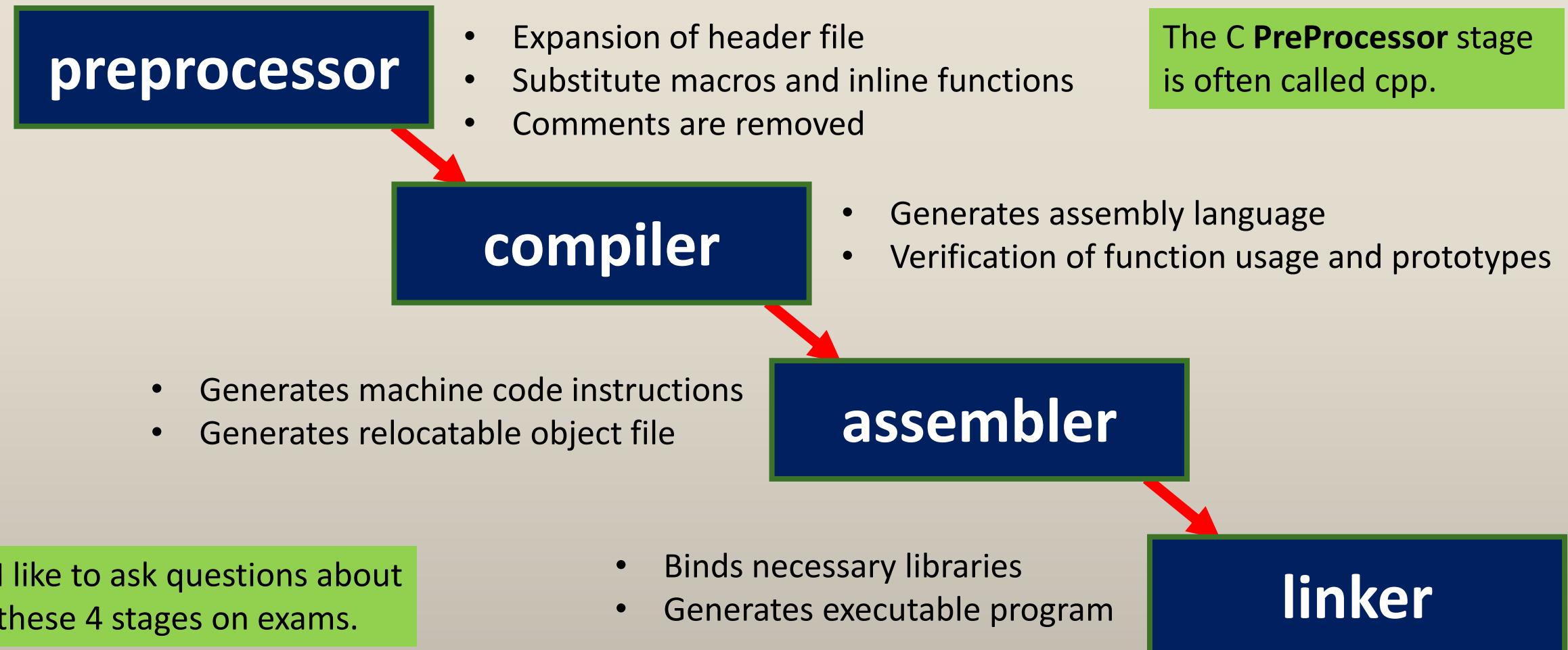


continue;

```
#include <stdio.h>

int main()
{
    printf("Hello World");
    return 0;
}
```

The 4 Stages of Compilation of a C Program





Vivaldi

The Four Seasons

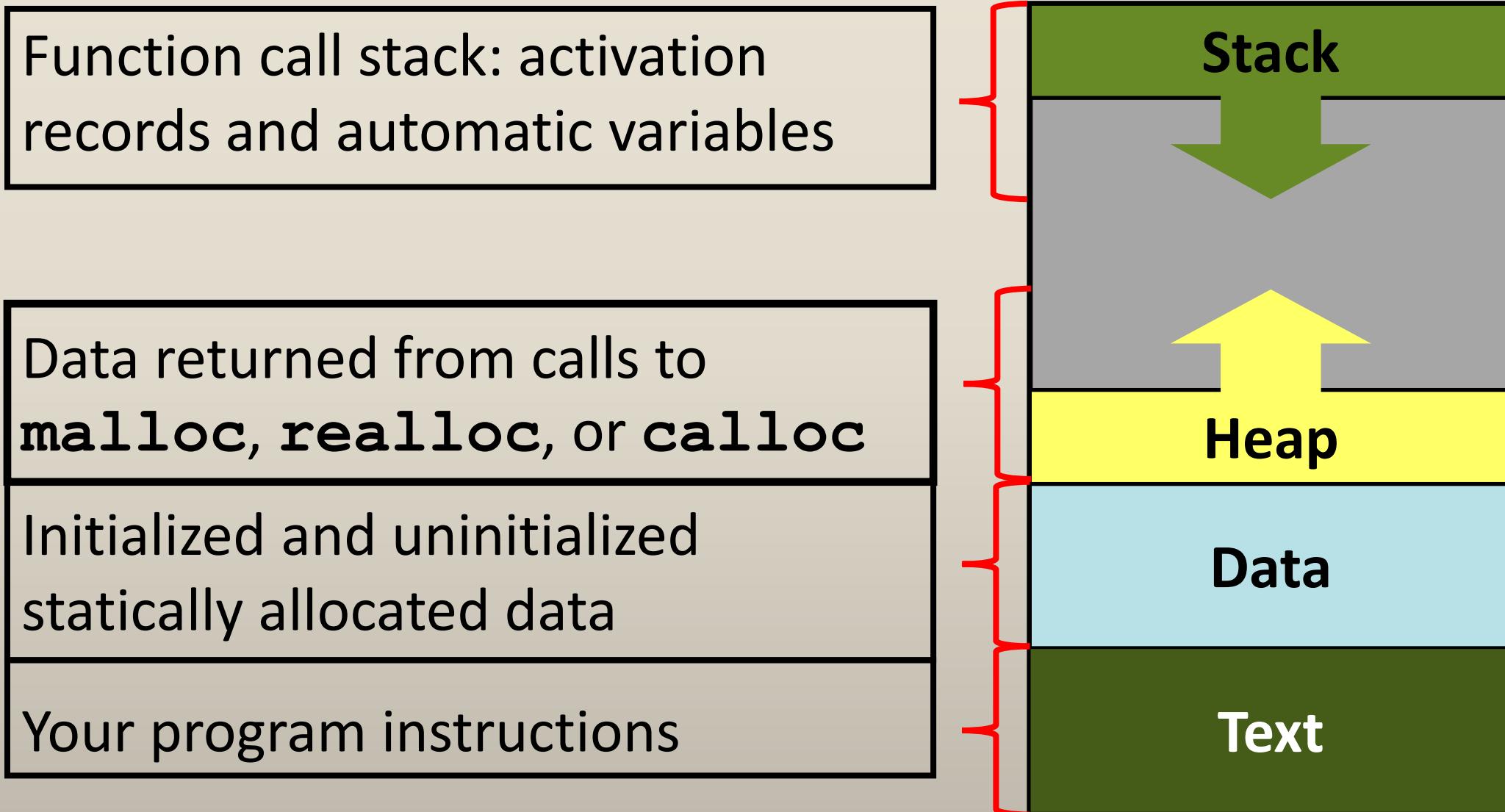


R. Jesse Chaney



o Op Sys

Memory Layout of a Process



The C Preprocessor – cpp

- Lines starting with a **#** character are interpreted by the C preprocessor as **preprocessor directives**.
- The C preprocessor is so much more than just header file inclusion.
- It performs macro expansion and conditional compilation.

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

If your use of cpp is limited to this, you are really missing out on a powerful feature of the C compiler.



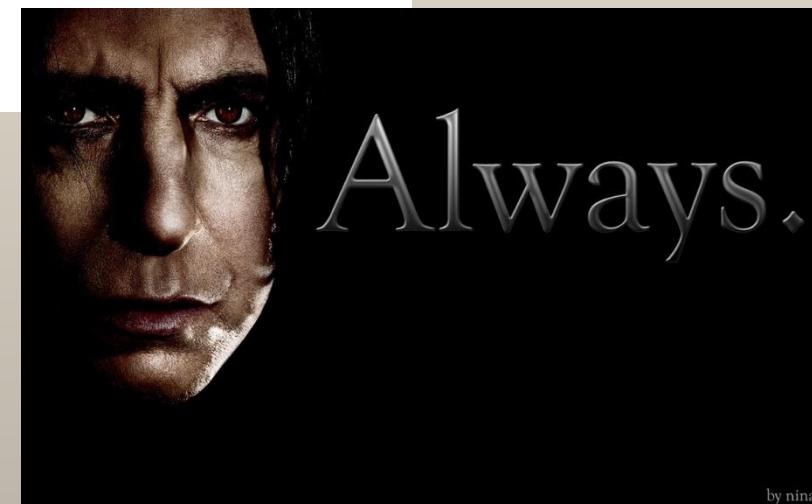
About Those Include Files

Notice the .h on the end of the #include preprocessor directive.

```
#include <stdio.h>
#include "my_include.h"
int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

System level include files **ALWAYS PRECEDE** your include files and are surrounded by less-than/greater-than symbols.

Your include files (the ones you create for your development) **ALWAYS FOLLOW** the system include files and are surrounded by double quotes.



String Literal Concatenation

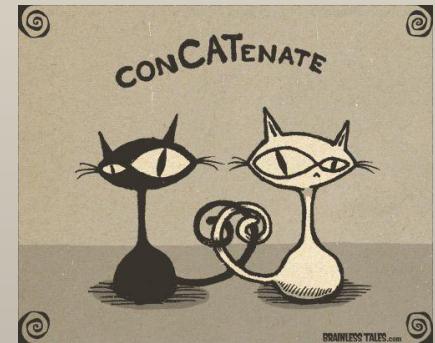
A minor function of the preprocessor is joining string literals together, "string literal concatenation" – automatically turning code like

```
printf("A long string " "with a longer string " "\n");
```

Into

String literals can be separated by spaces.

```
printf("A long string with a longer string \n");
```



Conditional Compilation

The use of `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` can make editing and debugging your code much easier.

```
#ifdef DEBUG
    fprintf(stderr, "debug message\n");
#endif // DEBUG
```

```
#if DEBUG >= 2
    fprintf(stderr, "debug message 2\n");
#endif // DEBUG > 2
```

```
#ifdef DEBUG
    fprintf(stderr, "debug message\n");
#else // not DEBUG
    fprintf(stderr, "happy message\n");
#endif // DEBUG
```

Use of these can really make your life a lot better. I highly recommend them.



Simple Macros

The use of simple macros in C can easily be compared to the use of `const`, and is in some ways not as good as using `const`.

However, in this class, we'll be using a lot of macros.

```
#define MY_NAME "R. Jesse Chaney" ← If you see this, DON'T do this!  
printf("My name is %s\n", "R. Jesse Chaney" );  
printf("My name is %s\n", MY_NAME );
```

Do the right/smart/proper/cool thing.
It's a macro, use it like one.



Almost Simple Macros

Here is one of my favorite uses of macros.

```
#ifdef NOISY_DEBUG
# define NOISY_DEBUG_PRINT fprintf(stderr, "%s %s %d\n" \
    , __FILE__ , __func__ , __LINE__ )
#else // not NOISY_DEBUG
# define NOISY_DEBUG_PRINT
#endif // NOISY_DEBUG
```

The line continuation character for macros.

Special macros defined by cpp or the compiler.

Define the macro to be nothing if NOISY_DEBUG is not defined.

I can sprinkle the macro NOISY_DEBUG_PRINT generously in my code. Any time I #define NOISY_DEBUG, I will see a trace of line numbers generated to stderr.

If I don't #define NOISY_DEBUG, no debugging messages are generated.

Send all *diagnostic* messages to stderr, not stdout.



Life is good®

Send all *diagnostic* messages to `stderr`, not `stdout`.

We will describe `stderr` in just a moment...

Macros Like Functions

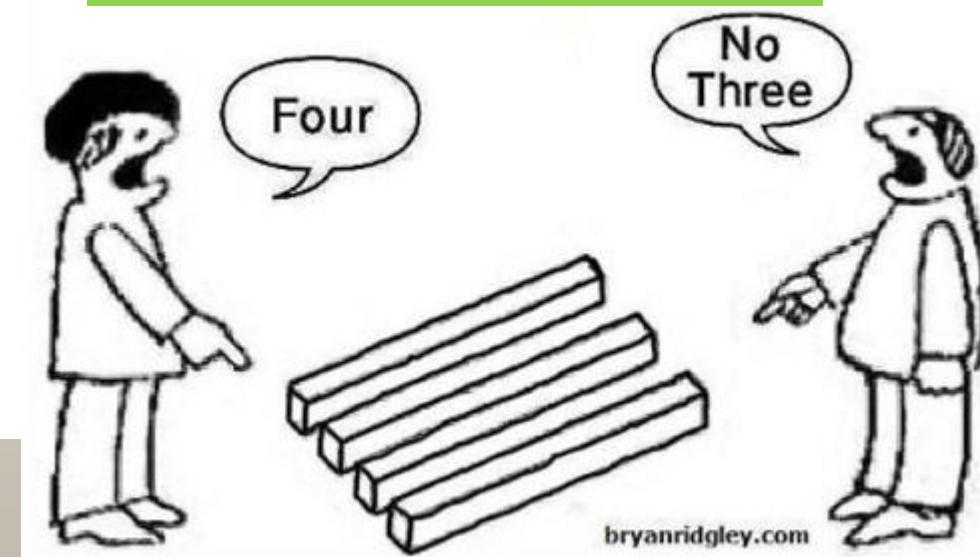
It is possible to define macros that **take arguments**. They will look like function calls when used. Here are a couple simple common ones.

```
#define ABSOLUTE_VALUE( x ) ( ((x) < 0) ? -(x) : (x) )
#define MIN(a,b) (((a) < (b)) ? (a) : (b))

int i1 = -7, i2 = 10;
float f1 = -7.0, f2 = 10.0;

i1 = ABSOLUTE_VALUE(i1);
f1 = ABSOLUTE_VALUE(f1);
i1 = MIN(i1,i2);
f1 = MIN(f1,f2);
```

Use LOTS of parentheses when you write macros like this.



The assert() Macro

```
#include <assert.h>

void assert(scalar expression);
```

If `expression` is false (compares equal to zero), `assert()` prints an error message to standard error and terminates the program by calling `abort(3)`.

- The assert macro provides a convenient way to abort the program while printing a message about where in the program the error was detected.
- You can disable the error checks performed by the `assert()` macro by recompiling with the macro `NDEBUG` defined.

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

```
int test assert( int x ) {
    assert( x <= 4 );
    return x;
}
```

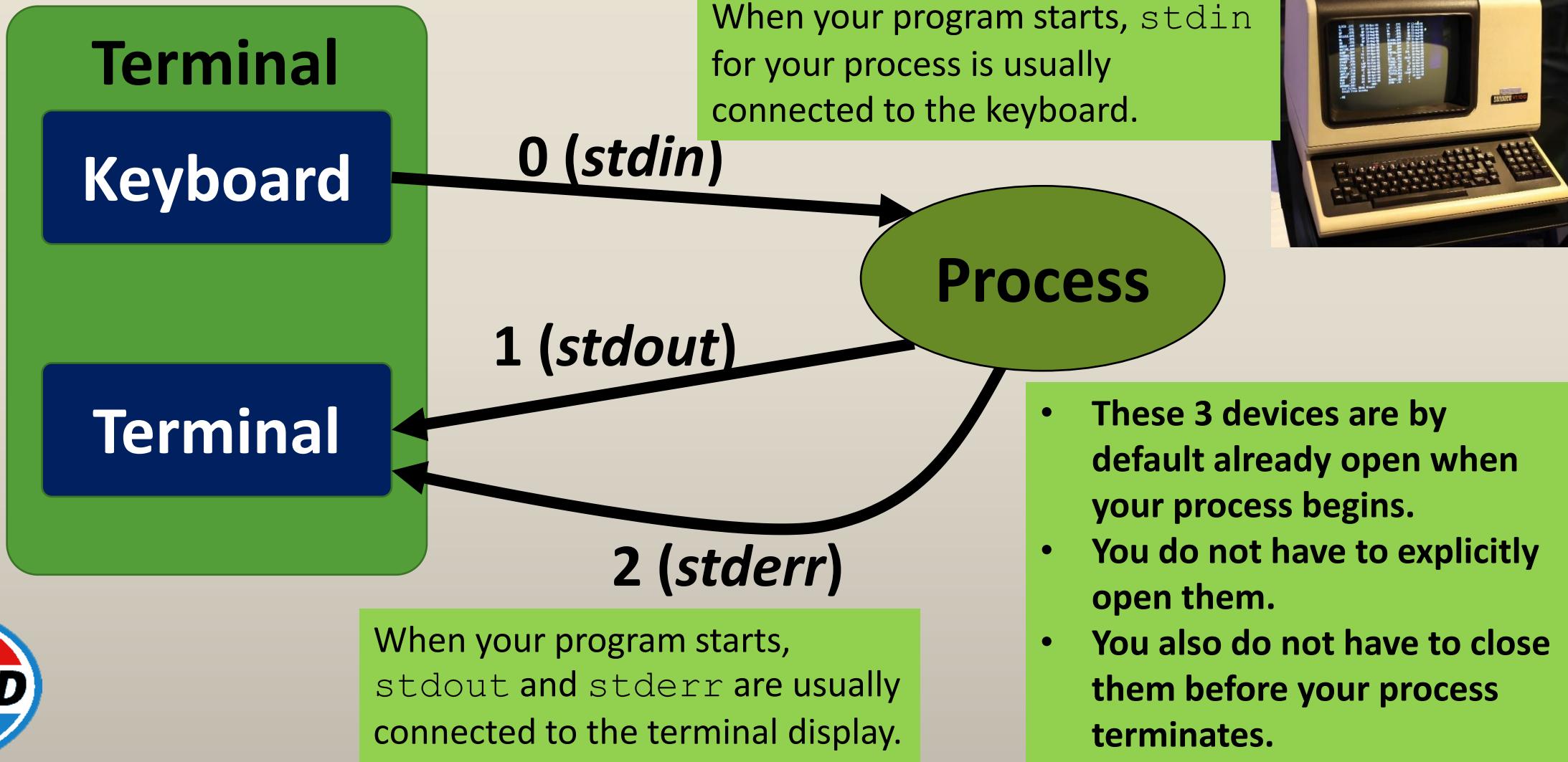
```
int main( ) {
    int i;
```

```
        for ( i = 0 ; i <= 9 ; i++ ) {
            test assert( i );
            printf( "i = %d\n", i );
        }
    return 0;
}
```

Use of the assert () macro. If the value of x is less than or equal to 4, nothing happens.

If the value of x is greater than 4, a message will be printed and the program will abort.

Standard I/O



The C stdio.h Header File

- In order to use the C input/output functions, you must include the stdio.h file.

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

- Do notice that you must include the .h portion of the header file name in the #include statement.
- **The C Book** has a nice description of this header file with some useful information about the general I/O mode in C.

http://publications.gbdirect.co.uk/c_book/chapter9/formatted_io.html



Standard I/O

FD Macro

0 `STDIN_FILENO`

1 `STDOUT_FILENO`

2 `STDERR_FILENO`

Stream

`stdin`

`stdout`

`stderr`

Device

keyboard

terminal

terminal

We'll talk about these
macros throughout the term.

FD stands for **file descriptor**.



Standard I/O



```
ada ~
rchaney # ps
  PID TTY      TIME CMD
791500 pts/3    00:00:00 bash
791538 pts/3    00:00:00 ps
ada ~
rchaney # 
```

When you `fprintf/write` something to `stderr`, it goes to the terminal display, by default.

- It can be **redirected** from the command line.

When you `read/scanf/getch` something from `stdin`, it comes from the keyboard, by default.

- It can be **redirected** from the command line.

When you `printf/write` something to `stdout`, it goes to the terminal display, by default.

- It can be **redirected** from the command line.

The 3 files descriptors, `stdin`, `stdout`, and `stderr`, are all open when your program starts. You do not have to open them. You **can** close or redirect them from within your program.

The C printf () Function

- A common statement you'll use to send text to the **terminal window** will be the `printf()` statement.
- The `printf()` statement **uses the already open `stdout` file stream**.
- While it is possible for `stdout` to not be open when your program starts, it is unlikely and takes a lot of effort (and reason).
- For your programs, you can assume `stdout` **is open** at the beginning of the program.
- It is also possible to **redirect** `stdout` to something other than the terminal display. We will cover this.



The C printf () Function

Syntax:

```
printf(<format_string>, <arg_list>);
```



- The `printf` function **requires a formatting string** for all variables you pass to the function.
- If formatting string contains any format specifiers an argument list must be supplied.
- There must be a matching argument for every format specifier in the format string.

The C printf() Function

printf function – for printing formatted output to the screen

- `printf("Hello World");`
- Supports use of **format specifiers** to determine the type of the data print.

Data Type	Format Specifier	Example
<code>int</code>	<code>%d or %i</code>	123
<code>int</code>	<code>%o</code>	unsigned octal value
<code>int</code>	<code>%x or %X</code>	unsigned hex value
<code>float</code>	<code>%f</code>	3.1400
<code>double</code>	<code>%lf</code>	12.4567878
<code>char</code>	<code>%c</code>	A
<code>string</code>	<code>%s</code>	Hello
<code>pointer</code>	<code>%p</code>	0x12345678

You **WANT** to remember these!!!



The C printf() Function

Requires the header file <stdio.h>

Print a variable:

```
#include <stdio.h>
int int_exp = 99;
...
printf( "%d\n", int_exp );
...
// Output
99
```

Notice the .h on the end of the #include preprocessor directive.

The variable being printed.

The format string.

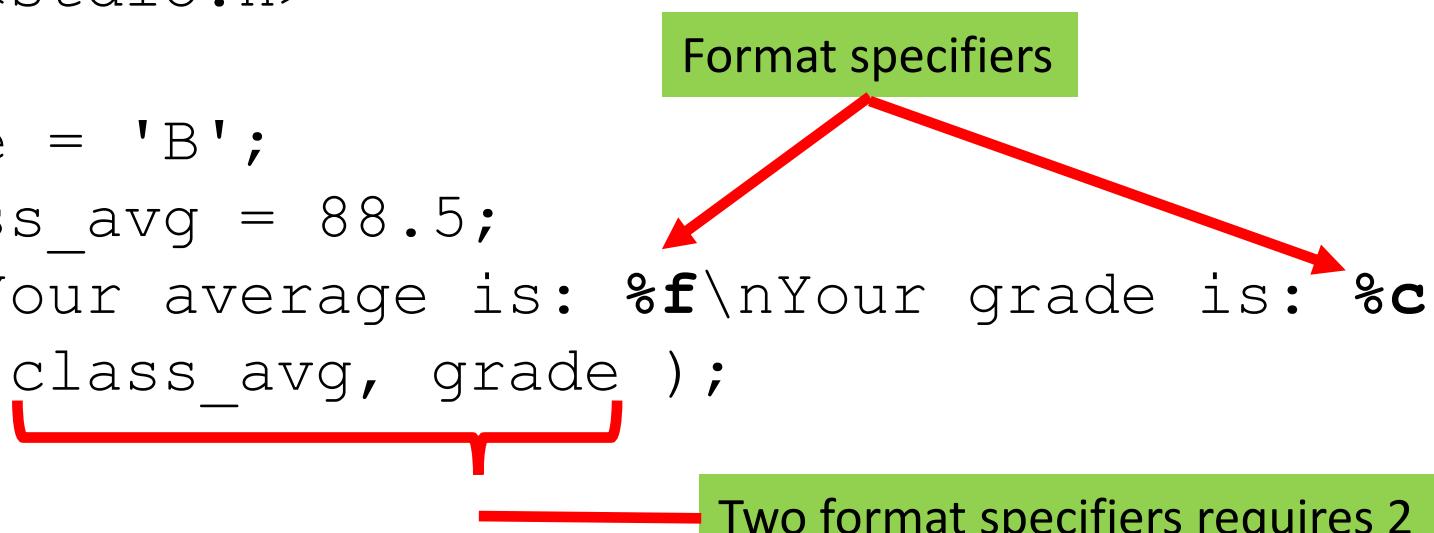
The format specifier for an integer data type.



The C printf() Function

Printing multiple pieces of data:

```
#include <stdio.h>
...
char grade = 'B';
float class_avg = 88.5;
printf( "Your average is: %f\nYour grade is: %c"
       , class_avg, grade );
...
// Output
Your average is: 88.500000
Your grade is: B
```



Format specifiers

Two format specifiers requires 2 pieces of data be passed.

FORMAT

Formatted Output with `printf()`

The general form for a format specifier is:

`%<flags><field width><precision><length>conversion`

Examples:

- `%5d` Print an integer with 5 total spaces, right justified
- `%6.2f` Print a decimal with 6 total places, including the decimal, with 2 places to the right of the decimal point.
- `%-25s` Print the string left justified with a total width of 25

Formatted Output with printf()

```
#include <stdio.h> // Needed for printf
int main( void )
{
    int int_exp = 123;
    float float_exp = 98.7653F;
    // Print an integer with 5 total spaces, right aligned
    printf( "%5d\n", int_exp );
    // Print a decimal with 6 total places including the decimal
    // with two places to the right
    printf( "%6.2f\n", float_exp );
    // Print the string literal left justified with a total width of 25
    printf( "%-25s\n", "Kevin" );
    // Print with 4 total spaces and two to the right of the decimal point
    printf( "%4.2f\n", .346 );
}
// Output
123
98.77
Kevin
0.35
```



The C `scanf()` Function

- `scanf()` – **reads from the keyboard** (or what ever is connected to `stdin`)
- Uses format specifiers previously discussed.
- Formatting string should **only** contain format specifiers and spaces.
- Each argument, **except for strings**, must be prefixed with an ampersand (&).
- The ampersand used in this context is called the **“address of”** operator.



The C `scanf()` Function

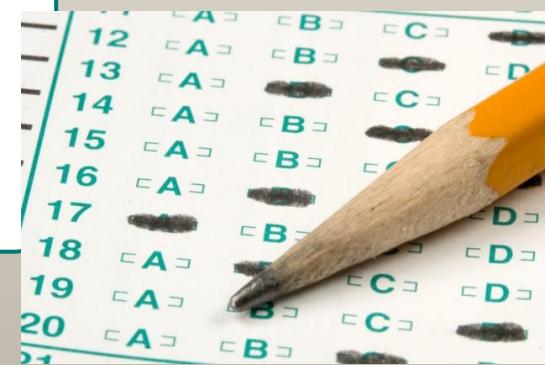
- Reading a value from the keyboard

```
int score = 0;  
  
scanf( "%d", &score ); // Don't forget the &
```

- Reading multiple values from the keyboard

```
int score1 = 0, score2 = 0;  
  
printf( "Enter two scores: " );  
scanf( "%d %d", &score1, &score2 ); // Place a space  
// between each  
// specifier
```

2 format specifiers requires 2 pieces of data be passed. Notice that each has an & in front of the variable name.



The C fopen () Function

- Sometimes you need to read from or write to more than just the keyboard or terminal. **You need to read and write files.**
- Then, you need to use the `fopen()` call to open the file, before you use it.
- The `fopen()` call returns a **FILE*** type.
- If `fopen()` fails, it returns a **NULL** pointer.

```
#include <stdio.h>
FILE *fopen(const char *pathname
            , const char *mode);
```

The name of the file to be opened, **as a string**.

Notice that mode is a `char *`, not just a `char`.



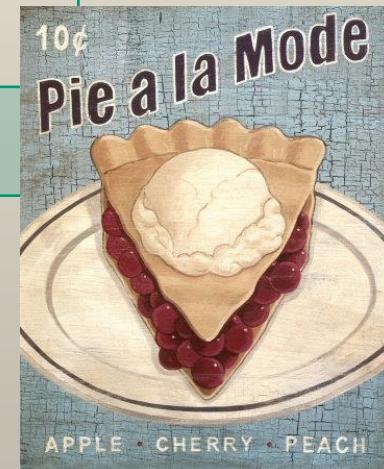
The C fopen () Function Modes

- The most common modes you'll use for the call to `fopen()` are listed in the table below.
- There are many other modes, especially for handling **binary files**.

Mode	Type of File	Read	Write	Create	Truncate
"r"	text	Yes	No	No	No
"w"	text	No	Yes	Yes	Yes
"a"	text	No	Yes	Yes	No

Notice the double quotes,
NOT single quotes.

For a file opened in append mode, **all** writes will occur at the end of the file, regardless of attempts to move the file position indicator with `fseek()`.



The C `fclose()` Function

- Of course, once you are done with a file (reading, writing, or both), you'll need to close it.
- Closing a file frees up important resources within the kernel.
- The kernel has a limited number of open files it can manage.
- Your process also has a limited number of open files it is allowed to have at any one time.
- **Don't pass a NULL pointer to `fclose()`.**

```
#include <stdio.h>
int fclose(FILE *stream);
```

The kind of thing returned
from `fopen()`.



The C fopen () Function

```
#include <stdio.h> Needed to access the fopen ()  
                           and fclose() calls.  
  
int main( void )  
{  
    FILE *fp;  
    char file_name[] = "file does not exist";  
  
    fp = fopen(file_name, "r");  
    if ( NULL == fp )  
    {  
        // file failed to open  
    }  
    else  
    {  
        // process file  
        fclose( fp );  
    }  
}
```

If fopen () fails, it returns a NULL pointer value.

Close the file when you
are done with it.

File I/O on Opened Files



- Once you've (successfully) opened the file, you'll need to perform operations on it.
- You can use the `fprintf()` and `fscanf()` calls.
- The `fprintf()` and `fscanf()` calls behave exactly like the `printf()` and `scanf()` calls, except that the first parameter to `fprintf()` and `fscanf()` is the open `FILE *` returned from a (successful) call to `fopen()`.

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

The kind of thing returned
from `fopen()`.

...

```
FILE *fp = NULL;  
int a, b, c;  
char str[50];  
fp = fopen(file_name, "r");  
if (NULL == fp)  
{  
    // file failed to open  
}  
else  
{  
    fscanf(fp, "%i %i %i %s", &a, &b, &c, str);  
    fclose(fp);  
}
```

Notice how the & is required for non-stringy variables and is not for the character array (aka string).

...

...

```
FILE *fp = NULL;  
int a = 1, b = 2, c = 3;  
fp = fopen(file_name, "w");  
if ( fp == NULL )  
{  
    // file failed to open  
}  
else  
{  
    fprintf(fp, "%06i %-7i %10i", a, b, c );  
    fclose( fp );  
}
```

What does the zero in front
of the width specifier do?

...

<http://www.cplusplus.com/reference/cstdio/printf/>

Other Functions for I/O

In addition to the `printf`, `fprintf`, `scanf`, and `fscanf` functions, there are a couple additional I/O functions that work with **strings**. These are useful when you know you will work with an entire line from the file (or keyboard). Lines from a file are delimited by a newline (`\n`) for `fgets()`.

```
fgets()           // get an entire line from a file
fputs()          // write the string to stream
```

The `fgets()` function can read from `stdin` and `fputs()` can write to `stdout`.



Let's Take Exceptions

A C++/Java/Python feature that you may miss while programming in C are exceptions.

Exceptions are a terrific language construct in C++/Java/Python to **manage anomalous conditions**.

In C, you will need to **check the return value** of the functions you call and determine if an error occurred and how to manage it.



The perror() Function

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

```
void perror(const char *s);
```

The perror() function is very useful.
Get used to using it.

The perror() function produces a message to **standard error** describing the last error encountered during a call to a system or library function.

...

```
perror( "Cannot open file " FILE_NAME );
```

Redirection of `stdin/stdout/stderr`

The shell (bash or other) and many UNIX commands take their input from standard input (`stdin`), write output to standard output (`stdout`), and write error output to standard error (`stderr`).

- By default, standard input is connected to the terminal keyboard and standard output and error to the terminal display.
- **The way of indicating an end-of-file on the standard input, a terminal, is usually `<Ctrl-d>`.**
- I've mentioned that you can redirect `stdin/stdout/stderr`.

Basic Redirection Operators

Character	Action	You will see these a LOT during the term.
>	Redirect standard output	
2>	Redirect standard error	
2>&1	Redirect standard error to standard output	
<	Redirect standard input	
	Pipe standard output to another command	
>>	Append to standard output	

You use these on the command line in the shell.

Some simple examples:

`$ who > names`

- Redirect standard output from the `who` command to a file named `names`. All `printf()` calls will automatically go into the `names` file.

`$ cat < file.txt`

- Redirect the file `file.txt` as the `stdin` to the `cat` command. All calls to `scanf()`/`gets()` come from the `file.txt` file.

`$ who | wc`

- The `stdout` from the `who` command is sent to the pipe (**the | character**) and is redirected as `stdin` for the `wc` command.

When used on UNIX/Linux command line, the vertical bar character is called a pipe.

```
1#include <stdio.h>
2
3#ifndef MAX_LINE_LEN
4#define MAX_LINE_LEN 1024
5#endif // MAX_LINE_LEN
6
7int
8main(void)
9{
10    char line[MAX_LINE_LEN] = {'\0'};
11    char *line_ptr = NULL;
12
13    while ((line_ptr = fgets(line, MAX_LINE_LEN, stdin)) != NULL) {
14        fputs(line, stdout);
15    }
16
17    return(0);
18}
```

Notice that there are no calls to open or close files.

This reads data from the **already open** `stdin` stream and writes that data to the **already open** `stdout` stream.

Read from the `stdin` stream.

Write to the `stdout` stream.

Can be found in `~rchaney/Classes/cs333/src/cat/my_cat1.c`

Examples how you can run my_cat1

```
./my_cat1 < passwd  
cat passwd | my_cat1
```

You can look at the source code.

You may also want to look at the source to my_cat2.c. The my_cat2.c program will allow you to have multiple files to cat on the command line.

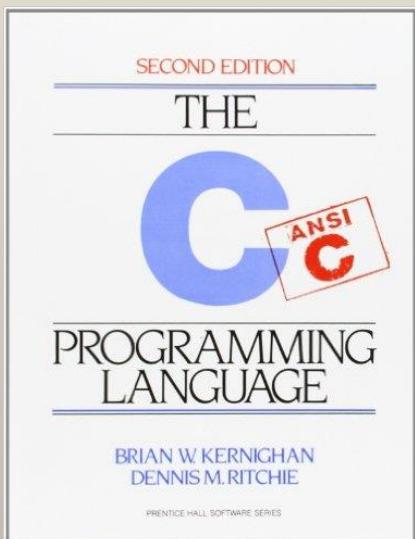
```
./my_cat2 passwd my_cat1.c my_cat2.c
```

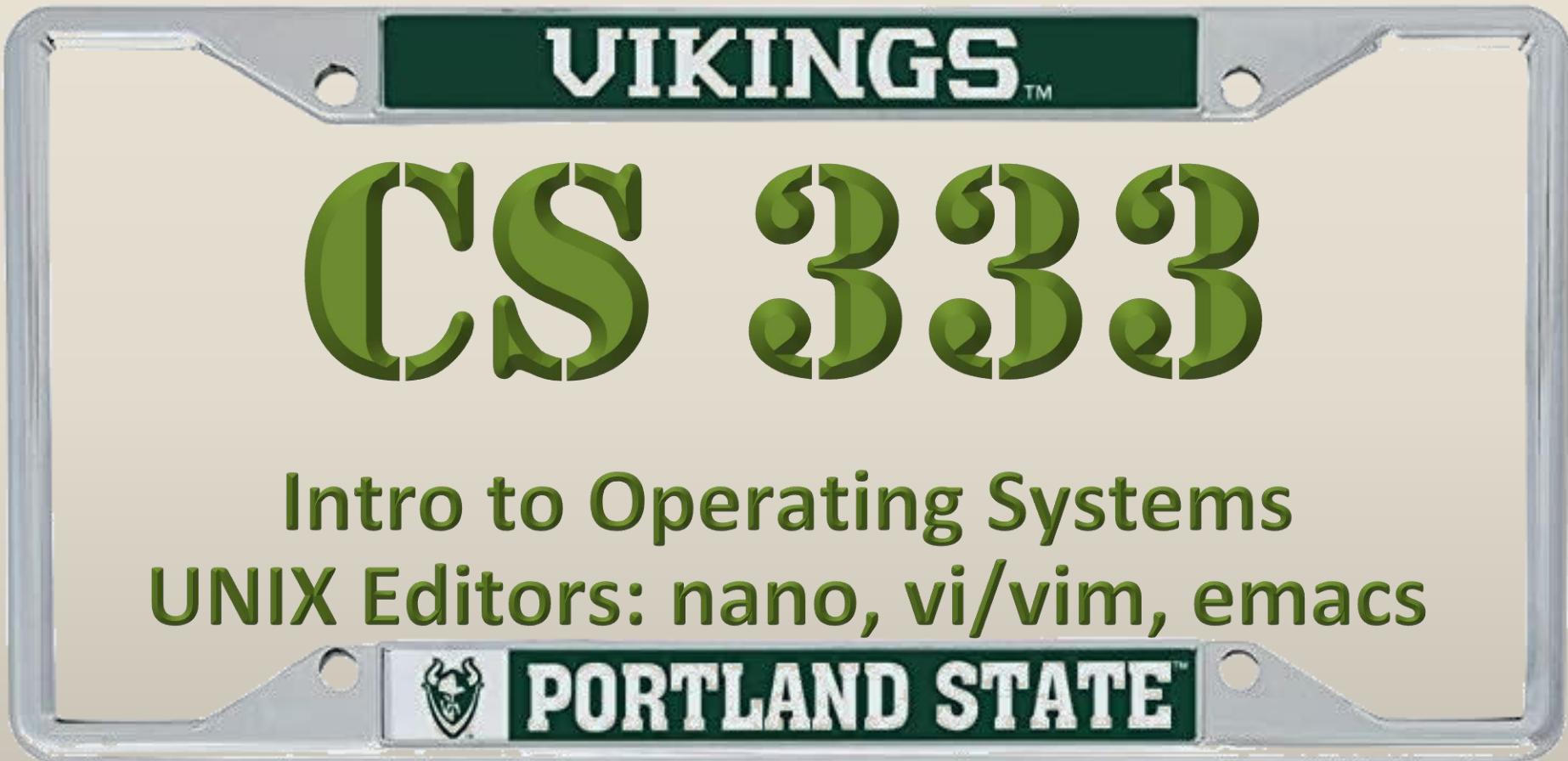


The C Programming Language

Some basic C capabilities:

- Structures and `typedef`
- Scope and extent
- Pointers
- Strings
- The C Preprocessor (aka `cpp`)
 - conditional compilation
 - macros
- `stdio`, `printf`, `fgets`, and buddies.





UNIX Editors:

- **nano**
- **vi/vim**
- **emacs**



I believe that it to your advantage to learn to use a native text based editor on Unix/Linux.

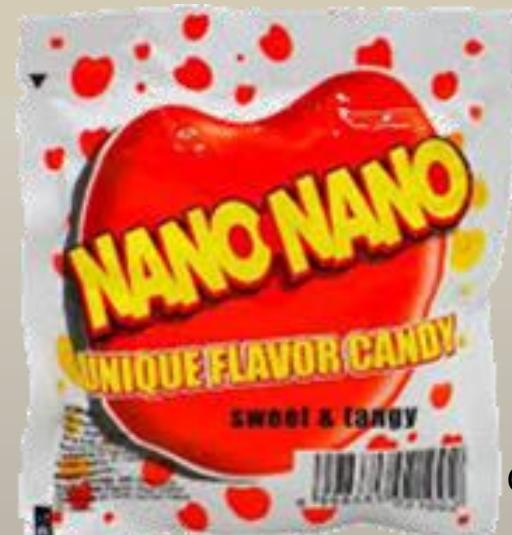
This contains a bit of information about 3 Linux editors:

- nano
- vi
- emacs

What it comes down to it, I really don't care what editor you use. I care about the code you write.

GNU nano

- GNU nano is a plain text editor for Unix-like computing systems or operating environments using a **command line interface**.
- Nano emulates the Pico text editor, part of the Pine email client, and provides additional functionality.
- GNU nano puts a two-line "shortcut bar" at the bottom of the screen, listing many of the commands available in the current context.



GNU nano 2.3.1 File: my_cat1.c

```
#include <stdio.h>

#ifndef MAX_LINE_LEN
# define MAX_LINE_LEN 1024
#endif // MAX_LINE_LEN

int main(int argc, char *argv[])
{
    char line[MAX_LINE_LEN];
    char *line_ptr;

    while ((line_ptr = fgets(line, MAX_LINE_LEN, stdin)) != NULL) {
        fputs(line, stdout);
    }
}
```

[Read 17 lines]

^G Get Help ^O WriteOut ^R Read Fil^Y Prev Pag^K Cut Text^C Cur Pos
^X Exit ^J Justify ^W Where Is^V Next Pag^U UnCut Te^T To Spell

GNU nano

If you want colorized code, as shown in the previous image, you can make a copy of my .nanorc file (from my home directory).

```
include /usr/share/nano/c.nanorc
```

```
include /usr/share/nano/makefile.nanorc
```

That's the end of the presentation on nano.

Though it is very basic, it does not require a lot from you.

There are files for other languages also available.

The Original Flame War

Hackles

By Drake Emko & Jen Brodzik



<http://huckles.org>

Copyright © 2003 Drake Emko & Jen Brodzik

https://en.wikipedia.org/wiki/Editor_war

vi/vim

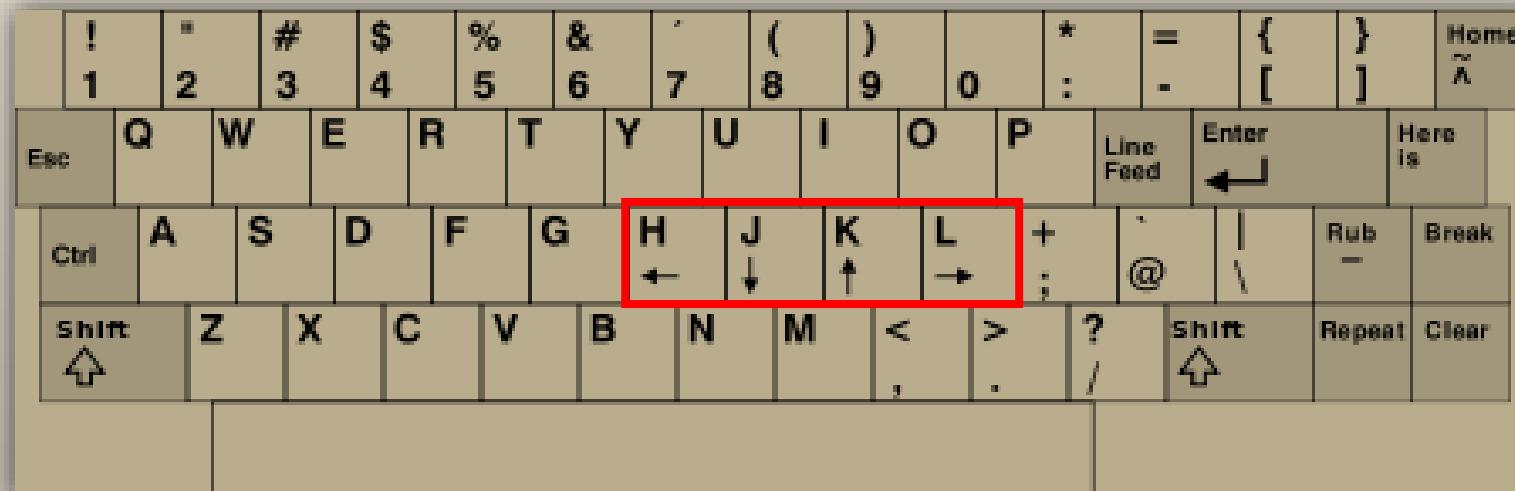
- "vi" is derived from the shortest unambiguous abbreviation for the `ex` command `visual` (`ex` is another UNIX editor)
- vim a contraction of Vi IMproved
- You will find vi in all UNIX/Linux implementations.
- Small and fast.

emacs

- **Editing MACroS**
- Non-modal interface.
- Extensible and customizable Lisp programming language variant.

- Regarding vi's modal nature, some emacs users *joke* that vi has two modes – **beep repeatedly and break everything**.
- vi users enjoy joking that emacs' key-sequences induce carpal tunnel syndrome, or mentioning one of many satirical expansions of the acronym EMACS, such as **Escape Meta Alt Control Shift** (a jab at emacs' reliance on modifier keys).
- As a poke at emacs' extensive programmability, vi advocates have been known to describe Emacs as "a great operating system, lacking only a decent editor".

- The original code for `vi` was written by Bill Joy in 1976, as the visual mode for a line editor called `ex` that Joy had written with Chuck Haley.
- `vi` was derived from a sequence of UNIX command line editors, starting with `ed`, which was a line editor designed to work well on teletypes, rather than display terminals.
- Joy developed the `vi` code on a ADM-3A terminal, which lacked many of the keys we now find common.



Bill Joy's words about vi and emacs:

*I think as mode-based editors go, it's (vi) pretty good.
One of the good things about EMACS, though, is its programmability and the modelessness. Those are two ideas which never occurred to me.*

<https://en.wikipedia.org/wiki/Vi>

More Joy (why vi doesn't have multiple windows, as emacs has):

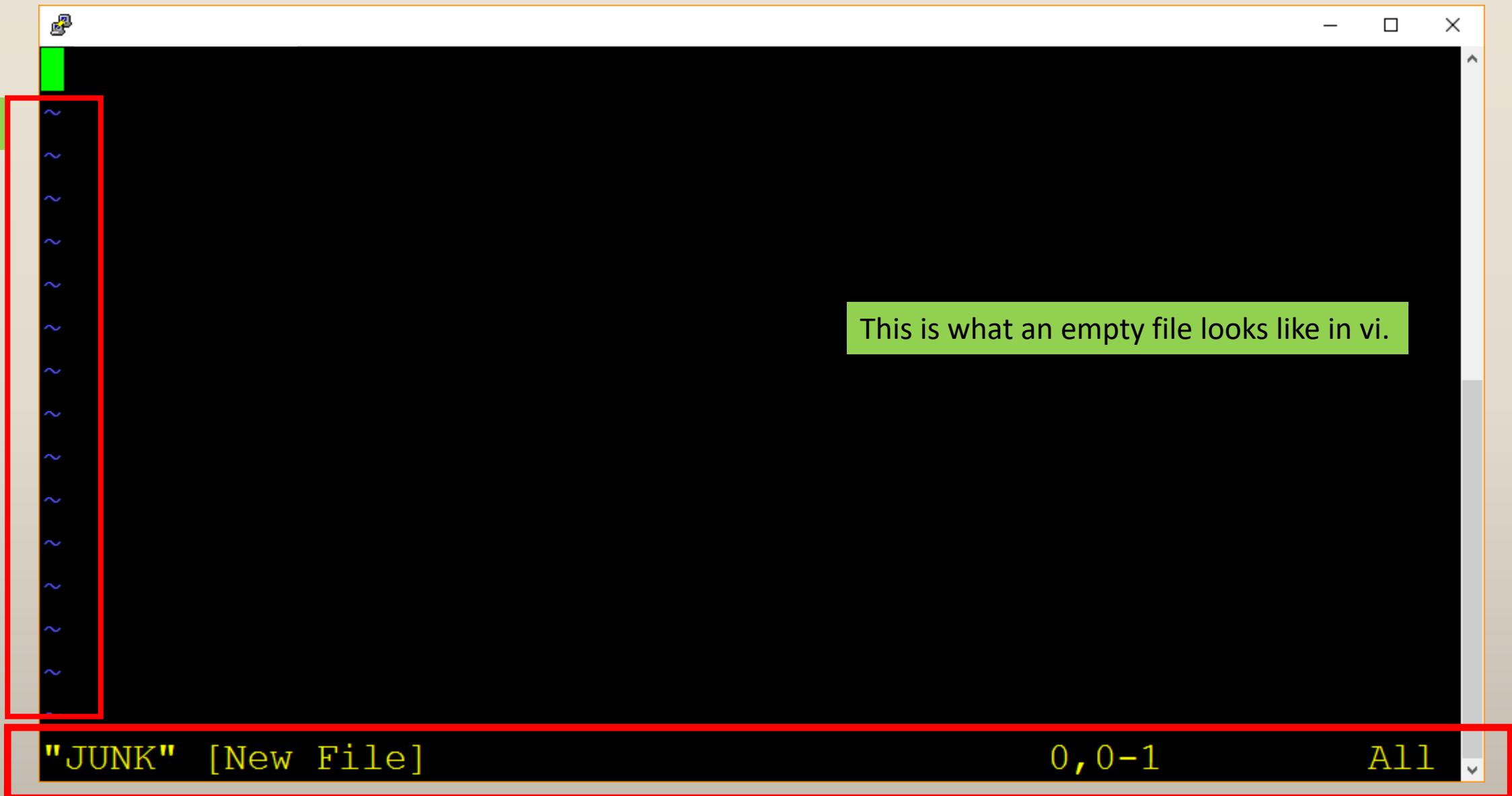
What actually happened was that I was in the process of adding multiwindows to vi when we installed our VAX, which would have been in December of '78. **We didn't have any backups and the tape drive broke.** I continued to work even without being able to do backups. And then **the source code got scrunched and I didn't have a complete listing.** I had **almost rewritten** all of the display code for windows, and that was when I gave up. After that, I went back to the previous version and just documented the code, finished the manual and closed it off. If that scrunch had not happened, vi would have multiple windows, and I might have put in some programmability – but I don't know.

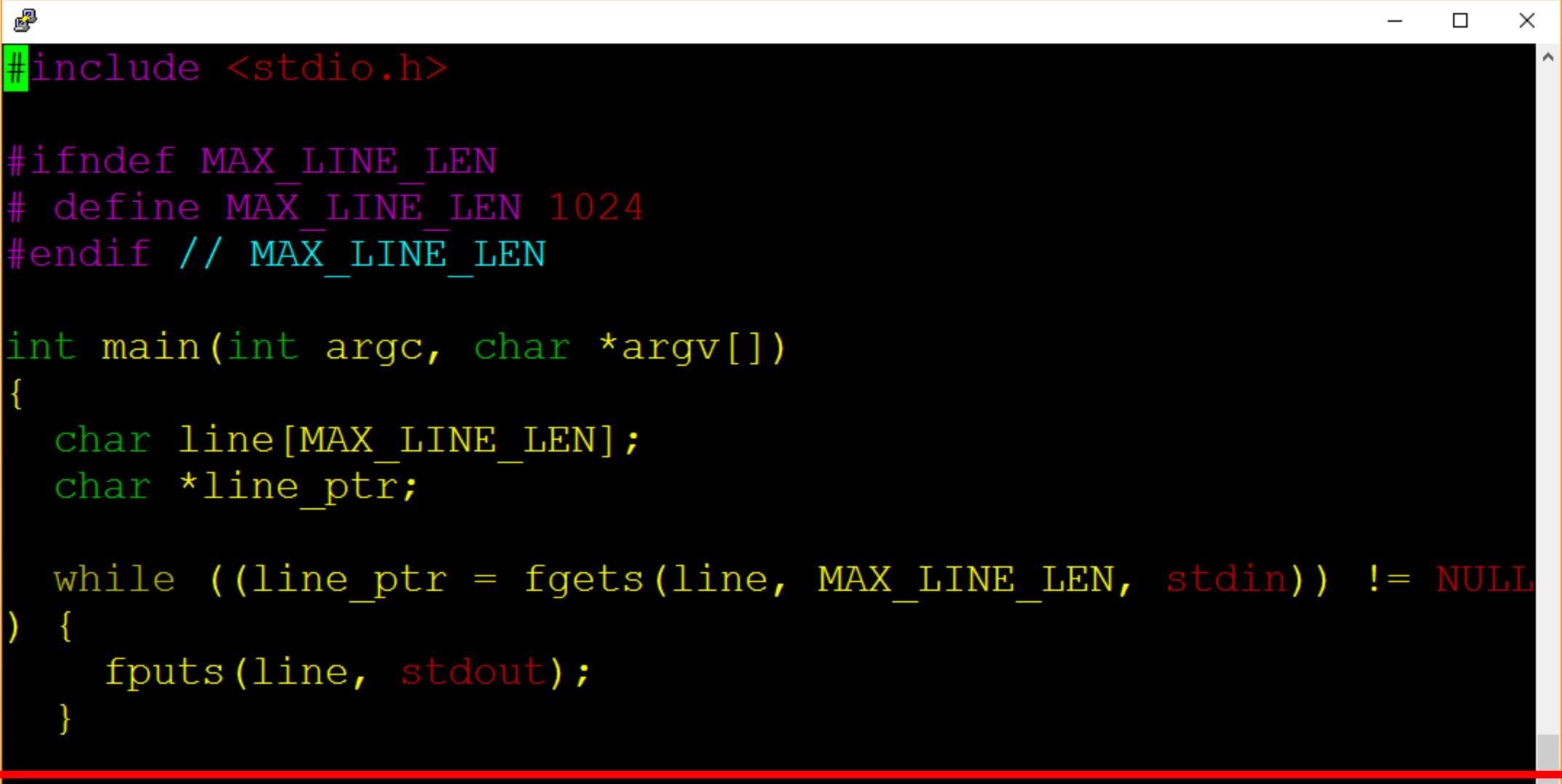
Starting vi

```
$ vi [filename]
```

```
$ vim [filename]
```

- If `filename` does not exist, a screen will appear with just a cursor at the top followed by tildes (~) in the first column.
- If `filename` does exist, the first few lines of the file will appear.
- The status line at the bottom of your screen shows error messages and provides information and feedback, including the name of the file.





```
#include <stdio.h>

#ifndef MAX_LINE_LEN
# define MAX_LINE_LEN 1024
#endif // MAX_LINE_LEN

int main(int argc, char *argv[])
{
    char line[MAX_LINE_LEN];
    char *line_ptr;

    while ((line_ptr = fgets(line, MAX_LINE_LEN, stdin)) != NULL)
    {
        fputs(line, stdout);
    }
}
```

"my_cat1.c" 17L, 284C 1,1 Top

vi Modes

Command Mode

- Command mode is the mode you are in when you start (default mode)
- Command mode is the mode in which commands are given to move around in the file, to make changes, and to leave the file
- Commands are case sensitive: j not the same as J
- Most commands do not appear on the screen as you type them. Some commands will appear on the last line: : / ?

vi Modes

Insert (or Text) Mode

- The mode in which text is created.
- You must press <Return> at the end of each line unless you've set wrap margin.
- There is more than one way to get into insert mode but only one way to leave: return to command mode by pressing the ESCAPE key, <Esc>

When in doubt about which mode you are in, start pounding on the <Esc> key.

Entering, Deleting, and Changing Text

From Command Mode

- i Enter text entry mode
- x Delete a character
- dd Delete a line
- r Replace a character
- R Overwrite text, press <esc> to end

I need a hat like this!



Eric was so proficient with the Unix text editor, he became known as the *vi-king*...

Exiting vi

To exit from vi, you must be in command mode

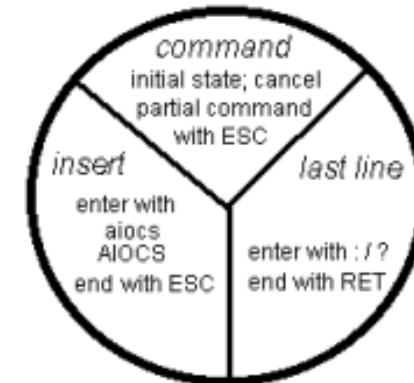
Press <Esc> if you are not in command mode.

You must press <Return> after commands that begin with a : (colon)

From Command Mode

- ZZ Write (if there were changes), then quit.
- :wq Write, then quit.
- :q Quit (will only work if file has not been changed).
- **:q!** **Quit without saving changes to file.**

vi states



CHANGE

cw word
cc line
C rest of line
s under cursor
S same as cc
r replace char

k line up
j line down
l right space
h left space
\$ end of line
G end of file
0 beginning of line

OTHER

u undo change
/ find down
? find up
. repeat
n next

DELETE

dw word
dd line
D rest of line
x under cursor
X before cursor
xp transpose

MOVE

scroll down ^D
word forward w
word backward b
end of word e
line n nG

vi commands

INSERT

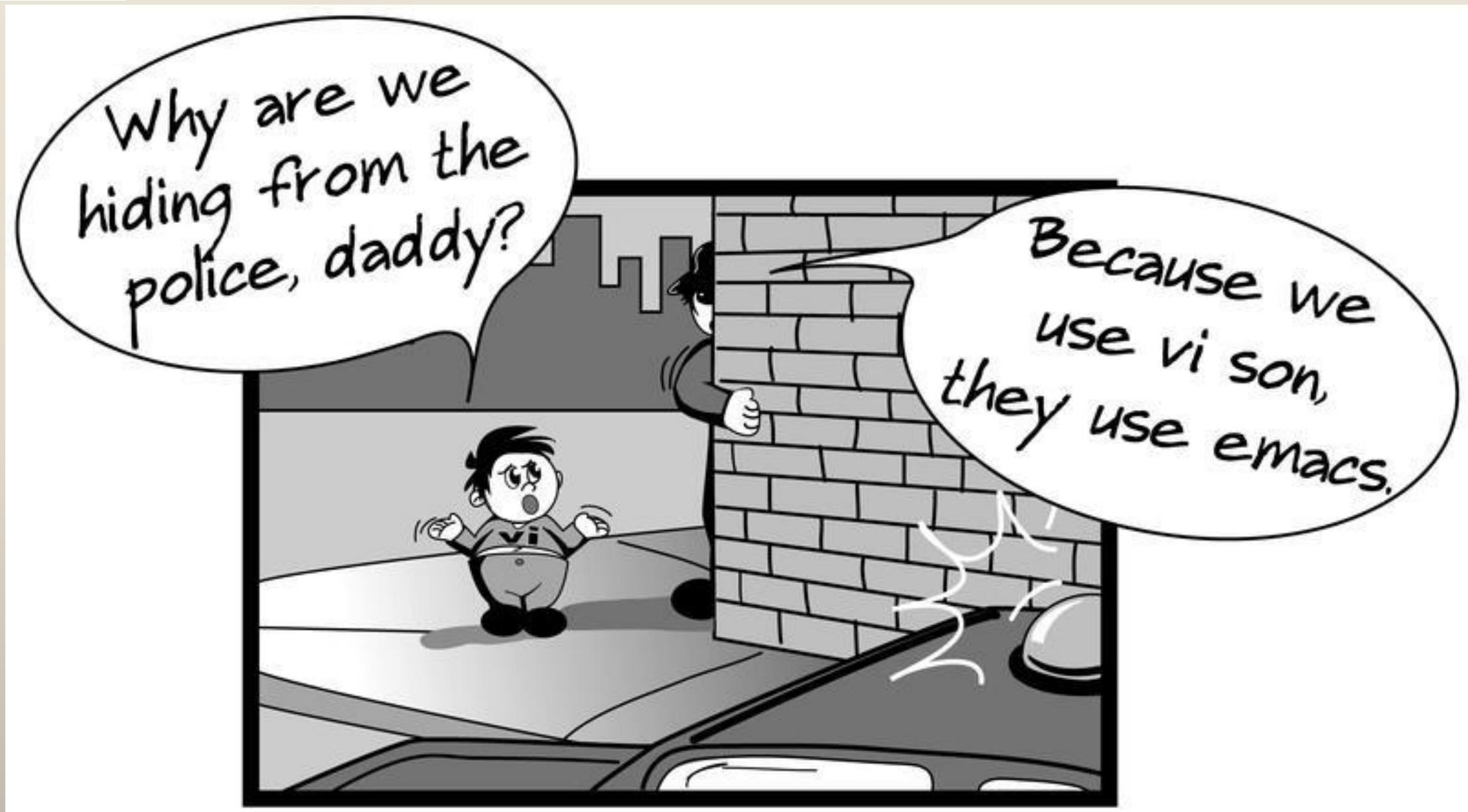
a after cursor	yank word yw
A at end of line	yank line yy
i before cursor	put p
I at beginning of line	
o open line below	
O open line above	

ex COMMANDS

:se nu	set numbers
:se nonu	no numbers
:r file	read in file
:! cmd	run cmd
:se wm=10	wrap words

SAVE/QUIT

:w	write buffer
:q	quit
:wq	write & quit
:q!	abandon buffer
ZZ	same as :wq
^Z	suspend vi



- emacs began at the Artificial Intelligence Laboratory at MIT.
 - In 1972, staff hacker Carl Mikkelsen added display-editing capability to TECO, by using a set of macros written in the TECO macro language.
- The collection of macros was organized by Richard Stallman and was called TecoEmacs (for Editor MACroS)
 - In 1984, Stallman rewrote emacs in C, creating GNU emacs and (later) the collection of GNU software.

emacs Saves the World

It can be *reasonably asserted* that the writing of GNU emacs and the creation of [FSF](#) that followed, **started the revolution** of Open Source software that we enjoy so much now.

The **first** example of free and open-source software is believed to be the A-2 system, developed at the UNIVAC division of Remington Rand in 1953, which was released to customers with its source code.



emacs – Terminal Editor or IDE?

Yes.

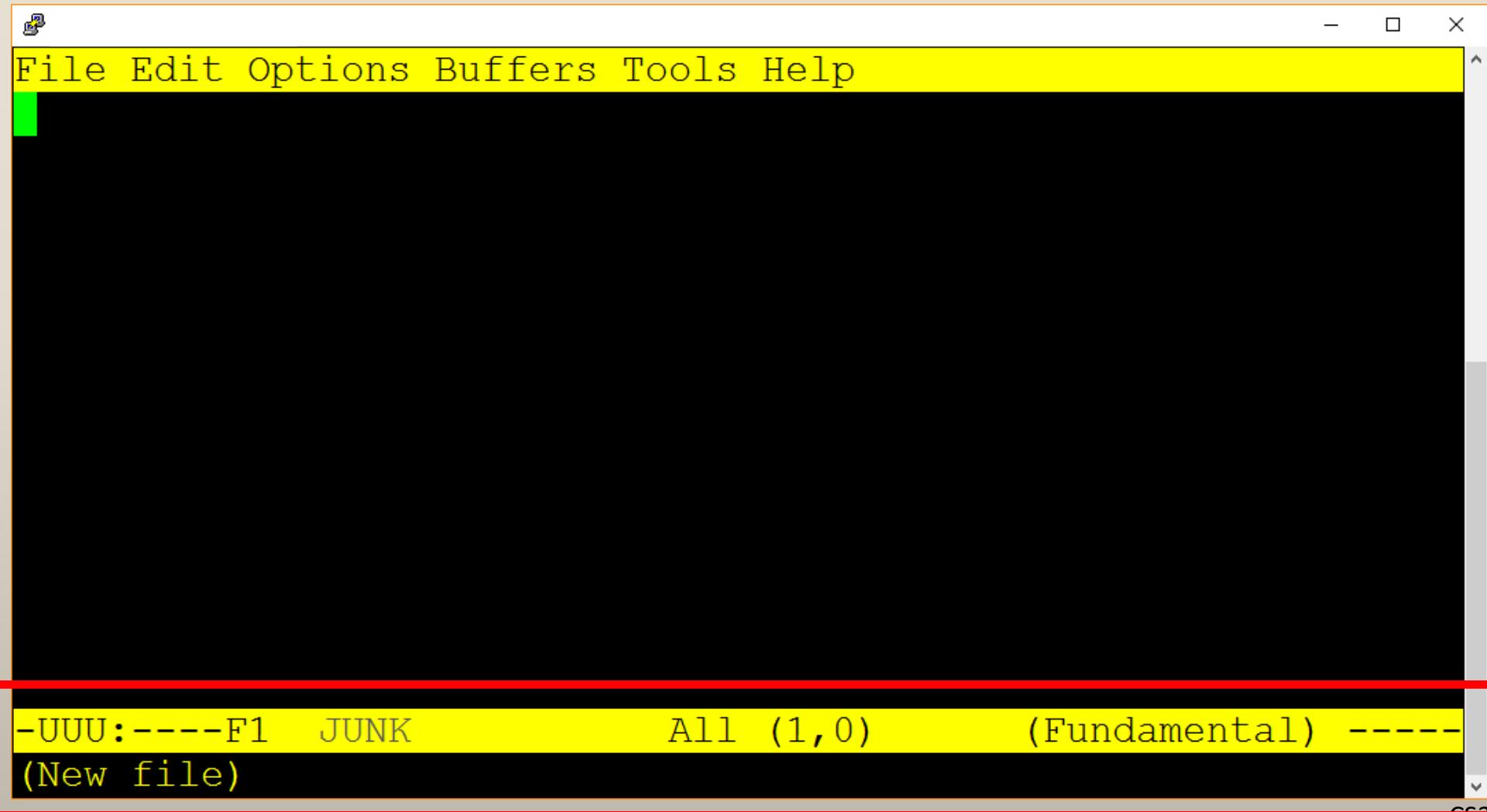
emacs can be used as strictly terminal based editor or as an IDE.

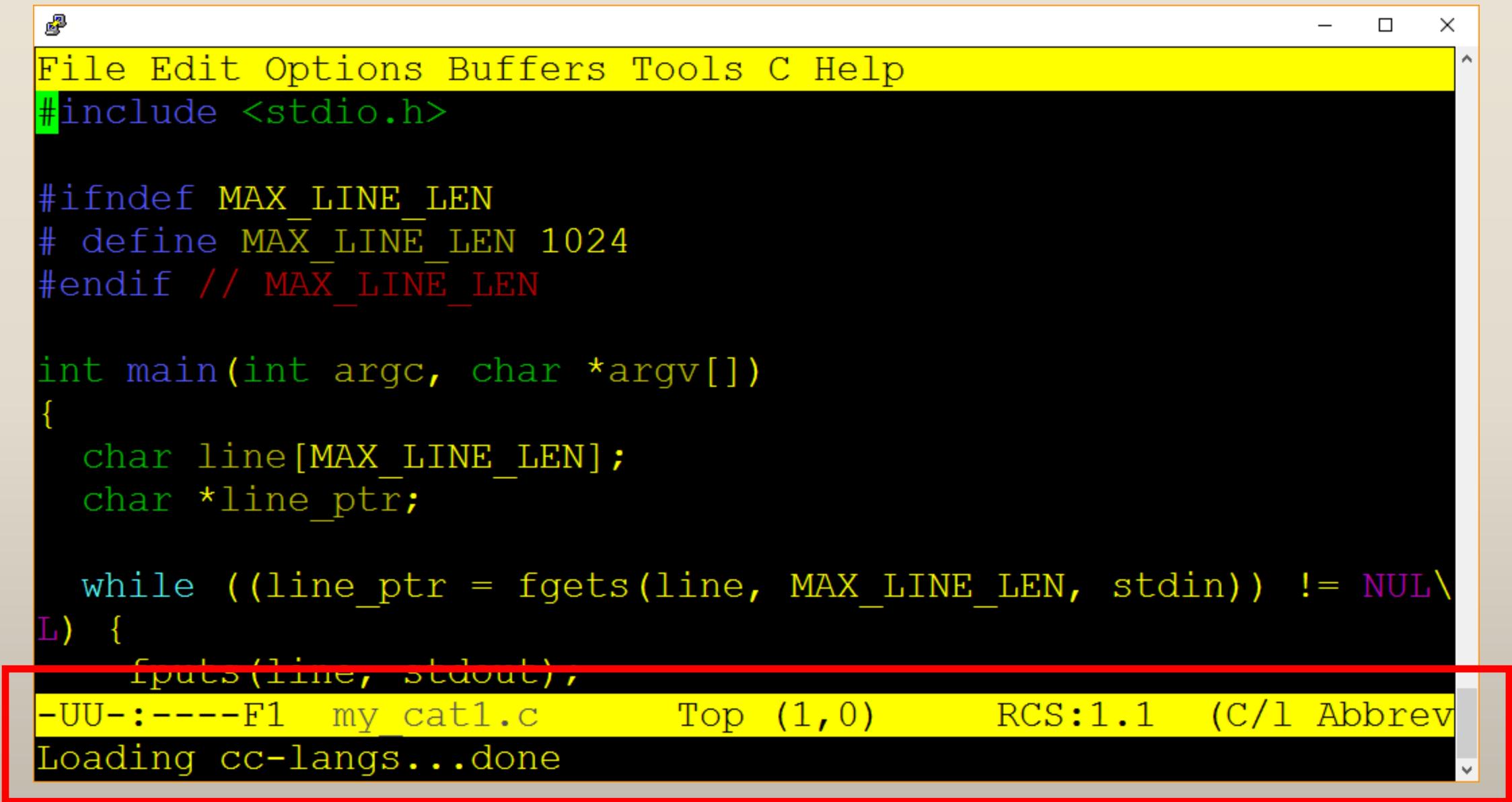
Compared to an IDE like Visual Studio, emacs may seem lacking. However, it is much more customizable than Visual Studio.

It is also free.

Starting emacs

```
$ emacs [filename]
```





A screenshot of a terminal window titled "File Edit Options Buffers Tools C Help". The window contains the following C code:

```
#include <stdio.h>

#ifndef MAX_LINE_LEN
# define MAX_LINE_LEN 1024
#endif // MAX_LINE_LEN

int main(int argc, char *argv[])
{
    char line[MAX_LINE_LEN];
    char *line_ptr;

    while ((line_ptr = fgets(line, MAX_LINE_LEN, stdin)) != NULL) {
        fputs(line, stdout);
    }
}
```

The terminal window shows the output of the program, which is the file "cat1.c" itself. The status bar at the bottom indicates:

-UU-:----F1 my cat1.c Top (1, 0) RCS:1.1 (C/l Abbrev)
Loading cc-lang...done

Where `vi` is modal (mo-dull) in nature, `emacs` is not.

- Rather `emacs` relies on **key-cords** and/or **key-sequences** for most/many commands.
- An example simple key-cord is the command that moves to the end of the current line:
 - **C-e** which means pressing the control key and the e key at the same time.
- An example of a key sequence is how to exit `emacs` (assuming there are no modified files)
 - **C-x C-c** which means pressing the control key with the x key, **releasing both**, then pressing the control key with the c key.

A key-cord in emacs is a combination of one or more of the modifier keys pressed with another key on the keyboard.

The modifier keys are:

- Control (abbreviation is C)
- Alt (abbreviation is A)
- Meta (abbreviation is M)
- Shift (abbreviation is S)

The meta key was common on the Lisp machines in use when emacs was developed at MIT.

On a Windows keyboard, **you can use the Alt key as the Meta key.**

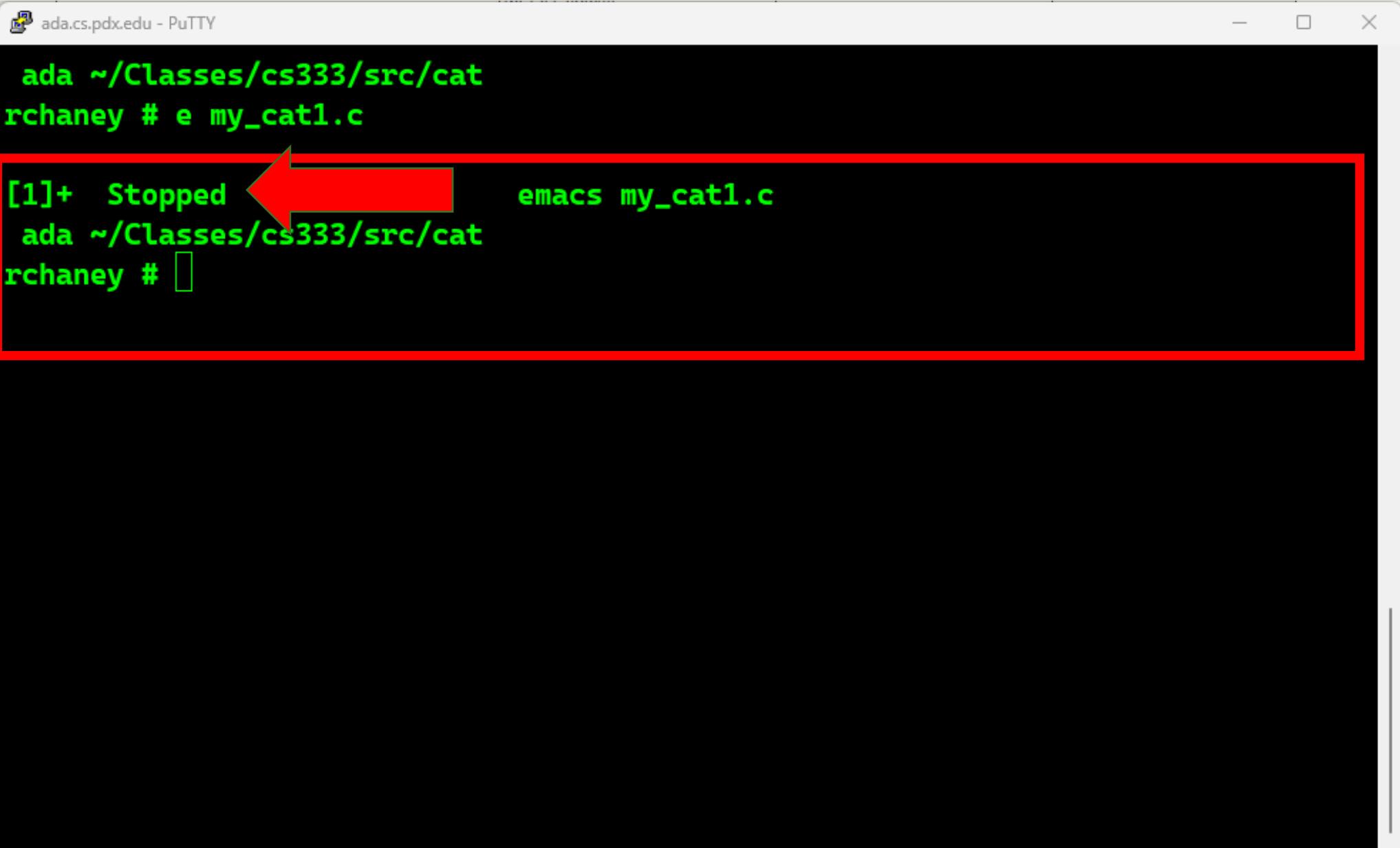
You can also simulate the meta key by pressing (and releasing) the escape key as a key-sequence.

Key-sequence	Command
C-x C-c	Exit emacs – if there are unsaved files, you will need to confirm. This stands for Control-x Control-c
C-x C-s	Save current file.
C-x s	Save all files.
C-x C-f	Open file.
C-space	Set mark.
C-w	Kill region
C-y	Yank region back from kill.
C-/ or C-x u	Undo previous edit.

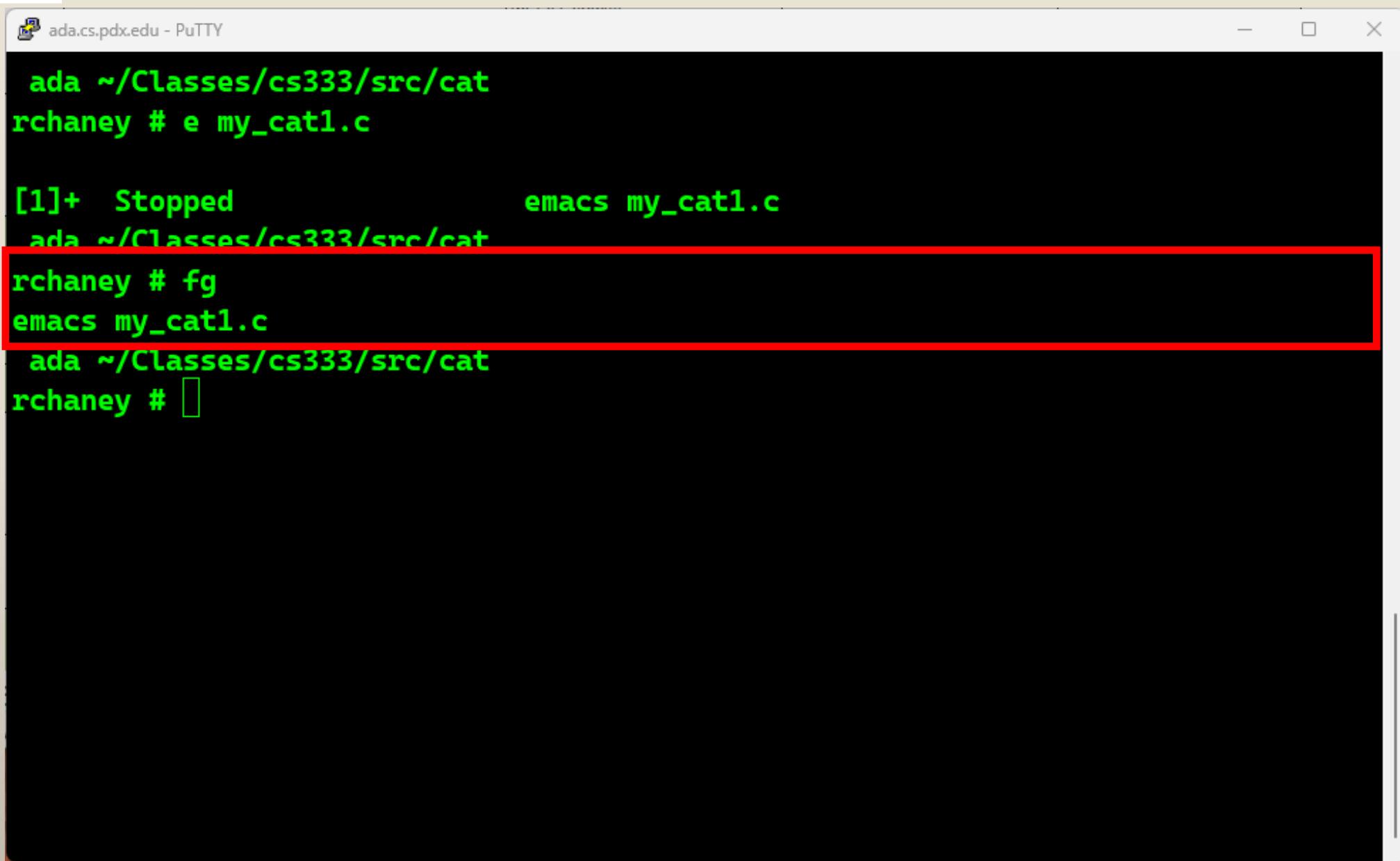
Key-sequence	Command
M-<	Beginning of file. This is Meta-GreaterThan
M->	End of file
M-g g	Go to line
M-w	Kill region without delete (aka copy)
C-a	Beginning of line
C-e	End of line
C-l	Center window at point
C-x 1	Close other windows
C-x 2	Split current window horizontally

A Background Note

- You will very likely try to use the Windows version of undo (the control-z key-cord) to undo some command in either vi or emacs (or both).
- **The control-z key-cord in UNIX is very different from the control-z in Windows.**
- In a Windows application, control-z is probably the undo command.
- In UNIX, a control-z means to “push the current foreground application into the background”.
- If you press control-z and find yourself suddenly at the shell prompt, **don’t rerun the editor**, return to your editor session by typing ‘fg’ (which stands for foreground) at the shell prompt.



```
ada ~/Classes/cs333/src/cat
rchaney # e my_cat1.c
[1]+  Stopped                 emacs my_cat1.c
ada ~/Classes/cs333/src/cat
rchaney # 
```



ada.cs.pdx.edu - PuTTY

```
ada ~/Classes/cs333/src/cat
rchaney # e my_cat1.c

[1]+  Stopped                  emacs my_cat1.c
ada ~/Classes/cs333/src/cat
rchaney # fg
emacs my_cat1.c
ada ~/Classes/cs333/src/cat
rchaney # 
```



One of the great things that Unix did was put the reference manual pages right into the system.

You can access all of the system commands, functions, and other information using the `man` command.

A common refrain you'll hear from me during the term will be

What does the man page tell you?

Or

Did you read the man page?



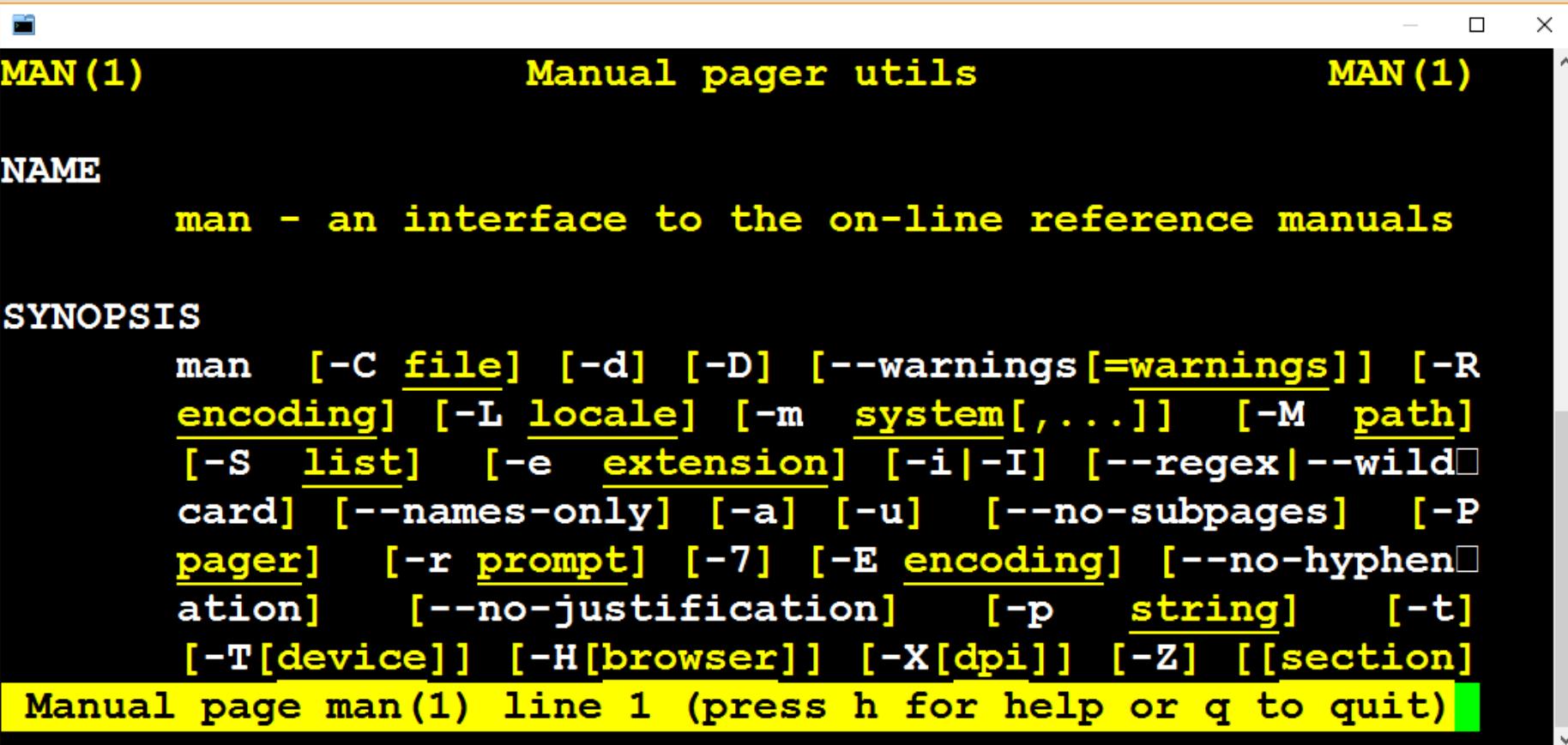
The Sections of Man Pages

Section	Contents	
1	Executable programs or shell commands.	You'll be spending most of your man time in sections 1, 2, and 3.
2	System calls (functions provided by the kernel).	
3	Library calls (functions within program libraries).	
4	Special files (usually found in /dev).	
5	File formats and conventions e.g. /etc/passwd.	
6	Games.	
7	Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7).	
8	System administration commands (usually only for root).	
9	Kernel routines [Non standard, but common for Linux].	

Man Up

One of the commands you may find yourself wanting to see the documentation for is man.

It is easy to do: 'man man'



The screenshot shows a terminal window with a black background and white text. The title bar reads "MAN(1) Manual pager utils MAN(1)". The window contains the following text:

```
NAME
    man - an interface to the on-line reference manuals

SYNOPSIS
    man [-C file] [-d] [-D] [--warnings[=warnings]] [-R
encoding] [-L locale] [-m system[,...]] [-M path]
[-S list] [-e extension] [-i|-I] [--regex|--wild-
card] [--names-only] [-a] [-u] [--no-subpages] [-P
] [-r prompt] [-7] [-E encoding] [--no-hyphen-
ation] [--no-justification] [-p string] [-t]
[-T[device]] [-H[browser]] [-X[dpi]] [-Z] [[section]]
Manual page man(1) line 1 (press h for help or q to quit)
```

Man Pages

Sometimes finding the right man pages can be a little challenging.



- For example, if you type `man exit` you get the man page for the bash shell, which is probably not what you want.
- In this case, you probably want to issue the command `man 3 exit` or possibly `man 2 exit`
- The same goes for the `read()` and `write()` system calls. Those are in section 2 of the man pages, though you can also find some of the information for those in section 3.

Man Pages Layout

All `man` pages follow a common layout that is optimized for presentation on a simple ASCII text display, possibly without any form of highlighting or font control. Sections may include:

NAME

The name of the command or function, followed by a one-line description of what it does.

SYNOPSIS

In the case of a command, a formal description of how to run it and what command line options it takes. For program functions, a list of the parameters the function takes and which header file contains its definition.

DESCRIPTION

A textual description of the functioning of the command or function.

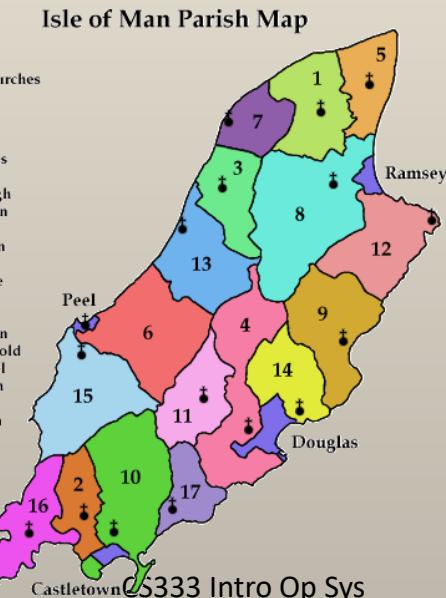
EXAMPLES

Some examples of common usage.

I like this section a lot!

SEE ALSO

A list of related commands or functions.





From section 3 in the
man pages.

EXIT(3)

Linux Programmer's Manual

EXIT(3)

NAME

exit - cause normal process termination

A short description
of the function.

SYNOPSIS

#include <stdlib.h>

The header file you need to
include to use the function.

void exit(int status);

DESCRIPTION

The **exit()** function causes normal process termination and the value of **status** & 0377 is returned to

Manual page **exit(3)** line 1 (press h for help or q to quit)

SYSCALLS (2) Linux Programmer's Manual SYSCALLS (2)

NAME syscalls - Linux system calls

SYNOPSIS Linux system calls.

DESCRIPTION

The system call is the fundamental interface between an application and the Linux kernel.

System calls and library wrapper functions

System calls are generally not invoked directly, but rather via wrapper functions in glibc (or perhaps some other library). For details of direct invocation of a system call, see intro(2). Often, but not always, the name of the wrapper function is the same as the name of the system call that it invokes. For example, glibc contains a function truncate() which invokes the underlying "truncate" system call.

Often the glibc wrapper function is quite thin, doing little work other than copying arguments to the right registers before invoking the system call, and then setting errno appropriately after the system call has returned. (These are the same steps that are performed by syscall(2), which can be used to invoke system calls for which no wrapper function is provided.) Note: system calls indicate a failure by returning a negative error number to the caller; when this happens, the wrapper function negates the returned error number (to make it positive).

Manual page syscalls(2) line 1 (press h for help or q to quit)

Finding the Right Man

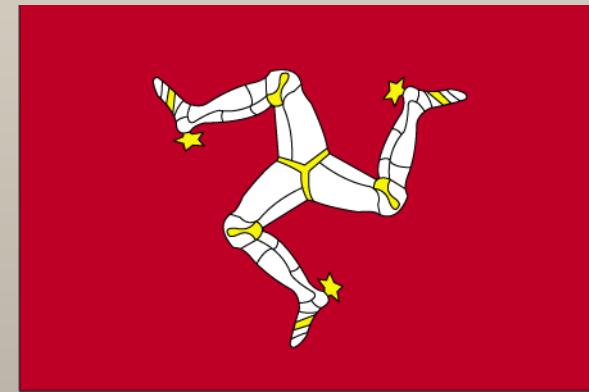
Man has a way to help you find what you are looking for in the man pages.
You can either

```
man -k <string>  
apropos <string>
```

man -k printf

Search the short descriptions and manual page names for the keyword
printf as regular expression. Print out any matches. Equivalent to

```
apropos -r printf
```



The flag of the Isle of Man

```
jesse.chaney@loki 03:38 PM $ man -k printf
asprintf (3)          - print to allocated string
ber_printf (3)         - OpenLDAP LBER simplified Basic Enc...
bprintf (9)           - Parse a format string and place arg...
bstr_printf (9)        - Format a string from binary argumen...
curl_mprintf (3)       - formatted output conversion
devm_kasprintf (9)     - Allocate resource managed space and...
devm_kvasprintf (9)    - Allocate resource managed space and...
dprintf (3)            - print to a file descriptor
format (n)             - Format a string in the style of spr...
fprintf (3)            - formatted output conversion
fprintf (3p)           - print formatted output
fwprintf (3)           - formatted wide-character output con...
fwprintf (3p)          - print formatted wide-character output
```

Man Commands

- From within a man page, **you can search** for text by first typing the backslash character ('/'), the text for which you are searching, and pressing Enter. You can use '?' to search backwards.
- You can repeat the search by pressing the 'n' key. 'N' will search backwards.
- You can scroll up and down using the arrow keys.
- You can page down by using the space-bar or page-down keys.
- The page-up key will scroll up one page.
- You can return to the top of the mane page with the 'g' key.
- You can go to the end of the man page with a 'G' key.
- The 'q' key will exit the man page.

Web Man

- There are a **LOT** of web sites that contain UNIX man pages.
- I encourage you to use the man pages on **ada.cs.pdx.edu** (aka **linux.cs.pdx.edu**) when you are working on your programs for this class.
- Many of the other man pages are for other flavors of UNIX or Linux or may actually be out of date (or be for a Mac).
- I hate to see students wasting time looking at out of date man pages **or man pages for a Mac**.
- **One VERY good place for man pages is:**

<https://www.kernel.org/doc/man-pages/>

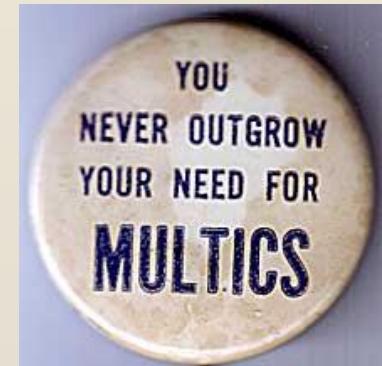
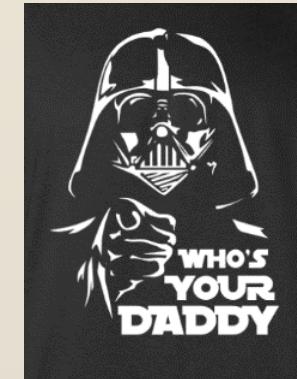


Multics

Multics

Multiplexed Information and Computing Service

- Bell Labs was involved in a project with MIT and General Electric to develop a time-sharing system.
- The majority of the systems at that time were strictly batch oriented.
- Virtually all modern operating systems are influenced by Multics.
- The project was started in 1964 in Cambridge, Massachusetts



Operating system genealogy places Multics as the **progenitor** of Unix



UNIX – The Early Days

UNIX was started at AT&T Bell Labs,
as a way to play a game!

One of the notable features of the UNIX system
is that its development was **not controlled by a
single vendor or organization.**

Dennis Ritchie



Ken Thompson

Digital Equipment PDP-7



The Programmed Data Processor (PDP) line (PDP-1 through PDP-16, though not PDP-13) basically created the category of mini-computer. For its time, it was a very successful series.

The PDP-1 was the system used to create the [first video game, Spacewar!](#) and the first word processor “Expensive Typewriter.”



UNIX Development

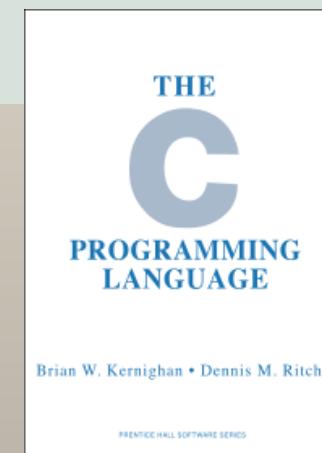
Some of the important features that UNIX inherited from Multics include:

- Tree structured file system
- A separate program for interpretation of commands (**the shell**)
- The notion of files are an unstructured stream of bytes.

UNIX Progresses

Initially **written in PDP assembly language**, it was ported (mostly) to a **new** programming language called C in 1972.

The definitive book “The C Programming Language” by Kernighan and Ritchie’s was published in 1978.



UNIX Editions

- Between 1969 and 1979, UNIX went through a number of releases, known as editions.
- Over the period of these releases, the use and reputation of UNIX began to spread within AT&T and then beyond AT&T.
- At this time, AT&T held a government-sanctioned monopoly on the US telephone system. The terms of AT&T's agreement with the US government prevented it from selling software, which meant that it could not sell UNIX as a product.
- Instead, **AT&T licensed UNIX for use in universities for a small distribution fee.**
 - The university distributions included documentation and the **kernel source code**.
 - For a university, UNIX offered an interactive multiuser operating system that was cheap but powerful, at a time when commercial operating systems were very expensive.

UNIX goes to School

- In 1975/1976, **Dennis Thompson spent a year as a visiting professor at the University of California at Berkeley.**
- He took the UNIX source code and ideas with him.
- Working with several graduate students (such as Bill Joy), several new tools and features were added to UNIX.
 - **C shell**
 - **vi** 
 - **an improved file system** (the Berkeley Fast File System)
 - **sendmail**
 - **virtual memory** (on a Digital Equipment VAX)

This is Beastie

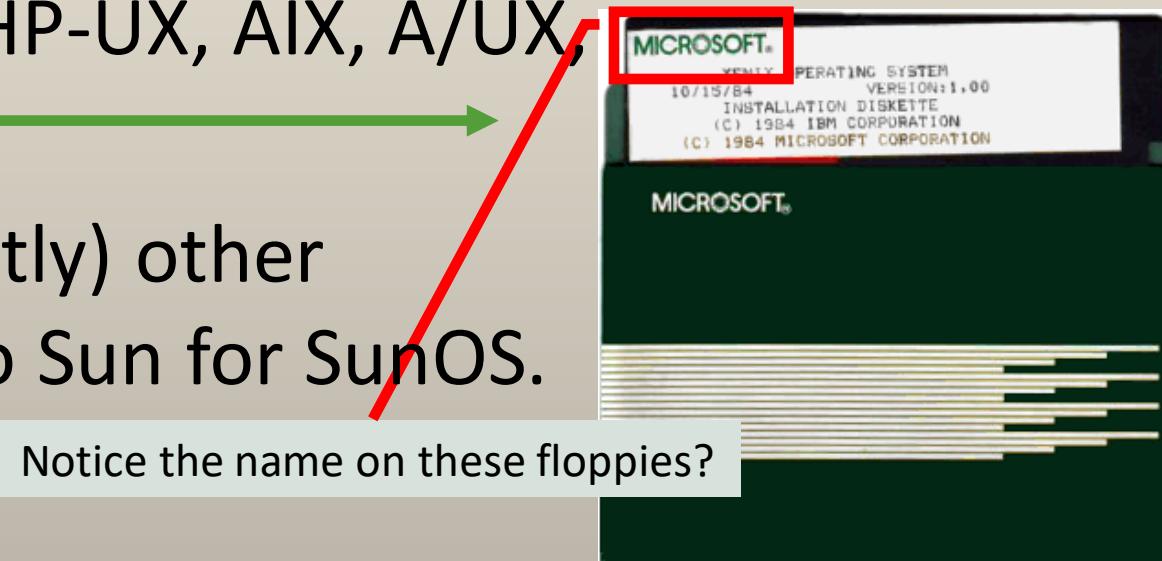


Berkeley Software
Distribution (**BSD**)

In 1983, 4.2BSD was released with **TCP/IP and sockets** for doing network programming!

Split Happens

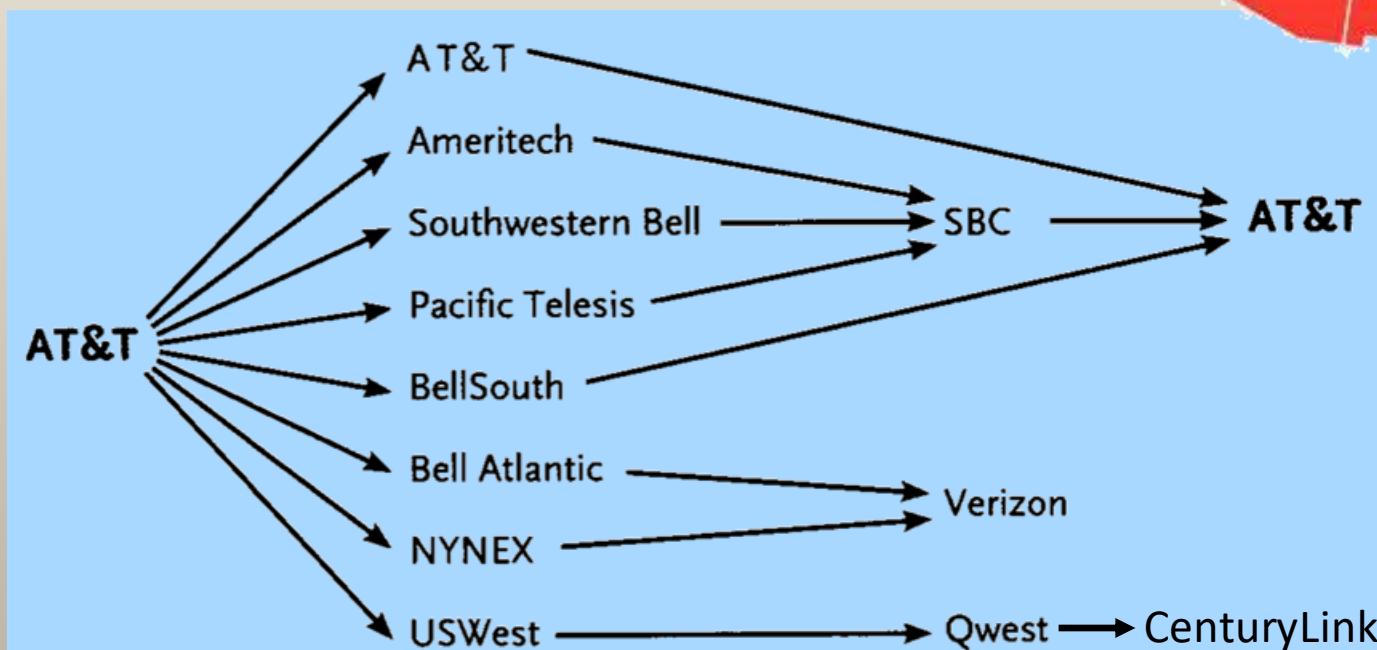
- In other news, US antitrust legislation forced the breakup of AT&T (called MA Bell at the time)
 - The additional effect was the company was now permitted to market and sell software, including UNIX.
- AT&T licensed UNIX to **commercial** vendors.
 - From this came systems like: HP-UX, AIX, A/UX, and XENIX.
 - Berkeley licensed BSD to (mostly) other academic institutions, but also Sun for SunOS.



January 8th 1982 – Breakup of the Bell System: AT&T agrees to divest itself of twenty-two subdivisions.

After an 8 year court case brought by the DOJ, AT&T proposed a settlement that would split the local phone service into regional telephone companies.

Part of the settlement proposed by AT&T was that it could begin to sell computer equipment and software.



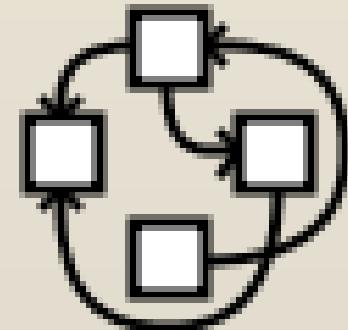
GNU Gets Going



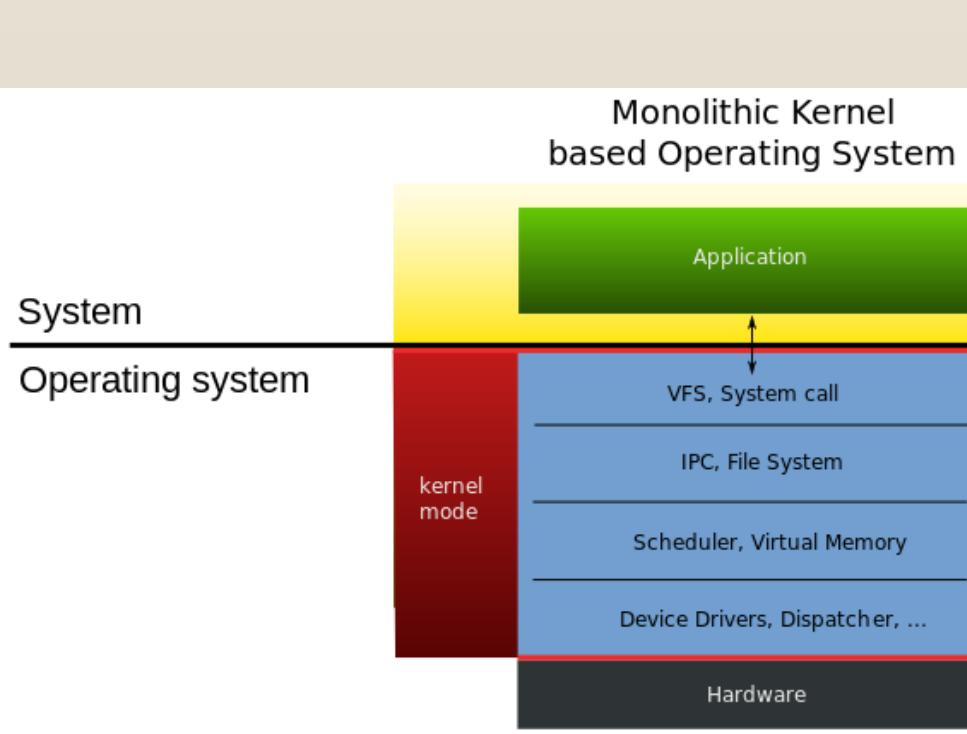
- Back at MIT, Richard Stallman is writing software that is given away **EMACS**
- Stallman believes that software (especially operating systems, editors, and compilers) should be at little or no cost.
- In 1985, Stallman founded the **Free Software Foundation** to support development on the GNU Project, to build freely available high quality software.

GNU's Almost There

- By the early 1990's the GNU Project **nearly** had a complete system, with all the UNIX-like utilities.
- **One significant thing was missing, a working kernel.**
- One had been started, GNU/HURD, to be based on the **Mach microkernel**.
 - However, GNU HURD was far from complete and further from being a releasable piece of software.
 - The meaning of "Hurd": "Hurd" stands for "Hird of Unix-Replacing Daemons". And, then, "Hird" stands for "Hurd of Interfaces Representing Depth".



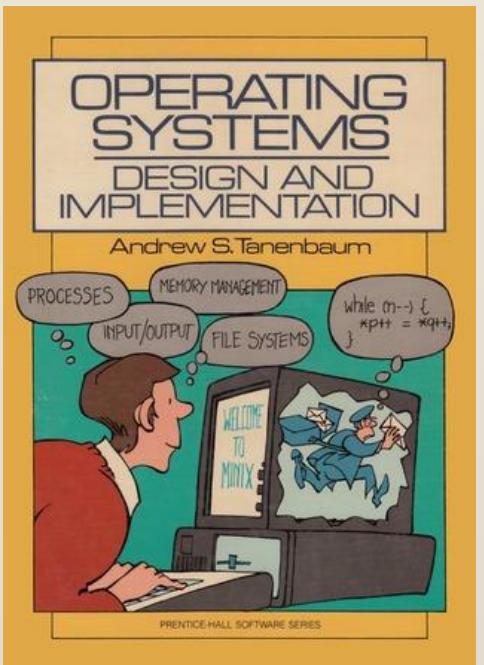
The Monolith vs Microkernel



https://en.wikipedia.org/wiki/GNU_Hurd

Minix leads to Linux

- In the course of his university studies, **Linus Torvalds** had come into contact with **Minix**, a small UNIX-like operating system kernel developed in the mid-1980s by Andrew Tanenbaum, a university professor in Holland.
- Minix was available as source code through one of Tanenbaum's book's about operating systems.
- Torvalds liked Minix, but it did not take advantage of the top popular hardware of the time, the **Intel 386**, and did not have all the capabilities he wanted.
- So, he set out to write a complete UNIX-like kernel.
- He also realized that he wanted others involved.



Linus' Missive

*Do you **pine** for the nice days of Minix-1.1, when men were men and wrote their own device drivers? Are you without a nice project and just dying to cut your teeth on a OS you can try to modify for your needs? Are you finding it frustrating when everything works on Minix? No more all-nighters to get a nifty program working? Then this post might be just for you. As I mentioned a month ago, **I'm working on a free version of a Minix-look-alike** for AT-386 computers. It has finally reached the stage where it's even usable (though may not be depending on what you want), and I am willing to put out the sources for wider distribution. It is just version 0.02 . . . but I've successfully run bash, gcc, gnu-make, gnu-sed, compress, etc. under it.*

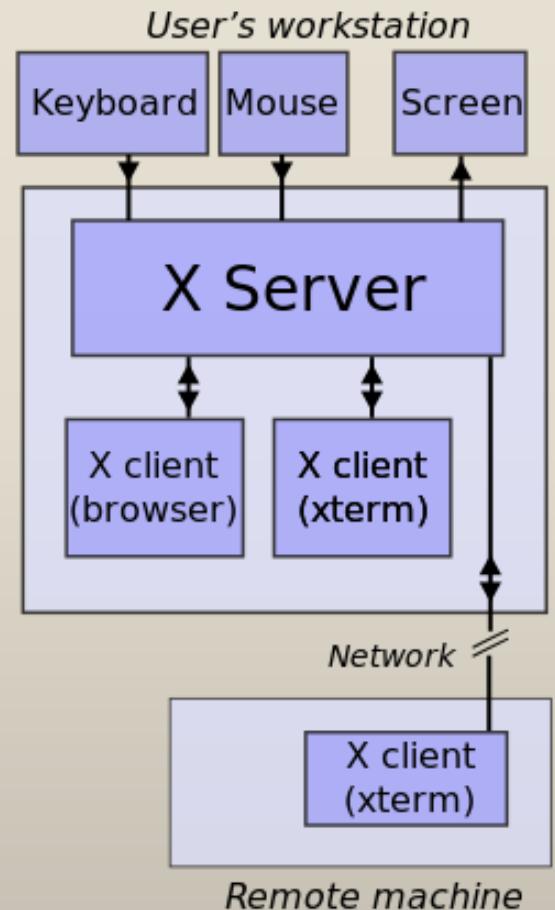
Oh Yex, There's X

- The original idea of the **X Window System** emerged at MIT in 1984 as a collaboration between Jim Gettys and Bob Scheifler.
- Project Athena, a joint project between **Digital Equipment Corporation (DEC)**, **MIT** and **IBM**, wanted to provide easy access to computing resources for all students using a platform-independent graphics system to link together its heterogeneous multiple-vendor systems.
- X (as the X Window System became known) became the first windowing system environment to offer **true hardware independence and vendor independence**.



The X Window System

- The X Window System provides the base technology for developing graphical user interfaces.
- At a very basic level, X draws the elements of the GUI on the user's screen and builds in methods for sending user interactions back to the application.
- X was been implemented from the start to use a client/server model, it is ideally suited for remote application deployment as well.
- **X lets you work from your computer directly with an application running on another computer.**

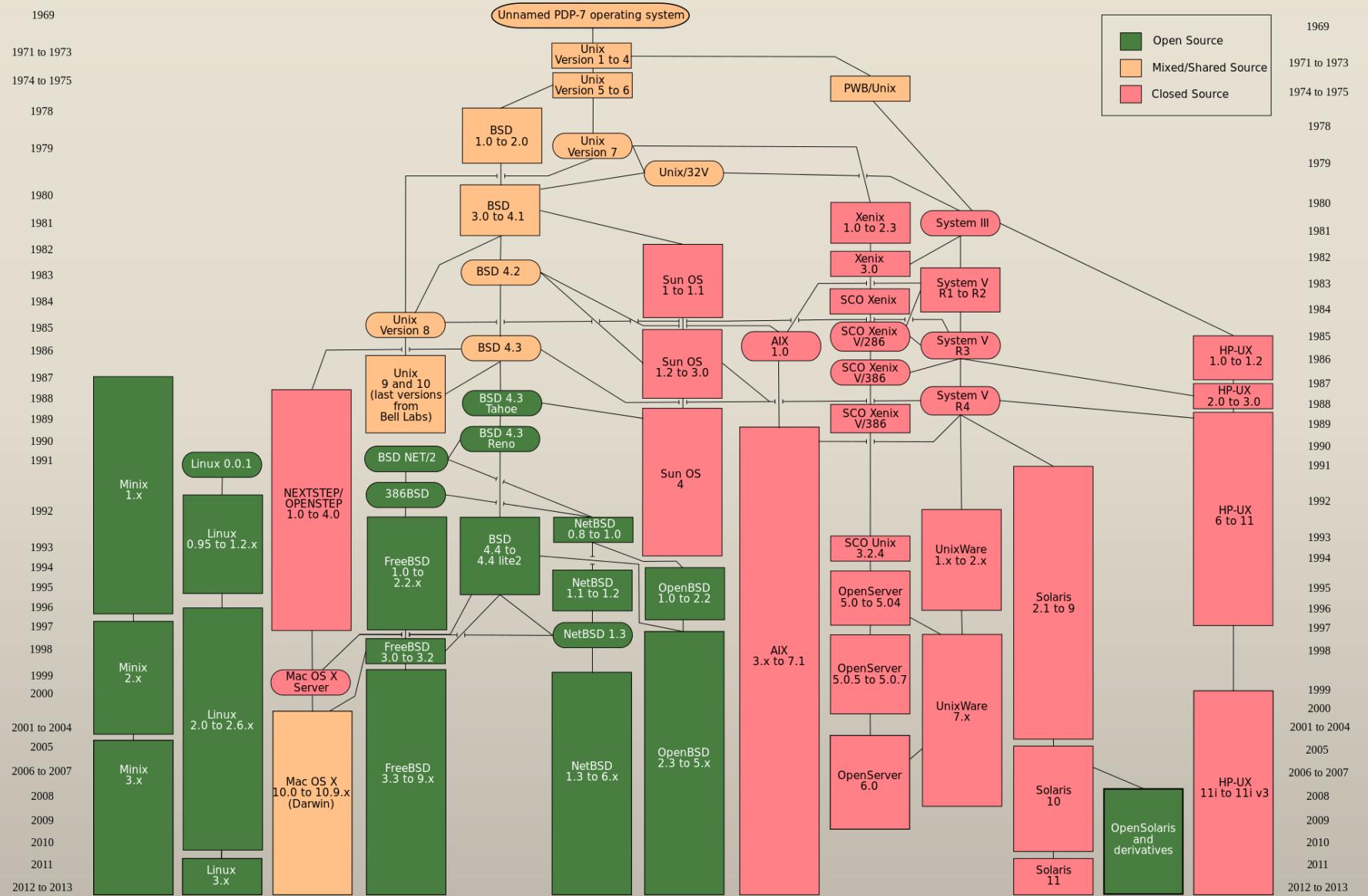


X Consortium and Beyond



- X was derived from W Window System (from Stanford).
- In January 1988, the MIT X Consortium formed as a non-profit vendor group to keep X as vendor independent.
- In its X11R6.3 release of the protocol, **X reached its zenith** and the **X Consortium dissolved at the end of 1996**, releasing the source code into X.Org.
 - The most recent release is X11R7.7, from June 2012.
- **When Linux needed to bring a graphical interface into the fold, X was already developed and available.**

UNIX Directions Over Time



HOW STANDARDS PROLIFERATE:

(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)

SITUATION:
THERE ARE
14 COMPETING
STANDARDS.

POSIX - Portable Operating System Interface

POSIX is more than just the kernel. It includes system calls, library functions, and many utilities.

It is a **family of standards** specified by the IEEE Computer Society for maintaining compatibility between operating systems.

Some POSIX compliant operating systems:

AIX

Solaris

HP-UX

macOS (aka OS X)

QNX

UnixWare



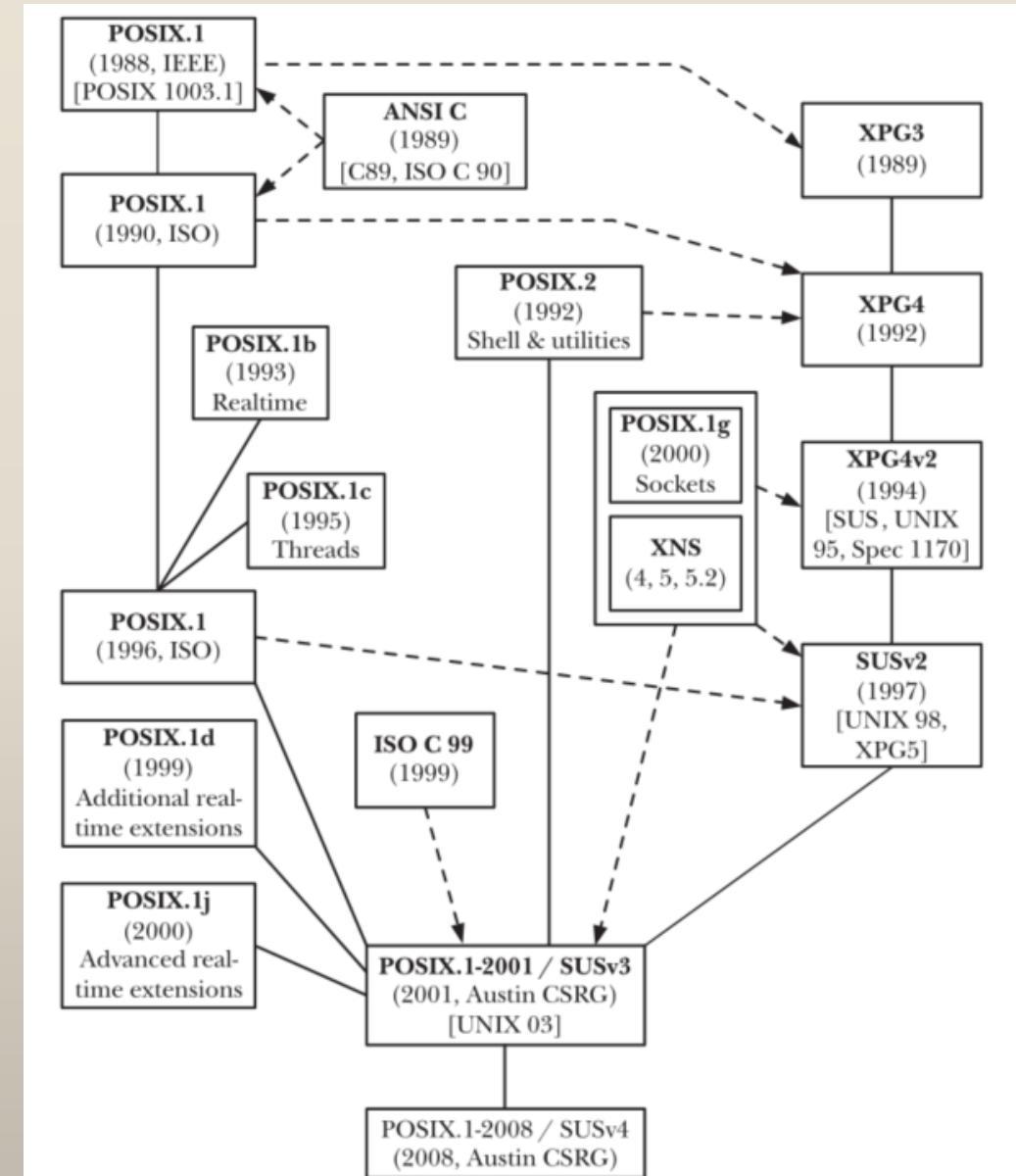
Mac OS X

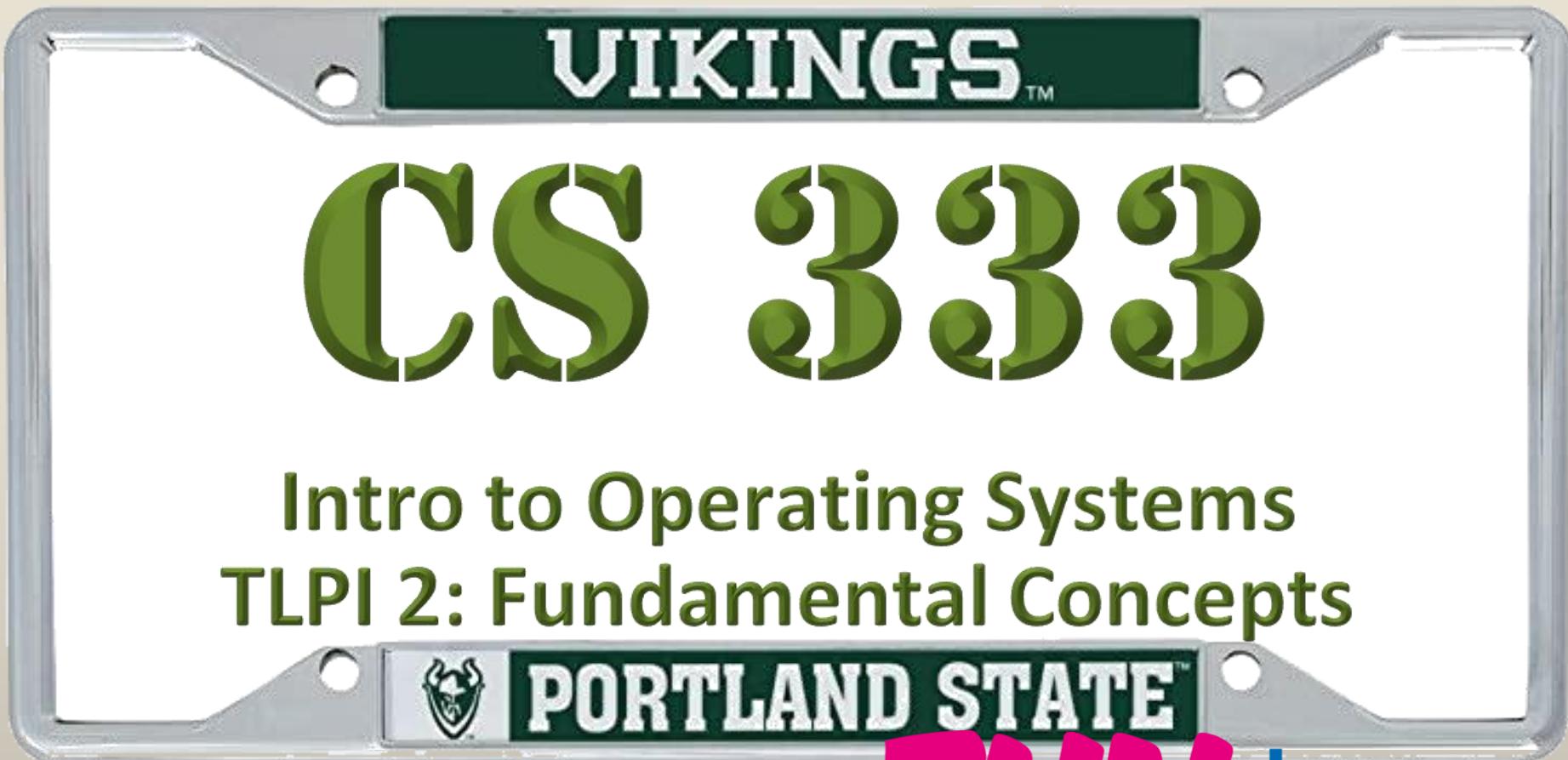
Is there an obvious **missing**
POSIX compliant OS?

SUSv3

Single UNIX Specification Version 3

SUSv3





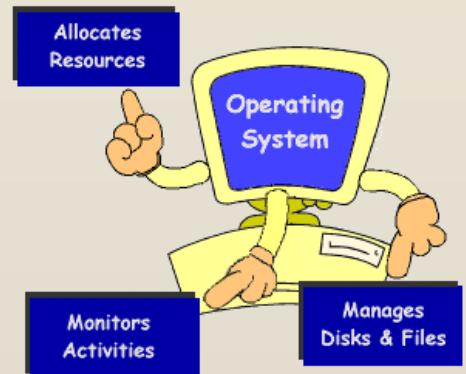
FUNdamental

Operating System

The term operating system is commonly used with **two** different meanings:

1. To denote the **entire package consisting of the central software managing a computer's resources and all of the accompanying standard software tools**, such as command-line interpreters, graphical user interfaces, file utilities, and editors.
2. More narrowly, to refer to the **central software that manages and allocates computer resources** (i.e., the CPU, RAM, and devices).

The term **kernel** is often used as a synonym for the second meaning, and it is with this meaning of the term operating system that we are concerned in this class.



The Kernel

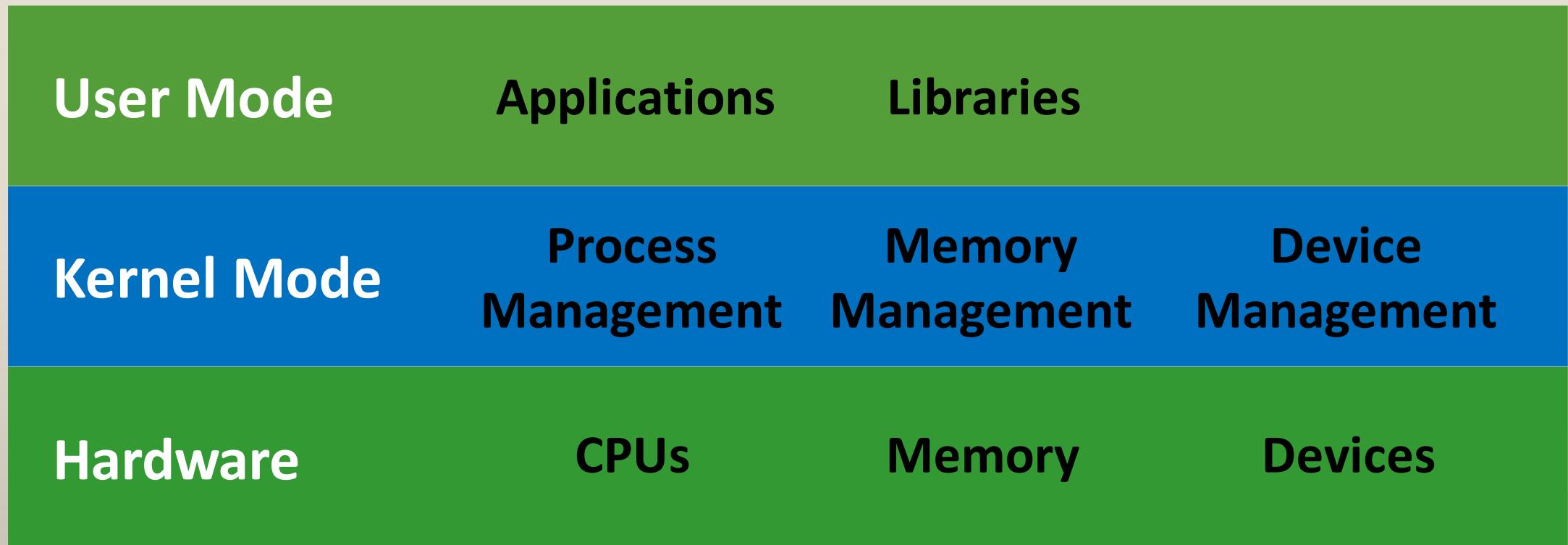
Tasks performed by the kernel

- Creation of and termination of processes.
- Process scheduling.
- Memory management.
- Provide a file system.
- Device access.
- Networking functions.
- System API.

This one is a big deal to you (FN-2187).



Kernel Mode vs User Mode

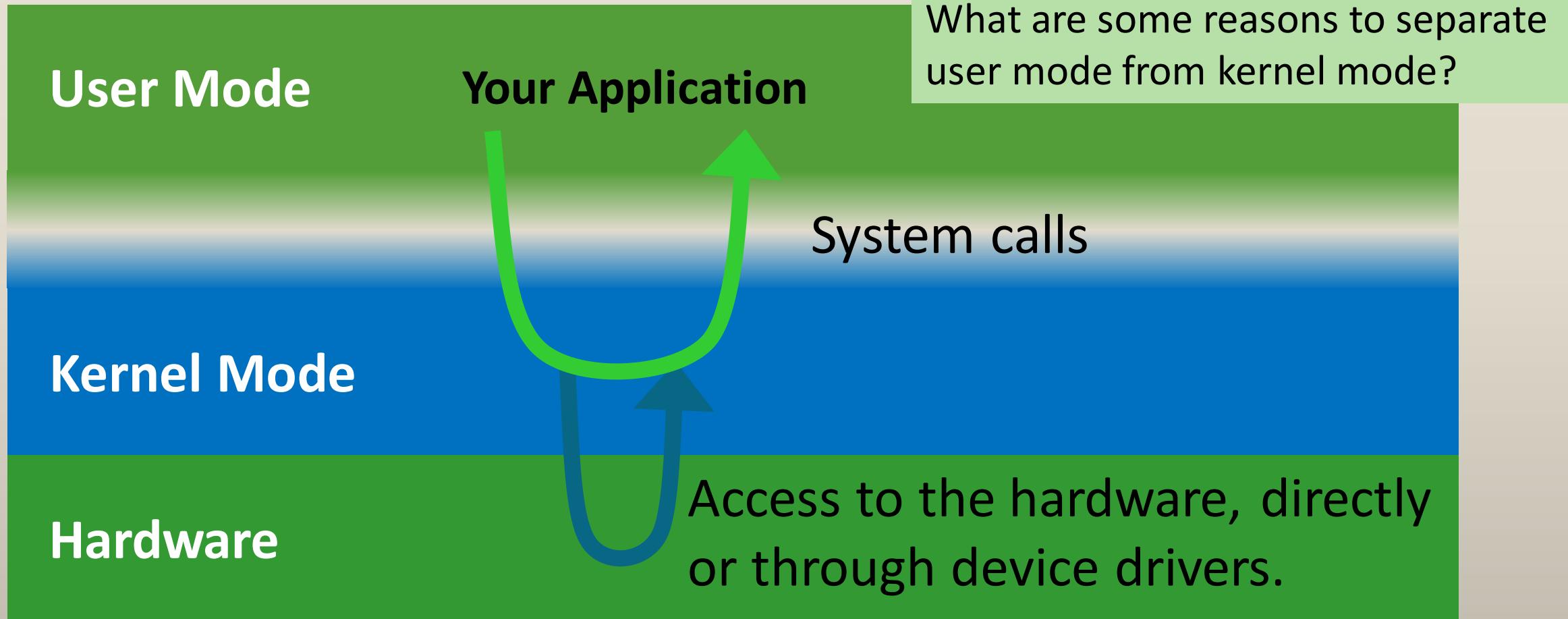


Kernel mode: the executing code has **complete and unrestricted access** to the hardware. It can execute any CPU instruction and reference any memory address. Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. **Crashes in kernel mode are catastrophic;** they will halt the entire system.



User mode: the executing code has **no ability to directly access hardware or reference memory.** Code running in user mode must delegate to system APIs to access hardware or memory. Due to the protection afforded by this sort of isolation, **crashes in user mode are recoverable.** Most code running on your computer executes in user mode.

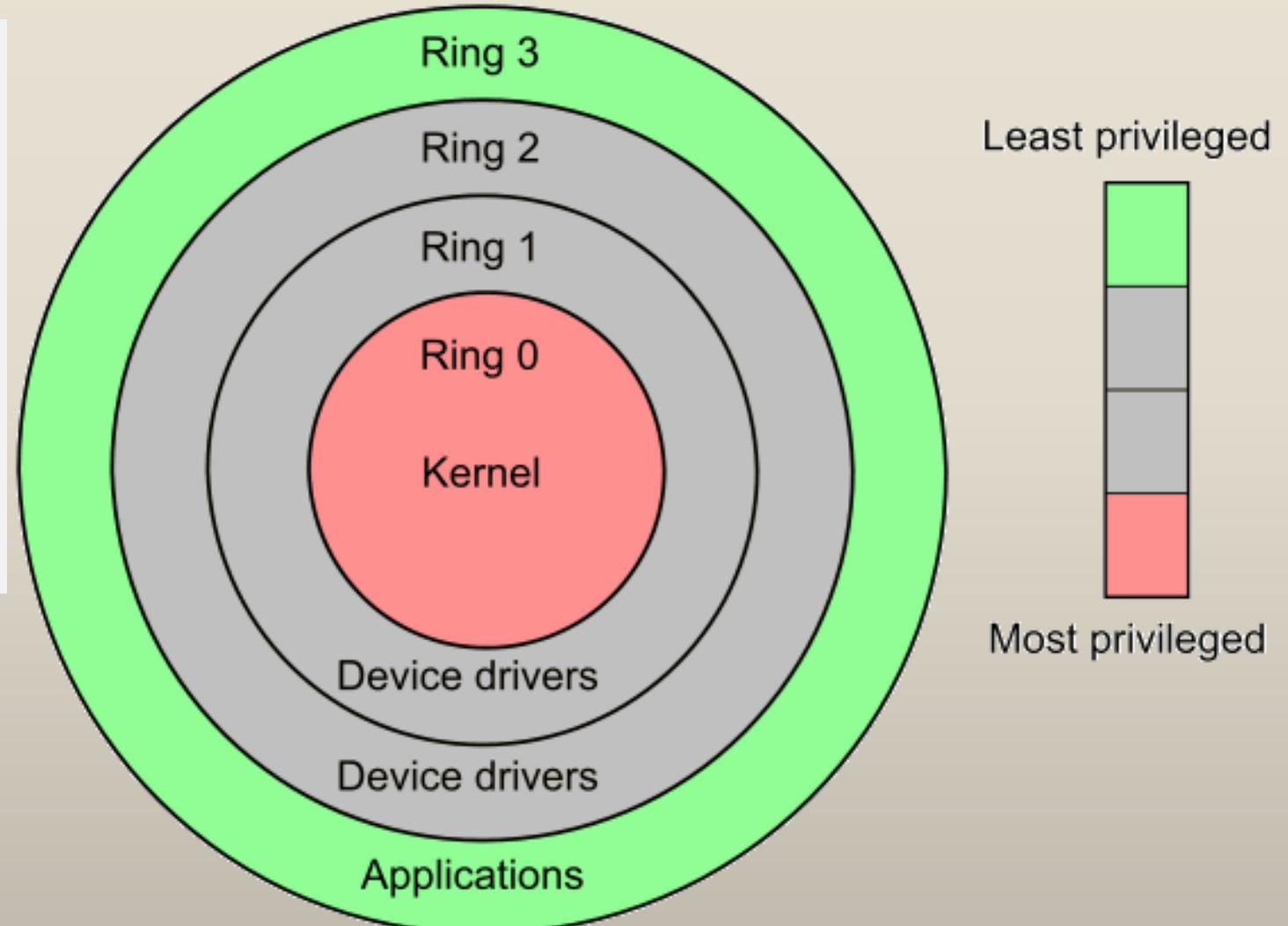
Those System API Calls



More CPU Modes

Sometimes a CPU has more modes than just kernel mode and user mode.

In this class, we do not go into additional modes.



The Shell

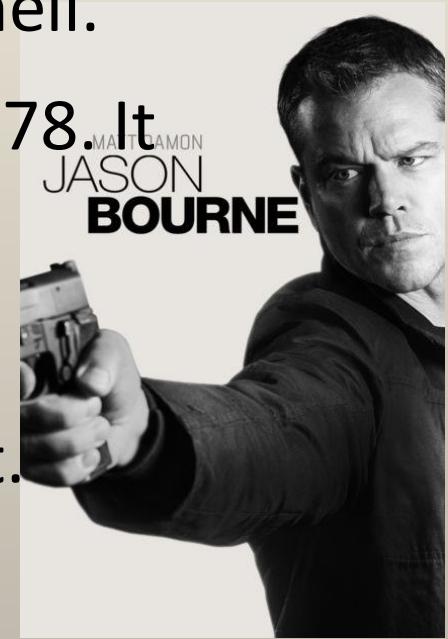


- A lot of your interaction with the system will be through **the shell**.
- The shell is designed to read commands from the user and execute the programs in response to the user input.
- Sometimes the shell is called a command interpreter.
- When you login to the system, you will be presented with a **shell prompt** (you'll have seen this by now).

The Shell, cont.

Some operating systems have a command interpreter that is an integral part of the kernel. On Unix, the shell is simply a user process. You can have a choice of the shell you use.

- sh – this was the original shell written by **Steve Bourne**, release in 1977. It is often called the Bourne shell or just shell.
- csh – this was written by **Bill Joy** at Berkeley, release in 1978. It has constructs more like the C Programming Language.
- tcsh – this is an enhanced version of csh, from CMU.
- **bash** – this is probably the shell you'll use. It is the default.
- There are others...



Users and Groups

Each user on a Unix system is **uniquely** identified.

- Every user of the system has a **unique login name** (**username** or **logname**) and a corresponding numeric user ID (**UID**).
- A user has a **home directory**.
- Each user has a login **default shell**.

It is possible, though strongly discouraged, that a single UID is associated with more than one logname.

The one time I saw this happen, it was a mistake that had to be fixed. It nearly drove a couple people insane.



Users and Groups



Each user may belong to several **groups**.

- For administrative purposes, especially for controlling access to files and other system resources, **Unix organizes users into groups.**
- In the early Unix implementations, a user could belong to only a single group. BSD extended this to more than one group.
- Examples of groups we have on our server are **them** and **internet-faculty**.
- As each user has a **UID** and **logname**, each group has a **GID** and **groupname**.

Users and Groups



There is a **special** user in Unix called the **superuser**.

- The super user has special privileges in the system.
- On typical UNIX systems, the superuser **bypasses all permission checks** in the system.
- The superuser can access any file in the system, regardless of the permissions on that file, and can send signals to any user process in the system.
- The **UID of the superuser is 0** and typically has a logname of **root**.

Users – Groups – Other

For the purposes of access and protection, **Unix divides the world into three categories:**

- **User** – this is you.
- **Group** – think of this as your “buds”.
- **Other** – this is everyone else.

Each file in the file system has permissions set for the access level for **User**, **Group**, and **Other**.

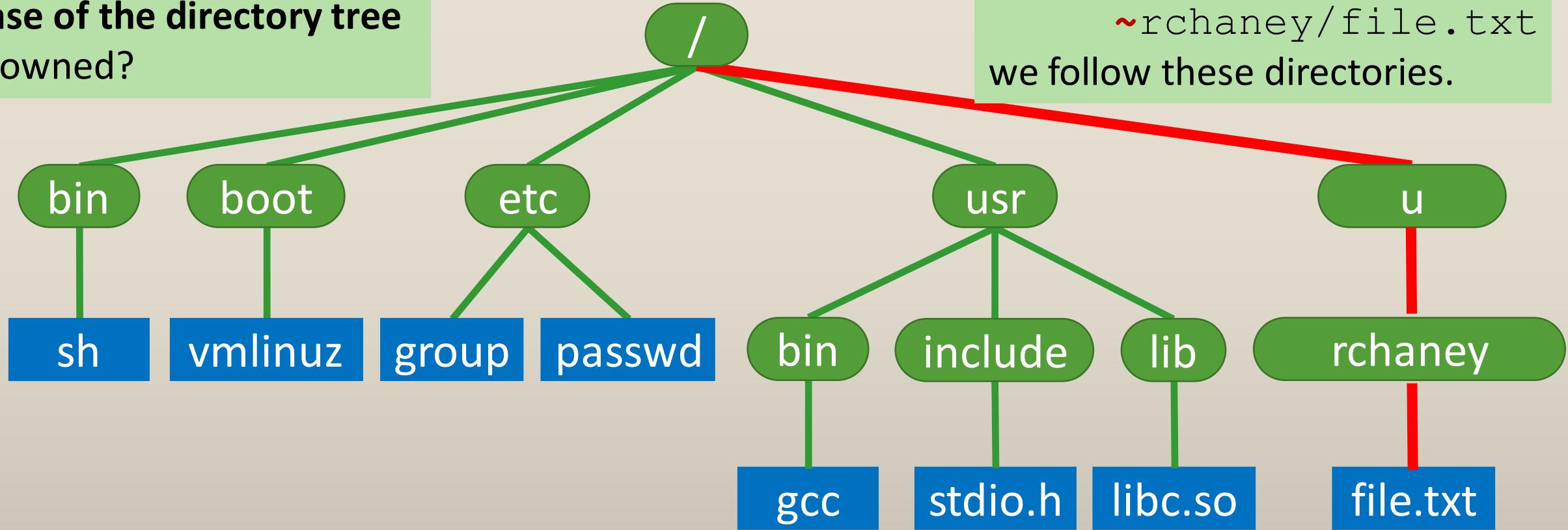


You will see this on an exam.

File Hierarchy

By whom do you think the **base of the directory tree** is owned?

If we want to follow the PATH to `~rchaney/file.txt` we follow these directories.



The **PATH** to the file.txt file is:
`/u/rchaney/file.txt`

File Types

Directory	d	rwxr-xr-x
Symbolic link	l	rwxrwxrwx
Named pipe	p	rw-rw----
Socket	s	rwxrwx---
Device special	b	rw-rw----
	c	rw-rw----
Regular file	-	rwxr--r--

You can see a description of the different file types by reading the man page on the `file` command.



Indicates the file *type*

When you do a '`ls -l`' in a directory, you will see a long listing showing these attributes (among other things).



Everything is a File

On the UNIX file system, **everything is a file.**

- Files are files.
- Directories are files.
- Devices are files.
- Processes are files.



The data structure that makes up a file on a UNIX file system is called an **inode**.

Contents of an inode

inode

- The size of the file
- Device ID.
- The User ID of the owner
- The Group ID of the owner
- The file mode (*permissions*)
- Additional system and user flags to control access
- Timestamps telling when the inode was last modified, and last accessed.
- A link count telling how many *hard* links point to the inode.
- *Pointers to the disk blocks that store the file's contents.*

What is conspicuous by its absence here?



Directory and inodes



Directory

Name & inode: numbers_10.txt **24601**

...

Name & inode: numbers_10.txt_LINK1 **24601**

Name & inode: numbers_10.txt_LINK2 **24601**

Name & inode: numbers_10.txt_LINK3 **24601**

A directory is a file whose data maps the names and inodes for files contained in the directory.

The data for a file can have multiple names.

Here, we have a file (inode) that has 4 entries in the directory.

Thus, it has 4 names. They all point to the same inode. They are hardlinks.

Since the inode contains the device id, hardlinks cannot span devices (volumes or logical drives).

File

inode: **24601**

Size: 30

uid: 50255

gid: 10993

Device id: 801h/2049d

Mode: _____

...

Data blocks: _____

Enter the Symbolic Link

Directory

Name & inode: numbers_10.txt

24601

...

Name & reference: **numbers_10.txt_SYM1** ./numbers_10.txt

- In a directory, a **symbolic link** is not mapped to an inode, but to a file path. That path can be on the same or different volume.
- **Unlike hardlinks, symbolic links** (sometimes called soft-links), can point to directories.
- A symbolic link can point to a non-existent file/directory.

File

inode: **24601**

Size: 30

uid: 50255

gid: 10993

Device id: _____

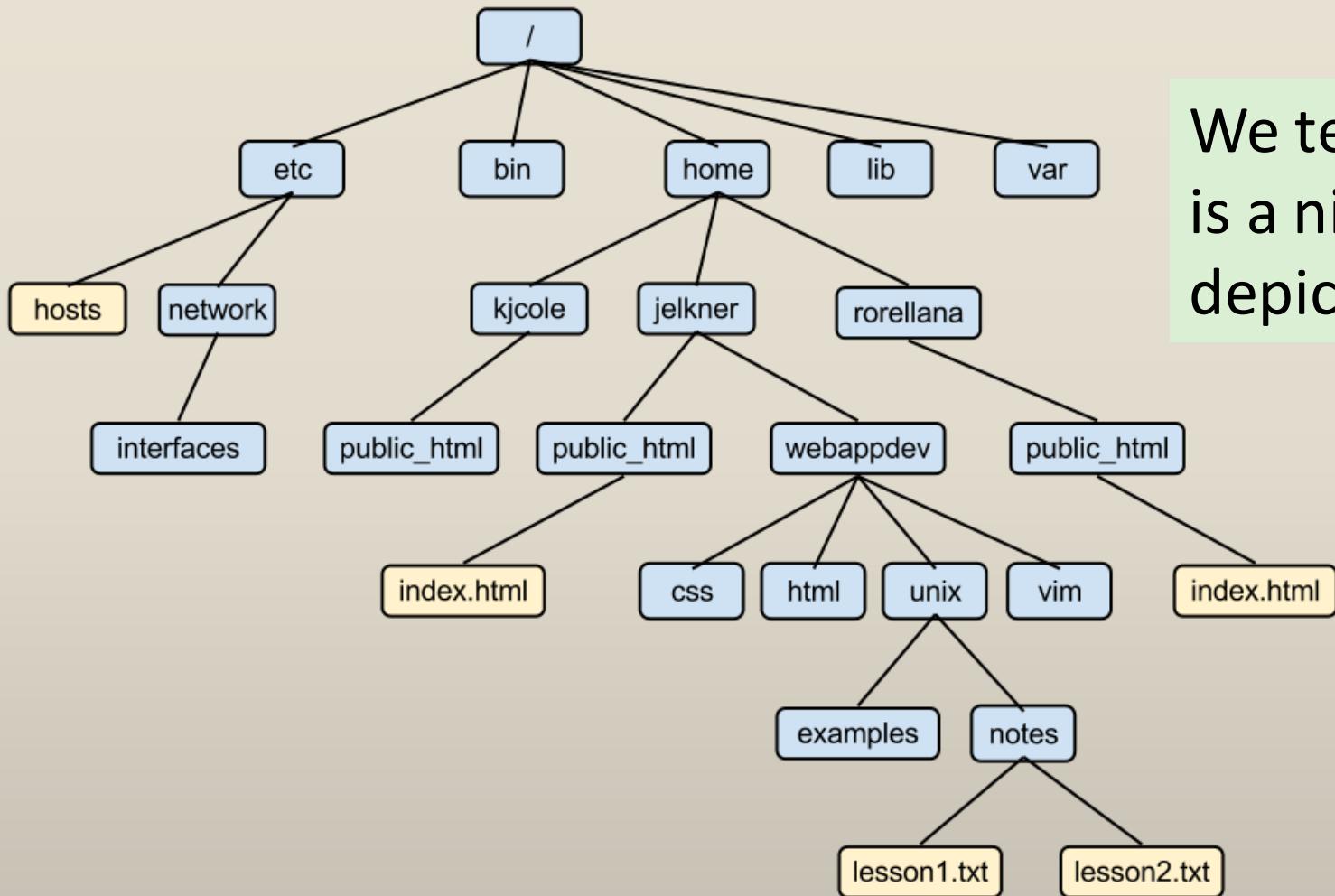
Mode: _____

...

Data blocks: _____



What We Tell You



We tell you that the UNIX file system is a nice neat tree and (usually) depict it as one.

Ahhh...



What We Don't Tell You

With all the links and symbolic links, neat and clean is a bit generous.



You Have Home and Current Working Directories



- Mentioned in the slide about a user's ID was a **home directory**.
- This is the directory in which the default shell will begin execution.
- **You can change your current directory using the 'cd' command.**
- The directory within the file system tree in which your shell is currently located is called your **current working directory** (or **present working directory**).
- When using your shell, the **~ (tilde)** can be used to represent your home directory.
 - To you, my home directory is `~rchaney`
 - Mark's home directory is `~mpj` (but he has not enabled access to his home directory)

Remember the
`pwd` command?

Everything is a File, part deux

Universality of I/O:

The same set of system calls are used to perform I/O on (nearly) all types of files, **including devices.**



Universality of I/O

The system calls (`open()`, `read()`, `write()`, `close()`, ...) are used to perform I/O on (nearly) **all types of files**, including devices.

A program employing these system calls will work on any type of file.

The kernel essentially provides a **single file type: a sequential stream of bytes**, which, in the case of disk files, disks, and tape devices, can be randomly accessed using the `lseek()` system call.



Standard I/O

We will spend a LOT of time talking about standard I/O.

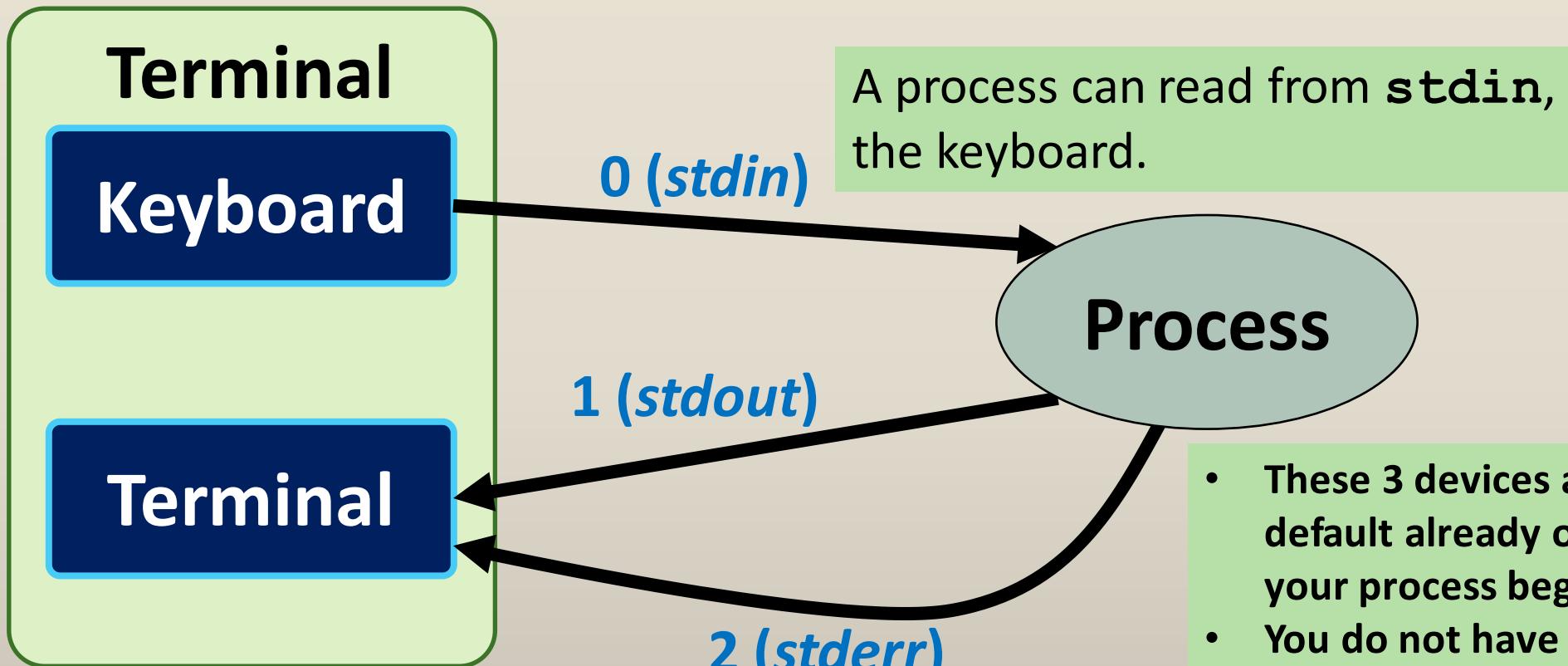
Standard I/O are the terminal keyboard input and terminal display.

There are 3 standard I/O devices (with defaults):

1. **stdin** standard in – the terminal keyboard input
2. **stdout** standard output – the terminal display
3. **stderr** standard error – also the terminal display



Standard I/O



- These 3 devices are by default already open when your process begins.
- You do not have to explicitly open them.
- You also do not have to close them before your process terminates.



These 3 devices are by default already open when your process begins. You do not have to explicitly open them.

Standard I/O

These 3 devices are by default already open when your process begins. You do not have to explicitly open them.

<u>FD</u>	<u>Macro</u>	<u>Stream</u>	<u>Device</u>
0	STDIN_FILENO	<i>stdin</i>	keyboard
1	STDOUT_FILENO	<i>stdout</i>	terminal
2	STDERR_FILENO	<i>stderr</i>	terminal

FD stands for file descriptor.

These 3 devices are by default already open when your process begins. You do not have to explicitly open them.

These 3 devices are by default already open when your process begins. You do not have to explicitly open them.

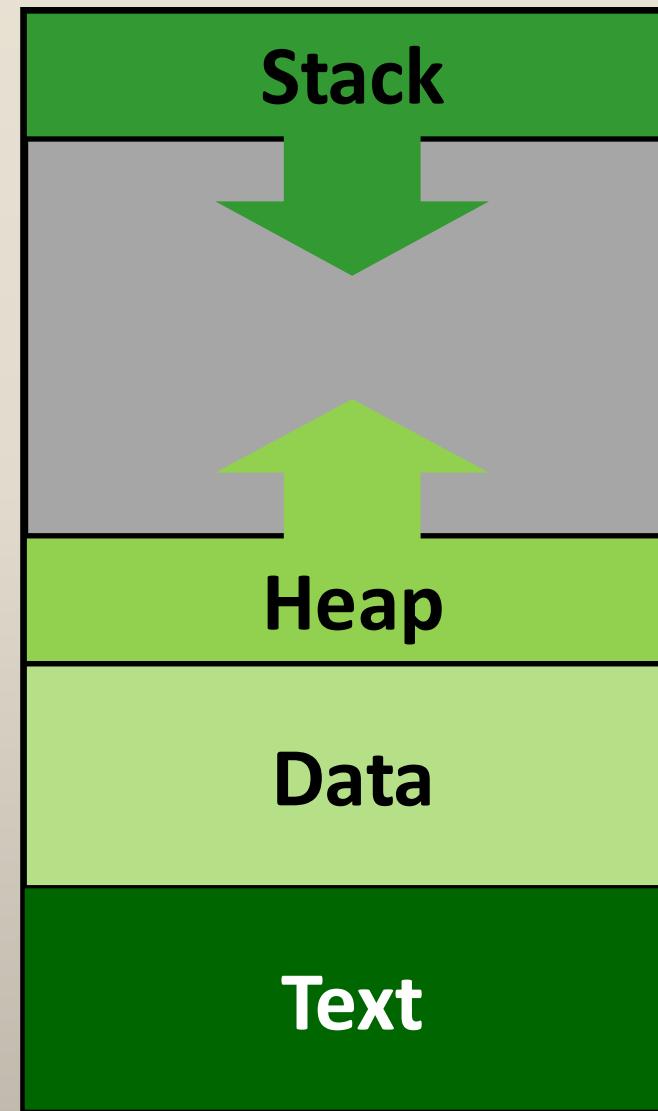
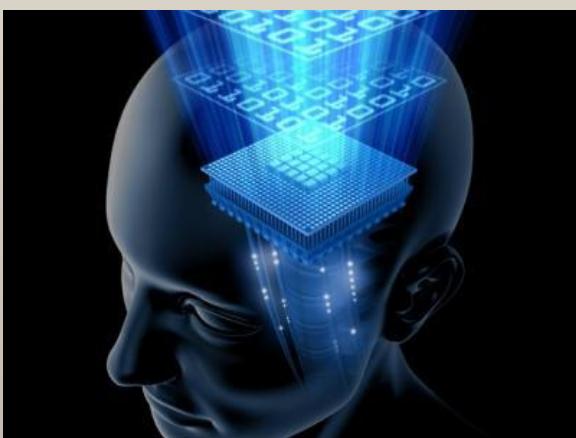
I'm trying to emphasize something.



Processes Memory Layout

A process is **an instance of a running program.**

You will see this on an exam.



Higher Addresses

Lower Addresses

Processes User and Group IDs

A process has an owner and a group, just like a file has an owner and a group.

Real user ID and Real group ID

A new process *inherits* these IDs from its parent process.

Effective user ID and Effective group ID

These are the ids under which a process is currently running.

Supplementary group IDs

Other information we won't be covering



The init process

When booting the system, the kernel creates a special process called `init`, the “**parent of all processes**.”

- The `init` process is a daemon process that continues running until the system is shut down.



Sometimes the `init` process is not actually called `init`. For example, on our server it is called `systemd`. However, for our purposes, we are going to call it `init` – the **parent of all processes**.

Dæmon Processes

Daemon or Dæmon or (wrongly) Demon

A dæmon process is a special-purpose process that is created and handled by the system in the same way as other processes, but which can be distinguished by the following characteristics:

- It is **long-lived**. A dæmon process is often started at system boot and remains in existence until the system is shut down.
- It runs in the background and has **no controlling terminal** from which it can read input or to which it can write output.
 - Sometimes called a service.

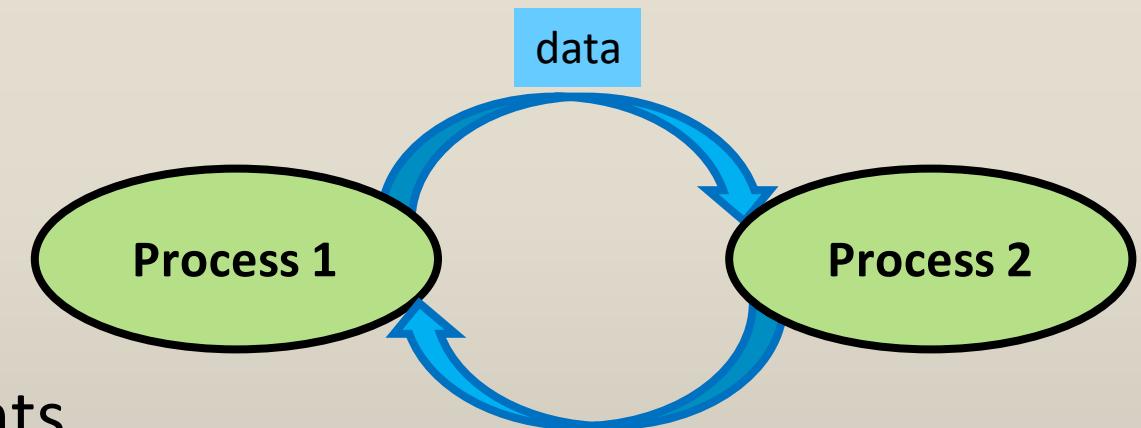


The combination character in dæmon is called a [ligature](#).

Inter-process Communication

We will be studying a lot of IPC

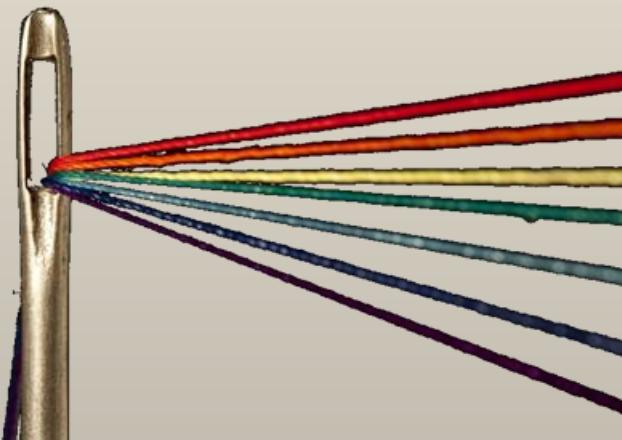
- Signals
- Pipes and FIFOs
- Sockets
- Semaphores
- Shared memory
- File monitoring events



Threads

In addition to using communication between processes, we'll also be doing some **multi-threaded** development.

Threads are like processes in some ways, but you can achieve **parallelism** within a **single process**.



Client-Server Architecture

A client-server application is one that is broken into two component processes:

- **a client**, which asks the server to carry out some service by sending it a request message; and
- **a server**, which examines the client's request, performs appropriate actions, and then sends a response message back to the client.





Systems Programming Concepts

Kernel:

The central software that manages and allocates computer resources (i.e., the CPU, RAM, and devices).



Systems Programming Concepts

System programming makes use of a layered architecture. **At the bottom of the programming layer is the kernel.**



Kernel

Hardware



Systems Programming Concepts

A system call is a controlled entry point into the kernel.

System Calls

Kernel

Hardware



Systems Programming Concepts

Some library functions are layered on top of system calls, just easier to use wrappers around systems calls.

Library Functions

System Calls

Kernel

Hardware



Systems Programming Concepts

Examples of applications are things like your web browser, Microsoft Excel, an email client, the shell.

Applications

Library Functions

System Calls

Kernel

Hardware



Systems Programming Concepts

You program
in here



Applications

Library Functions

System Calls

Kernel

Hardware

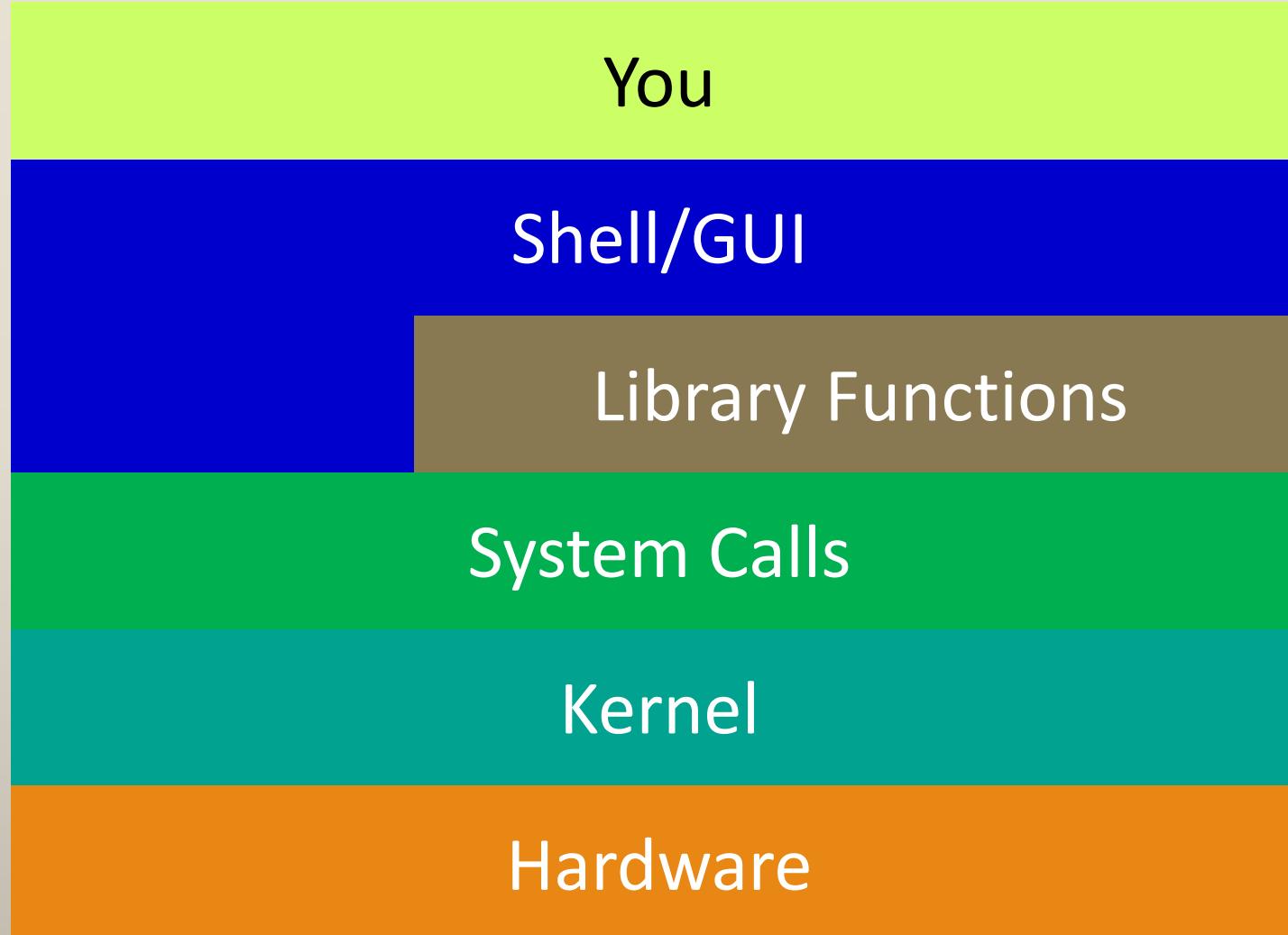
You operate
in here



Interacting with the System

A shell is a special-purpose program designed to read commands typed by a user and execute appropriate programs in response to those commands

```
Using username "chaneyr".  
chaneyr@os-class.engr.oregonstate.edu's password:  
Last login: Wed Jan 7 07:51:16 2015 from 10-162-141-69.wireless.oregonstate.edu  
This system is strictly for use by faculty, students, and staff of  
the College of Engineering, Oregon State University.  
Unauthorized access is prohibited - violators will be prosecuted.  
Use should be consistent with the OSU Acceptable Use Policy  
as well as College of Engineering policies and guidelines.  
Refer to http://engr.oregonstate.edu/computing/faqs/coe\_swp/index.html  
  
Quotas are used for home directories, incoming email, and printing.  
For details, check:  
http://engr.oregonstate.edu/computing/faqs/quotas.html  
  
If you have any problems with this machine, please mail support@engr.orst.edu  
  
(~) chaneyr@os-class 11:31 AM ~
```



System Calls

A **system call** is a **controlled entry point into the kernel**, allowing a process to request that the kernel perform some action on the process's behalf.

- The kernel makes a range of services accessible to programs via the system call application programming interface (API).
- These services include, creating a new process, performing I/O, and creating a pipe for interprocess communication.

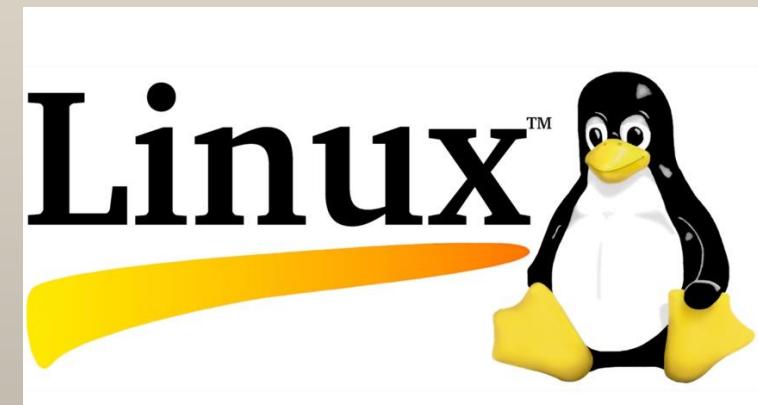
Every time you perform I/O,
you are making system calls.

System Calls

- A system call changes the processor state from user mode to kernel mode, so that the CPU can access protected kernel memory.
- The set of system calls is fixed. Each **system call is identified by a unique number.**
- Each system call may have a set of arguments that specify information to be transferred from user space to kernel space and vice versa.

Executing a System Call

1. The application program makes a system call by **invoking a wrapper function in the C library.**
2. The wrapper function must make all of the system call arguments available to the system call trap-handling routine. The wrapper function copies the arguments to these registers.



Executing a System Call

3. Since all system calls enter the kernel in the same way, the kernel needs some method of identifying the system call. **The wrapper function copies the system call number into a specific CPU register.**
4. The wrapper function executes a trap machine instruction, causing the processor to **switch from user mode to kernel mode** and execute code at a specific memory location the system's **trap vector**.

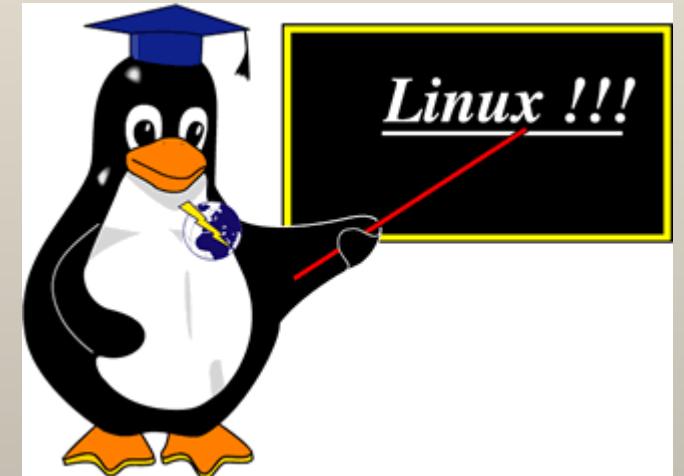


Executing a System Call

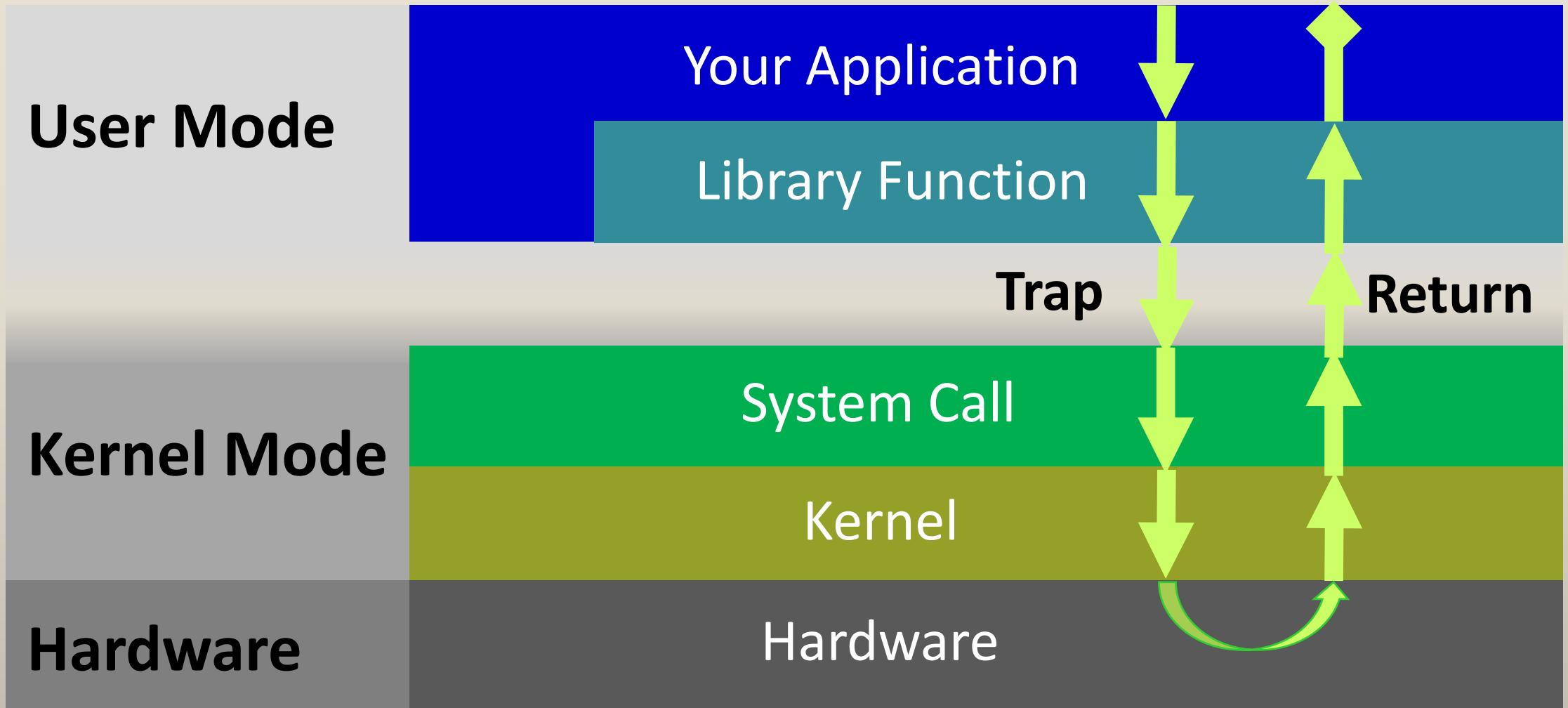
5. In response to the trap, the **kernel invokes its `system_call()` routine to handle the trap**. This handler:
 - a. Saves register values onto the kernel stack.
 - b. Checks the validity of the system call number.
 - c. Invokes the appropriate system call service routine, found by using the system call number to index a table of all system call service routines. Finally, the service routine returns a result status to the `system_call()` routine.
 - d. Restores register values from the kernel stack and places the system call return value on the stack.
 - e. Returns to the wrapper function, returning the processor to user mode.

Executing a System Call

6. If the return value of the system call service routine indicates an error, **the wrapper function sets the global variable `errno` using this value**. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

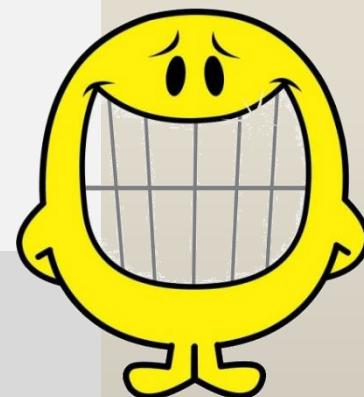


Executing a System Call



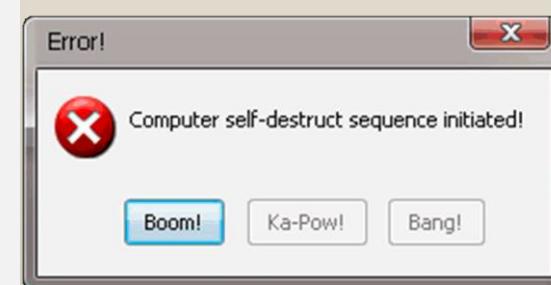
Executing Library Functions

- A library function is simply one of the many/MANY of functions that are contained in the standard C library.
- **Many library functions don't make use of system calls** (the string manipulation functions for example).
- **Some library functions are layered on top of system calls.**
 - The `fopen()` library function uses the `open()` system call to actually open/create a file.
- **Frequently, library functions are designed to provide a more caller-happy interface** than the basic system calls.
 - The `malloc()` and `free()` functions perform bookkeeping tasks that make them an easier way to manage memory than the underlying `brk()` system call.



Handling Errors from System and Library Calls

- Almost every system call and library function **returns some type of status**, indicating whether the call succeeded or failed.
- **The status value should always be checked to see if the call succeeded or failed.**
- Many hours can be frittered away because a check was not made on the status return of a system call or library function.
- The man page for each system call documents the possible return values of the call, showing which return value(s) indicate an error.



Handling Errors from System and Library Calls

- When a system call fails, it sets the global integer variable **errno** to a positive value that identifies the specific error.
- The <errno.h> header file provides a declaration of errno, as well as a set of constants for the various error numbers.
- The **perror()** function prints the string pointed to by its msg argument, followed by a message corresponding to the current value of errno.

NAME

perror - print a system error message

SYNOPSIS

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

```
void perror(const char *s);
```

```
#include <errno.h>
```

System Data Types



- Various implementation **data types are represented using standard C types, for example, process IDs, user IDs, and file offsets.**
- Although it is possible to use the C primitive data types, such as `int` and `long`, to declare variables for the information, this reduces portability across UNIX systems.
- The sizes of these primitive types vary across UNIX implementations
 - **A `long` may be 4 bytes on one system and 8 bytes on another.**
 - A process ID might be an `int` on one system but a `long` on another.
- Even a single UNIX implementation the types used to represent information may differ between releases of the implementation.
 - Examples on Linux are user and group IDs. On Linux 2.2 and earlier, these values were represented in 16 bits. On Linux 2.4 and later, they are 32-bit values.

System Data Types

Data Type	SUSv3 type requirement	Description
<code>clock_t</code>	integer or real-floating	System time in clock ticks
<code>uid_t</code>	integer	Numeric user identifier
<code>gid_t</code>	integer	Numeric group identifier
<code>mode_t</code>	integer	File permissions and type
<code>off_t</code>	signed integer	File offset or size
<code>pid_t</code>	signed integer	Process ID, process group ID, or session ID
<code>sig_atomic_t</code>	integer	Data type that can be atomically accessed
<code>size_t</code>	unsigned integer	Size of an object in bytes
<code>ssize_t</code>	signed integer	Byte count or (negative) error indication
<code>time_t</code>	integer or real-floating	Calendar time in seconds since the Epoch

```
struct stat {  
    dev_t      st_dev;      /* ID of device containing file */  
    ino_t      st_ino;      /* inode number */  
    mode_t     st_mode;     /* protection */  
    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;   /* blocksize for file system I/O */  
    blkcnt_t   st_blocks;   /* number of 512B blocks allocated */  
    time_t     st_atime;    /* time of last access */  
    time_t     st_mtime;    /* time of last modification */  
    time_t     st_ctime;    /* time of last status change */  
};
```

System Data Types

NAME

`lseek - move the read/write file offset`

SYNOPSIS

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

```
off_t lseek(int fildes, off_t offset, int whence);
```

What is the type of the `offset` argument to the `lseek()` function?

What is the return type from the `lseek()` function?

`function shall set the file offset for the open file description associated with the file descriptor fildes, as follows:`

- * If `whence` is `SEEK_SET`, the file offset shall be set to `offset` bytes.
- * If `whence` is `SEEK_CUR`, the file offset shall be set to its current location plus `offset`.
- * If `whence` is `SEEK_END`, the file offset shall be set to the size of the file plus `offset`.

System Data Types

NAME

chmod, fchmod - change permissions of a file

SYNOPSIS

```
#include <sys/stat.h>
```

```
int chmod(const char *path, mode_t mode);  
int fchmod(int fd, mode_t mode);
```

What is the type of the mode argument to
the chmod() and fchmod() functions?

System Data Types

NAME

getpid, getppid - get process identification

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t getpid(void);
pid_t getppid(void);
```

What is the return type from
the getpid() function?

DESCRIPTION

getpid() returns the process ID of the calling process. (This is often used by routines that generate unique temporary filenames.)

getppid() returns the process ID of the parent of the calling process.

ERRORS

These functions are always successful.

Why would these functions
ALWAYS successfully return?



VIKINGS™

CS 333

Intro to Operating Systems
TLPI 4: File I/O - The Universal I/O Model



PORTLAND STATE™

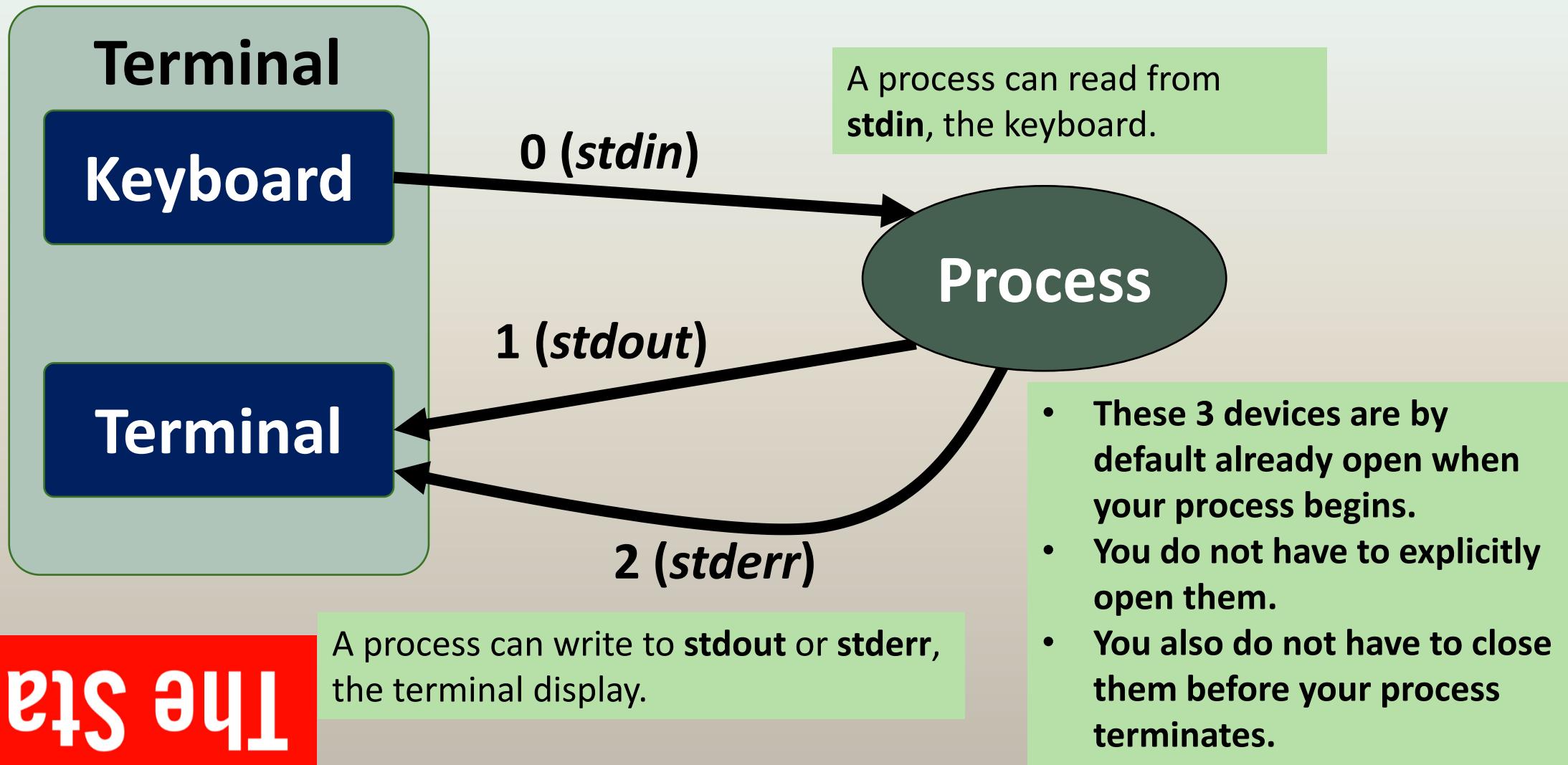
Everything is a File

Universality of I/O:

The same set of system calls are used to perform I/O on all types of files, **including devices.**



Standard I/O



Standard I/O

FD	POSIX
0	STDIN_FILENO
1	STDOUT_FILENO
2	STDERR_FILENO

The file descriptors (FD) **you've heard so much about.**

Use the POSIX mnemonics whenever possible.

Streams
stdin
stdout
stderr

Device
keyboard
display
display



The device with which the FD/stream is associated.

You've been using these, with fgets() and such. You'll continue to use these if you are **printing diagnostic output or output to the terminal.**

Opening a File

- The 3 devices that are **Standard I/O** are (by default) open when your process begins.
- Sometimes, you need to open other files, for either input or output. It may be an existing file, or you may be creating a new file.
- So far, we've used `fopen()` for this purpose.
- But, that is the ***streams*** interface.
- How do we get to the **file descriptor** (fd)?



The open () Call



NAME

open, ~~creat~~ - open and possibly create a file or device

SYNOPSIS

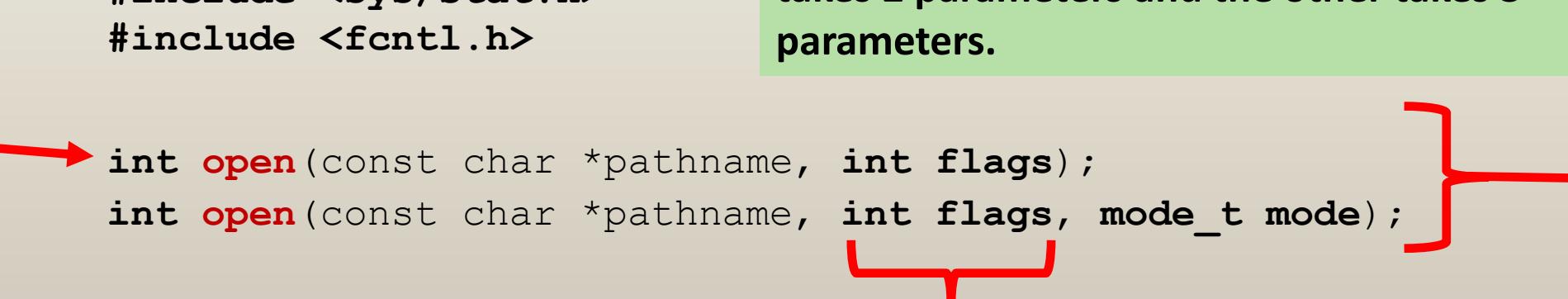
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Notice that there are 2 different calls, one takes 2 parameters and the other takes 3 parameters.

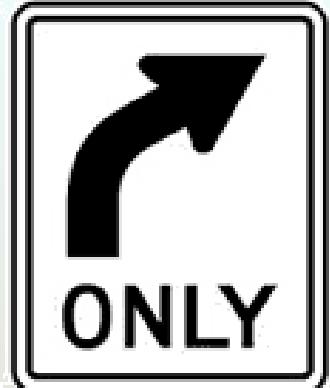
Returns an **int**.
Returns a -1 if an error occurred.
The returned value is a **file descriptor**.

This is straight out of the man page for open `man 2 open`.



The flags parameter must include one of the following access modes: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**.

The argument flags **must** include **one** of the following access modes:



- `O_RDONLY`
- `O_WRONLY`
- `O_RDWR`

Trying to use (`O_RDONLY` | `O_WRONLY`) in place of `O_RDWR` is an error.

Open for read and write.

These request opening the file read-only, write-only, or read/write mode.

In addition, zero or more file **creation flags** the file status flags can be **bitwise-or'd** in flags.

- O_CREAT
- O_TRUNC
- O_APPEND

If it does not exist, create it.

If it does exist, truncate it to zero bytes.

When opening an existing file, open it in append mode.
Before every `write()`, the file offset is positioned at the end of the file.

```
int openFlags =  
    O_CREAT | O_WRONLY | O_TRUNC;
```

Bitwise or, not logical or.



O_EXCL

Ensure that **this call creates the file**: if this flag is specified in conjunction with O_CREAT, and pathname **already exists, then open () will fail.**

If O_CREAT and O_EXCL are used with symbolic links, the links are not followed. If *pathname* is a symbolic link, open () just fails.

I like to ask about the O_EXCL flag on exams.



```
int open(const char *pathname, int flags, mode_t mode);
```

The **mode** parameter to `open()` specifies the access permissions to set **in case a new file is created.**

The permissions of the created file are:

`(mode & ~umask)`.



Your `umask` is an odd thing. You probably want to investigate the man page for `umask()`.

I wonder if it would be a good idea to ask for a description of the `umask` on an exam?

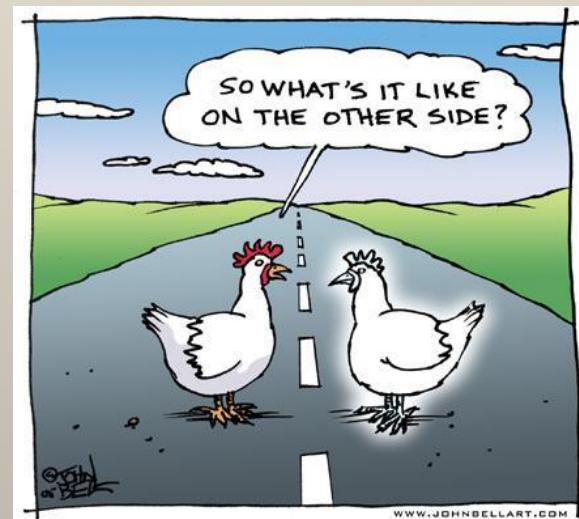
Users – Groups – Other

For the purposes of access and protection, **Unix divides the world into three categories:**

- **User** – this is you.
- **Group** – think of this as your “buds”
- **Other** – this is everyone else.

For each of the 3 categories, there are 3 permissions:

- **read**
- **write**
- **execute**



Mode (aka Permissions) for a File

- Think of the permissions on a file as a group of 0's and 1's.
- You'll have a group of three 0's/1's for each of the 3 categories (user-group-other).
- The collection of the **3 bits** for each category is an **octal value**.
 - Remember octal requires 3 bits to represent a value, unlike hexadecimal, which requires 4 bits.

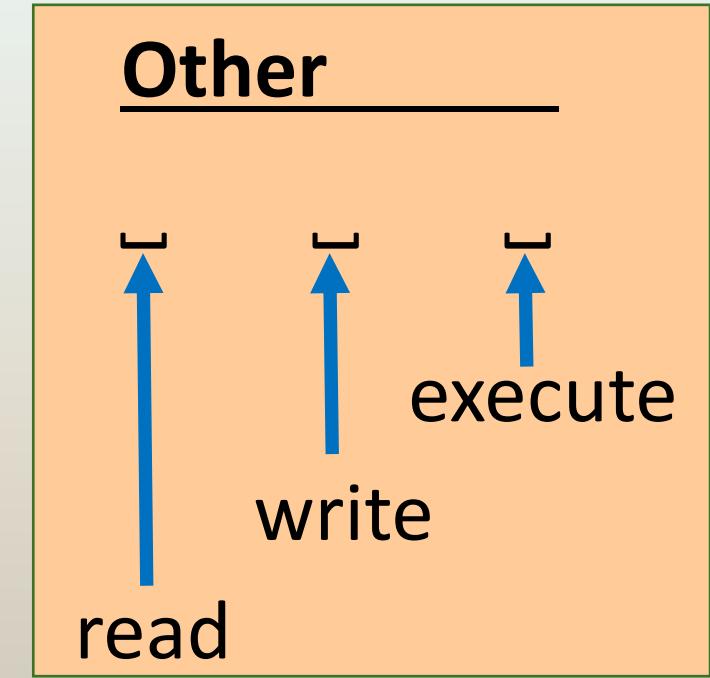
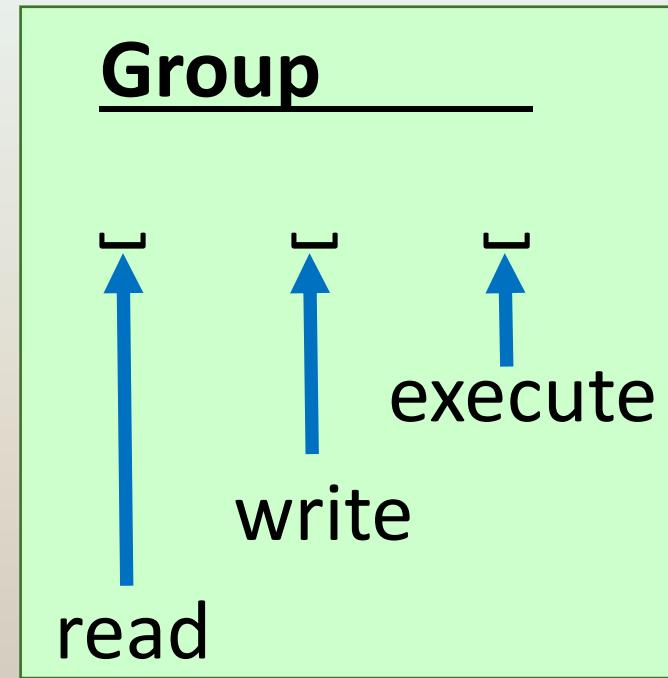
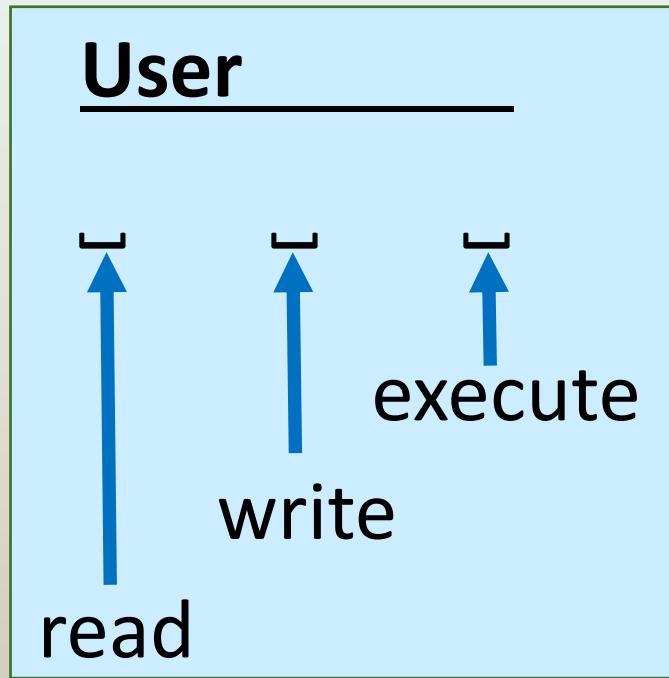
The original systems on which UNIX was developed (DEC PDPs) used octal instead of hex for most things.

Mode (aka Permissions) for a File

Octal Value	Binary Representation
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111



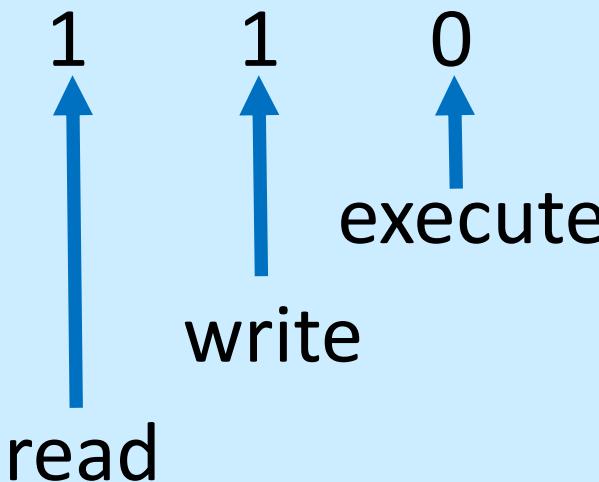
Mode (aka Permissions) for a File



Each _ represents a 0 or a 1, indicating if that permission (read-write-execute) is enabled for that category.

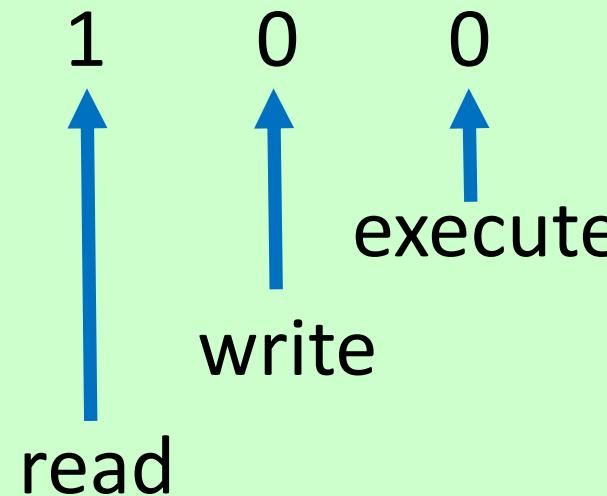
Mode (aka Permissions) for a File

User



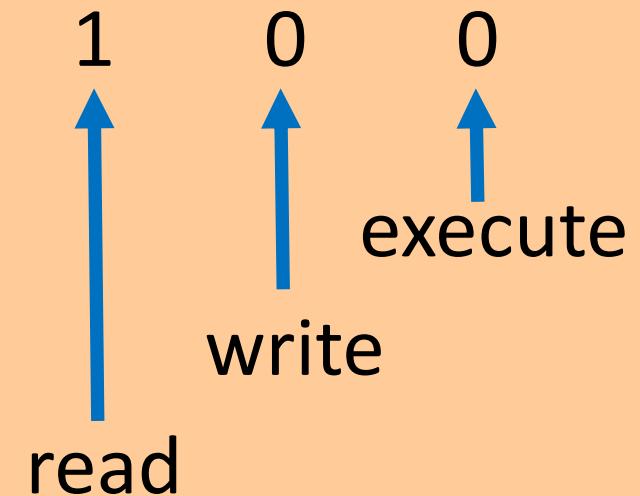
The *user* bits = 6

Group



The *group* bits = 4

Other



The *other* bits = 4

The file permissions shown here are **644**, allowing the user to read and write the file, group members can read the file, and others can only read the file.

Mode (aka Permissions) for a File

User

1	1	1	execute
↑	↑	↑	
read	write		

Group

1	0	1	execute
↑	↑	↑	
read	write		

Other

1	0	0	execute
↑	↑	↑	
read	write		

The *user* bits = 7

The *group* bits = 5

The *other* bits = 4

The file permissions shown here are **754**, allowing the user to read, write, and execute the file, group members can read and execute the file, and others can only read the file.



When setting the mode, the following macros can be bitwise-or'd together.

Notice this is a shorthand for the following 3 bitwise-or'd together.

- S_IRWXU **00700** user (file owner) has read, write and execute permission
 - S_IRUSR **00400** user has read permission
 - S_IWUSR **00200** user has write permission
 - S_IXUSR **00100** user has execute permission
-
- S_IRWXG **00070** group has read, write and execute permission
 - S_IRGRP **00040** group has read permission
 - S_IWGRP **00020** group has write permission
 - S_IXGRP **00010** group has execute permission
-
- S_IRWXO **00007** others have read, write and execute permission
 - S_IROTH **00004** others have read permission
 - S_IWOTH **00002** others have write permission
 - S_IXOTH **00001** others have execute permission

```
mode_t filePerms =
```

```
S_IRUSR  
|_S_IWUSR  
|_S_IRGRP  
|_S_IWGRP  
|_S_IROTH  
|_S_IWOTH;  
/* rw-rw-rw- */
```



User bits

Group bits

Other bits

These are **bitwise-or'd** together.

These are grouped into user, group, and other.



chmod Calculator

Owner	Group	Public
Read <input type="checkbox"/>	Read <input type="checkbox"/>	Read <input type="checkbox"/>
Write <input type="checkbox"/>	Write <input type="checkbox"/>	Write <input type="checkbox"/>
Execute <input type="checkbox"/>	Execute <input type="checkbox"/>	Execute <input type="checkbox"/>

Linux

0666

-rw-rw-rw-

Permissions:**Chmod Calculator**

Chmod Calculator is a free utility to calculate the numeric (octal) or symbolic value for a set of file or folder permissions in Linux servers.

How to use

Check the desired boxes or directly enter a valid numeric value (e.g. 777) or symbolic notation (e.g. rwxrwxrwx) to see its value in other formats.

File Permissions

File permissions in Linux file system are managed in three distinct user classes: user/owner, group and others/public. Each class can have read, write and execute permissions. File permission can be represented in a symbolic or numeric (octal) format.

<https://chmod-calculator.com/>

```

// Open existing file for reading
fd = open("startup.txt", O_RDONLY);
if (fd == -1)
    perror("open failed: " "startup.txt");

// Open new or existing file for reading and writing, truncating to zero
// bytes; file permissions read+write for owner, nothing for all others
fd = open("myfile.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
if (fd == -1)
    perror("open failed: " "myfile.txt");

// Open new or existing file for writing;
// writes should always append to end of file
fd = open("w.log", O_WRONLY | O_CREAT | O_APPEND,
           S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
if (fd == -1)
    perror("open failed: " "w_log.log");

```



NAME

read - read from a file descriptor

Read bytes from an **open file descriptor.**

SYNOPSIS

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

Notice `ssize_t` vs `size_t`.

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

Returns the **number of bytes actually read, zero indicates end of file.**

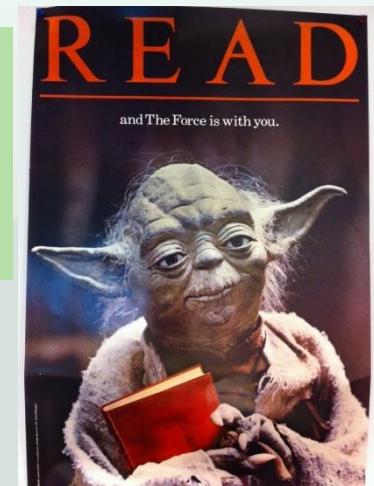
On error, -1 is returned, and `errno` is set appropriately.

I love asking this question on exams.

```
#define MAX_READ 20
char buff[MAX_READ];

if (-1 == read(STDIN_FILENO, buff, MAX_READ))
    perror("read failed");
```

Notice the excellent use of the `#define` macro to make the code easier to maintain.



Read from **standard-in**, up to 20 bytes, into the character buffer called `buff`.

Write bytes to an **open file descriptor**.

NAME

write - write to a file descriptor

SYNOPSIS

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

Notice ssize_t vs size_t.

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

Returns the **number of bytes actually written**, zero indicates nothing was written.

On error, -1 is returned, and errno is set appropriately

I love asking this question on exams.

Close an open file
descriptor.

NAME

close - close a file descriptor

SYNOPSIS

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

```
int close(int fd);
```



Random Access to a File

- Typically, we read a file from the beginning to the end, access is sequential through the file.
 - The FPM (**File Position Marker**, aka **file pointer**) is advanced each time we read a chunk of data from the file.
- Sometimes we need to access random portions of the file. We read the file non-sequentially.
 - We accomplish moving to random places in the file using the **lseek()** function.
 - The **lseek()** function is used to change the **byte offset** into the file (the FPM).



seek

Change the current bytes offset into a file.

NAME

lseek - reposition read/write file offset

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
```

```
off_t lseek(int fd, off_t offset
            , int whence);
```



Allows you to jump to a different location in the file, including jump backwards.

Whence:

The valid values for the whence parameter

- **SEEK_SET** - The offset is set to offset bytes.
- **SEEK_CUR** - The offset is set to its current location **plus** offset bytes (this could move you backward in the file).
- **SEEK_END** - The offset is set to the size of the file **plus** offset bytes (you can seek past the end of the file).

I always ask about the whence flags on exams.



You can create a holey file.

Random Access to a File

```
off_t loc = lseek(in_fd, 0, SEEK_END);
```

```
off_t loc = lseek(in_fd, -10, SEEK_END);
```

```
off_t loc = lseek(in_fd, 1000, SEEK_END);
```



Create a holey file?

```
off_t loc = lseek(in_fd, 100, SEEK_CUR);
```

```
off_t loc = lseek(in_fd, -100, SEEK_CUR);
```

```
off_t loc = lseek(in_fd, 0, SEEK_SET);
```

```
off_t loc = lseek(in_fd, 100, SEEK_SET);
```

Move to an absolute
byte offset from the
beginning of the file.

Standard Input

- A very important thing to know about standard input, once you read it, **the data you read has been consumed** from standard input.
- You cannot get that back (as you might were you reading from a file on disk).
- Imagine you are typing from the keyboard as standard input. You don't want to have to type everything over again.
- **You cannot seek backward on standard input.** **the Standard**

Some Examples

The following slides show some short examples of the use of some of the functions we've just covered.

- my_cat3
- my_tee
- my_seek

The above examples can be found in
`~rchaney/Classes/cs333/src/cat`



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

#ifndef MAX_BUFFER_LEN
#define MAX_BUFFER_LEN 1024
#endif // MAX_BUFFER_LEN

int main(int argc, char *argv[])
{
    char buffer[MAX_BUFFER_LEN];
    ssize_t bytes_read;

    while ((bytes_read = read(STDIN_FILENO, buffer, MAX_BUFFER_LEN)) > 0) {
        write(STDOUT_FILENO, buffer, bytes_read);
    }

    return(0);
}
```

Notice that there are no calls to open or close files.

This reads data from the already open, file descriptor 0, and writes that data to the already open file descriptor 1.

Read from the STDIN_FILENO file descriptor.



Write to the STDOUT_FILENO file descriptor.

Can be found in ~rchaney/Classes/cs333/src/cat/my_cat3.c

Examples how you can run my_cat3

```
./my_cat3 < passwd  
cat passwd | my_cat3
```

Redirection

Inter-process
communication via a pipe.



You can look at the source code.

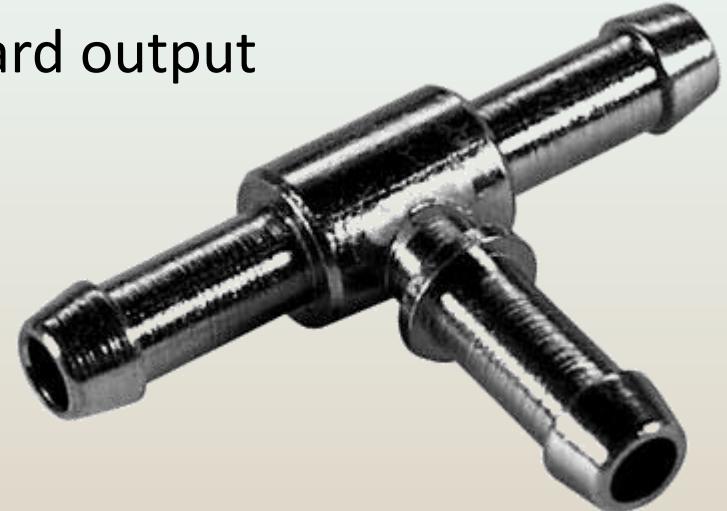
You may also want to look at the source to `my_cat4.c`.

The `my_cat4.c` program will allow you to have multiple files on the command line.

```
./my_cat4 passwd my_cat1.c my_cat2.c
```

NAME

tee - read from standard input and write to standard output and files



SYNOPSIS

tee [OPTION]... [FILE]...

DESCRIPTION

Copy standard input to each FILE, and also to standard output.

```
cat /etc/passwd | tee JUNK1 JUNK2 JUNK3 JUNK4
```

This command will read from standard input, write that data to standard output, and write the data into each of the files named on the command line.



Loop through all the file names listed on the command line (argv).

```
for (i = 1; i < argc; i++) {  
    int fd = open(argv[i], O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR | S_IRGRP);  
    if (fd >= 0) {  
        tee_fd[i] = fd;  
    }  
    else {  
        // Could not open the file for output.  
        tee_fd[i] = -1;  
        fprintf(stderr, "Could not open file for output %s\n", argv[i]);  
    }  
}
```

Open a file given on the command line.

```
cat /etc/passwd | ./my_tee JUNK1 JUNK2 JUNK3 JUNK4
```

Can be found in ~rchaney/Classes/cs333/src/cat/my_tee.c

```
while ((bytes_read = read(STDIN_FILENO, buffer, MAX_BUFFER_LEN)) > 0) {  
    write(STDOUT_FILENO, buffer, bytes_read);  
    for (i = 1; i < argc; i++) {  
        if (tee_fd[i] >= 0) {  
            write(tee_fd[i], buffer, bytes_read);  
        }  
    }  
}
```

1. Loops through all the data arriving from file descriptor STDIN_FILENO (0),
2. Writes the data to STDOUT_FILENO, and
3. Writes the to each of the file descriptors from files listed on the command line.

```
cat passwd | ./my_tee JUNK1 JUNK2 JUNK3 JUNK4
```



```
for (i = 1; i < argc; i++) {  
    if (tee_fd[i] >= 0) {  
        close(tee_fd[i]);  
    }  
}
```

Loops through all the file descriptors open from the command line close all the files that were opened.

```
cat passwd | ./my_tee JUNK1 JUNK2 JUNK3 JUNK4
```



Seek into a File

The following example takes 3 command line parameters:

1. a file name,
2. an offset into the file, and
3. the number of bytes from the file to read.

Once the file is opened, the program does an **`lseek()`** to the specified offset from the beginning of the file, and then reads the specified number of bytes from the file and writes them to `stdout`.



```
file_name = argv[1];
seek_to = strtol(argv[2], NULL, 10);
num_bytes = strtol(argv[3], NULL, 10);
```

The name of the file.

The offset to seek into the file.

The number of bytes of the file to read.

```
fd = open(file_name, O_RDONLY);
if (fd >= 0) {
    int oset = lseek(fd, seek_to, SEEK_SET);
    if (oset < 0) {
        fprintf(stderr, "lseek failed\n");
        perror("lseek failed");
    }
    bytes_read = read(fd, buffer, num_bytes);
    if (bytes_read > 0) {
        write(STDOUT_FILENO, buffer, bytes_read);
    }
}
else {
    fprintf(stderr, "could not open file to read: %s\n", file_name);
}
```

Use `lseek()` to go to an absolute byte offset from the beginning of the file.

Read bytes from that location in the file.

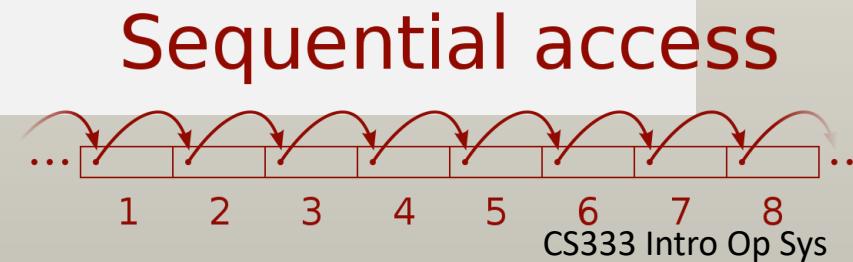
Write the bytes to stdout.

Can be found in `~rchaney/Classes/cs333/src/cat/my_seek.c`



Sequential Programming

- So far, you've probably mostly studied sequential programming.
 - One thing happens at a time.
 - The next thing to happen is “my” next statement or instruction.
 - The processing is very predictable and stable.
 - You can, fairly easily, predict how long it will take to complete a task.



Parallelism

- A parallel program is one that uses multiple hardware units in order to complete computation more quickly.
- Different parts of the computation are delegated to computational units that execute at the same time (in parallel), so that results may be delivered earlier than if the computation had been performed sequentially.
- Throw hardware at it!



You can fold, stuff, and stamp a letter and envelope in about 15 seconds.

Problem

- You have 5,000 letters/envelopes/stamps to process.
- This will take you nearly 21 hours to complete.
- You don't want it to take 21 hours.
- Find friends to help.
 - Throw hardware at it.



Divide the work so that the three sub tasks that can be done in parallel (an assembly line).

Build a pipeline.

Fold



Stuff



Stamp



- Throw hardware at it!

A software pipeline consists of a chain of processing elements, arranged so that the output of each element is the input of the next.

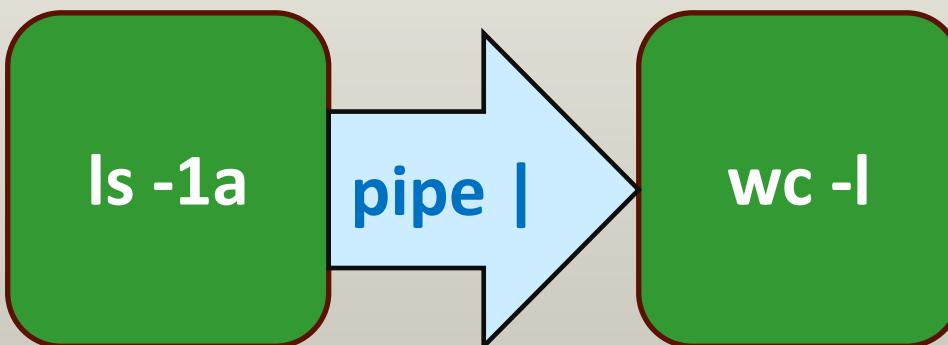
- The elements of a pipeline may be called filters.
- Usually some amount of buffering is provided between consecutive elements.
- Typically, a pipeline is linear and one-directional.



A Unix command line pipeline is built connecting processes using the **pipe** character, the ‘|’ (vertical bar above the return key).

The output from one process is sent as the input to the next process in the pipeline.

This is part of Unix **stdio**, the standard output (**stdout**) is connected to the standard input (**stdin**) of the next process.



A simple Unix pipeline:

```
ls -1a | wc -l
```

Shows how many files are in a directory.
The output from `ls` is used as input to the `wc` command.

A Bigger Command Pipeline in Unix

The `count_procs.bash` script in `~rchaney/bin`

```
ps -Ao user
| grep -vs '^root\|^gdm'
| sort
| uniq -c
| sort -nr
| head
```

It represents 6 processes, all chained together using *pipes* for inter-process communication.

- We can easily build a pipeline as a Unix command line.
- Here we use 6 individual commands connected through pipes with Standard I/O.
- Output from a left-hand command is input to a right-hand command.

Divide the work so that portions can be completed in parallel.



Another way to divide the work. Divide the initial stack of paper into sixths.

You could even pipeline within each task!

Concurrency

- Concurrency is a program-structuring technique in which there are **multiple streams of execution**.
- The streams of execution may execute "at the same time"; the user may perceive the streams as operating **interleaved** in time.
- Whether they actually execute at the same time is detail.
 - A concurrent program can execute on a single processor through interleaved execution, or on multiple physical processors.
- When we talk about (or implement) concurrency, we bring in the idea of **shared resources**.
 - Vital aspects of the computation that are **accessed by all the streams of execution**.

Concurrency

: Correctly and efficiently manage access to shared resources.

Stream of execution 1

Stream of execution 2

Stream of execution 3

Stream of execution 4

Stream of execution 5

Stream of execution 6



There is a single stack of unfolded letters, a single stack of folded letters, and a single roll of stamps.

These are (carefully) shared between multiple streams of execution.

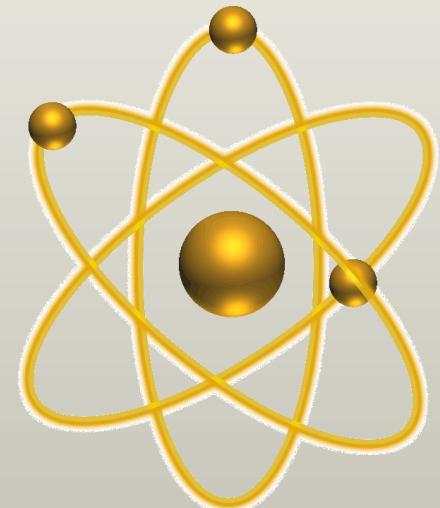
Race Conditions

- A **race condition** is a situation where **the result produced by two (or more) processes or threads (*streams of execution*) operating on a shared resource depends in an unexpected way on the relative order in which the processes are scheduled on the CPU(s).**



Atomicity

- Atomic operations provide **functions or instructions that execute without interruption.**
- In assembly language, atomic operations complete within a single clock cycle.
- For a system function, the **kernel guarantees that all of the steps in the system call are completed as a single operation**, without being interrupted by another process or thread.



Atomicity and Race Conditions

A

\$1,000

Read Balance

\$100

Read deposit

Add balance and deposit

Write new balance

B

\$1,000

Read Balance

\$1,000,000

Read deposit

Add balance and deposit

Write new balance

What you want at the end is \$1,001,100



Are you satisfied with this result?

A

Read Balance

\$1,000

Read Deposit

\$100

Add Balance
and Deposit

\$1,100

Write new
balance

\$1,100



\$1,100

B

\$1,000

Read Balance

\$1,000,000

Read Deposit

\$1,001,000

Add Balance
and Deposit

\$1,001,000

Write new
balance

Make sure each function completes atomically.



A

Read Balance: \$1,000

Read deposit: \$100

Add balance and deposit: \$1,100

Write new balance: \$1,100

B

Read Balance: \$1,100

Read deposit: 1,000,000

Add balance and deposit: \$1,001,100

Write new balance: **\$1,001,100**

Played only for entertainment purposes

O_EXCL flat on the open () call.

- Ensure that the call to create a file **actually creates the file**.
- If this flag is specified in conjunction with O_CREAT, and pathname already exists, then open () will fail.
- Using the terms we just introduced, **atomically** create and open the file on the file system.
- If the calling process is not the process to create the file, fail.
 - **The shared resource is the file system.**

Duplicating File Descriptors

Using the shell I/O redirection syntax **2>&1** informs the shell that you want to connect standard error (file descriptor 2) to the same place to which standard output (file descriptor 1) is going. The following command will send both standard output and standard error to the file results.log:

```
$ ./myscript > results.log 2>&1
```

The shell accomplishes the redirection of standard error into standard out by duplicating file descriptor 2 so that it refers to the same open file descriptor as file descriptor 1.

DUPLICATE

Order matters. The following will **not** work:

```
./myscript 2>&1 > results.log
```

The Open File Table

The **kernel** maintains a system-wide table of all open file descriptors.

- Often referred to as the **open file table**
 - Entries called **open file handles**.
 - An open file descriptor stores information related to a file open in a process, including:
 - the current file offset;
 - status flags specified when opening the file;
 - the file access mode; and
 - a reference to the `inode` object for the file.
- 
- OpenTable®**

The Open File Table



So far, we had a 1-to-1 correspondence between an open file descriptor and a file (or device from `stdio`).

Sometimes, it is useful to have more than 1 file descriptor referencing a single file.

Two different file descriptors that refer to the same open file descriptor share a file offset value (through the kernel Open File Table).

- If the file offset is changed through one file descriptor, the change is visible through the other file descriptor.
- This applies both when the two file descriptors belong to the same process and when they belong to different processes.

NAME

dup, dup2 - duplicate an open file descriptor

SYNOPSIS

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

```
int dup(int oldfd);  
int dup2(int oldfd, int newfd);
```

I recommend you use `dup2()` over `dup()`.

- The `dup()` system call creates a copy of the file descriptor `oldfd`, using the **lowest-numbered unused** file descriptor for the new descriptor.
- The `dup2()` system call performs the same task as `dup()`, but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in `newfd`. If the file descriptor `newfd` was previously open, it is silently closed before being reused.

Use of dup2 ()

Your Process

File Desc	Name
0	stdin
1	stdout
2	stderr

Kernel shared with all processes

Your process starts with the stdio file descriptors open and allocated in the **kernel** Open File Table.

Use of dup2()

Your Process

File Descriptor Table	
File Desc	Name
0	stdin
1	stdout
2	stderr
3	ifd
4	ofd

Kernel shared with all processes

Open File Table			
Index	Offset	Flags	inode
0			
...			
23			
...			
73			
...			
119			
120			

Open 2 files, one for reading and one for writing (using the open() function).

Use of dup2 ()

Your Process

File Descriptor Table

File Desc	Name
0	stdin
1	stdout
2	stderr
3	ifd
4	ofd

Kernel shared with all processes

Open File Table

Index	Offset	Flags	inode
...			
23			
...			
73			
0	3		
...			
119			
120			

dup2 (3 ,

Use the dup2 () function to associate stdin (fd = 0) with the newly opened file for reading.

Use of dup2 ()

Your Process

File Descriptor Table	
File Desc	Name
0	stdin
1	stdout
2	stderr
3	ifd
4	ofd

Kernel shared with all processes

Open File Table			
Index	Offset	Flags	inode
...			
...			
73			
...			
119			
120			

`dup2(4, 1)`

Use the `dup2()` function to associate `stdout` (`fd = 1`) with the newly opened file for writing.

Use of dup2()

Your Process

File Descriptor Table

File Desc	Name
0	stdin
1	stdout
2	stderr
3	old file descriptor
4	ofd

Kernel shared with all processes

Open File Table

Index	Offset	Flags	inode
...			
...			
73			
...			
119			
120			

close(3)

You can now close the old file descriptor 3. This does not close file descriptor 1.

Use of dup2()

Your Process

File Descriptor Table	
File Desc	Name
0	stdin
1	stdout
2	stderr
	close(4)

Kernel shared with all processes

Open File Table			
Index	Offset	Flags	inode
...			
73			
...			
119			
120			

You can now close the old file descriptor 4. This does not close file descriptor 2.

Use of dup2()

Your Process

File Descriptor Table

File Desc	Name
0	stdin
1	stdout
2	stderr

Kernel shared with all processes

Open File Table

Index	Offset	Flags	inode
...			
...			
73			
...			
119			
120			

Now, when your process reads from stdin, it will in fact read from the file you opened for reading. When it writes to stdout, it will write to the file you opened for writing.

The dup2 () call



```
int dup2(int oldfd, int newfd);
```



dup2 () makes **newfd** be the copy of **oldfd**, closing **newfd** first if necessary:

- If **oldfd** is not a valid file descriptor, then the call fails, and **newfd** is not closed.
- If **oldfd** is a valid file descriptor, and **newfd** has the same value as **oldfd**, then dup2 () does nothing, and returns **newfd**.

The dup () call

```
int dup(int oldfd);
```



dup () uses the **lowest-numbered unused** descriptor for the new descriptor.

On success, dup () returns the **new** descriptor.

On error, -1 is returned, and errno is set appropriately.

I prefer dup2()

The dup3 () call

```
int dup3(int oldfd, int newfd, int flags);
```

dup3 () is the same as dup2 (), except:

- The caller can force the **close-on-exec** flag to be set for the new file descriptor by specifying **O_CLOEXEC** in the flags argument.
- If oldfd equals newfd, then dup3 () fails with the error EINVAL.

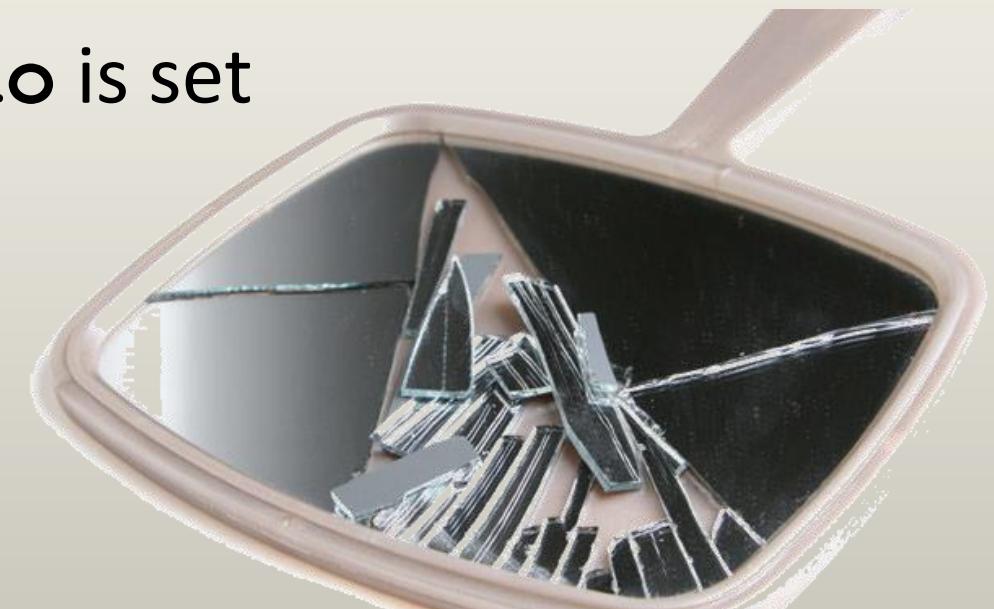


Rare special cases.

dup(), dup2(), and dup3(): Errors

On success, the `dup()`, `dup2()`, and `dup3()` system calls return the **new** descriptor.

On error, **-1** is returned, and **errno** is set appropriately.



The `pread()` and `pwrite()` calls

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

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
```

Returns **number of bytes read**, 0 on EOF, or -1 on error

```
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

Returns **number of bytes written**, or -1 on error

When you use the `read()` and `write()` calls on a file descriptor, the file offset is changed to reflect a different location in the file (the kernel does this automatically).

Sometimes it is useful to read from or write to a file and **not update the file offset** (such as in a multi-threaded application where multiple threads access a shared file in different locations).

Truncating a File to Length

```
#include <unistd.h>
#include <sys/types.h>

int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

Both return 0 on success, or -1 on error

- The `truncate()` and `ftruncate()` functions cause the regular file named by `path` or referenced by `fd` to be truncated to a size of **precisely length bytes**.
- If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as null bytes.
- The file offset is not changed.

Temporary Files

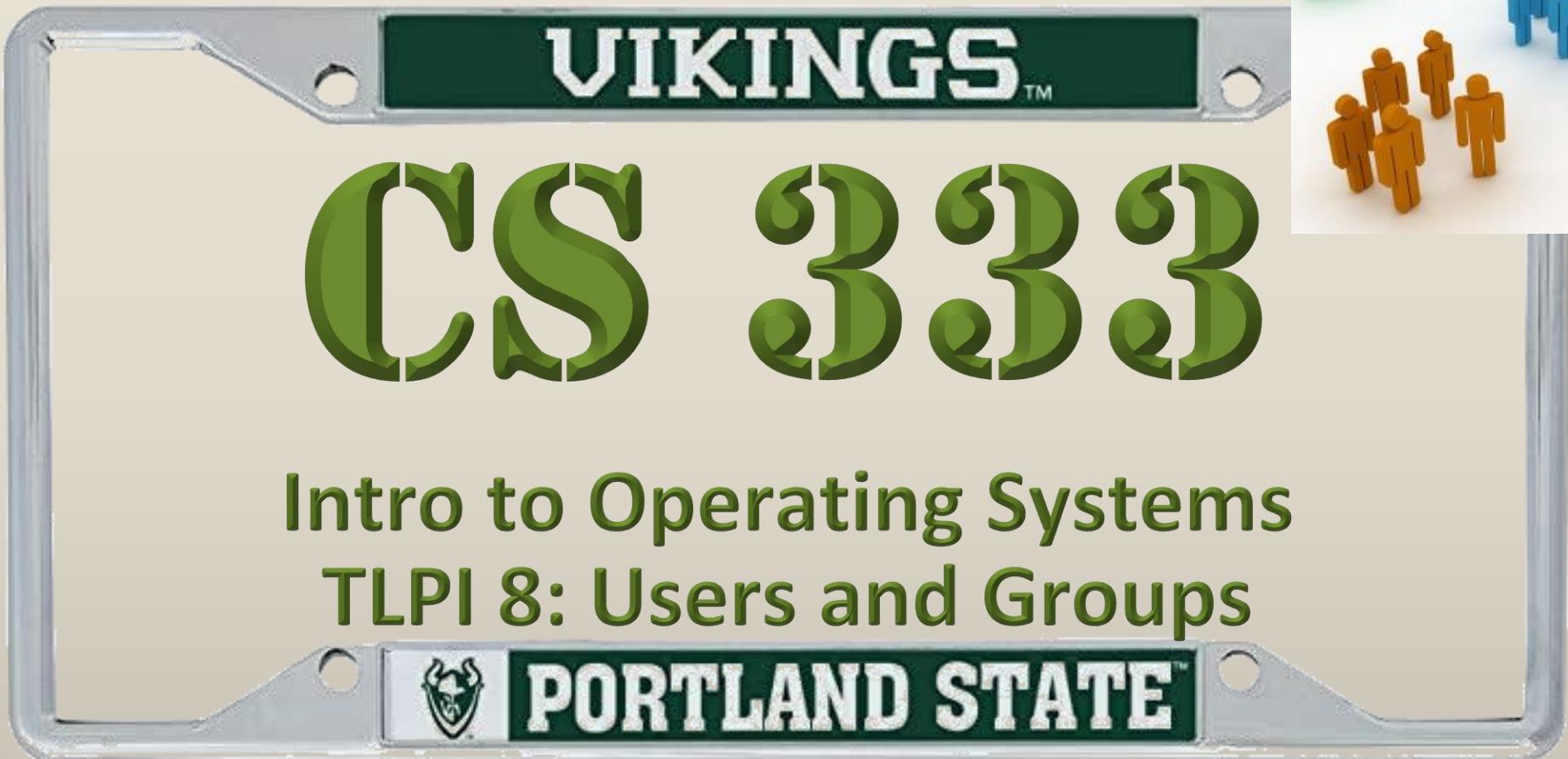
```
#include <stdio.h>

FILE *tmpfile(void);
```

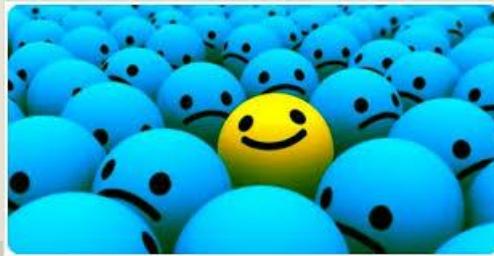
Returns file pointer on success, or NULL on error

Occasionally programs need to create temporary files that are used only while the program is running, **the files are automatically removed when the program has a normal termination.**

- For example, many compilers create temporary files during the compilation process (the C compiler may produce temporary pre-processed files or assembly language files).



Users and Groups



- Every user has a **unique login name** and an associated numeric **user identifier (UID)**.
- While it is very rare, it is possible to have duplicate UID entries in the /etc/passwd file.
- Users can belong to **one or more groups**.
- Each group also has a **unique name** and a **group identifier (GID)**.
- The **primary purpose** of user IDs and group IDs is to **determine ownership** of various system resources and to control the permissions granted to processes accessing those resources.

The Password File

The system password file, `/etc/passwd`, contains one line for each user account on the system.

```
rchaney:*:18781:300:Jesse Chaney:/u/rchaney:/bin/bash
```

Login name

User ID

Group ID

Comment,
sometimes called
the GECOS field.

Login shell

Home directory

Encrypted
password

Some early Unix systems at Bell Labs used **GECOS** machines for print spooling and various other services, so this field was added to carry information on a user's GECOS identity.

General **C**omprehensive **O**perating **S**ystem

The Password File

The typical format for the comment or **GECOS field** is a comma-delimited list with this order:

- The user's full name (or application name)
- Building and room number or contact person
- Office telephone number
- Home telephone number
- Any other contact information (pager number, fax, external e-mail address, etc.)



The Password File

If you are using a system such as **Network Information System (NIS)** or **Lightweight Directory Access Protocol (LDAP)** to distribute passwords in a network environment, part or all of this information resides on a remote system.

That's why you won't see your account entry in the `/etc/passwd` file on the Unix server.

LDAP - Lightweight Directory Access Protocol



The Shadow Password File

- Historically, **UNIX systems maintained all user information**, including the encrypted password, in /etc/passwd.
- As you can imagine, this **presented a security problem**.
- Since various unprivileged system utilities need to have read access to information in the password file (though not necessarily the password), it had to be made readable to all users.

Programs like who read the password file.



The Shadow Password File

- The shadow password file, `/etc/shadow`, was devised as a method of preventing such attacks.
- All of the **non-sensitive user information resides in the publicly readable password file**, while **encrypted passwords are maintained in the shadow password file** (and it has very limited access).

```
----- 1 root root 1.2K Oct 6 04:37 /etc/shadow
```



The Group File

- The set of groups to which a user belongs is defined by the combination of the group ID field in the user's password entry and the groups under which the user is listed in the group file.
- **In early UNIX implementations, a user could be a member of only one group at a time.**
- A user's initial group membership at login was determined by the group ID field of the password file and could be changed thereafter using the newgrp command



The Group File

- 4.2BSD introduced the concept of multiple simultaneous group memberships, which was later standardized under POSIX.
- Under this BSD schema, the group file listed the additional group memberships for each user.
- The **group file, /etc/group, contains one line for each group** in the system. Each line consists of four colon-separated fields



The Group File

The group file, `/etc/group`, contains one line for each group in the system. Each line consists of four colon-separated fields

`them:*:300:rchaney,mpj,dmcgrath,...`

Group name

Encrypted
password

Group ID

User List: This is a **comma-separated list of names of users** who are members of this group. (This list consists of usernames rather than user IDs because of the possibility user IDs are not unique in the password file. A group can have a LOT of members in it.

This is an entry from the `/etc/group` file from a different institution.

Retrieving User and Group Information

- Just because much of the information about the users and groups is maintained outside of the `/etc/passwd` and `/etc/group` files, it does not mean you cannot **fetch** the information *programmatically*.



Retrieving User Information

NAME

getpwent, endpwent - get password file entry

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwent(void);

void endpwent(void);
```

The `getpwent()` function returns a pointer to a structure containing the broken-out fields of a record from the password database (e.g., **the local password file `/etc/passwd`, NIS, and LDAP**). The first time `getpwent()` is called, it returns the first entry; thereafter, it returns successive entries.

Retrieving User Information

The `passwd` structure is defined in `<pwd.h>` as follows:

```
struct passwd {  
    char    *pw_name;          /* username */  
    char    *pw_passwd;        /* user password */  
    uid_t   pw_uid;           /* user ID */  
    gid_t   pw_gid;           /* group ID */  
    char    *pw_gecos;         /* user information */  
    char    *pw_dir;           /* home directory */  
    char    *pw_shell;         /* shell program */  
};
```

Retrieving User Information

```
struct passwd *p;
while((p = getpwent()) != NULL) {
    printf("%s:%s:%d:%d:%s:%s:%s\n"
           , p->pw_name
           , p->pw_passwd // This is probably not very useful.
           , p->pw_uid
           , p->pw_gid
           , p->pw_gecos
           , p->pw_dir
           , p->pw_shell);
}
endpwent();
```

Fetches an entry from
the password database.

This function is used to close the password database
after all processing has been performed.

Retrieving Group Information

NAME

getrent, setrent, endrent - get group file entry

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>

struct group *getrent(void);

void endrent(void);
```

The `getrent()` function returns a pointer to a structure containing the broken-out fields of a record in the group database (e.g., **the local group file `/etc/group`, NIS, and LDAP**). The first time `getrent()` is called, it returns the first entry; thereafter, it returns successive entries.

Retrieving Group Information

The group structure is defined in <grp.h> as follows:

```
struct group {
    char    *gr_name;          /* group name */
    char    *gr_passwd;        /* group password */
    gid_t   gr_gid;           /* group ID */
    char   **gr_mem;          /* group members */
};
```

Retrieving Group Information

```
while((g = getgrent()) != NULL) {
    printf("%s:%s:%d:"
           , g->gr_name
           , g->gr_passwd
           , g->gr_gid
           );
    for (i = 0; NULL != g->gr_mem[i] ; i++) {
        printf("%s"
               , (i == 0) ? "" : ", "
               , g->gr_mem[i]);
    }
    printf("\n");
}
endrent();
```

Fetches an entry from the group database.

This function is used to close the group database after all processing has been performed.

Not Retrieving User/Group Information

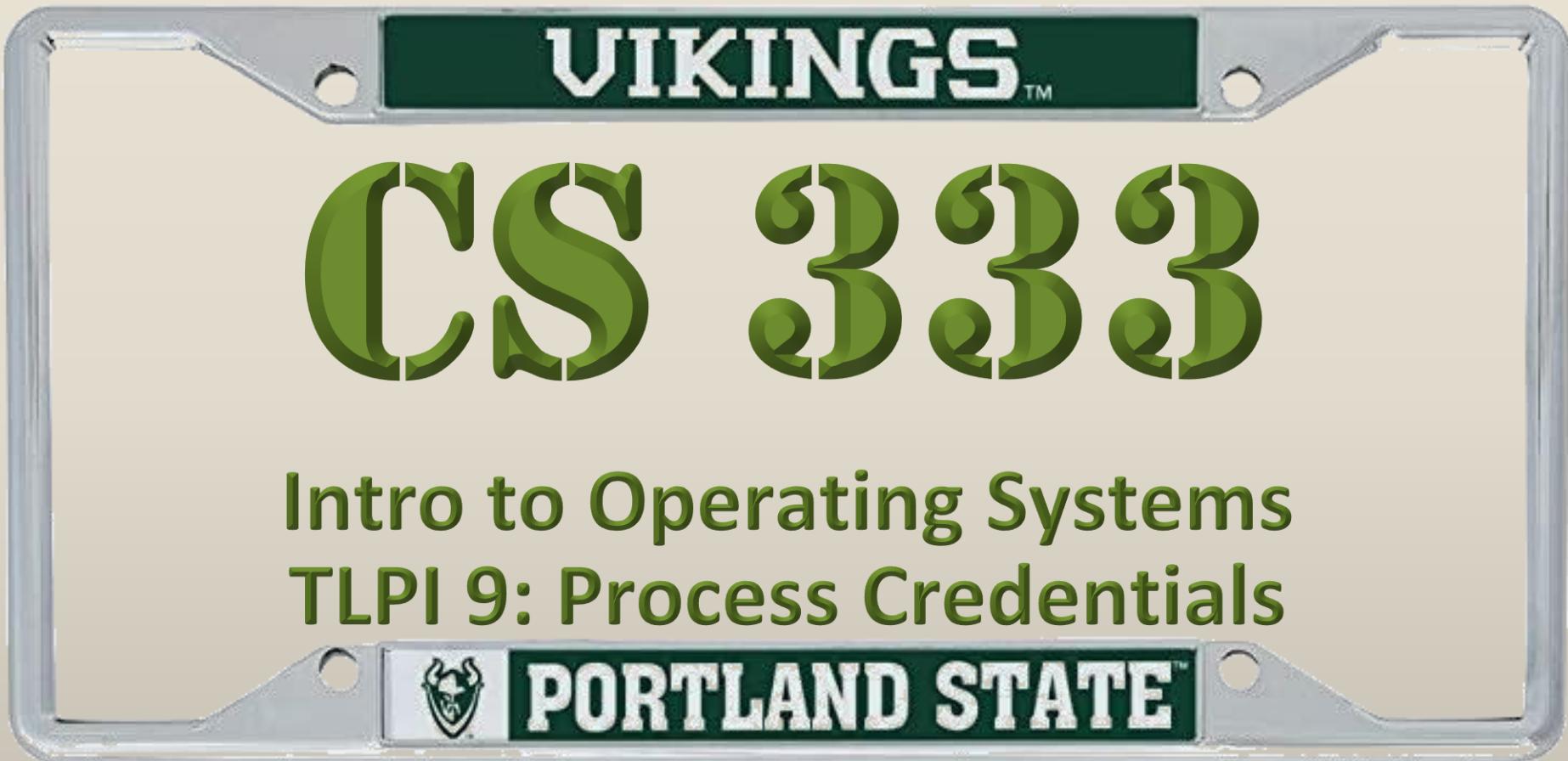
On some systems (such as ours), access to the list of users and groups through LDAP is tightly controlled.

You may not be able to access entries with `getpwent()` or `getgrent()`.

However, there are other things that can be done...

Ya just gotta be sly and have some good pointers.





A process is an instance of a running program.

A program is a file on the disk.

The file on disk has a **uid** and a **gid** associated with it, just like all the other files.

Process Credentials

Every process has a set of associated numeric user identifiers (UIDs) and group identifiers (GIDs). These are referred to as process credentials.

These identifiers are as follows:

- **real user ID and group ID;**
- **effective user ID and group ID;**
- saved set-user-ID and saved set-group-ID;
- file-system user ID and group ID (Linux-specific);
- supplementary group IDs.



Real vs Effective

The **real user ID** and **group ID** identify the user and group to which the process belongs when it is started.

When a new process is created, the **new process inherits** these identifiers from its parent process.

The **effective user ID and group ID** are used to determine the **permissions granted to a process when it tries** to perform various operations.



Set-User-ID and Set-Group-ID Programs

A set-user-ID (**setuid**) program allows a process to gain privileges it would not normally have, by setting the process' **effective** user ID to the same value as the user ID (owner) of the executable file.

If an executable file is owned by root (superuser) and has the set-user-ID permission bit enabled, then the process gains superuser privileges whenever that program is run, no matter which user starts the program.

```
-rwsr-xr-x. 1 root root 32K Aug 16 11:47 su*
--s--x--x. 1 root root 140K Jun 27 2018 sudo*
```



Set-User-ID and Set-Group-ID Programs

The same sort of thing applies to set-group-id (**setgid**) programs.

The ssh-agent program will always automatically run as the group nobody, no matter by whom it is started.

```
-rwxr-sr-x. 1 root mail      20K Nov 28 2017 lockfile*
---x--s--x. 1 root nobody 374K Apr 10 2018 ssh-agent*
```



Retrieving and Modifying Real and Effective IDs

- The `getuid()` and `getgid()` system calls return, respectively, the **real user ID** and **real group ID** of the calling process.
- The `geteuid()` and `getegid()` system calls perform the corresponding tasks for the **effective IDs**.
- The `setuid()` system call **changes the effective user ID**.
- The `setgid()` system call **changes the effective group ID**.

Modifying setuid/getgid from the Shell

You can use the command `chmod` to set or clear the setuid/setgid.

However, there are some restrictions.

As a mortal user, you very limited for how you can set the setuid/setgid bits on a file or within your process. You need to have superuser privileges to do most of the interesting/scary stuff.

Most systems disallow setuid/setgid bits to be set on any kind of shell script (or script in general). Only binary programs can have those bits set.



Intro to Operating Systems

TLPI 10: Time



Time

Within a program, we are often interested in two different kinds of time:

- **Real time:** This is the time as measured either from some standard point or from some fixed point in the life of a process. Measuring elapsed time is useful for a program that takes periodic actions or makes regular measurements from some external input device.
- **Process time:** This is the amount of CPU time used by a process. Measuring process time is useful for checking or optimizing the performance of a program or algorithm.

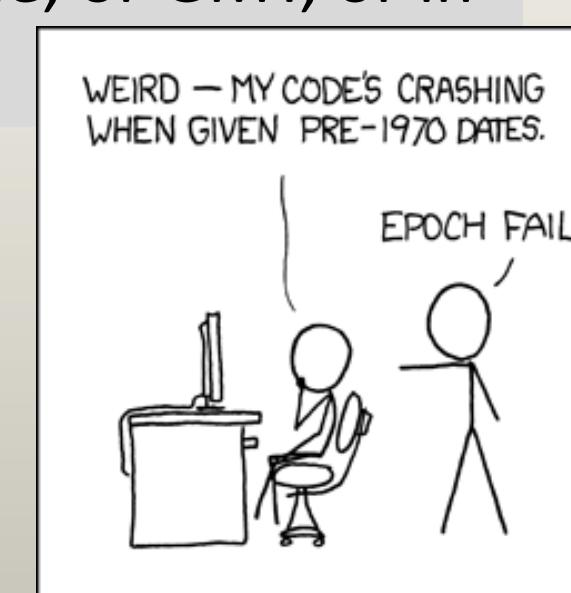


Calendar Time

- Regardless of geographic location of the system, UNIX systems represent time internally as a measure of **seconds since the Epoch**.
 - That is, since **midnight on the morning of 1 January 1970, Universal Coordinated Time**.
 - (UTC, previously known as Greenwich Mean Time, or GMT, or in the military as Zulu).

UNIX EPOCH
00:00:00 (UTC),
Thursday, 1 January 1970

I always like to ask when did the UNIX epoch begin.



Calendar Time

- Calendar time is stored in variables of type `time_t` (which is a **32-bit signed integer**).
- On 32-bit Linux systems, `time_t`, can represent dates in the range 13 December 1901 20:45:52 to **19 January 2038 03:14:07**.
- Many current 32-bit UNIX systems face a **theoretical Year 2038** problem, which they may encounter before 2038, if they do calculations based on dates in the future.
- This is also called the **Epochalypse**.

I always like to ask “What is the 2038 problem?”



Calendar Time

The `time()` system call returns the number of seconds since the Epoch.

```
#include <time.h>
```

```
time_t time(time_t *timep) ;
```

Returns number of seconds since the Epoch, or `(time_t) -1` on error



Calendar Time

The `gettimeofday()` system call returns the **calendar time** in the **buffer pointed to by `tv`**.



```
#include <sys/time.h>
```

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

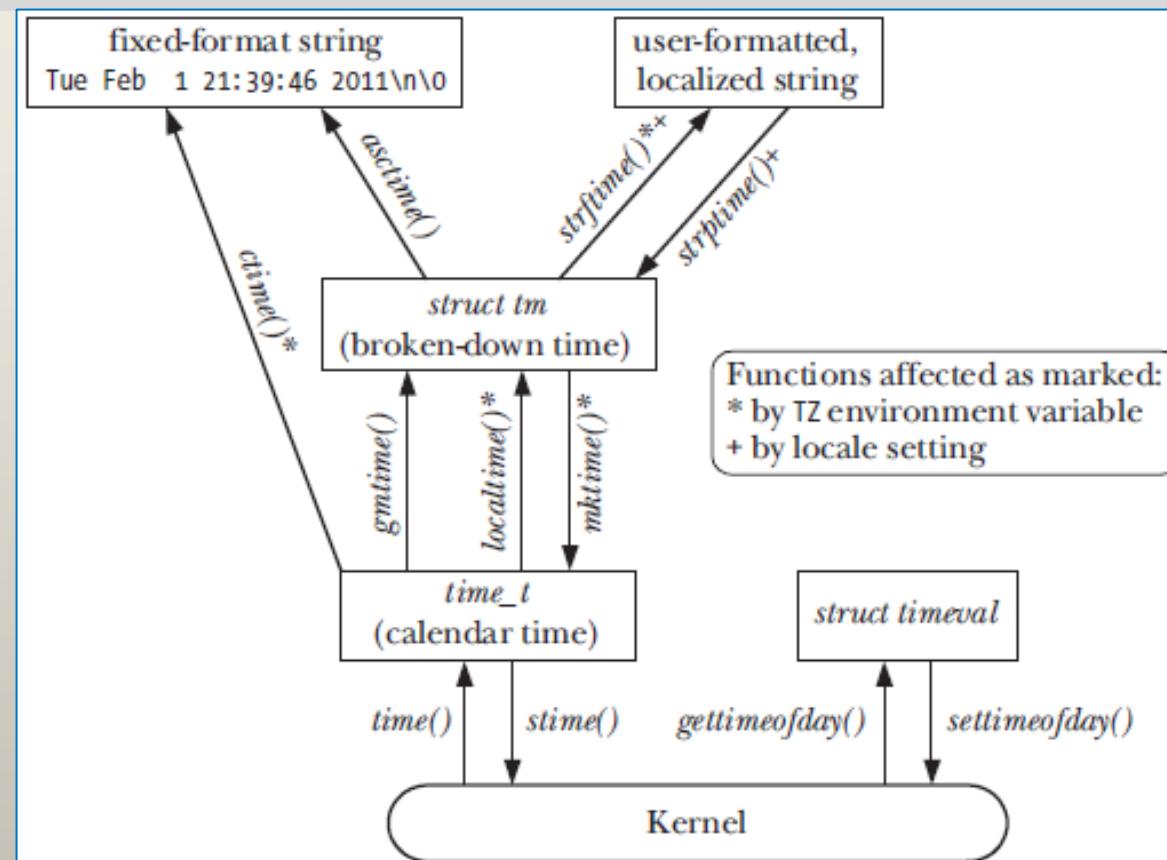
Returns 0 on success, or -1 on error

The `tv` argument is a pointer to a structure of the following form:

```
struct timeval {  
    time_t tv_sec; /* Seconds since 00:00:00, 1 Jan 1970 UTC */  
    suseconds_t tv_usec; /* Additional microseconds (long int) */  
};
```

Time-Conversion Functions

Speaking technically, there are **oodles** of functions for converting from the Unix internal time format into other time formats, including human readable forms.



Converting to a Simple Printable Form

The `ctime()` function provides a simple method of converting a `time_t` value into printable form.

```
#include <time.h>
char *ctime(const time_t *timep);
```

- Returns pointer to **statically allocated** string terminated by newline and \0 on success, or NULL on error.
- `ctime()` returns a 26-byte string containing the date and time in a standard format, as illustrated by the following example:

Wed Jun 8 14:22:34 2011



Broken-Down Time

The `gmtime()` and `localtime()` functions convert a `time_t` value into a so-called broken-down time. The broken-down time is placed in a **statically allocated structure** whose address is returned as the function result.

- The `gmtime()` function converts a calendar time into a broken-down time **corresponding to UTC**.
- The `localtime()` function **takes into account timezone and DST settings** to return a broken-down time corresponding to **the system's local time**.



Broken-Down Time

```
struct tm {  
    int tm_sec;      /* Seconds (0-60) */  
    int tm_min;      /* Minutes (0-59) */  
    int tm_hour;     /* Hours (0-23) */  
    int tm_mday;     /* Day of the month (1-31) */  
    int tm_mon;      /* Month (0-11) */  
    int tm_year;     /* Year since 1900 */  
    int tm_wday;     /* Day of the week (Sunday = 0) */  
    int tm_yday;     /* Day in the year (0-365; 1 Jan = 0) */  
    int tm_isdst;    /* Daylight saving time flag  
                      > 0: DST is in effect;  
                      = 0: DST is not effect;  
                      < 0: DST information not available */  
};
```



I like to ask “What is broken-down time?”

Displaying Time

The `strftime()` function formats the broken-down time `tm` according to the format specification format and places the result in the character array `s` of size `max`.

The format specification is a null-terminated string and may contain special character sequences called conversion specifications

NAME

`strftime` - format date and time

SYNOPSIS

```
#include <time.h>
```

```
size_t strftime(char *s, size_t max
               , const char *format
               , const struct tm *tm);
```



Displaying Time

Specifier	Description
%a	The abbreviated weekday name according to the current locale.
%A	The full weekday name according to the current locale.
%b	The abbreviated month name according to the current locale.
%B	The full month name according to the current locale.
%c	The preferred date and time representation for the current locale.
%C	The century number (year/100) as a 2-digit integer. (SU)
%d	The day of the month as a decimal number (range 01 to 31).
%D	Equivalent to %m/%d/%y. (Yecch—for Americans only. Americans should note that in other countries %d/%m/%y is rather common. This means that in international context this format is ambiguous and should not be used.) (SU)

Printable From Back into Broken-down Time

NAME

`strptime` - convert a string representation of time to a time `tm` structure

SYNOPSIS

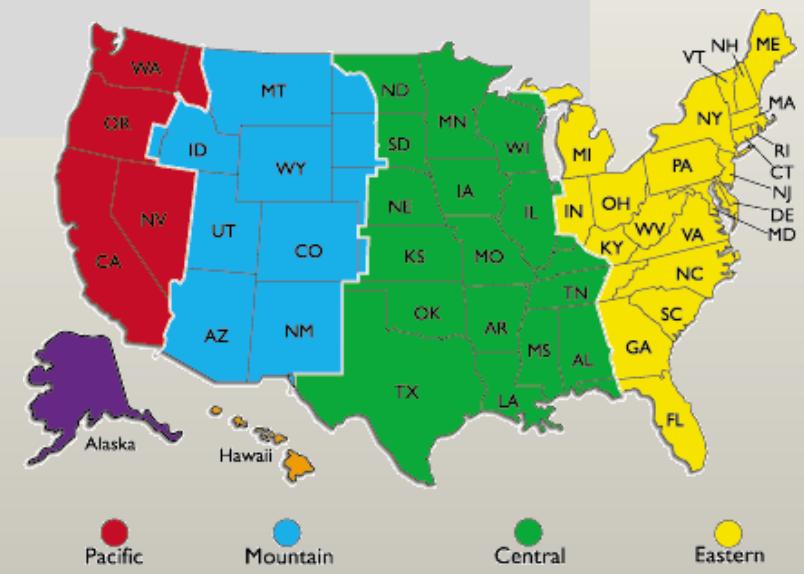
```
#define _XOPEN_SOURCE      /* See feature_test_macros(7) */
#include <time.h>

char *strptime(const char *s, const char *format, struct tm *tm);
```

The `strptime()` function is the converse function to `strftime()` and converts the character string pointed to by `s` to values which are stored in the `tm` structure pointed to by `tm`, using the format specified by `format`.

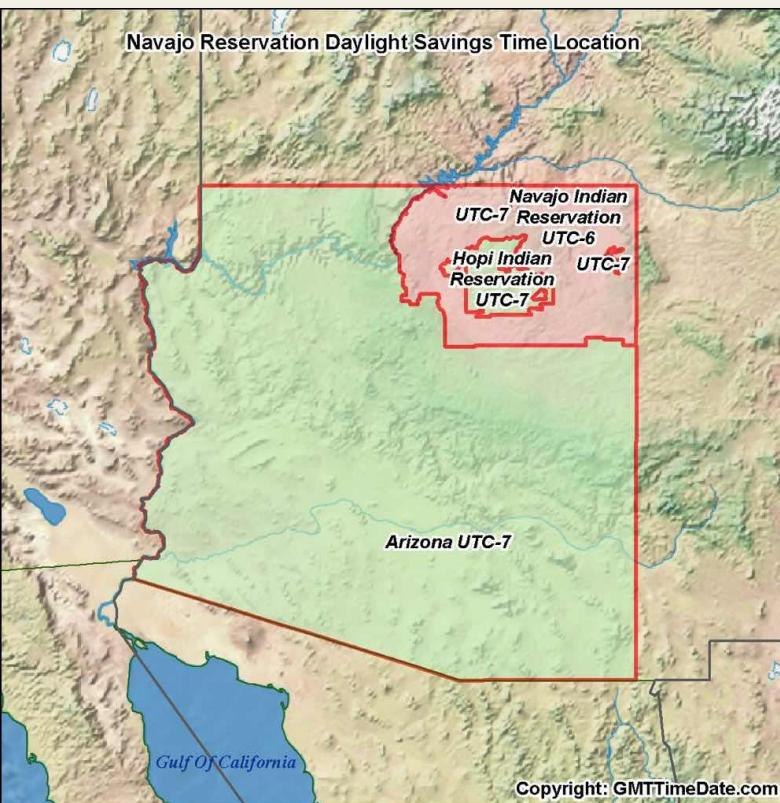
Timezones

- Different countries (and sometimes even different regions within a single country) operate on different timezones and DST regimes.
- Programs that input and output times must take into account the timezone and DST regime of the system on which they are run.
- The details are handled by the C library.



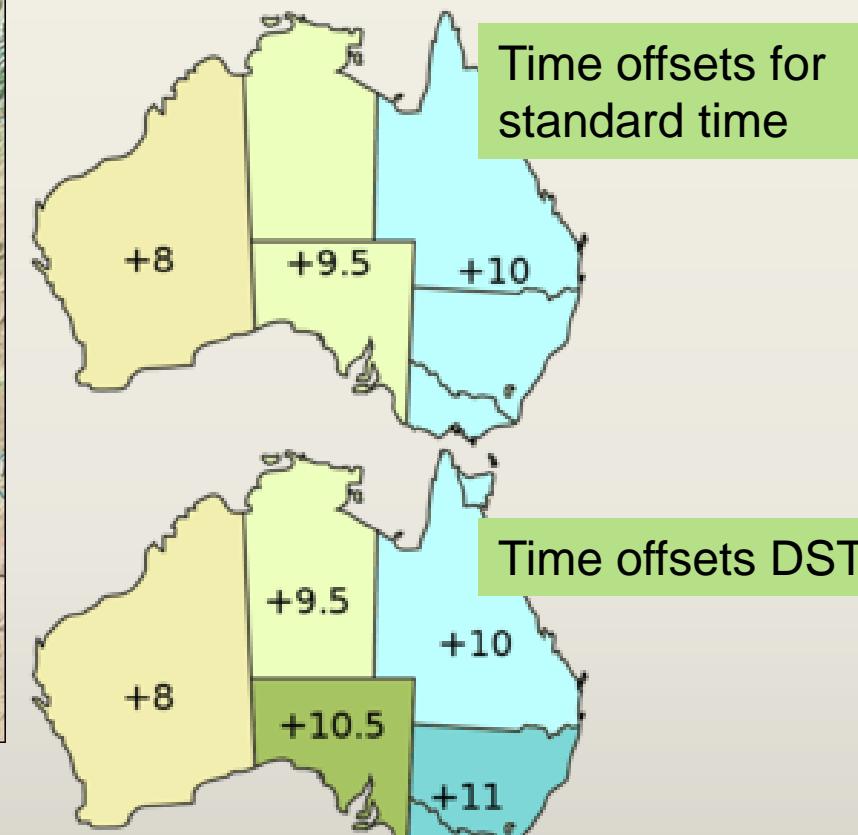


Some Timezones



THE
TWILIGHT
ZONE

R. Jesse Chaney



A compromise between Western and Central time (**UTC+8:45**, without DST), unofficially known as Central Western Standard Time, is used in one area in the southeastern corner of Western Australia and **one roadhouse in South Australia**.



IST – India Standard Time (IST) is 5:30 hours ahead of Coordinated Universal Time.

Nepal Standard Time (UTC+05:45)
DST not observed.

How India's single time zone is hurting its people

12 February 2019

India stretches 3,000km (1,864 miles) from east to west, spanning roughly 30 degrees longitude. This corresponds with a **two-hour difference in mean solar times** - the passage of time based on the position of the sun in the sky.

The US equivalent would be New York and Utah sharing one time zone.
Except that in this case, it also affects more than a billion people ...

...

Mr. Jagnani says that back of the envelope estimates suggested that India would accrue annual human capital gains of over \$4.2bn (0.2% of GDP) if the country switched from the existing single time zone to the proposed two time zone policy ...

<https://www.bbc.com/news/world-asia-india-47168359>

Timezone Definitions

- All the cruft about timezones information is **both voluminous and volatile**.
- Rather than encoding it directly into programs or libraries, the system maintains this information in files in standard formats.
- These files reside in the directory /usr/share/zoneinfo.
- Each file in this directory contains information about the timezone regime in a particular country or region.



Timezone Info

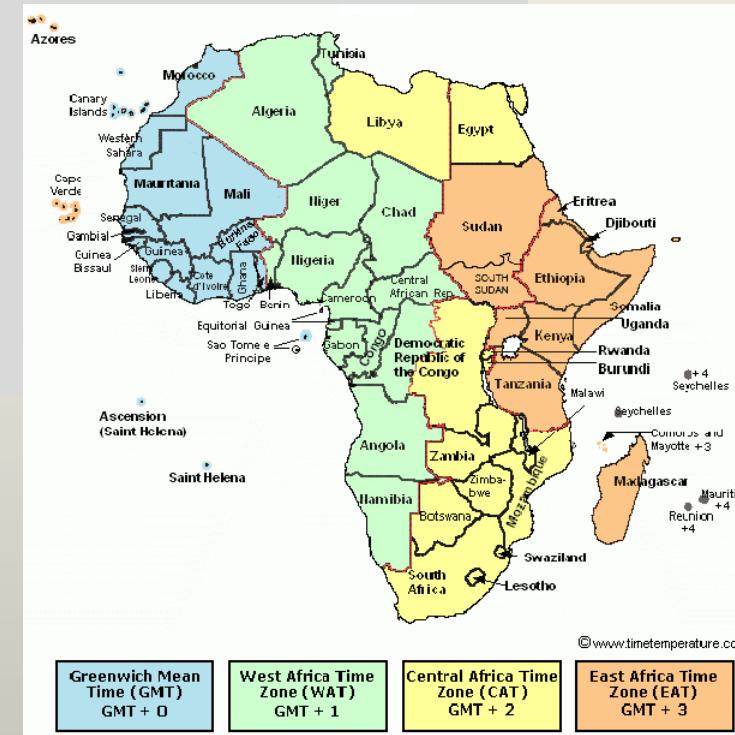
drwxr-xr-x.	2	root	root	4.0K	Aug	9	08:43	Africa/
drwxr-xr-x.	6	root	root	8.0K	Aug	9	08:43	America/
drwxr-xr-x.	2	root	root	187	Aug	9	08:43	Antarctica/
drwxr-xr-x.	2	root	root	26	Aug	9	08:43	Arctic/
drwxr-xr-x.	2	root	root	4.0K	Aug	9	08:43	Asia/
drwxr-xr-x.	2	root	root	196	Aug	9	08:43	Atlantic/
drwxr-xr-x.	2	root	root	4.0K	Aug	9	08:43	Australia/
drwxr-xr-x.	2	root	root	59	Aug	9	08:43	Brazil/
-rw-r--r--	1	root	root	2.1K	Mar	27	2017	CET
-rw-r--r--	1	root	root	2.3K	Mar	27	2017	CST6CDT
drwxr-xr-x.	2	root	root	161	Aug	9	08:43	Canada/
drwxr-xr-x.	2	root	root	45	Aug	9	08:43	Chile/
...						
-rw-r--r--	4	root	root	2.4K	Mar	27	2017	Navajo

Specifying the Timezone

When you need to set the timezone for running a program, you set the TZ environment variable to a string consisting of a colon (:) followed by one of the timezone names defined in /usr/share/zoneinfo.

```
TZ=":Pacific/Auckland" ./show_time  
TZ=":Navajo" ./show_time
```

The show_time application is another handy example that resides in
~rchaney/Classes/cs333/src/time



Locales

- Several thousand languages are spoken across the world, of which a significant percentage are regularly used on computer systems.
- Different countries use different conventions for displaying information such as numbers, currency amounts, dates, and times.
 - For example, in most European countries, a comma, rather than a decimal point, is used to separate the integer and fractional parts of (real) numbers, and most countries use formats for writing dates that are different from the MM/DD/YY format used in the United States.
- SUSv3 defines a **locale** as the “subset of a user’s environment that depends on language and cultural conventions.”



low.local

The logo consists of the word "low" in orange and ".local" in grey, all in a lowercase sans-serif font, arranged vertically on a white background.

The System Clock

- Most systems now use a Network Time Protocol (NTP) daemon to keep the system clock aligned with worldwide clocks.
- Abrupt changes in the system time of the sort caused by calls to `settimeofday()` can have **deleterious** effects on applications.
- **When making small changes to the time, it is usually preferable to use the `adjtime()` library function, which causes the system clock to gradually adjust to the desired value.**



The Software Clock

- The accuracy of various time-related system calls described in this book is limited to the resolution of the system software clock, which measures time in units called **jiffies**.
- The size of a jiffy is defined by the **constant HZ** within the kernel source code.
- This is the unit in which the kernel allocates the CPU to processes under the round-robin time-sharing scheduling algorithm

I like to have an exam question such as, “In what unit is the software clock on UNIX measured.”



Leap Seconds

A leap second is a one-second adjustment that is occasionally applied to Coordinated Universal Time (UTC) in order to keep its time of day close to the mean solar time.

Without the correction, time measured by the Earth's rotation drifts away from atomic time because of irregularities in the Earth's rate of rotation.

Right now, the official U.S. time is:

23:59:60

How is this possible???

12-hr 24-hr

Click arrows
to change
time zone

Saturday, December 31, 2016
UTC
Corrected for network delay

Problems?
Questions?

Leap Seconds

- The irregularity and unpredictability of UTC leap seconds is problematic for several areas, **especially computing**.
- For example, to compute the elapsed time in seconds between two given UTC past dates requires the consultation of a table of leap seconds, which needs to be updated whenever a new leap second is announced.
- In addition, it is not possible to compute accurate time intervals for UTC dates that are more than about six months in the future.



LEAP

Year	June 30	Dec 31
1992	+1	0
1993	+1	0
1994	+1	0
1995	0	+1
1997	+1	0
1998	0	+1
2005	0	+1
2008	0	+1
2012	+1	0
2015	+1	0
2016	0	+1
2017	0	0
2018	0	0
2019	0	0
2020	0	0



Dr. Markus Kuhn (University of Cambridge) lists the following **arguments against leap seconds**:

- Leap seconds could cause disruptions where computers are tightly synchronized with UTC.
- Leap seconds are a rare anomaly, which is a concern for safety-critical real-time systems (e.g. air-traffic control concepts entirely based on satellite navigation).
- Universal Time (UT1), which is defined by Earth's rotation, is not significant in most people's daily lives.

His **arguments in favor of leap seconds** include:

- There have been no credible reports about serious problems caused by leap seconds.
- Some computerized systems that work with leap seconds are costly to modify (eg. antennas that track satellites).
- Computer errors caused by leap seconds can be avoided simply by using International Atomic Time instead of UTC.
- Desktop computers and network servers have no trouble coping with leap seconds.
- Humankind has defined time by the Earth's rotation for over 5000 years – this tradition should not be given up because of unfounded worries of some air-traffic control engineers.
- **Abandoning leap seconds would make sundials obsolete.**

Leap for Joy

After many years of discussions by different standards bodies, in November 2022, at the 27th General Conference on Weights and Measures, **it was decided to abandon the leap second by or before 2035.**



https://en.wikipedia.org/wiki/Leap_second

Hold the Joy

A suggested possible future measure would be to **let the discrepancy increase to a full minute**, which would take 50 to 100 years, and then **have the last minute of the day taking two minutes in a "kind of smear" with no discontinuity.**

https://en.wikipedia.org/wiki/Leap_second



Leap Smear

Since 2008, instead of applying leap seconds to our servers using clock steps, **we have "smeared" the extra second across the hours before and after each leap.** The leap smear applies to all Google services, including all our APIs.

...

We encourage anyone smearing leap seconds to use a 24-hour linear smear from noon to noon UTC.



<https://developers.google.com/time/smear>

Process Time

Process time is the **amount of CPU time** used by a process since it was created.

For recording purposes, the kernel separates CPU time into the following two components:

- **User CPU time** is the amount of time spent **executing in user mode**. Sometimes referred to as virtual time, this is the time that it appears to the program that it has access to the CPU.
- **System CPU time** is amount of time spent **executing in kernel mode**. This is the time that the kernel spends executing system calls or performing other tasks on behalf of the program



Measuring Elapsed Time

The UNIX command **time** will track and display the duration and resources a command takes to complete.

When command finishes, **time** writes a message **to standard error** giving timing statistics about the program run.

The statistics consist of:

- the elapsed real time (wall clock time) between invocation and termination,
- the user CPU time (user mode time), and
- the system CPU time (kernel mode time).

Some shells have a built-in time command that provides less information. To access the full command, specify its full pathname (/usr/bin/time).

NAME

times - get process times

SYNOPSIS

```
#include <sys/times.h>
```

```
clock_t times(struct tms *buf);
```

DESCRIPTION

times() stores the current process times in the struct tms that buf points to. The struct tms is as defined in <sys/times.h>:

```
struct tms {  
    clock_t tms_utime;      /* user time */  
    clock_t tms_stime;      /* system time */  
    clock_t tms_cutime;       /* user time of children */  
    clock_t tms_cstime;       /* system time of children */  
};
```

Measuring Elapsed Time in Your Code

```
#define MICROSECONDS_PER_SECOND 1000000.0D

struct timeval tv0;
struct timeval tv1;

gettimeofday (&tv0, NULL);

...
...
...

gettimeofday (&tv1, NULL);

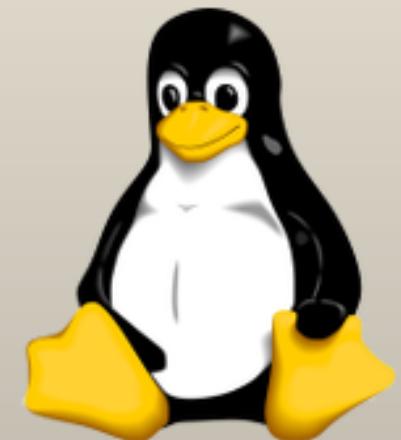
double run_time =
    (((double) (tv1.tv_usec - tv0.tv_usec))
     / MICROSECONDS_PER_SECOND)
    + ((double) (tv1.tv_sec - tv0.tv_sec));
```



The Linux /proc Filesystem

More information can be found by looking at:

- TLPI Chapter 12
- `man proc`
- `info proc`



What is /proc?

- A **pseudo-filesystem** that acts as an **interface to internal data structures in the kernel**.
- Real time, **it resides in the virtual memory**.
- Tracks the processes running on the machine and the state of the system.
- A new /proc file system is created every time your Linux machine reboots.



What can you do with /proc?

- Can be used to **read** information about the system
- Can be used to **modify** certain kernel parameters at runtime.

Other features

- /proc file system does not exist on any particular media (disk drive).
- The contents of the /proc file system can be read by anyone who has the necessary permissions.

History

- The idea of a Process Filesystem
 - Used for reporting process information only.
 - Seen in other UNIX implementations, such as Solaris.
- /proc extends the concept.
- A similar implementation available for other various flavors of BSD, including FreeBSD.
- /proc for Linux is the most actively developed.



/proc vs /sys

- Over time, the number of files in **procfs** and their layout has become rather chaotic.
- In Linux 2.6, **sysfs** was introduced to export a subset of the data in an ordered way.
- In contrast, **sysfs** exports a very ordered hierarchy of files relating to devices and the way they are connected to each other.
- Many of the legacy system information and control points are still accessible in **procfs**, but all **new busses and drivers should expose their info and control points via sysfs**.

<https://subscription.packtpub.com/book/programming/9781784392536/5/ch05lvl1sec49/the-proc-and-sysfs-filesystems>
<https://unix.stackexchange.com/questions/4884/what-is-the-difference-between-procfs-and-sysfs>

The Problem:

- A modern kernel is highly complex.
- Linux kernel has device drivers built-in.
- An **enormous amount of status information**.
- Many runtime configurable parameters.
- How could the system allow controlled access to kernel data and parameters and provide a familiar interface that programmers can easily adopt?



The Solution:

- Create **pseudo-filesystem** to represent status information and configuration parameters as **files**.
- Provides a unified ‘API’ for collecting status information and configuring drivers.
- Control access through UNIX permissions.
- No new libraries needed – **simple filesystem calls are all that is necessary**.
- Quick, easy access via command line.
- Not version or configuration-specific.



The layout of /proc

Two major subdivisions

- Read-only files/directories
- Configurable settings in /proc/sys/

Hierarchical Subdirectories for

- Network
- SCSI
- IDE
- Device Drivers
- Etc...

ls -la /proc

-r--r--r-- 1 root	root	0 Apr 3 13:52	ioports
dr-xr-xr-x 24 root	root	0 Mar 30 19:19	irq/
-r--r--r-- 1 root	root	0 Apr 3 13:52	kallsyms
-r----- 1 root	root	128T Mar 30 19:19	kcore
-r--r--r-- 1 root	root	0 Apr 3 13:52	keys
-r--r--r-- 1 root	root	0 Apr 3 13:52	key-users
-r----- 1 root	root	0 Mar 30 19:19	kmsg
-r----- 1 root	root	0 Apr 3 13:52	kpagecggroup
-r----- 1 root	root	0 Apr 3 13:52	kpagecount
-r----- 1 root	root	0 Apr 3 13:52	kpageflags
-r--r--r-- 1 root	root	0 Apr 4 22:42	loadavg
-r--r--r-- 1 root	root	0 Apr 3 13:52	locks
-r--r--r-- 1 root	root	0 Mar 30 20:18	mdstat
-r--r--r-- 1 root	root	0 Mar 30 19:19	meminfo
-r--r--r-- 1 root	root	0 Apr 3 13:52	misc
-r--r--r-- 1 root	root	0 Mar 30 19:19	modules
lrwxrwxrwx 1 root	root	11 Mar 30 19:19	mounts -> self/mounts
-rw-r--r-- 1 root	root	0 Apr 3 13:52	mtrr
lrwxrwxrwx 1 root	root	8 Mar 30 19:19	net -> self/net/
-r----- 1 root	root	0 Apr 3 13:52	pagetypeinfo
-r--r--r-- 1 root	root	0 Apr 3 13:52	partitions
dr-xr-xr-x 5 root	root	0 Mar 30 19:19	pressure/
-r--r--r-- 1 root	root	0 Apr 3 13:52	schedstat
dr-xr-xr-x 5 root	root	0 Apr 3 13:52	scsi/
lrwxrwxrwx 1 root	root	0 Mar 30 19:19	self -> 159761/

Some files in /proc

- **cmdline**
Kernel command line
- **cpuinfo**
Information about the processor(s). (Human readable)
- **devices**
List of device drivers configured into the currently running kernel (block and character).
- **dma**
Shows which DMA channels are being used at the moment.
- **execdomains**
List of the execution domains - execdomains is related to security

/proc/kcore

This file represents the virtual memory of the system and is stored in the core file format.

Unlike most /proc files, kcore does display a size. This value is given in bytes and is equal to the size of **virtual memory** (RAM) used plus 4KB.

Its contents are designed to be examined by a debugger, such as gdb, the GNU Debugger.

Only the **root** user has the rights to view this file.

```
-r----- 1 root      root    128T Mar 30 19:19 kcore
```

128 terabytes!

Some other files in /proc

- **loadavg**
 - Provides a look at load average
 - The first three columns measure **CPU utilization of the last 1, 5, and 15 minute periods.**
 - The fourth field consists of two numbers separated by a slash
 - The first of these is the **number of currently runnable processes.**
 - The value after the slash is the **number of processes that currently exist on the system.**
 - The fifth field is the **PID of the process that was most recently created on the system.**

- **locks**
 - Displays the files currently locked by the kernel
- **meminfo**
 - One of the more commonly used /proc files
 - It reports back plenty of valuable information about the current utilization of memory on the system
- **stat**
 - Keeps track of a variety of different statistics about the system since it was last restarted

- **uptime**

Contains information about how long the system has gone since its last restart. The file contains two numbers: the uptime of the system (seconds), and the amount of time spent in idle process (seconds).

- **version**

- Tells the versions of the Linux kernel and `gcc`, as well as the version of Linux is installed on the system.

Here's a look at the top of /proc/meminfo on babbage

```
rchaney # cat meminfo
MemTotal:      65844568 kB
MemFree:       33334444 kB
MemAvailable:  62669628 kB
Buffers:        1091812 kB
Cached:         28136072 kB
SwapCached:      0 kB
Active:          8506932 kB
Inactive:       22333456 kB
```

Process Information

- Each process has a `/proc` directory identified by its PID - `/proc/PID/`
- Symbolic link `/proc/self/` points to the process reading the file system
- Allows access to
 - Process status
 - Process memory information
 - Links to cwd, exe, root dir
 - CPU and Memory Map information

A typical process directory

- **cmdline** : it contains the entire command line used to invoke the process. The contents of this file are the command line arguments with all the parameters (without formatting/spaces).
- **cwd** : symbolic link to the current working directory
- **environ** : contains all the process-specific environment variables
- **exe** : symbolic link of the executable

- **fd** : this directory contains the list file descriptors as opened by the particular process.
- **limits** : this file displays the soft limit, hard limit, and units of measurement for each of the process's resource limits.
- **root** : symbolic link pointing to the directory which is the root file system for the particular process
- **status** : information about the process

Process Information (Example)

```
babbage /proc/self
rchaney # cat status
Name: bash
Umask: 0077
State: S (sleeping)
Tgid: 154331
Ngid: 0
Pid: 154331
PPid: 154318
TracerPid: 0
Uid: 18781 18781 18781 18781
Gid: 300 300 300 300
FDSize: 256
```

Kernel Information

APM	Buses	CPUs	Available Devices	DMA Channels
Filesystems	Device Drivers	Frame Buffer Devices	IDE Subsystem	Interrupts
Memory Map	I/O Ports	ISA PnP	Kernel Core Image	Kernel Symbols
Kernel Messages	Load Averages	Kernel Locks	Memory	Loaded Modules
Mounted Filesystems	Networking	Partitions	RTC	SCSI
Statistics	Swap Space	SysV IPC	TTY Drivers	Uptime

/proc/sys subdirectories

- **ctrl-alt-del** : Controls whether [Ctrl]-[Alt]-[Delete] will gracefully restart the computer using init (value 0) or force an immediate reboot without syncing the dirty buffers to disk (value 1).
- **domainname** : Allows you to **configure** the system's domain name, such as domain.com.
- **hostname** : Allows you to **configure** the system's host name, such as host.domain.com.
- **threads-max** : Sets the maximum number of threads to be used by the kernel.

Configuring the Kernel

Read-**write** entries in /proc/sys/

Allow for tuning, monitoring, and optimization of a running kernel.

Modifiable only by root

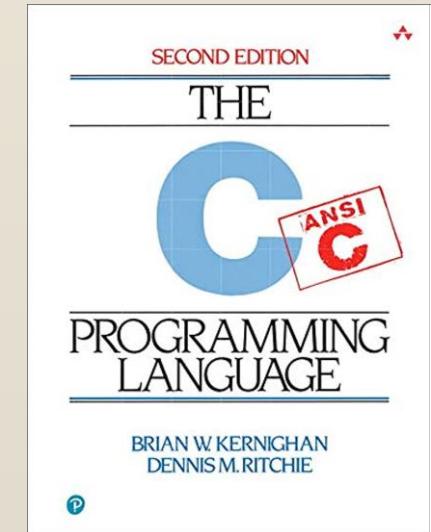
Parameters may be changed simply via 'echo'

```
# cat /proc/sys/fs/file-max  
6514751  
  
# echo 8192 > /proc/sys/fs/file-max  
# cat /proc/sys/fs/file-max  
8192
```

Programming for /proc

- Simple filesystem representation allows for easy programming
- C calls

```
uptimefp = fopen (PROC_DIR "uptime");
fgets (line, sizeof (line), uptimefp);
new.uptime =
    (atof (strtok (line, " ")) * (unsigned long) HZ);
```



- Web interfaces

```
<html><body>
<? if ($fp = fopen('/proc/sys/kernel/hostname', 'r')) {
    $result = trim(fgets($fp, 4096));
    echo gethostbyaddr(gethostbyname($result)); } ?>
</body></html>
```



- Shell scripts – bash, Perl, Python,



Pros and Cons

- **Advantages**
 - Coherent, intuitive interface to the kernel
 - Great for tweaking and collecting status info
 - Easy to use and use in programs

- **Disadvantages**
 - Certain amount of overhead, must use fs calls
 - User can possibly cause system instability

PROS	CONS
①	1)
②	2)
③	3)



Device Special Files



Devices can be divided into two types:

1. Character devices handle data on a character-by-character basis.

- Terminals and keyboards are examples of character devices.

Device special files can be divided into two types. What are they?

2. Block devices handle data a block at a time. The size of a block depends on the type of device, but is typically some multiple of 512 bytes.

- Examples of block devices include disks and tape drives.

Device Special Files

- Within the kernel, each device type has a corresponding **device driver**, which handles all I/O requests for the device.
- A **device driver** is a **unit of kernel code that implements a set of special operations** that (usually) correspond to input and output actions on an associated piece of hardware.
- **Device drivers are custom written code that allow you to act as though the device is just a file.**



Device Special Files

Device files appear within the file system, just like other files, usually under the `/dev` directory.

The superuser can create a device file using the `mknod` command, and the same task can be performed in a privileged program using the `mknod()` system call.

/dev/random and /dev/urandom

The character special files **/dev/random** and **/dev/urandom** provide an interface to the **kernel's random number generator**.

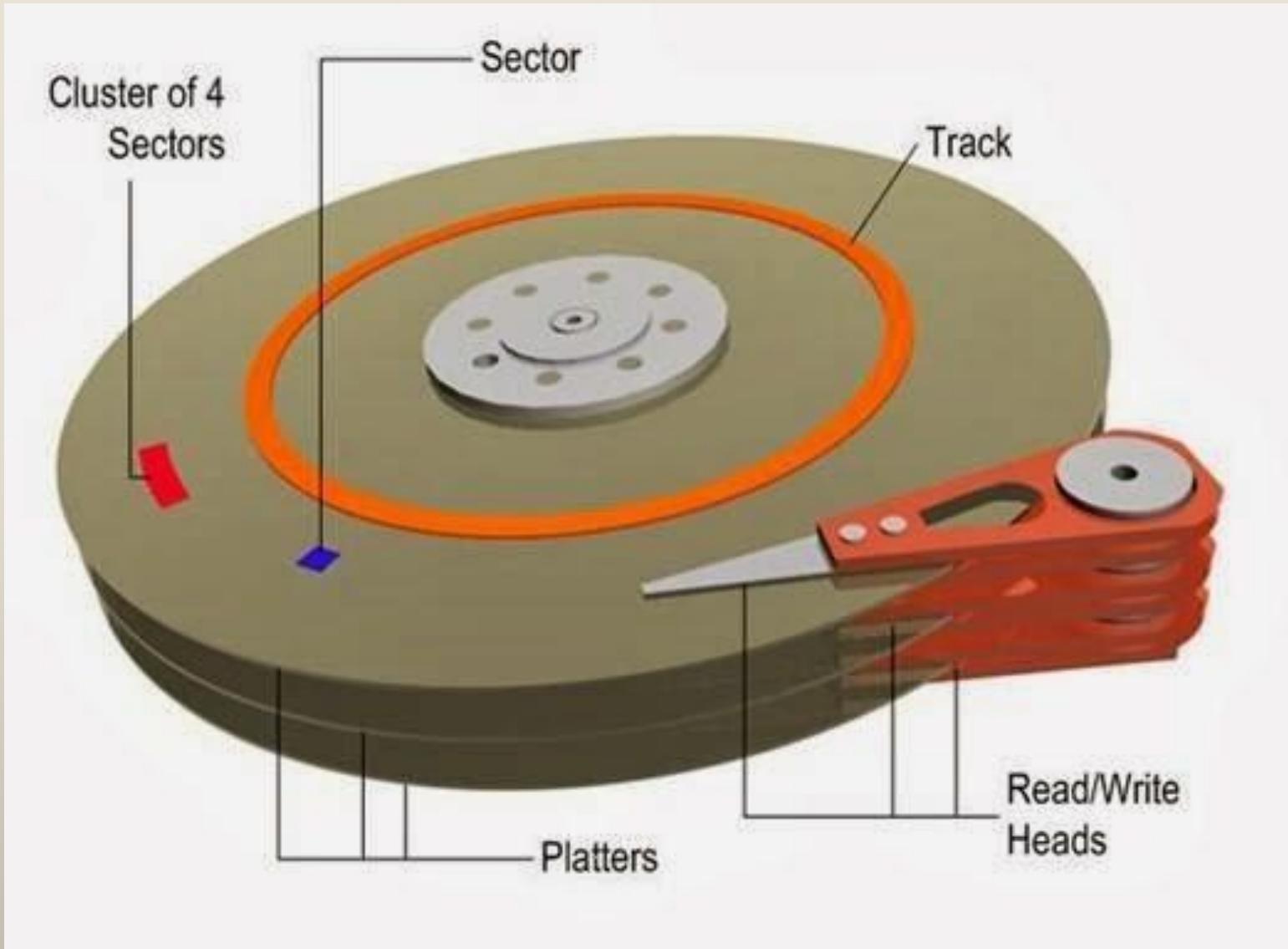
- **The random number generator gathers environmental noise from device drivers and other sources into an entropy pool.**
- The generator also keeps an estimate of the number of bits of noise in the entropy pool.
- From this entropy pool random numbers are created.

Disk Drives



- A disk drive is a mechanical device consisting of one or more **platters** that rotate at high speed.
- Physically, information on the disk surface is located on a **set of concentric circles called tracks**.
- Tracks themselves are divided into a number of **sectors**, each of which consists of a series of physical blocks.
- Physical **blocks** are typically 512 bytes (or a multiple of 512), and represent the smallest unit of information that the drive can read or write.







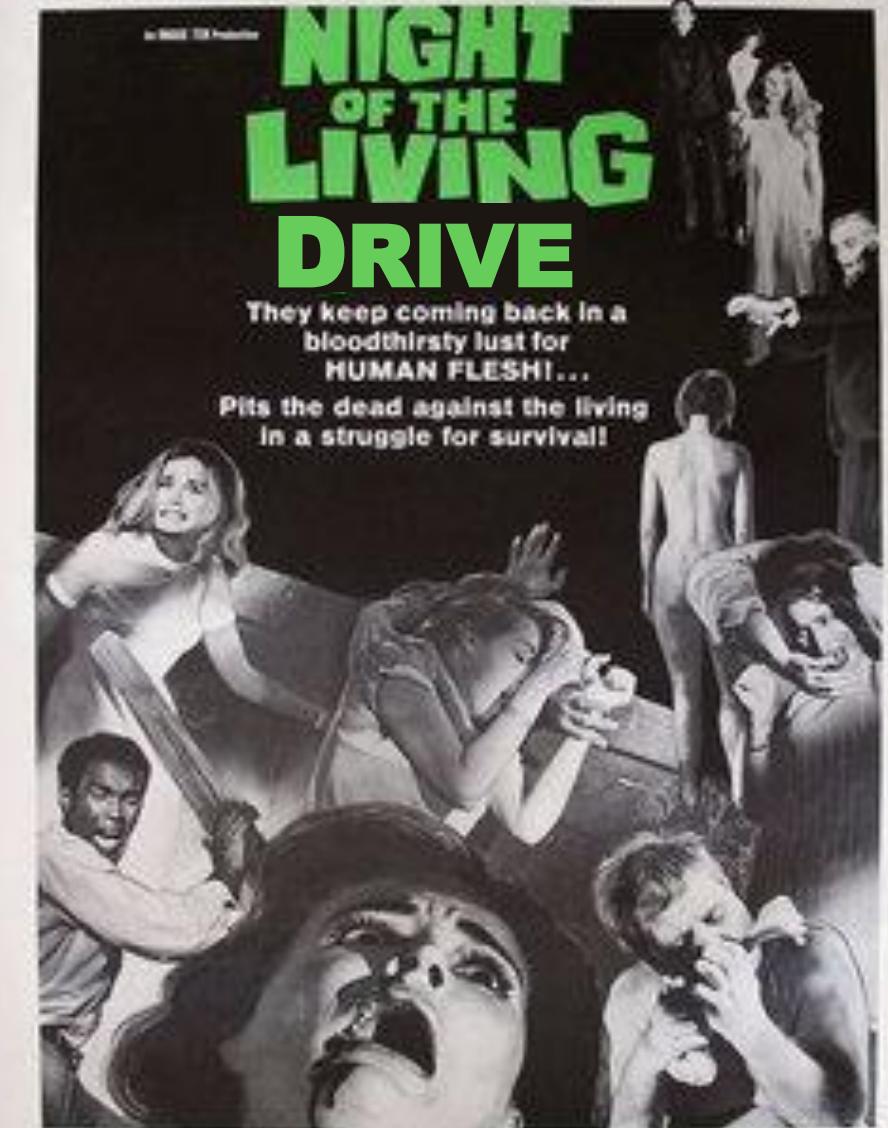
Digital Equipment Corporation's RP-04

walking drives: n.

An occasional failure mode of magnetic-disk drives back in the days when they were huge, clunky washing machines. Those old dinosaur parts carried terrific angular momentum; the combination of a misaligned spindle or worn bearings and stick-slip interactions with the floor could cause them to ‘walk’ across a room, lurching alternate corners forward a couple of millimeters at a time. **There is a legend about a drive that walked over to the only door to the computer room and jammed it shut; the staff had to cut a hole in the wall in order to get at it!**

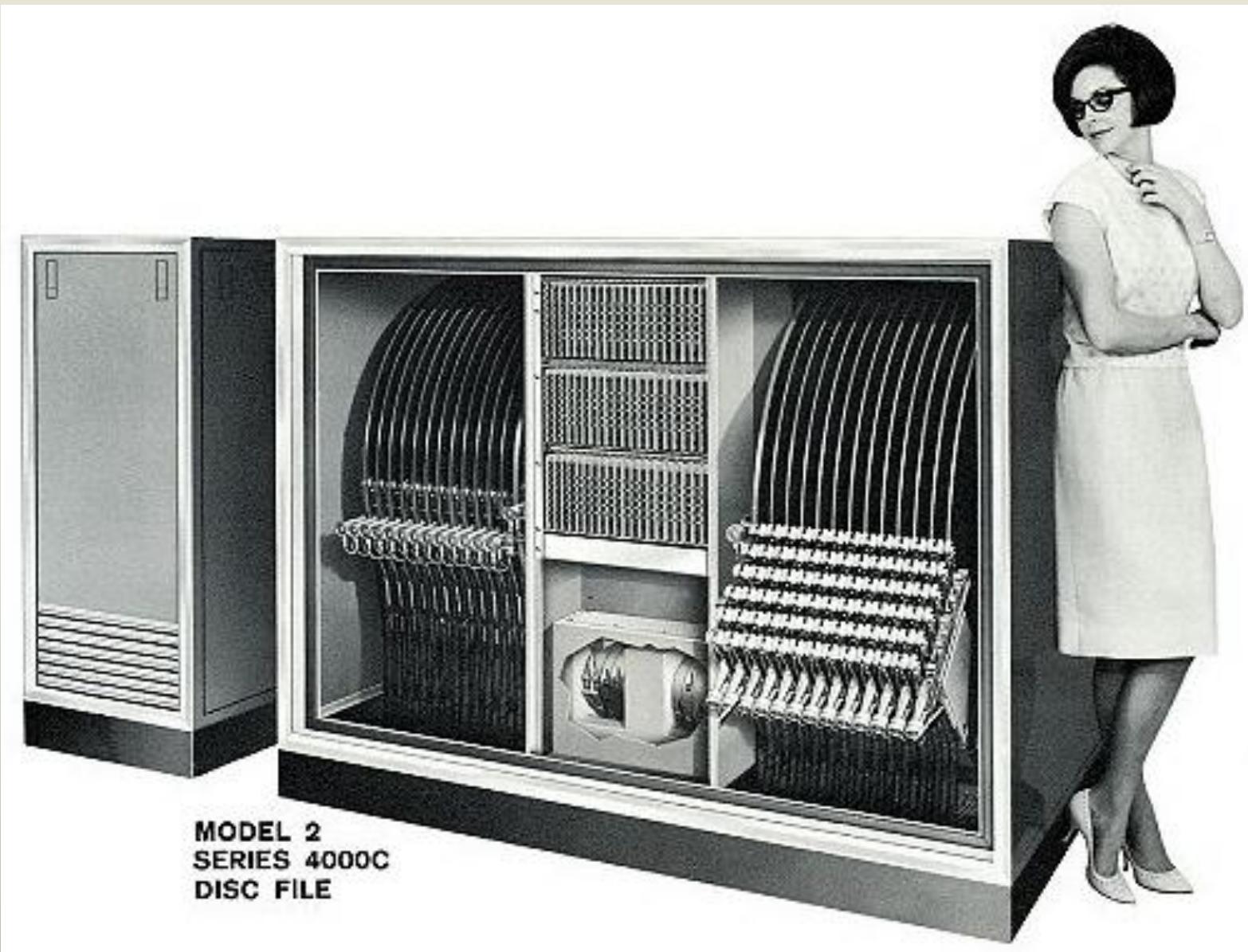
Disk drive walking could also be induced by certain patterns of drive access (a fast seek across the whole width of the disk, followed by a slow seek in the other direction). **Some bands of old-time hackers figured out how to induce disk-accessing patterns that would do this to particular drive models and held disk-drive races.**

THEY WON'T STAY DEAD!



Starring JUDITH O'DEA · DIANE JONES · MARIJN EASTMAN · KARL HARDMAN · JUDITH RIDLEY · KEITH WAYNE

Produced by Robert W. Rosen and Alan Arkin · Directed by George A. Romero · Screenplay by John A. Russo · A Major Media International Production · Released by Continental



Disk Drives

Reading and writing information on the disk takes significant time.

1. The disk head must first move to the appropriate track (**seek time**).
2. The drive must wait until the appropriate sector rotates under the head (**rotational latency**).
3. The required blocks must be transferred (**transfer time**) from the disk to a kernel buffer.

I often ask what are the 3 delays when reading data from disk drives.



Disk Drive Fragmentation

Internal Fragmentation: the space wasted inside of allocated memory blocks because of restriction on the allowed sizes of allocated blocks. Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

External Fragmentation: the free sectors are widely scattered through the disk drive.

Data Fragmentation: the sectors for a file are widely scattered over the disk drive. This means the disk must will have many seeks and a lot of rotational latency to read the disk.

Describe 3 types of disk fragmentation.

Disk Partitions

Each disk is divided into one or more (non-overlapping) **partitions**. Each partition is treated by the kernel as a **separate device residing under the /dev directory**.

A disk partition may hold any type of information, but usually contains one of the following:

1. a file system holding **regular files and directories**
2. a data area accessed as a **raw-mode device** (some database management systems use this technique)
3. a **swap area** used by the kernel for memory management.

File Systems

One of the strengths of Linux is that it supports a wide variety of file systems, including the following:

- the traditional ext2 file systems
- various native UNIX file systems such as the Minix, System V, and BSD file systems
- Microsoft's FAT, FAT32, and NTFS file systems;
- the ISO 9660 CD-ROM file system
- Apple Macintosh's HFS
- a range of network file systems, including NFS
- IBM and Microsoft's SMB, Novell's NCP, and the Coda file system
- a range of **journaling file systems**, including ext3, ext4, Reiserfs, JFS, XFS, and Btrfs

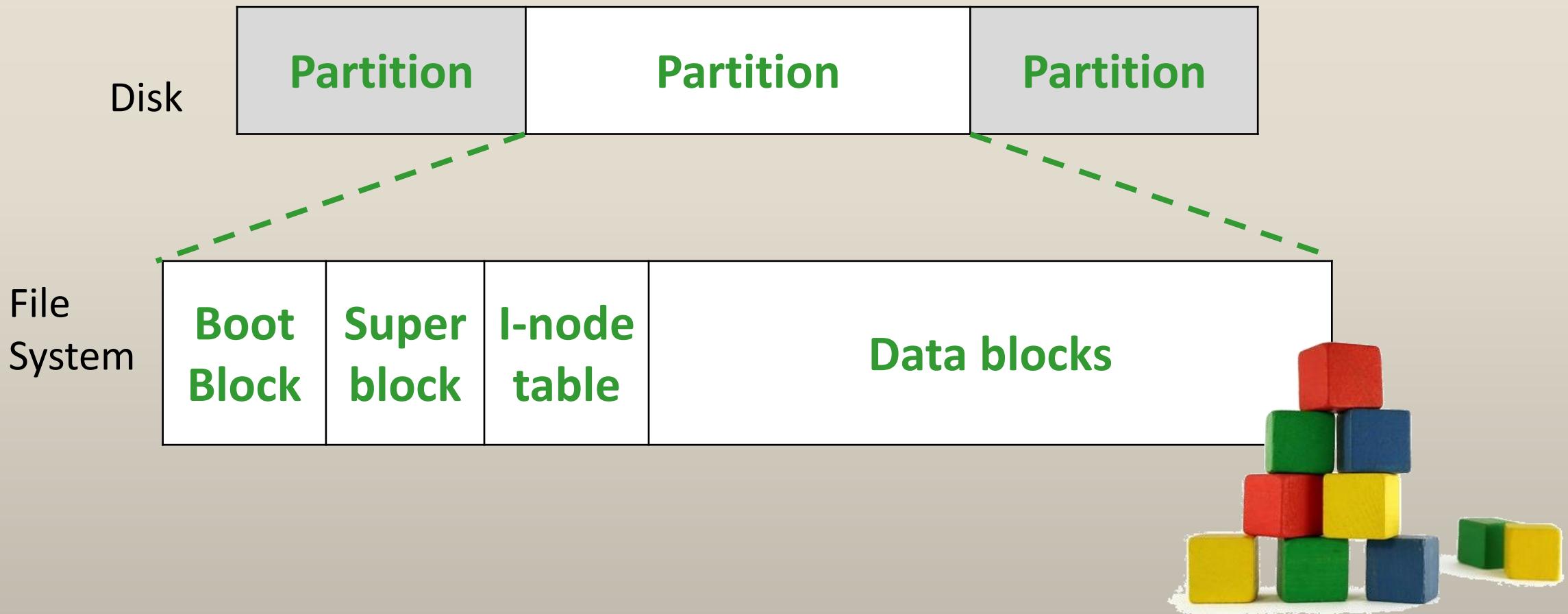
The ext2 File System

For many years, the most widely used file system on Linux was **ext2**, the **Second Extended File System**, which is the successor to the original Linux file system, ext.

The basic unit for allocating space in a file system is a logical block, which is some multiple of contiguous physical blocks on the disk device on which the file system resides. The logical block size on **ext2** is 1024, 2048, or 4096 bytes.

The ext2 File System

A physical disk drive.



The ext2 File System

Boot block: This is **always the first block in a file system.**

Superblock: This is a single block, immediately following the boot block, which **contains parameter information about the file system**, including:

- the size of the i-node table;
- the size of logical blocks in this file system; and
- the size of the file system in logical blocks.

I-node table: Each file or directory in the file system has a unique entry in the i-node table.

Data blocks: The great majority of space in a file system is used for the blocks of data that form the files and directories residing in the file system.

Superblocks

- The primary copy of the superblock is stored at an offset of 1024 bytes from the start of the device, and it is essential to mounting the filesystem.
- Since it is so important, backup copies of the superblock are stored in block groups throughout the filesystem.
- All fields in the superblock (as in all other ext2 structures) are stored on the disc in little endian format.

The ext2 File System – i-nodes

i-node

- The size of the file in bytes
- Device ID.
- The User ID of the file's owner.
- The Group ID of the file.
- The file mode (*permission bits*).
- Additional system and user flags to further protect the file.
- Timestamps telling when the i-node was last modified, the file content last modified, and last accessed.
- A link count telling how many *hard* links point to the i-node.
- *Pointers to the disk blocks that store the file's contents.*

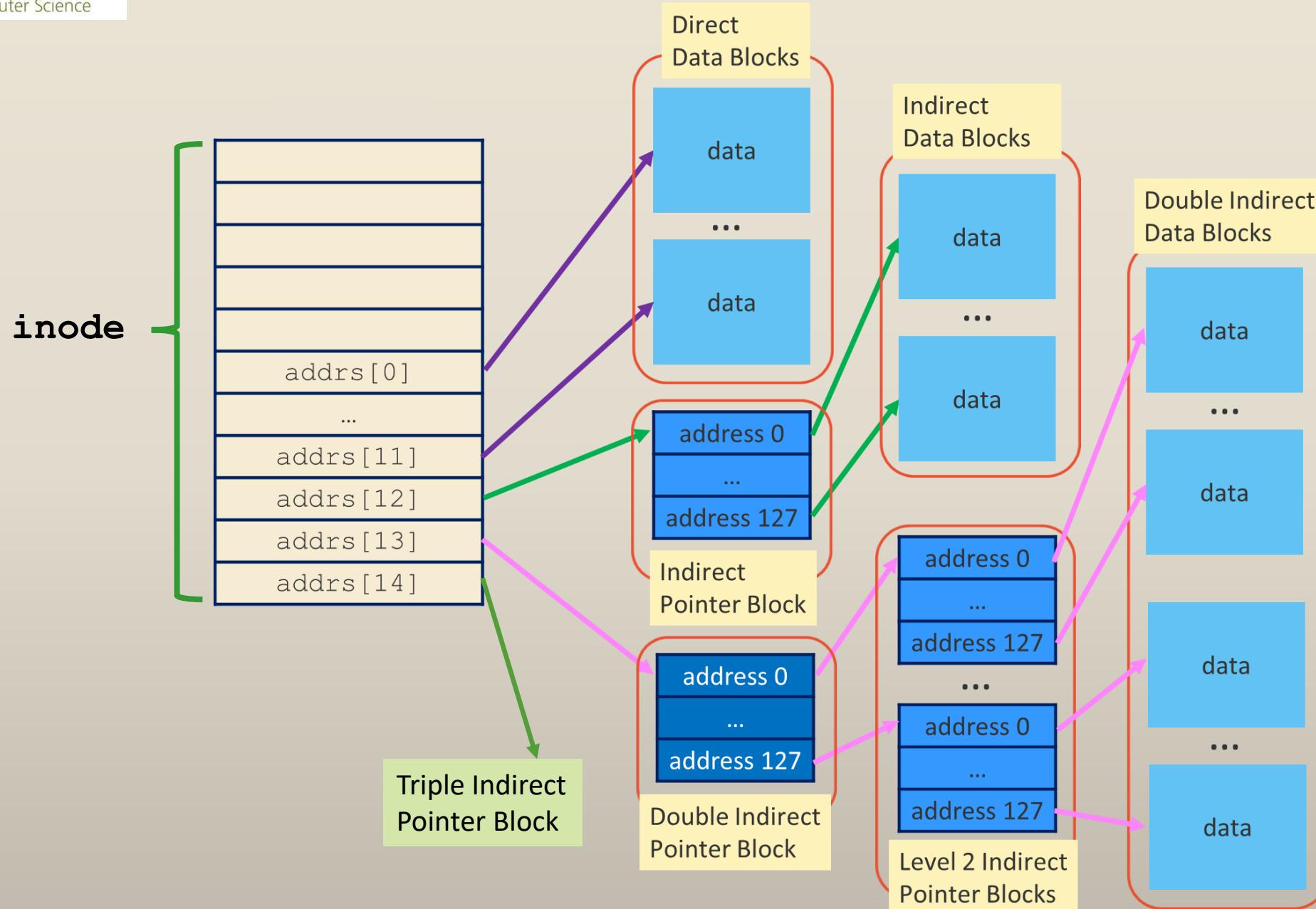
The ext2 File System – i-nodes

- Like most UNIX file systems, the ext2 file system **does not store the data blocks of a file contiguously** or even in sequential order (though it does attempt to store them *close* to one another).
- To locate the file data blocks, the kernel maintains a set of **pointers in the i-node**.

The ext2 File System – i-nodes

Under **ext2**, each i-node contains 15 pointers.

- The **first 12** of these pointers point to the location in the file system of the **first 12 blocks** of the file.
- The **13th** pointer is a pointer to a block of pointers that give the locations of the thirteenth and subsequent data blocks of the file.
- The **fourteenth** pointer is a **double indirect pointer** – it points to blocks of pointers that in turn point to blocks of pointers that in turn point to data blocks of the file.
- The **last pointer** in the i-node is a **triple-indirect pointer**.



The ext2 File System – i-nodes

This layout is designed to satisfy a number of requirements.

- The **i-node structure to be a fixed size**,
- It **allows for files of an arbitrary size**.
- It allows the file system to store the blocks of a file non-contiguously.
- for small files, which form the most of files on most systems, this allows the file data blocks to be accessed rapidly via the direct pointers of the i-node.

Journaling File Systems

The ext2 file system is a good example of a traditional UNIX file system, and

- It suffers from a classic limitation of such file systems: after a system crash, a file-system consistency check (**fsck**) must be performed on reboot in order to ensure the integrity of the file system.
 - At the time of the system crash, a file update may have been only partially completed, and the file-system metadata may be in an inconsistent state, so that the file system might be further damaged if these inconsistencies are not repaired.



The Dreaded `fsck`

A crash/shutdown that may have created issues with the file system is called dirty.

The system utility `fsck` (for "file system consistency check") is a tool for checking the consistency of a file system.

A consistency check requires **examining the entire file system**.

- On a small file system, this may take anything from several seconds to a few minutes.
- On a large file system, this may require several hours, which is a serious problem for systems that must maintain high availability

Journaling File Systems

Journaling file systems eliminate the need for lengthy file-system consistency checks after a system crash.

A journaling file system logs (journals) all **metadata** updates to a special on-disk journal file before they are actually carried out.

- The updates are logged in groups of related metadata updates (*transactions*).
- The file system is treated in the same manner that a database treats data, with transactions.

Journaling File Systems

In the event of a system crash or unclean shutdown in the middle of a transaction, on system reboot,

- The log can be used to rapidly redo any incomplete updates and bring the file system back to a consistent state.
- In database terminology, a journaling file system ensures that file metadata transactions are always committed as a complete unit (atomic).

Journaling File Systems

The most notable disadvantage of journaling is that it adds time to file updates.

In addition to actually writing data buffers to the disk, you have to keep all the journal information correct.

Depending on the implementation, a journaling file system may only keep track of stored metadata, resulting in improved performance at the expense of increased possibility for data corruption.

A journaling file system may track both stored data and related metadata, though some implementations allow selectable behavior

Journaling File Systems

Examples of Unix/Linux Journaling File Systems include:

- **Reiserfs**: the first of the journaling file systems to be integrated into the kernel.
- The **ext3** file system was a result of a project to add journaling to ext2 with minimal impact. The migration path from ext2 to ext3 is very easy.
- **JFS** from IBM.
- The **ext4** file system, the successor to ext3. The ext4 filesystem can support volumes with sizes up to 1 exbibyte (EiB) and files with sizes up to 16 tebibytes.

Single Directory Hierarchy and Mount Points

- All files from all file systems reside under a single directory tree.
- **At the base of this tree is the root directory, / (slash).**
- Other file systems are mounted under the root directory and appear as subtrees within the overall hierarchy.



Mount Points

The superuser can add a mount point to the file system using the command:

```
mount device directory
```

This command attaches the file system on the named device into the directory hierarchy at the specified directory—the file system's **mount point**.

Looking at the i-node, files on different mount points would have different device id's.

Mount Points

You can see what file systems are mounted onto the Unix system by issuing the `df -h` command.

On babbage, there are a **BUNCH** of mounted file systems:

Filesystem	Size	Used	Avail	Use%	Mounted on
tmpfs	6.3G	1.8M	6.3G	1%	/run
/dev/vda1	98G	39G	54G	42%	/
tmpfs	32G	0	32G	0%	/dev/shm
tmpfs	5.0M	0	5.0M	0%	/run/lock
tmpfs	32G	0	32G	0%	/run/qemu
tmpfs	6.3G	152K	6.3G	1%	/run/user/124
vol-spinel.cat.pdx.edu:/spinel/home/01/janaka/ubuntu	9.7T	364G	9.4T	4%	/u/janaka
tmpfs	6.3G	148K	6.3G	1%	/run/user/1004
tmpfs	6.3G	176K	6.3G	1%	/run/user/4703
vol-spinel.cat.pdx.edu:/spinel/home/15/giampiet/ubuntu	9.6T	190G	9.4T	2%	/u/giampiet
tmpfs	6.3G	176K	6.3G	1%	/run/user/15179
vol-spinel.cat.pdx.edu:/spinel/home/05/jgreever/ubuntu	9.5T	179G	9.4T	2%	/u/jgreever
vol-spinel.cat.pdx.edu:/spinel/stash/cat/doc	400G	19G	382G	5%	/cat/admin
vol-spinel.cat.pdx.edu:/spinel/home/06/rchaney/ubuntu	9.5T	139G	9.4T	2%	/u/rchaney
tmpfs	6.3G	148K	6.3G	1%	/run/user/18781
vol-spinel.cat.pdx.edu:/spinel/pkgs/pkgs	9.4T	10M	9.4T	1%	/pkgs/pkgs
vol-spinel.cat.pdx.edu:/spinel/home/06/rchaney	9.5T	139G	9.4T	2%	/home/rchaney



VIKINGS™
CS 333

Intro to Operating Systems Monitoring File Events



Ever write an application that needs to be notified when a new data file arrives or when a file is changed?

Wonder how you would do that?



File Monitoring Events!!!

- **Monitor files or directories** in order to determine whether events/changes have occurred for the monitored objects.
- TLPI Chapter 19
- The **inotify interface is a Linux only API.**
- *inotify* stands for inode-notify



- The `inotify` API provides a mechanism for **monitoring** filesystem events.
- The `inotify` interface can be used to **monitor individual files**, or to **monitor directories**.
- When a directory is monitored, `inotify` will return events for the directory itself, **and** for files inside the directory.





Limitations imposed by `inotify` include the following:

- `inotify` **does not support recursively watching directories**; a separate `inotify` watch must be created for each subdirectory.
- `inotify` does report some but not all events in `/sysfs` and `/procfs`.
- Notification via `inotify` requires the **kernel to be aware of all relevant filesystem events**
 - This is **not always possible for networked filesystems such as NFS** where changes made by one client are not immediately broadcast to other clients.
 - **Rename events are not handled directly**, `inotify` issues two separate events that must be examined and matched in a context of potential race conditions.

Key Steps

1. Call the `inotify_init()` to **create the inotify instance**.
2. **Notify the kernel what files and directories** you wish to monitor, using the `inotify_add_watch()` call.
3. **Perform a `read()` operation on each inotify file descriptor**.
4. Call the `inotify_rm_watch()` function and **close the file descriptor** when done.



`inotify_init()` creates an instance of the `inotify` subsystem in the kernel and returns a **file descriptor** on success and -1 on failure.

```
// Initialize inotify() system. We have to do
// this before we can use the inotify interface.
fd = inotify_init();
```

Note there are no parameters.

- When you call the `inotify_init()` function, it **returns a file descriptor**.
- It is a file descriptor just like the file descriptor returned by the `open()` call.
- It gets back, again, to the **Universality of I/O**.



Add Watch



```
int inotify_add_watch(  
    int fd  
    , const char *pathname  
    , uint32_t mask  
) ;
```

Adds a new watch item or modifies an existing watch item in the watch list for the `inotify` instance referred to by the file descriptor `fd`.

```
wd = inotify_add_watch(fd  
    , "/u/rchaney"  
    , IN_MODIFY | IN_CREATE | IN_DELETE) ;
```

```
wd = inotify_add_watch(fd, ".emacs"  
    , IN_OPEN | IN_CLOSE) ;
```

Add Watch



- Each watch must provide a pathname and a list of pertinent events, where each event is specified by a constant, such as `IN_MODIFY`.
- To monitor more than one event, simply use the bitwise or — the single pipe (`|`) operator in C — between each event.
- If `inotify_add_watch()` succeeds, the call returns a unique identifier for the registered watch; otherwise, it returns -1.
- Use returned the identifier to alter or remove the associated watch.

Sample inotify Event Types

IN_ACCESS File was accessed.

IN_ATTRIB Metadata changed.

IN_CLOSE_WRITE File opened for writing was closed.

IN_CLOSE_NOWRITE File not opened for writing was closed.

IN_CREATE File/directory created in a watched directory.

IN_DELETE File/directory deleted from a watched directory.

IN_DELETE_SELF Watched file/directory was itself deleted.

IN MODIFY File was modified.

IN_MOVE_SELF Watched file/directory was itself moved.

IN_MOVED_FROM File moved out of watched directory.

IN_MOVED_TO File moved into watched directory.

IN_OPEN File was opened.

Among others.

The inotify Event

```
struct inotify_event {  
    int wd;                  /* Watch descriptor on which event occurred */  
    uint32_t mask;            /* Bits describing event that occurred */  
    uint32_t cookie;          /* Cookie for related events (for rename()) */  
    uint32_t len;              /* Size of 'name' field */  
    char name[];              /* Optional null-terminated filename */  
};
```

The **cookie** field is used to tie related events together.

It is used to link together an `IN_MOVED_FROM` and an `IN_MOVED_TO` event.



```
#define EVENT_SIZE    (sizeof (struct inotify_event))  
#define BUF_LEN       (1024 * (EVENT_SIZE + 16))  
  
char buffer[BUF_LEN];  
  
length = read(fd, buffer, BUF_LEN);
```

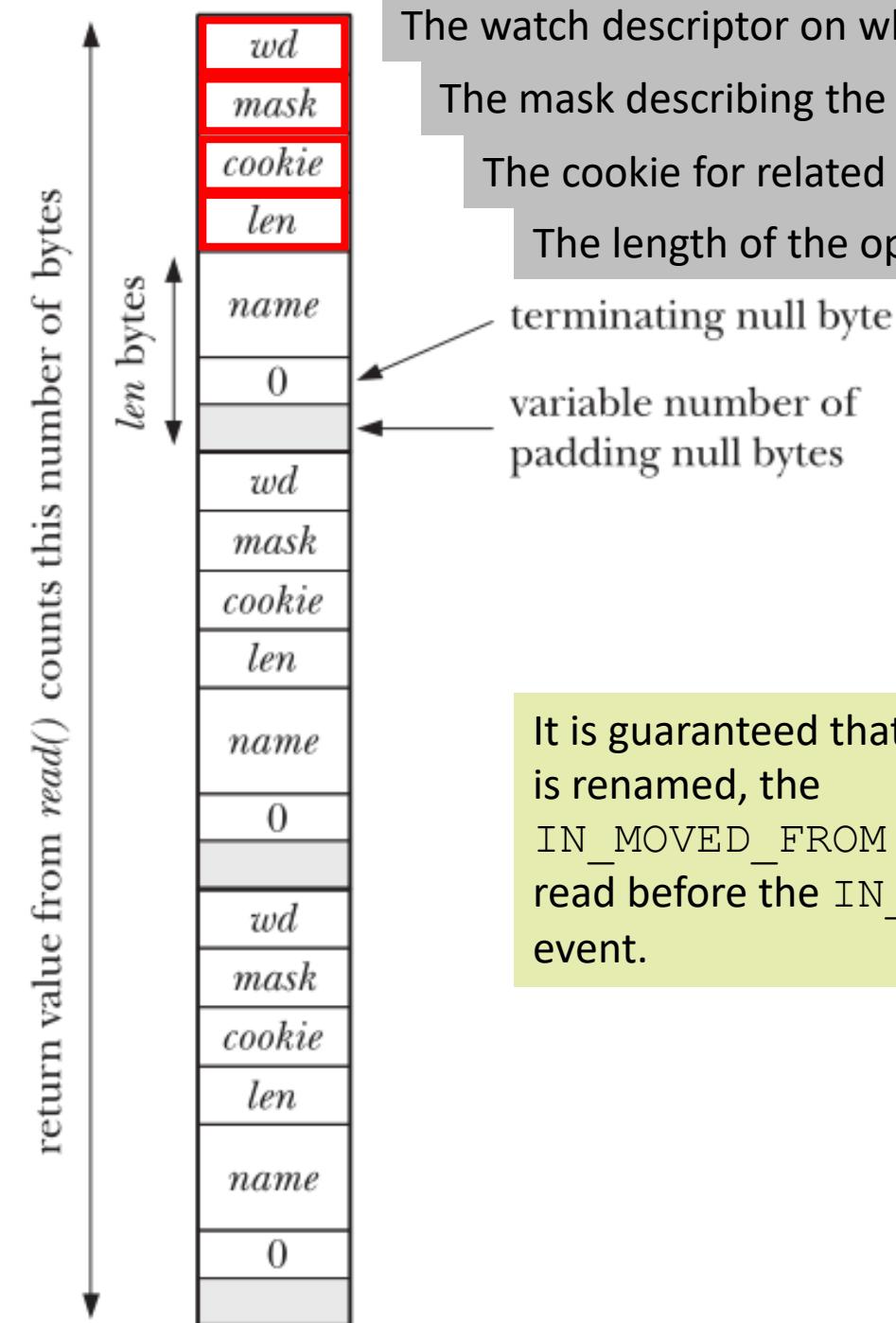


The file descriptor from `inotify_init()` can also be monitored using `select()`.

The events read from an `inotify` file descriptor form an **ordered queue**.

The `len` field indicates how many bytes are actually allocated for the name field.

- This field is necessary because there may be additional padding bytes between the end of the string stored in `name` and the start of the next `inotify_event` structure contained in the buffer returned by `read()`.



The watch descriptor on which the event occurred.

The mask describing the event.

The cookie for related events.

The length of the optional name field.

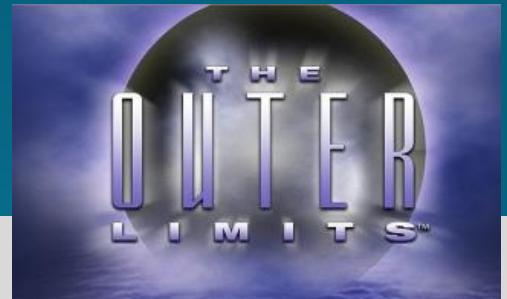
terminating null byte

variable number of padding null bytes

It is guaranteed that when a file is renamed, the `IN_MOVED_FROM` event will be read before the `IN_MOVED_TO` event.



Queue Limits and /proc Files



`max_queued_events`

The upper limit on the number of events that can be queued on the new *inotify* instance. If this limit is reached, then an IN_Q_OVERFLOW event is generated and **excess events are discarded**, typically 16,384.

`max_user_instances`

This is a limit on the number of *inotify* instances that can be created per real user ID, typically 128.

`max_user_watches`

A limit on the number of watch items that can be created per real user ID, typically 8,192.

The kernel does place some limits on the operation of the *inotify* mechanism

When you are done with the `inotify` file descriptor, call the `inotify_rm_watch()` function and close the file descriptor to free up the resources.



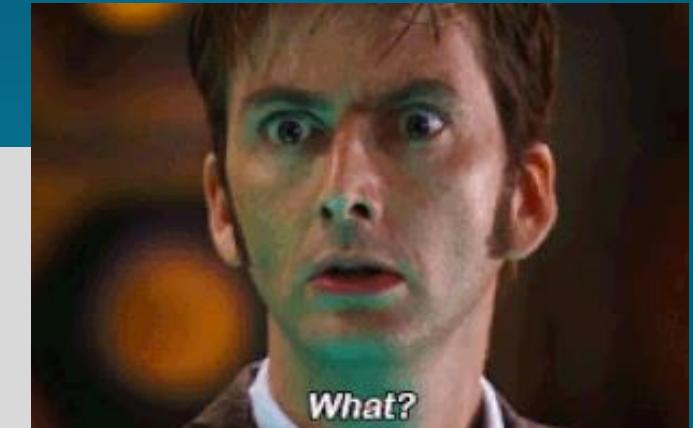
That “other” OS

Straight out of the MSDN documentation for the FileSystemWatcher Class:

“Note that a FileSystemWatcher **does not raise an Error event when an event is missed or when the buffer size is exceeded**, due to dependencies with the Windows operating system.”



Should this word be changed to deficiencies?



I've been burned by the FileSystemWatcher in Windows.

And the Other Other OS...



Apple OSX has the FSEvents and kqueues.

- The file system events (FSEvents) API is not designed for finding out when a particular file changes. For such purposes, the **kqueues** mechanism is more appropriate.
- The file system events (FSEvents) API is designed for passively monitoring a **large tree of files for changes**. The most obvious use for this technology is for backup software.
- The **kernel queues API** provides a way for an application to **receive notifications** whenever a given file or directory is modified in any way, including changes to the file's contents, attributes, name, or length.

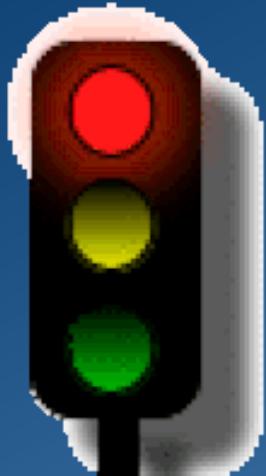
I've never used either of the Apple APIs for monitoring file and directory changes.



Intro to Operating Systems

TLPI 20: Signals - Fundamental Concepts

Unix Signals



- What are the various different signals and their purposes.
- When the kernel may generate a signal for a process, the system calls that one process may use to send a signal to another process.
- How a process responds to a signal and how a process can change its response to a signal.
- The use of a process signal mask to block signals, and the associated notion of pending signals.
- How a process can suspend itself and wait for the delivery of a signal.

Unix Signals

Signals have been around since the **1970s** Bell Labs Unix and have been more recently specified in the POSIX standard.

A signal is an **asynchronous** notification sent to a process or to a specific thread within the same process in order to notify it that **some sort of event has occurred**.

When a signal is sent, the operating system **interrupts** the target process' normal flow of execution to deliver the signal.

Queue the 70's music.



What is a Signal?

You are strolling along getting stuff done. On either side of you sits a segmentation fault or worse. Then, out of the blue, someone comes along and pokes you in the shoulder. You really don't know by whom you were poked, just that you were poked. That's a signal.



No, *Really* what is a Signal?

Time flies like wind.



Signal arrives

A signal is also often called
a **software interrupt**.

I love to ask the question “In Unix, a signal is also known
as a _____”

Fruit flies like



We now resume your regularly
scheduled programming.

Signal processing
code

AND NOW BACK TO
OUR REGULARLY
SCHEDULED
PROGRAMMING

Unix Signals



Unix Signals fall into two broad categories.

1. The first set constitutes the traditional or **standard signals**, which are used by the kernel to notify processes of events.
 - On Linux, the **standard signals** are numbered from **1 to 31**.
2. The other set of signals consists of the **real-time signals (33-64)**.

I love to ask the question “The standard signals on Linux are numbered ____ to ____”

Unix Signals

Some frequently seen signals for UNIX



Signal Name	Signal Value	Signal Meaning
SIGSEGV	11	Generated when a program makes an <i>invalid</i> memory reference. The reference may be invalid because the referenced page doesn't exist, the process tried to update a location in read-only memory, or the process tried to access a part of kernel memory while running in user mode
SIGPIPE	13	Generated when a process tries to write to a pipe, a FIFO, or a socket for which there is no corresponding reader process.
SIGINT	2	The user types the terminal interrupt character (usually Control-C), the terminal driver sends this signal to the foreground process
SIGCHLD	17	Sent by the kernel to a parent process when one of its children terminates.
SIGKILL	9	This is the <i>sure kill signal</i> . It cannot be blocked, ignored, or caught by a handler , and will always “terminate” a process.

Always use the mnemonic, not the number.

See “man 7 signal” or
“kill -l” for a complete list.

WAIT WAIT...
DON'T TELL ME![®]

[D] Network

Unix Signals, more

Signal Name	Signal Value	Signal Meaning
SIGQUIT	3	When the user types the quit character (usually Control-\) on the keyboard, this signal is sent to the foreground process
SIGTERM	15	This is the standard signal used for terminating a process and is the default signal sent by the <code>kill</code> and <code>killall</code> commands.
SIGUSR1	30,10,16	This signal is available for programmer-defined purposes.
SIGUSR2	31,12,17	This signal is available for programmer-defined purposes.
SIGPWR	29,30,19	This is the power failure signal. On systems that have an UPS, it is possible to set up a daemon process that monitors the backup battery level in the event of a power failure.
SIGBUS	7,10	This signal (“bus error”) is generated to indicate certain kinds of memory access errors.
SIGALRM	14	The kernel generates this signal upon the expiration of a real-time timer set by a call to <code>alarm()</code> or <code>setitimer()</code> .

Always use the mnemonic, not the number.

Default Actions of a Signal

1. The signal is **ignored**; meaning, it is discarded by the kernel and has no effect on the process.
2. The process is **terminated (killed)**. This is sometimes referred to as abnormal process termination, as opposed to the normal process termination that occurs when a process terminates using `exit()`.
3. A **core dump** file is generated and the process is terminated. A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger in order to inspect the state of the process at the time that it terminated.
4. The process is **stopped** — execution of the process is suspended (**Control-Z**).
5. Execution of the process is **resumed** after previously being stopped.



I love to ask the question “What are the possible **default** actions for a signal?”

Change Action of a Signal

A program can set one of the following **dispositions** for a signal:

1. The **default action should occur**.
 - This is useful to undo an earlier change of the disposition of the signal to something other than its default.
2. The **signal is ignored**.
 - This is useful for a signal whose default action would be to terminate the process.
3. A **signal handler is executed**.



I love to ask the question “What are the possible dispositions for a signal?”

Change Action of a Signal

```
int main(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGINT\n");

    while( 1 )
        sleep(1);
    return 0;
}

void sig_handler(int signo)
{
    if (SIGINT == signo)
        printf("received SIGINT\n");
}
```

Establishing a signal handler for the SIGINT signal.

You **DO NOT** explicitly call the signal handler. The kernel will invoke the handler as part of the delivery of the signal.



The signal handler.



Action of a Signal

Kernel

The kernel keeps track of what action a signal should perform in your process.

Your Process' Signal Table

Signal	Action
SIGHUP	terminate
SIGINT	terminate
SIGQUIT	terminate
...	
SIGUSR1	<code>usr1Handler</code>
SIGUSR2	SIG_IGN

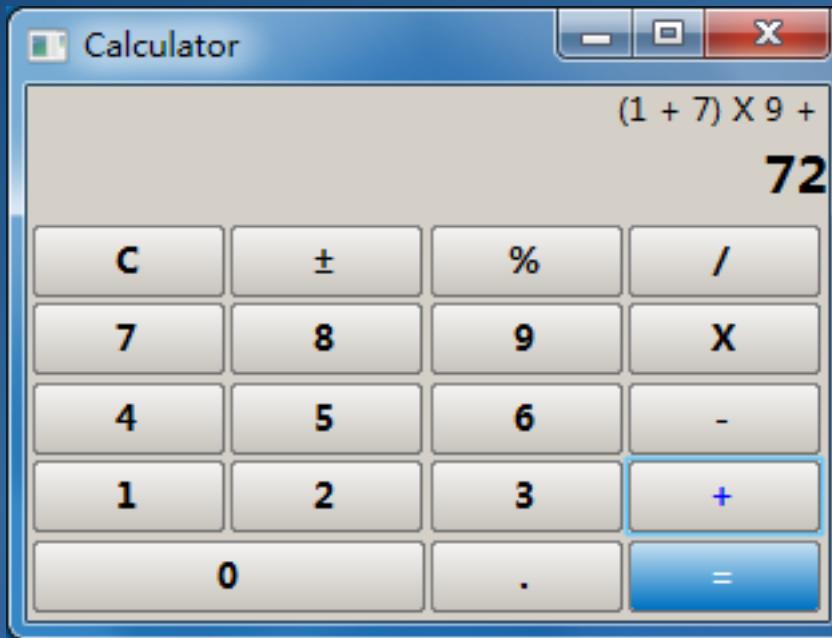
Your Process



`usr1Handler`

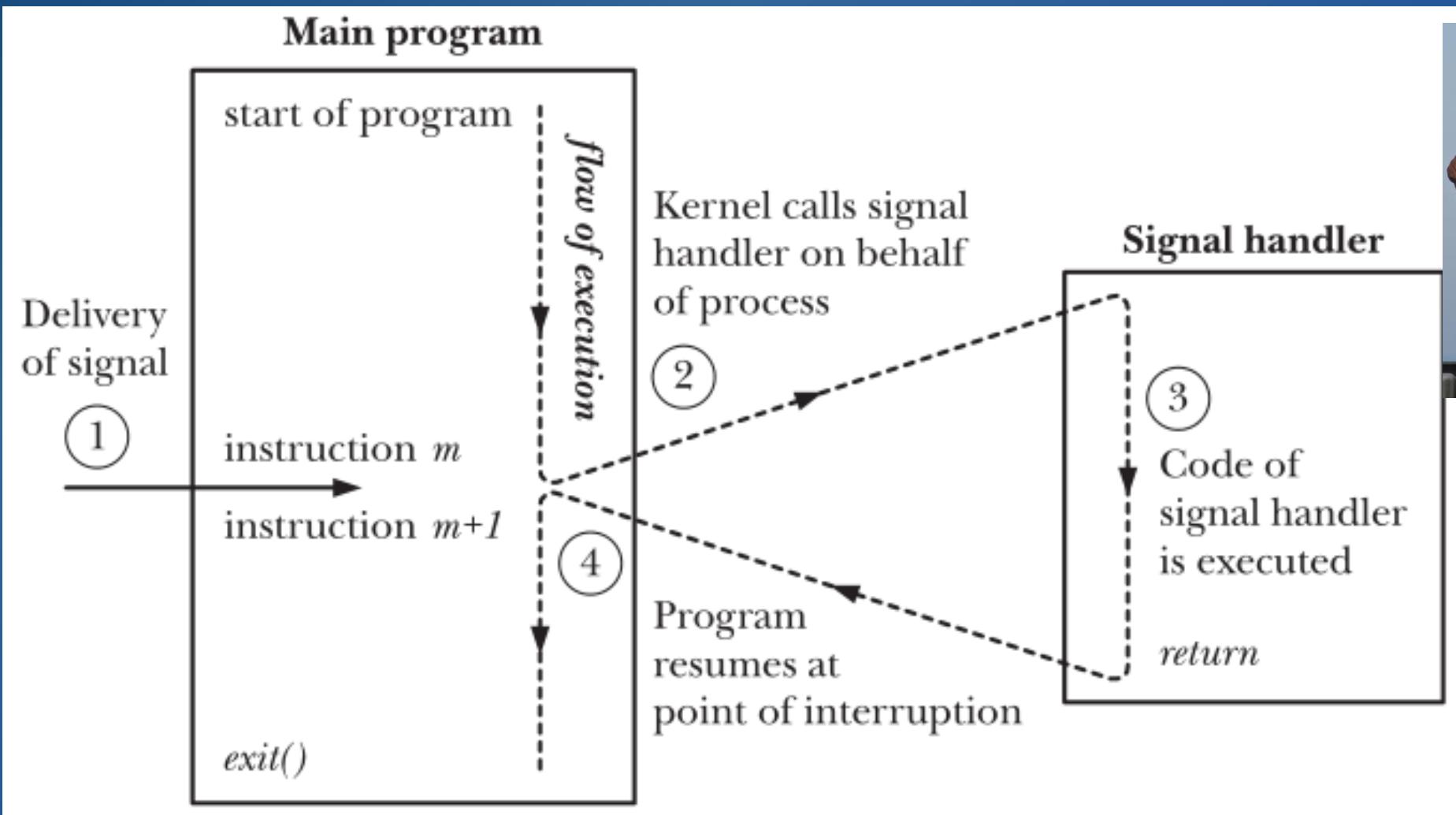
When a signal is sent to your process, it goes via the kernel. The kernel performs the task to determine: if the process should be terminated, the signal ignored, or if a function should be called within your process.

Signal Dispatch



- This is similar to how you program a graphical interface.
- You establish event handlers.
- You don't explicitly call the event handlers.
- An event dispatcher does that for you.
- The kernel is the dispatcher for signals sent to your process.

Signal Handlers



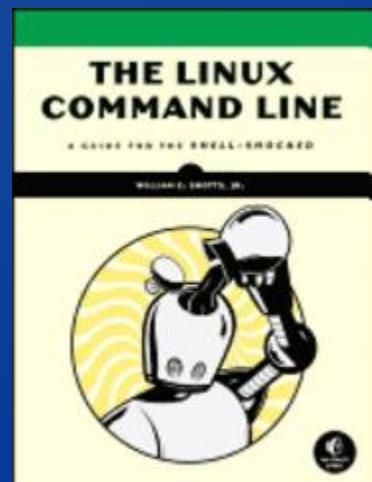
Sending Signals from the Command Line

```
kill [-s signal|-p] [-q sigval] [-a] [--] pid...
kill -l [signal]
```

The command **kill** sends the specified signal to the specified process or process group.

If no signal is specified, the **SIGTERM** signal is sent. The **SIGTERM** signal will terminate a processes which does not catch this signal.

For other processes, it may be necessary to use the **SIGKILL** (9) signal, since this signal cannot be caught, blocked, or ignored.

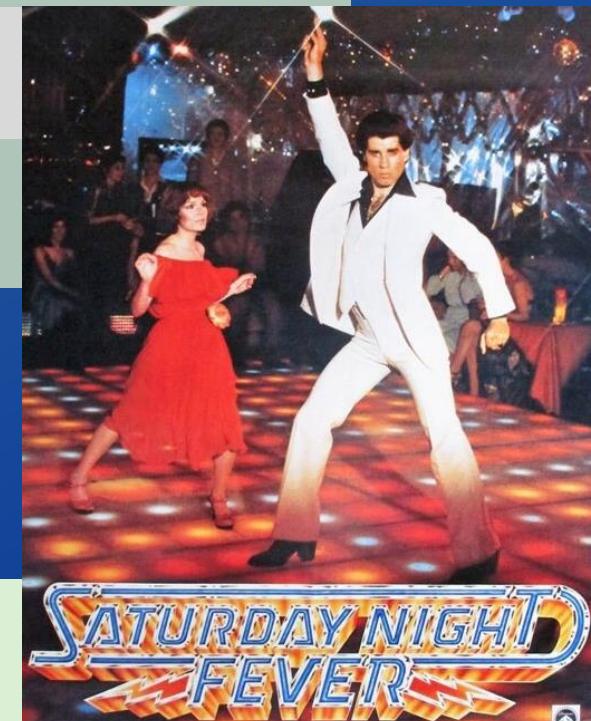
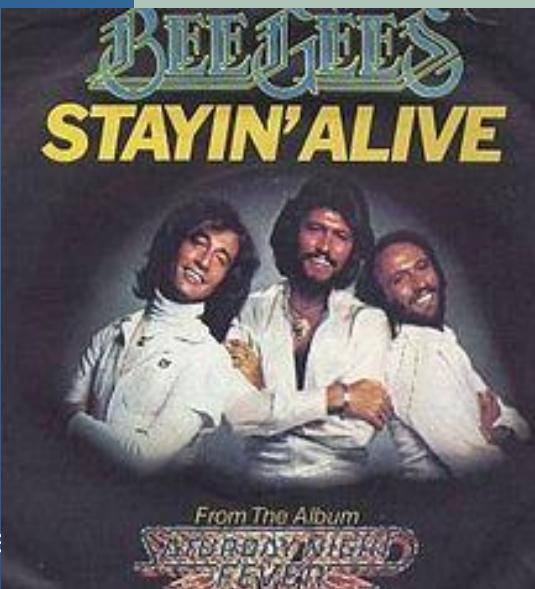


Sending Signals from a Program

```
#include <signal.h>  
  
int kill(pid_t pid, int sig);
```

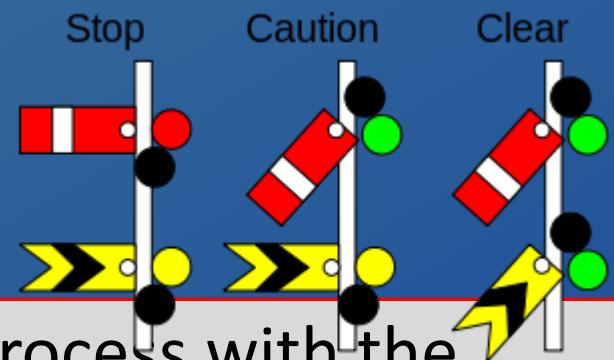
This is how you send a signal to a process, from another process.

The term **kill** was chosen because the default action of most of the signals on early UNIX implementations was to terminate the process.



I love to ask the question “How do you send a signal to another process from a program?”

Kill and pids



- If **pid is greater than 0**, the signal is sent to the process with the process ID specified by pid.
- If **pid equals 0**, the signal is sent to every process in the same **process group** as the calling process, including the calling process itself.
- If **pid is less than -1**, the signal is sent to all of the processes in the process group whose ID equals the absolute value of pid.
- If **pid equals -1**, the signal is sent to every process for which the calling process has permission to send a signal, except `init` ($\text{pid} = 1$) and the calling process.

Blaa, blaa, blaa. I really only care about the first one.

Sending Signals

An *unprivileged* process can send a signal to another process if the real or effective user ID of the sending process matches the real user ID or saved setuser-ID of the receiving process.

If a process doesn't have permissions to send a signal to the requested pid, then **kill () fails**, setting errno to EPERM.

Sending ...



Existence of a Process

How do you answer that age old question of the universe?

Is there a process with a specific pid ?

Send a kill () with a **signal id of 0** (zero, the null signal).

Remember that the list of normal signals is 1 to 31. So, 0 is, technically, not a signal.

I love to ask the question “How do you test for the existence of a specific pid?”



Other Ways to Send Signals

```
#include <signal.h>
```

I love to ask the question “What is an easy way for a process to send a signal to itself?”

```
int raise(int sig) ; // sends a signal to the process itself
```

```
int pthread_kill(pthread_self() , sig) ;
```

```
int killpg(pid_t pgrp , int sig ) ;
```

// sends a signal to all of the members of a process group



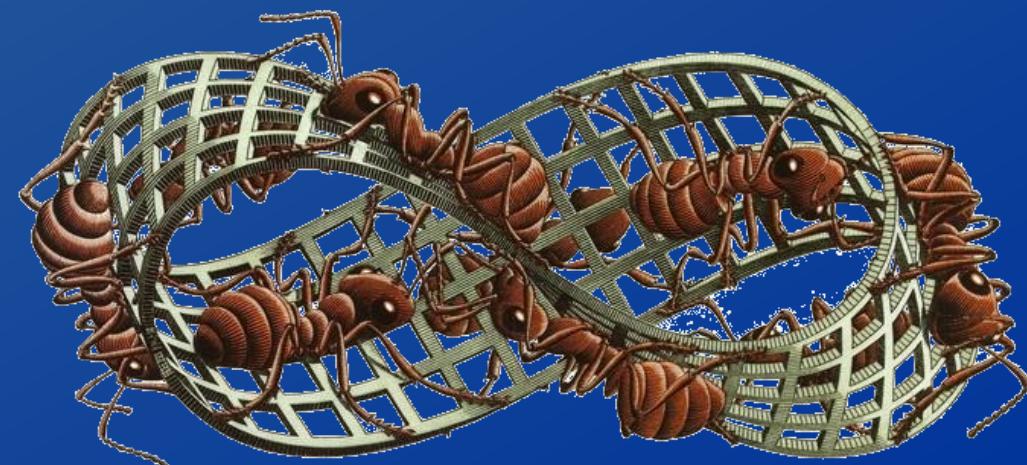
Signals Are Not Queued

You **cannot** reliably count how many times a (regular) signal has been received.

(Regular) Signals are not queued!!!

If the same signal is generated multiple times while it is blocked (e.g., in its signal handler), then it is recorded in the set of pending signals, and later delivered, **just once**.

I love to ask the question “How do you reliably count the number of times a signal has arrived?”



Changing Signal Dispositions

```
#include <signal.h>

void (*signal(int sig
, void (*func)(int)))(int);
```

- If the call succeeds, `signal()` will return the value of `func` for the most recent call to `signal()` for the specified signal `sig` (the old signal handler).
- On error, `SIG_ERR` shall be returned and a positive value shall be stored in `errno`.



Be the Change

```
#include <signal.h>
void *sig_ret = NULL;
void psignal(int sig, const char *s);

sig_ret = signal(SIGINT, handler_simple);
sig_ret = signal(SIGTERM, handler_simple);
sig_ret = signal(SIGKILL, handler_simple); ← This one will fail. Why?
... ... ...
```

```
void handler_simple(int sig) {
    psignal(sig, "\nCaught handler_simple");
    printf("Caught a signal %d: %s\n"
           , sig, sys_siglist[sig]);
}
```

I love to ask this question.

A very simple
handler.

Changing Signal Dispositions

```
#include <signal.h>

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

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

The struct sigaction

The function sigaction()

If `act` is non-NULL, the new action for signal `signum` is installed from `act`. If `oldact` is non-NULL, the previous action is saved in `oldact`.



Changing Signal Dispositions

```
struct sigaction new_handler;
struct sigaction old_handler;
int rhett = 0;

new_handler.sa_handler = handler_simple;
// Restart the system calls, if possible.
new_handler.sa_flags = SA_RESTART;

// Block every signal during the handler.
sigfillset(&new_handler.sa_mask);

rhett = sigaction(SIGINT
                  , &new_handler
                  , &old_handler);
```



```
struct sigaction new_handler;
struct sigaction old_handler;
int rhett = 0;

new_handler.sa_sigaction = whey_better;
// Restart the system calls, if possible.
new_handler.sa_flags = SA_RESTART | SA_SIGINFO;

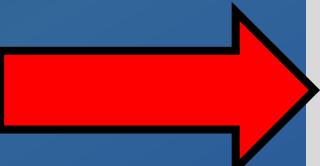
// Block every signal during the handler.
sigfillset(&new_handler.sa_mask);

rhett = sigaction(SIGINT
                  , &new_handler
                  , &old_handler);
```

This is not `sa_handler`.



This makes **more** information available to your signal handler.



```
void whey_better(int sig, siginfo_t *siginfo, void *ucontext)
{
    psignal(sig, "\nCaught whey_better!");
    switch (siginfo->si_code) {
        // Was the signal sent by:
        // the user, the kernel, a timer, ...
        case SI_USER:
        case SI_KERNEL:
    }
    if (siginfo->si_code == SI_USER) {
        printf("\tsignal sent by user %d pid %d\n"
               , siginfo->si_uid, siginfo->si_pid);
    }
}
```



This lets you know which user and pid sent the signal.

Waiting for a Signal

```
#include <unistd.h>

int pause(void);
```

Calling `pause()` **suspends** the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

Simple Signal

```
void sig_handler(int signo)
{
    if (signo == SIGINT)
        printf("\n    received SIGINT\n");
}

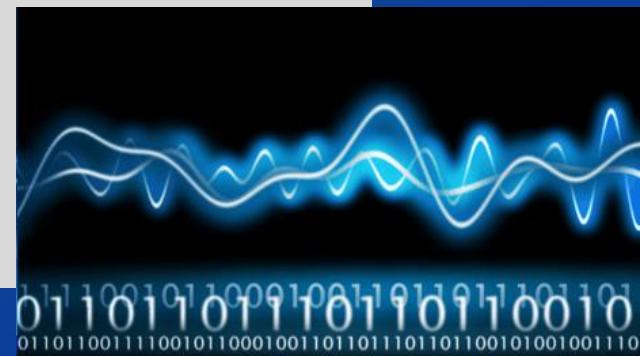
int main(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGINT\n");

    while(pause())
        ;
    return 0;    How would you terminate
}                                this process?
```

Don't forget these.

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

How would you terminate this process?



Ignore and Restore

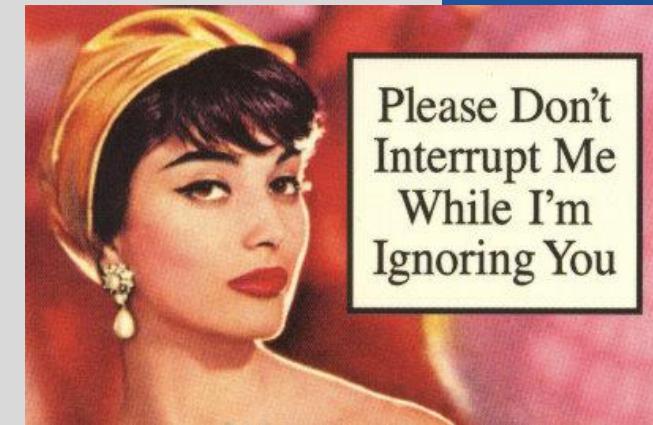
```
int main(void)
{
    ...
    if (signal(SIGINT, SIG_IGN) == SIG_ERR)
```

This means ignore the signal completely.

I love to ask the question “How do you configure a signal to be ignored?”

```
...
if (signal(SIGINT, SIG_DFL) == SIG_ERR)
...
}
```

This means restore the default action to the signal.



I love to ask the question “How do you restore a signal to its default action?”

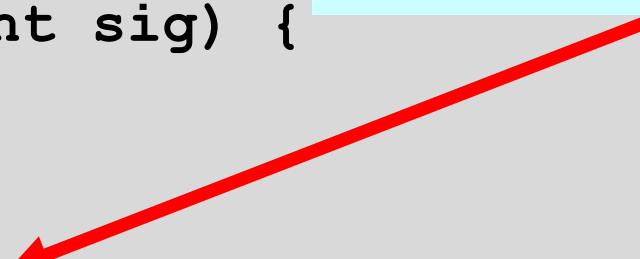
Those SIGCHLD Thingies

```
int main(void)
{
    (void) signal(SIGCHLD, sigchld_handler);
    ...
    fork();
    ...
}
void sigchld_handler(int sig) {
    pid_t cpid;
    int status;

    while ((cpid = waitpid(-1, &status, WNOHANG)) > 0) {
        printf("SIGCHLD handler: child pid: %d  exit status: %d\n"
               , cpid, WEXITSTATUS(status));
    }
}
```

Why loop calling `waitpid()`?

Calling `waitpid()` with `WNOHANG`, not `wait()`, in the signal handler.



Loading Image

Please Wait...

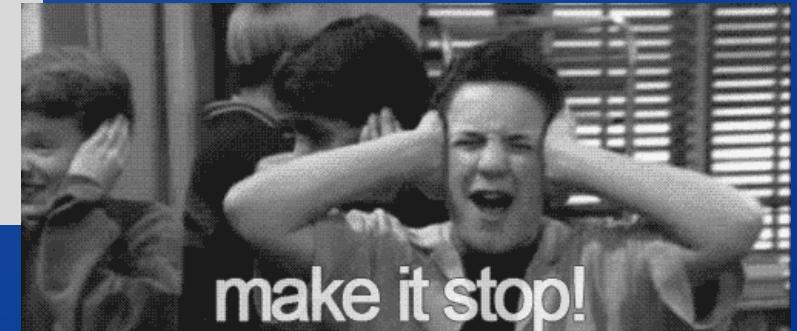
Blocking Signals

The kernel maintains a signal mask for each process - a set of signals whose delivery to the process is currently **blocked** (or masked out).

```
#include <signal.h>

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

If a signal that is blocked is sent to a process, delivery of that signal is delayed until it is unblocked by being removed from the process signal mask.



Blocking Signals

```
#include <signal.h>

sigset_t newMask, oldMask;
sigemptyset(&newMask);
sigaddset(&newMask, SIGINT);
sigaddset(&newMask, SIGTERM);
sigprocmask(SIG_BLOCK, &newMask, &oldMask);
```



Pending Signals

If a process receives a signal that it is currently blocked, that signal is added to the process' set of **pending** signals.

```
#include <signal.h>

int sigpending(sigset_t * set);
```

`sigpending()` returns the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked).



When (and if) the signal is later unblocked, it is then delivered to the process.

Pending Signals

```
#include <signal.h>
#include <stddef.h>

sigset_t waiting_mask;

sigpending(&waiting_mask);
if (sigismember(&waiting_mask, SIGINT)) {
    /* User has sent a SIGINT to the process. */
}
```





Simple Signal Handlers

It is **preferable** to write **simple signal handlers**.

An important reason for this is to **reduce the risk of creating race conditions**.

1. The signal handler sets a global flag and returns. The main program periodically checks this flag and, if it is set, takes appropriate action.
2. The signal handler performs some type of cleanup and then either terminates the process or uses a **nonlocal goto** to unwind the stack and return control to a predetermined location in the main program.

simplicity

Reentrant and Async-Signal-Safe Functions

- Not all system calls and library functions can be **safely** called from a signal handler.
- An async-signal-safe function is one that the implementation **guarantees to be safe when called from within a signal handler**.
- A function is async-signal-safe either because it is **reentrant** or because it is **not interruptible** by a signal handler.



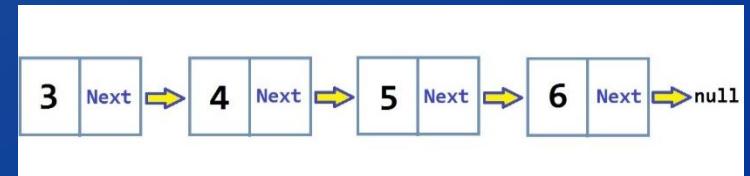
Non-Async-Signal-Safe

The `malloc()` and `free()` functions maintain a **linked list** of free memory blocks available for allocation from the heap.

If a call to `malloc()` in the main program is interrupted by a signal handler that also calls `malloc()`, then the **linked list can be corrupted**.

For this reason, **the `malloc()` family of functions, and other library functions that use them, are non-reentrant**.

As good exam question, explain why `malloc()` is/is not an async-signal-safe function.



Async-Signal-Safe Handlers

You have two choices when writing signal handlers:

1. Ensure that the code of the signal handler itself is reentrant and that it calls only async-signal-safe functions.
2. Block delivery of signals while executing code in the main program that calls unsafe functions or works with global data structures also updated by the signal handler.



Interruption of System Calls

Consider the following scenario:

1. We establish a handler for some signal.
2. We make a **blocking** system call, for example, a `read()` from a terminal device, which blocks until input is supplied.
3. While the system call is blocked, the signal for which we established a handler is delivered, and its signal handler is invoked.



Interruption of System Calls

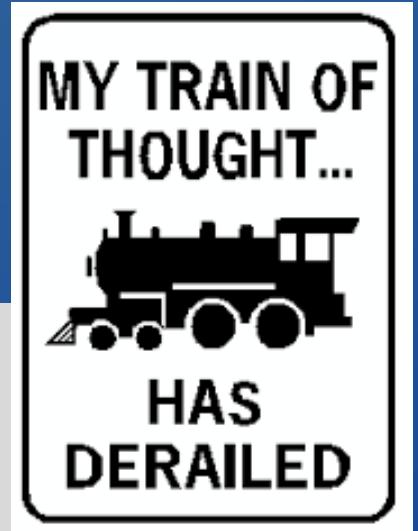
What happens with the system function after the signal handler returns?

By default, the **system call fails** with the error EINTR (“Interrupted function”).



This Code Helps

```
ssize_t cnt = 0;    Just keep restarting when a EINTR occurs.  
  
while (((cnt = read(fd, buf, BUF_SIZE)) == -1)  
      && (EINTR == errno)) {  
    continue; /* Do nothing loop body */  
}  
  
if (-1 == cnt) { /* read() failed with other than EINTR */  
    exit(EXIT_FAILURE);  
}
```



Automatic Restart

Having signal handlers interrupt system calls can be ***inconvenient***, since we must add code to each blocking system call (assuming that we want to restart the call in each case).

Soooo, we can specify the `SA_RESTART` flag when establishing the signal handler with `sigaction()`, so that system calls are **automatically (auto-magically!!!)** restarted by the kernel on the process' behalf.

This means that we don't need to handle a possible `EINTR` error return for these system calls.

At least it should, in theory, mostly, sorta, sometimes...



Automatic Restart, almost

Unfortunately, not all blocking system calls automatically restart as a result of specifying `SA_RESTART`.



The I/O system calls are interruptible, and hence automatically restarted by `SA_RESTART` **only when operating on a “slow” device.**

- **Sloooooooow devices** include **terminals, pipes, FIFOs, and sockets**. On these file types, various I/O operations may block.
- **Disk files don’t fall into the category of slow devices**, because disk I/O operations generally can be immediately satisfied via the buffer cache.



The Real-time Signals

- Real-time signals provide an increased range of signals that can be used for application-defined purposes.
- **Real-time signals are queued.**
- When sending a real-time signal, it is possible to specify data (an integer or pointer value) that accompanies the signal.
- The **order of delivery of different real-time signals is guaranteed.**

The real-time signals are really outside the scope of this class. ☹

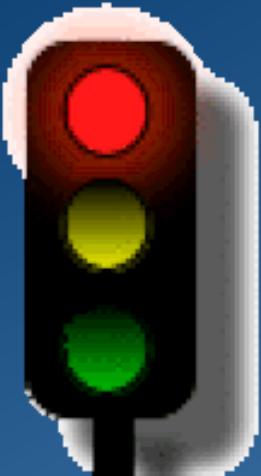




Intro to Operating Systems

TLPI 20: Signals - Fundamental Concepts

Unix Signals



- What are the various different signals and their purposes.
- When the kernel may generate a signal for a process, the system calls that one process may use to send a signal to another process.
- How a process responds to a signal and how a process can change its response to a signal.
- The use of a process signal mask to block signals, and the associated notion of pending signals.
- How a process can suspend itself and wait for the delivery of a signal.

Unix Signals

Signals have been around since the **1970s** Bell Labs Unix and have been more recently specified in the POSIX standard.

A signal is an **asynchronous** notification sent to a process or to a specific thread within the same process in order to notify it that **some sort of event has occurred**.

When a signal is sent, the operating system **interrupts** the target process' normal flow of execution to deliver the signal.

Queue the 70's music.



What is a Signal?

You are strolling along getting stuff done. On either side of you sits a segmentation fault or worse. Then, out of the blue, someone comes along and pokes you in the shoulder. You really don't know by whom you were poked, just that you were poked. That's a signal.



No, *Really* what is a Signal?

Time flies like wind.



Signal arrives

A signal is also often called
a **software interrupt**.

I love to ask the question “In Unix, a signal is also known
as a _____”

Signal processing
code

Fruit flies like



We now resume your regularly
scheduled programming.



AND NOW BACK TO
OUR REGULARLY
SCHEDULED
PROGRAMMING

Unix Signals



Unix Signals fall into two broad categories.

1. The first set constitutes the traditional or **standard signals**, which are used by the kernel to notify processes of events.
 - On Linux, the **standard signals** are numbered from **1 to 31**.
2. The other set of signals consists of the **real-time signals (33-64)**.

I love to ask the question “The standard signals on Linux are numbered ____ to ____”

Unix Signals

Some frequently seen signals for UNIX



Signal Name	Signal Value	Signal Meaning
SIGSEGV	11	Generated when a program makes an <i>invalid</i> memory reference. The reference may be invalid because the referenced page doesn't exist, the process tried to update a location in read-only memory, or the process tried to access a part of kernel memory while running in user mode
SIGPIPE	13	Generated when a process tries to write to a pipe, a FIFO, or a socket for which there is no corresponding reader process.
SIGINT	2	The user types the terminal interrupt character (usually Control-C), the terminal driver sends this signal to the foreground process
SIGCHLD	17	Sent by the kernel to a parent process when one of its children terminates.
SIGKILL	9	This is the <i>sure kill signal</i> . It cannot be blocked, ignored, or caught by a handler , and will always “terminate” a process.

Always use the mnemonic, not the number.

See “man 7 signal” or
“kill -l” for a complete list.

WAIT WAIT...
DON'T TELL ME![®]

[D] Network

Unix Signals, more

Signal Name	Signal Value	Signal Meaning
SIGQUIT	3	When the user types the quit character (usually Control-\) on the keyboard, this signal is sent to the foreground process
SIGTERM	15	This is the standard signal used for terminating a process and is the default signal sent by the <code>kill</code> and <code>killall</code> commands.
SIGUSR1	30,10,16	This signal is available for programmer-defined purposes.
SIGUSR2	31,12,17	This signal is available for programmer-defined purposes.
SIGPWR	29,30,19	This is the power failure signal. On systems that have an UPS, it is possible to set up a daemon process that monitors the backup battery level in the event of a power failure.
SIGBUS	7,10	This signal (“bus error”) is generated to indicate certain kinds of memory access errors.
SIGALRM	14	The kernel generates this signal upon the expiration of a real-time timer set by a call to <code>alarm()</code> or <code>setitimer()</code> .

Always use the mnemonic, not the number.

Default Actions of a Signal

1. The signal is **ignored**; meaning, it is discarded by the kernel and has no effect on the process.
2. The process is **terminated (killed)**. This is sometimes referred to as abnormal process termination, as opposed to the normal process termination that occurs when a process terminates using `exit()`.
3. A **core dump** file is generated and the process is terminated. A core dump file contains an image of the virtual memory of the process, which can be loaded into a debugger in order to inspect the state of the process at the time that it terminated.
4. The process is **stopped** — execution of the process is suspended (**Control-Z**).
5. Execution of the process is **resumed** after previously being stopped.



I love to ask the question “What are the possible **default** actions for a signal?”

Change Action of a Signal

A program can set one of the following **dispositions** for a signal:

1. The **default action should occur**.
 - This is useful to undo an earlier change of the disposition of the signal to something other than its default.
2. The **signal is ignored**.
 - This is useful for a signal whose default action would be to terminate the process.
3. A **signal handler is executed**.



I love to ask the question “What are the possible dispositions for a signal?”

Change Action of a Signal

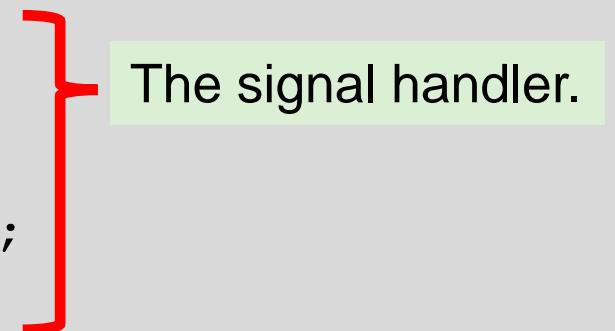
```
int main(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGINT\n");

    while( 1 )
        sleep(1);
    return 0;
}

void sig_handler(int signo)
{
    if (SIGINT == signo)
        printf("received SIGINT\n");
}
```

Establishing a signal handler for the SIGINT signal.

You **DO NOT** explicitly call the signal handler. The kernel will invoke the handler as part of the delivery of the signal.



Action of a Signal

Kernel

The kernel keeps track of what action a signal should perform in your process.

Your Process' Signal Table

Signal	Action
SIGHUP	terminate
SIGINT	terminate
SIGQUIT	terminate
...	
SIGUSR1	<code>usr1Handler</code>
SIGUSR2	SIG_IGN

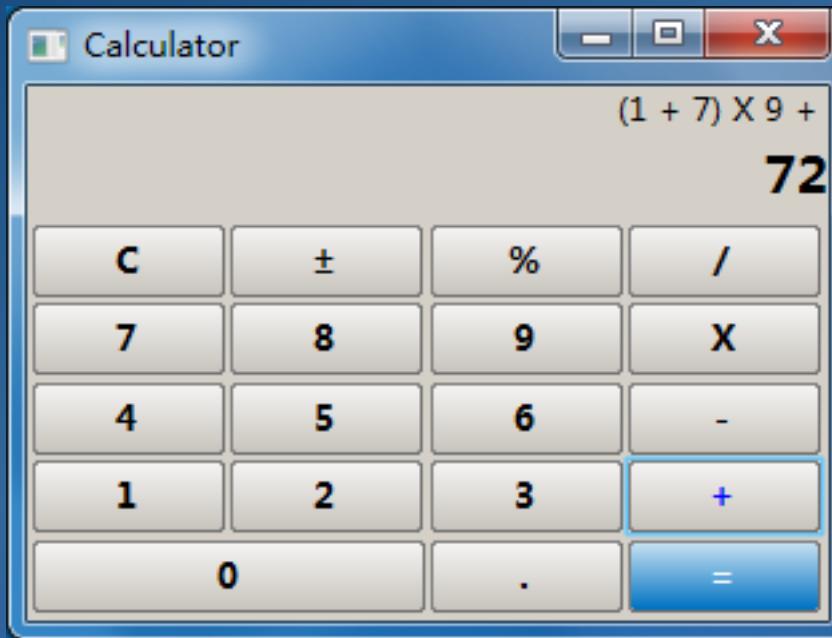
Your Process



`usr1Handler`

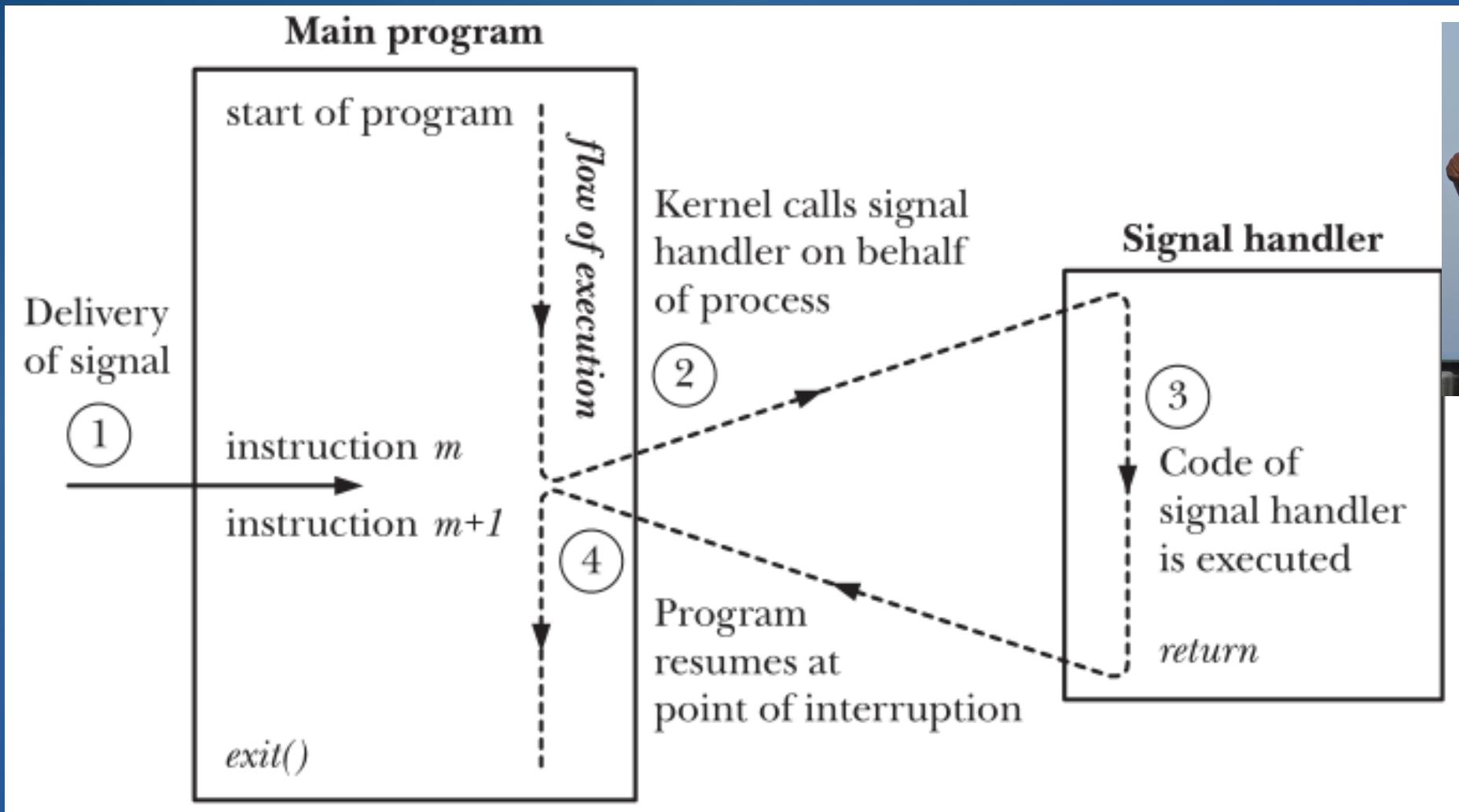
When a signal is sent to your process, it goes via the kernel. The kernel performs the task to determine: if the process should be terminated, the signal ignored, or if a function should be called within your process.

Signal Dispatch



- This is similar to how you program a graphical interface.
- You establish event handlers.
- You don't explicitly call the event handlers.
- An event dispatcher does that for you.
- The kernel is the dispatcher for signals sent to your process.

Signal Handlers



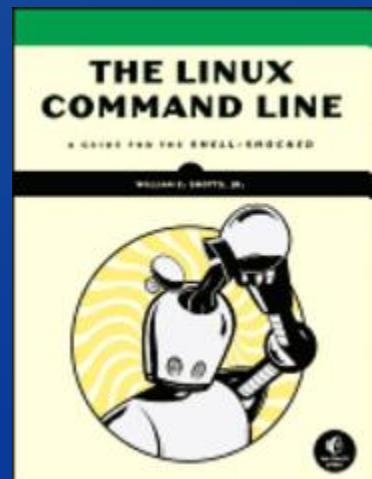
Sending Signals from the Command Line

```
kill [-s signal|-p] [-q sigval] [-a] [--] pid...
kill -l [signal]
```

The command **kill** sends the specified signal to the specified process or process group.

If no signal is specified, the **SIGTERM** signal is sent. The **SIGTERM** signal will terminate a processes which does not catch this signal.

For other processes, it may be necessary to use the **SIGKILL** (9) signal, since this signal cannot be caught, blocked, or ignored.

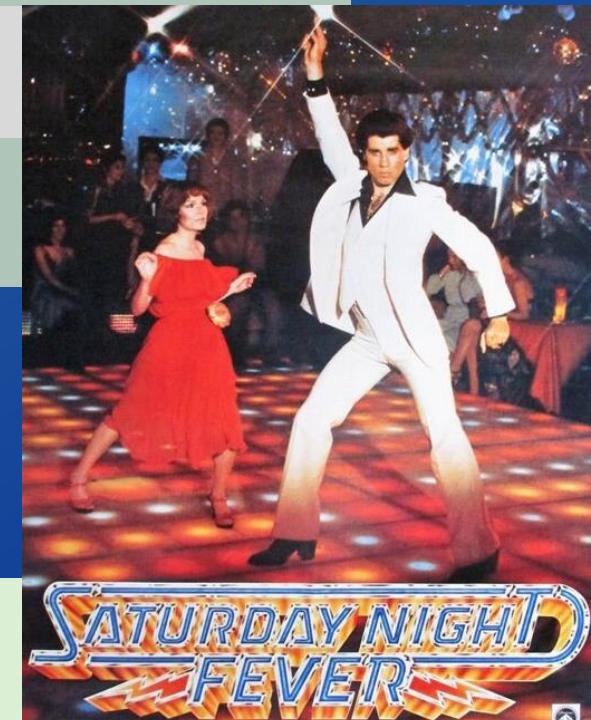
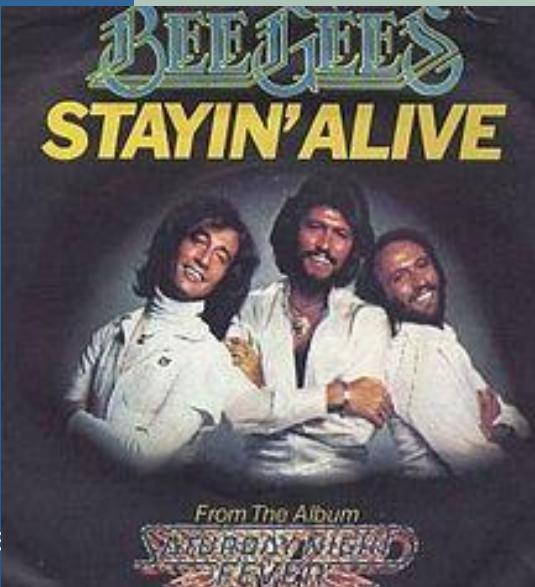


Sending Signals from a Program

```
#include <signal.h>  
  
int kill(pid_t pid, int sig);
```

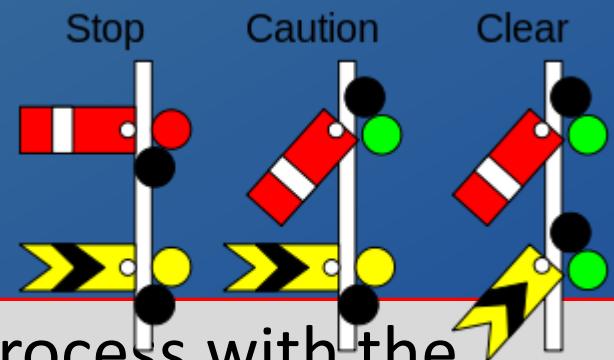
This is how you send a signal to a process, from another process.

The term **kill** was chosen because the default action of most of the signals on early UNIX implementations was to terminate the process.



I love to ask the question “How do you send a signal to another process from a program?”

Kill and pids



- If **pid is greater than 0**, the signal is sent to the process with the process ID specified by pid.
- If **pid equals 0**, the signal is sent to every process in the same **process group** as the calling process, including the calling process itself.
- If **pid is less than -1**, the signal is sent to all of the processes in the process group whose ID equals the absolute value of pid.
- If **pid equals -1**, the signal is sent to every process for which the calling process has permission to send a signal, except `init` ($\text{pid} = 1$) and the calling process.

Blaa, blaa, blaa. I really only care about the first one.

Sending Signals

An *unprivileged* process can send a signal to another process if the real or effective user ID of the sending process matches the real user ID or saved setuser-ID of the receiving process.

If a process doesn't have permissions to send a signal to the requested pid, then **kill () fails**, setting errno to EPERM.

Sending ...



Existence of a Process

How do you answer that age old question of the universe?

Is there a process with a specific pid ?

Send a kill () with a **signal id of 0** (zero, the null signal).

Remember that the list of normal signals is 1 to 31. So, 0 is, technically, not a signal.

I love to ask the question “How do you test for the existence of a specific pid?”



Other Ways to Send Signals

```
#include <signal.h>
```

I love to ask the question “What is an easy way for a process to send a signal to itself?”

```
int raise(int sig) ; // sends a signal to the process itself
```

```
int pthread_kill(pthread_self() , sig) ;
```

```
int killpg(pid_t pgrp , int sig ) ;
```

// sends a signal to all of the members of a process group



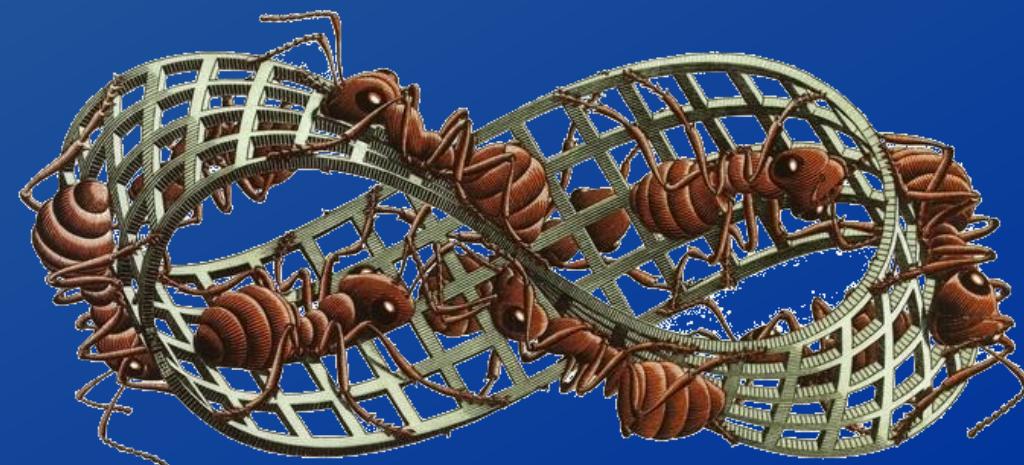
Signals Are Not Queued

You **cannot** reliably count how many times a (regular) signal has been received.

(Regular) Signals are not queued!!!

If the same signal is generated multiple times while it is blocked (e.g., in its signal handler), then it is recorded in the set of pending signals, and later delivered, **just once**.

I love to ask the question “How do you reliably count the number of times a signal has arrived?”



Changing Signal Dispositions

```
#include <signal.h>

void (*signal(int sig
, void (*func)(int)))(int);
```

- If the call succeeds, `signal()` will return the value of `func` for the most recent call to `signal()` for the specified signal `sig` (the old signal handler).
- On error, `SIG_ERR` shall be returned and a positive value shall be stored in `errno`.



Be the Change

```
#include <signal.h>
void *sig_ret = NULL;
void psignal(int sig, const char *s);

sig_ret = signal(SIGINT, handler_simple);
sig_ret = signal(SIGTERM, handler_simple);
sig_ret = signal(SIGKILL, handler_simple); ← This one will fail. Why?
... ... ...
```

```
void handler_simple(int sig) {
    psignal(sig, "\nCaught handler_simple");
    printf("Caught a signal %d: %s\n"
           , sig, sys_siglist[sig]);
}
```

I love to ask this question.

A very simple
handler.

Changing Signal Dispositions

```
#include <signal.h>

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

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

The struct sigaction

The function sigaction()

If `act` is non-NULL, the new action for signal `signum` is installed from `act`. If `oldact` is non-NULL, the previous action is saved in `oldact`.



Changing Signal Dispositions

```
struct sigaction new_handler;
struct sigaction old_handler;
int rhett = 0;

new_handler.sa_handler = handler_simple;
// Restart the system calls, if possible.
new_handler.sa_flags = SA_RESTART;

// Block every signal during the handler.
sigfillset(&new_handler.sa_mask);

rhett = sigaction(SIGINT
                  , &new_handler
                  , &old_handler);
```



```
struct sigaction new_handler;
struct sigaction old_handler;
int rhett = 0;

new_handler.sa_sigaction = whey_better;
// Restart the system calls, if possible.
new_handler.sa_flags = SA_RESTART | SA_SIGINFO;

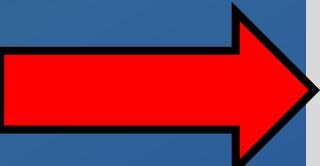
// Block every signal during the handler.
sigfillset(&new_handler.sa_mask);

rhett = sigaction(SIGINT
                  , &new_handler
                  , &old_handler);
```

This is not `sa_handler`.



This makes **more** information available to your signal handler.



```
void whey_better(int sig, siginfo_t *siginfo, void *ucontext)
{
    psignal(sig, "\nCaught whey_better!");
    switch (siginfo->si_code) {
        // Was the signal sent by:
        // the user, the kernel, a timer, ...
        case SI_USER:
        case SI_KERNEL:
    }
    if (siginfo->si_code == SI_USER) {
        printf("\tsignal sent by user %d pid %d\n"
               , siginfo->si_uid, siginfo->si_pid);
    }
}
```



This lets you know which user and pid sent the signal.

Waiting for a Signal

```
#include <unistd.h>

int pause(void);
```

Calling `pause()` **suspends** the calling thread until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

Simple Signal

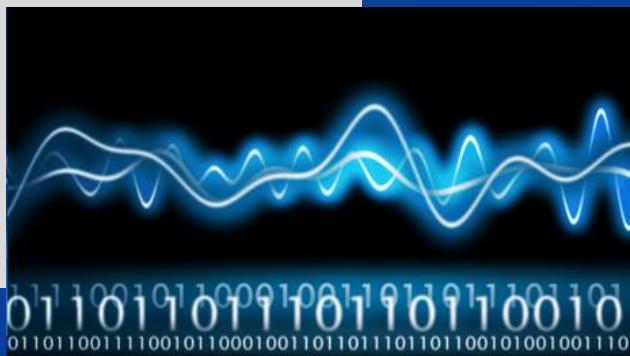
```
void sig_handler(int signo)
{
    if (signo == SIGINT)
        printf("\n    received SIGINT\n");
}

int main(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
        printf("\ncan't catch SIGINT\n");

    while(pause())
        ;
    return 0;    How would you terminate
}                                this process?
```

Don't forget these.

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



Ignore and Restore

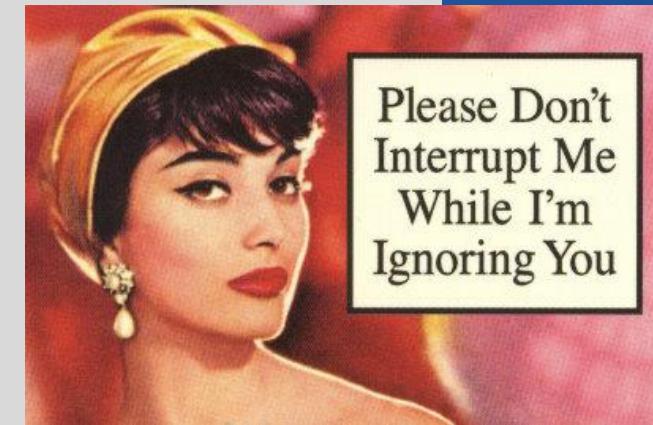
```
int main(void)
{
    ...
    if (signal(SIGINT, SIG_IGN) == SIG_ERR)
```

This means ignore the signal completely.

I love to ask the question “How do you configure a signal to be ignored?”

```
...
if (signal(SIGINT, SIG_DFL) == SIG_ERR)
...
}
```

This means restore the default action to the signal.



I love to ask the question “How do you restore a signal to its default action?”

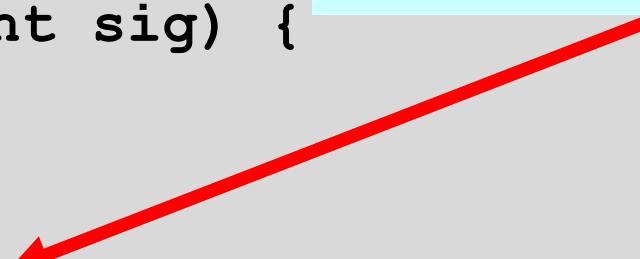
Those SIGCHLD Thingies

```
int main(void)
{
    (void) signal(SIGCHLD, sigchld_handler);
    ...
    fork();
    ...
}
void sigchld_handler(int sig) {
    pid_t cpid;
    int status;

    while ((cpid = waitpid(-1, &status, WNOHANG)) > 0) {
        printf("SIGCHLD handler: child pid: %d  exit status: %d\n"
               , cpid, WEXITSTATUS(status));
    }
}
```

Why loop calling `waitpid()`?

Calling `waitpid()` with `WNOHANG`, not `wait()`, in the signal handler.



Loading Image

Please Wait...

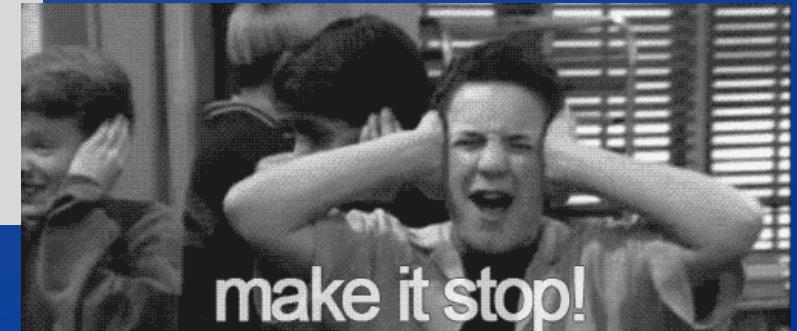
Blocking Signals

The kernel maintains a signal mask for each process - a set of signals whose delivery to the process is currently **blocked** (or masked out).

```
#include <signal.h>

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

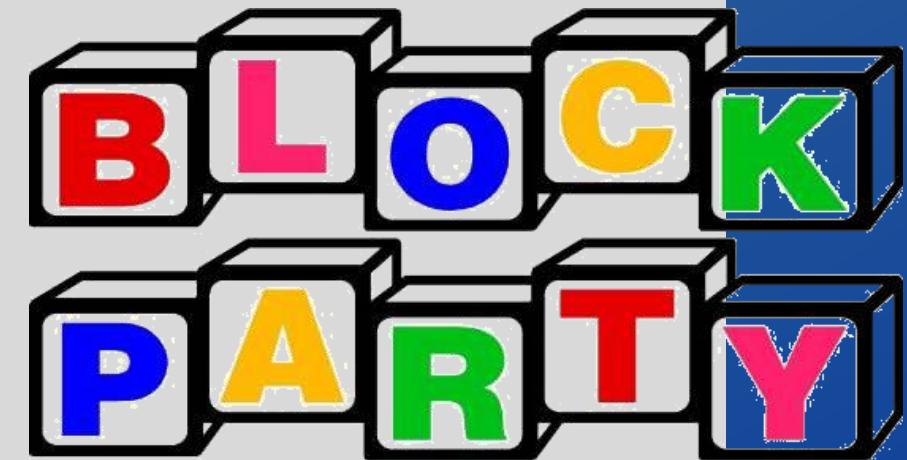
If a signal that is blocked is sent to a process, delivery of that signal is delayed until it is unblocked by being removed from the process signal mask.



Blocking Signals

```
#include <signal.h>

sigset_t newMask, oldMask;
sigemptyset(&newMask);
sigaddset(&newMask, SIGINT);
sigaddset(&newMask, SIGTERM);
sigprocmask(SIG_BLOCK, &newMask, &oldMask);
```



Pending Signals

If a process receives a signal that it is currently blocked, that signal is added to the process' set of **pending** signals.

```
#include <signal.h>

int sigpending(sigset_t * set);
```

`sigpending()` returns the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked).



When (and if) the signal is later unblocked, it is then delivered to the process.

Pending Signals

```
#include <signal.h>
#include <stddef.h>

sigset_t waiting_mask;

sigpending(&waiting_mask);
if (sigismember(&waiting_mask, SIGINT)) {
    /* User has sent a SIGINT to the process. */
}
```





Simple Signal Handlers

It is **preferable** to write **simple signal handlers**.

An important reason for this is to **reduce the risk of creating race conditions**.

1. The signal handler sets a global flag and returns. The main program periodically checks this flag and, if it is set, takes appropriate action.
2. The signal handler performs some type of cleanup and then either terminates the process or uses a **nonlocal goto** to unwind the stack and return control to a predetermined location in the main program.

simplicity

Reentrant and Async-Signal-Safe Functions

- Not all system calls and library functions can be **safely** called from a signal handler.
- An async-signal-safe function is one that the implementation **guarantees to be safe when called from within a signal handler**.
- A function is async-signal-safe either because it is **reentrant** or because it is **not interruptible** by a signal handler.



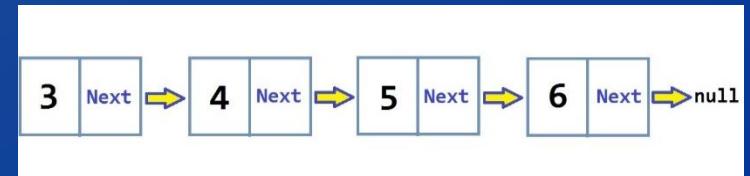
Non-Async-Signal-Safe

The `malloc()` and `free()` functions maintain a **linked list** of free memory blocks available for allocation from the heap.

If a call to `malloc()` in the main program is interrupted by a signal handler that also calls `malloc()`, then the **linked list can be corrupted**.

For this reason, **the `malloc()` family of functions, and other library functions that use them, are non-reentrant**.

As good exam question, explain why `malloc()` is/is not an async-signal-safe function.



Async-Signal-Safe Handlers

You have two choices when writing signal handlers:

1. Ensure that the code of the signal handler itself is reentrant and that it calls only async-signal-safe functions.
2. Block delivery of signals while executing code in the main program that calls unsafe functions or works with global data structures also updated by the signal handler.



Interruption of System Calls

Consider the following scenario:

1. We establish a handler for some signal.
2. We make a **blocking** system call, for example, a `read()` from a terminal device, which blocks until input is supplied.
3. While the system call is blocked, the signal for which we established a handler is delivered, and its signal handler is invoked.



Interruption of System Calls

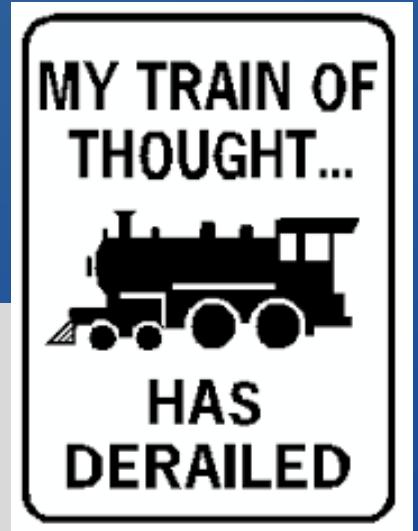
What happens with the system function after the signal handler returns?

By default, the **system call fails** with the error EINTR (“Interrupted function”).



This Code Helps

```
ssize_t cnt = 0;    Just keep restarting when a EINTR occurs.  
  
while (((cnt = read(fd, buf, BUF_SIZE)) == -1)  
      && (EINTR == errno)) {  
    continue; /* Do nothing loop body */  
}  
  
if (-1 == cnt) { /* read() failed with other than EINTR */  
    exit(EXIT_FAILURE);  
}
```



Automatic Restart

Having signal handlers interrupt system calls can be ***inconvenient***, since we must add code to each blocking system call (assuming that we want to restart the call in each case).

Soooo, we can specify the `SA_RESTART` flag when establishing the signal handler with `sigaction()`, so that system calls are **automatically (auto-magically!!!)** restarted by the kernel on the process' behalf.

This means that we don't need to handle a possible `EINTR` error return for these system calls.

At least it should, in theory, mostly, sorta, sometimes...



Automatic Restart, almost

Unfortunately, not all blocking system calls automatically restart as a result of specifying `SA_RESTART`.



The I/O system calls are interruptible, and hence automatically restarted by `SA_RESTART` **only when operating on a “slow” device.**

- **Sloooooooow devices** include **terminals, pipes, FIFOs, and sockets**. On these file types, various I/O operations may block.
- **Disk files don’t fall into the category of slow devices**, because disk I/O operations generally can be immediately satisfied via the buffer cache.



The Real-time Signals

- Real-time signals provide an increased range of signals that can be used for application-defined purposes.
- **Real-time signals are queued.**
- When sending a real-time signal, it is possible to specify data (an integer or pointer value) that accompanies the signal.
- The **order of delivery of different real-time signals is guaranteed.**

The real-time signals are really outside the scope of this class. ☹





VIKINGS™

CS 333

Intro to Operating Systems
TLPI 24: Process Creation (procreation)



PORTLAND STATE™

Process Creation and Management

- The `fork()` system call allows one process, the parent, to create a new process, the child.
- The `exit(status)` library function terminates a process, **freeing all resources** (memory, open file descriptors, ...) used by the process available for subsequent reallocation by the kernel.
- The `wait(&status)` system call allows a **parent process to synchronize with a child** and determine the child's exit status.
- The `exec(...)` system call **loads a new program** into a process' memory space **(overwriting the old one)**.

What is a Process?

- An instance of a running program.
- Identified by a unique PID (Process ID).
 - Each PID is unique within a system, but may be **recycled**.
- Created by a parent process as its child process.
- One process can be parent process of multiple child processes.
- A process can be killed or stopped by sending it a signal.

You WILL see this as a question on an exam!



Creating a New Process: `fork()`

In many applications, creating multiple processes can be a useful way of dividing up a task.

Dividing tasks up as multiple processes can make application design simpler.

It may also permit greater concurrency.



The `fork()` system call **creates a new process**.

The child process is, **almost, an exact duplicate** of the calling process, the parent.

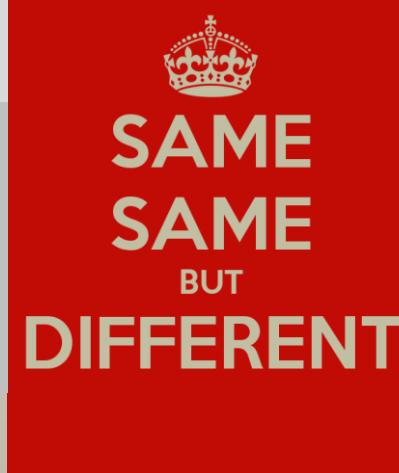
The key to understanding `fork()` is to realize that after it has (successfully)

returned, **two** processes exist, and each process execution continues from the point where `fork()` returns.



Creating a New Process: `fork()`

- Following a call to `fork()`, the **two** processes (**parent and child**) are **executing the same program** text (program).
 - However, **the two processes have separate copies of the text, stack, data, and heap segments.**
- The child's stack, data, and heap segments are initially exact duplicates of the corresponding parts the parent's memory.
- After the `fork()`, each process can modify the variables in its stack, data, and heap segments without affecting the other process.



How can We Tell Parent from Child?

- Within the code of a program, we can distinguish the two processes via **the value returned from `fork()`**.
- **For the parent, `fork()` returns the process ID of the newly created child.**
 - This is useful because the parent may create several child processes.
- **For the child, `fork()` returns 0.**
 - If necessary, the child can obtain its own process ID using `getpid()`, and the process ID of its parent using `getppid()`.

Process Attributes

Parent Process Attributes that **are Not Inherited by Child**

- PID
- PPID (parent PID)



Parent Process Attributes that **are Inherited by Child**

- Resource limits
- Real UID and GID
- Effective UID and GID
- Current working directory
- Open file descriptors
- umask value
- Environment variables

The Call to `fork()`

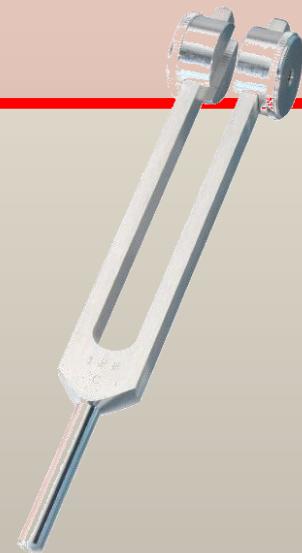
```
#include <unistd.h>  
  
pid_t fork(void);
```

On success,

- the **PID of the child process is returned in the parent**, and
- 0 is returned in the child.

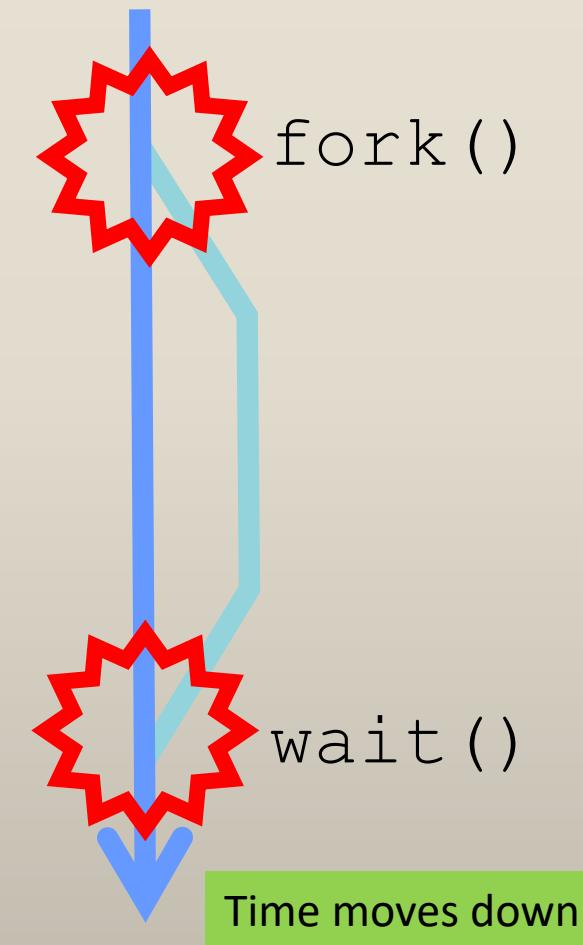
On failure,

- -1 is returned in the parent,
- no child process is created, and
- errno is set appropriately.

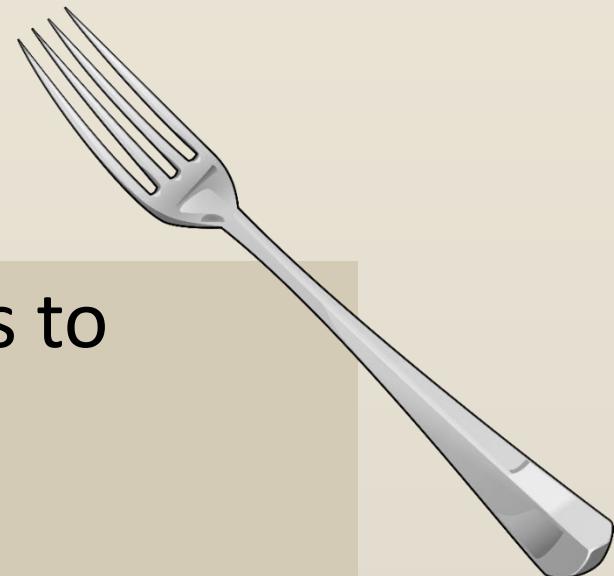


Fork and Wait vs Fork and Join

Fork and Wait are the same
as Fork and Join



What is Fork?

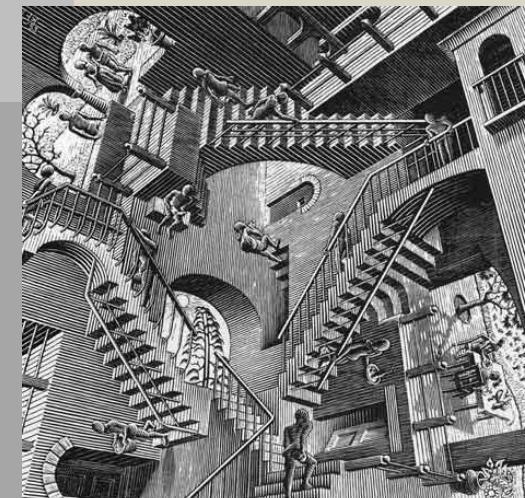


A call to the **fork()** function allows a process to ***create a new process***.

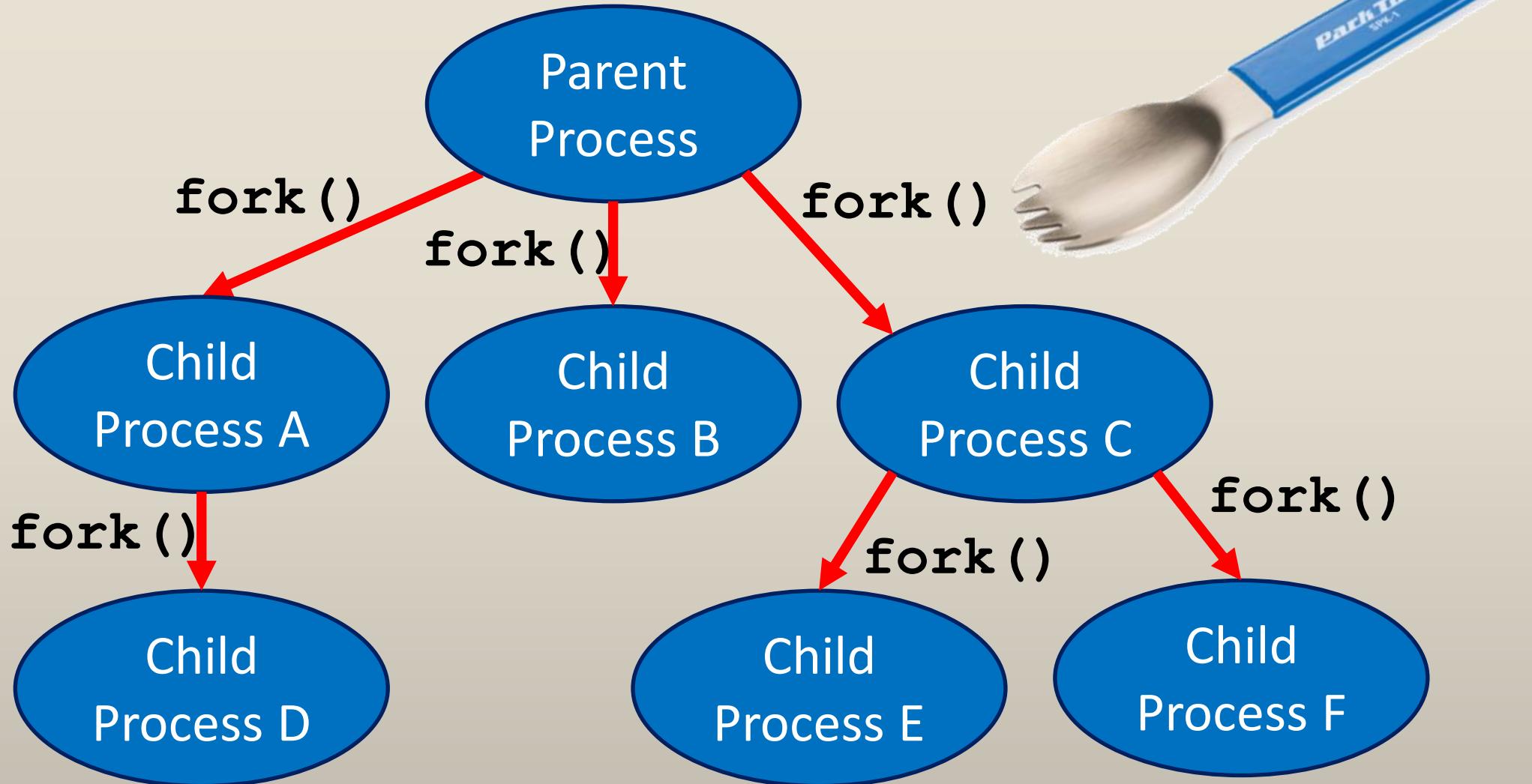
1. The **process that calls fork()** is called the **parent process**.
2. The **newly created process** is called a **child process**.
3. Specifically, the **child process is called a child of the parent/creating process**.

Process Creation Steps

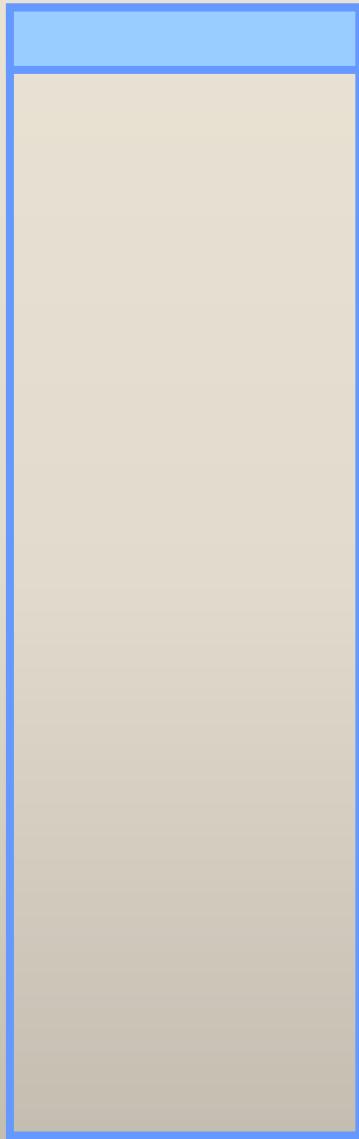
- A process forks a child by first **replicating its own process image**.
- Child may `exec()` the image of another program.
The call to `exec()` **overwrites** the memory of the calling process.
- The parent process may,
 - wait for child to complete execution (foreground execution)
 - continue with other tasks (background execution).
- When the child process terminates, parent picks up exit status of child.
- The kernel removes entry for exited process from process table.



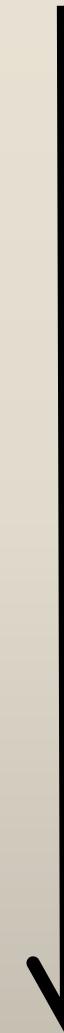
Relationships between Processes



*Parent process
running program A*



time



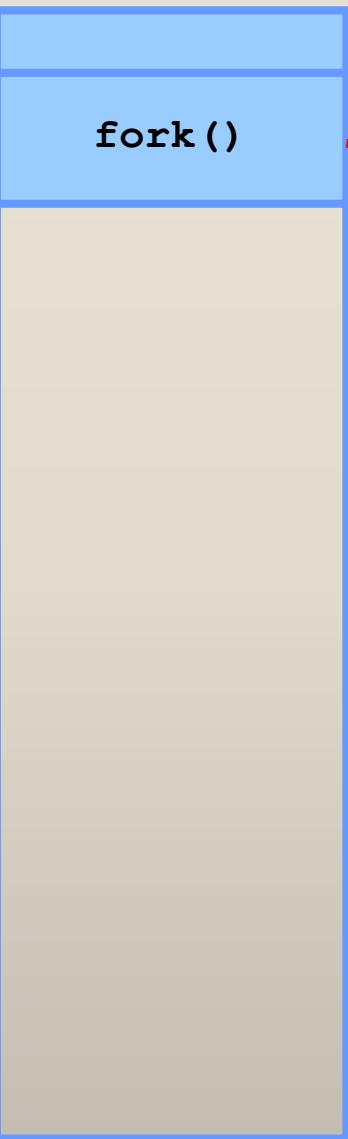
*Parent process
running program A*



time



*Parent process
running program A*



Memory of parent
copied to child

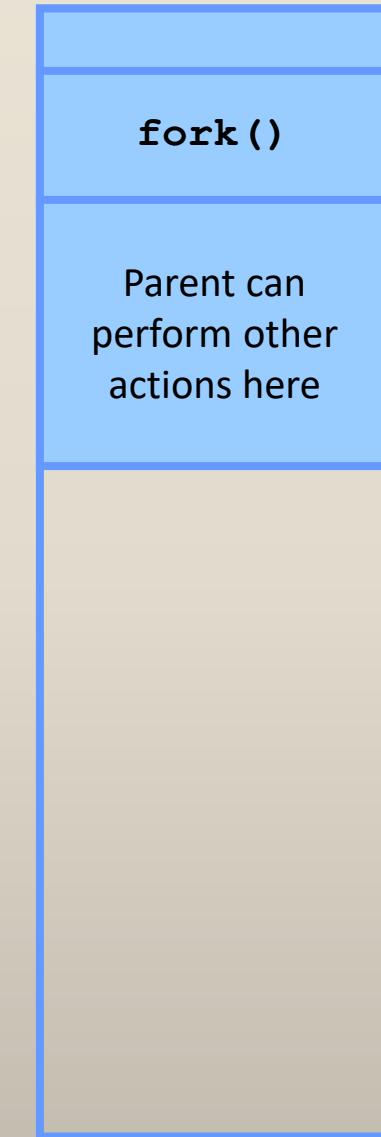
*Child process
running program A*



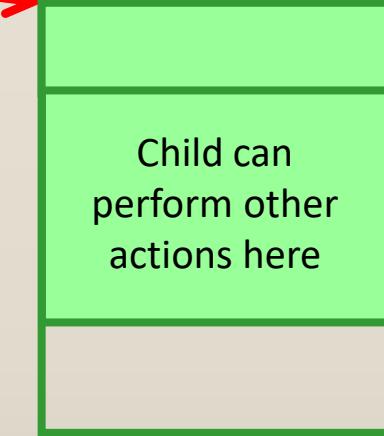
time



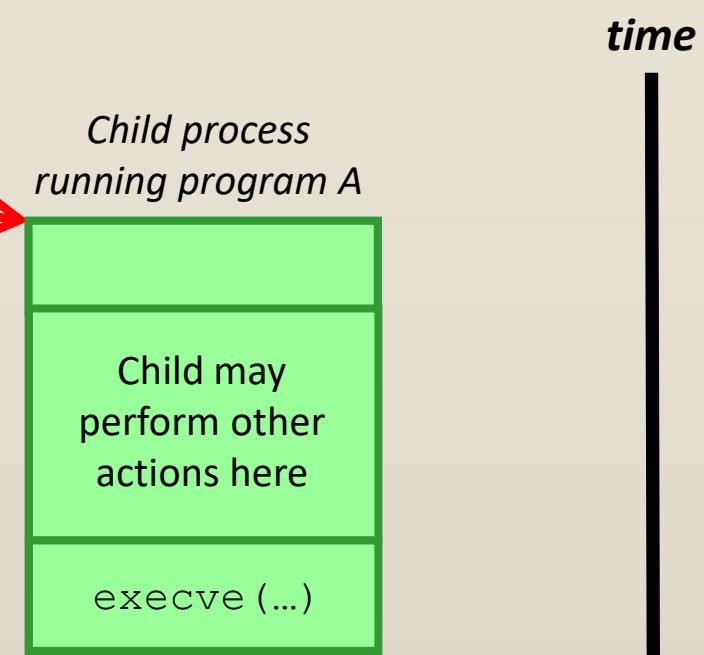
*Parent process
running program A*



*Child process
running program A*



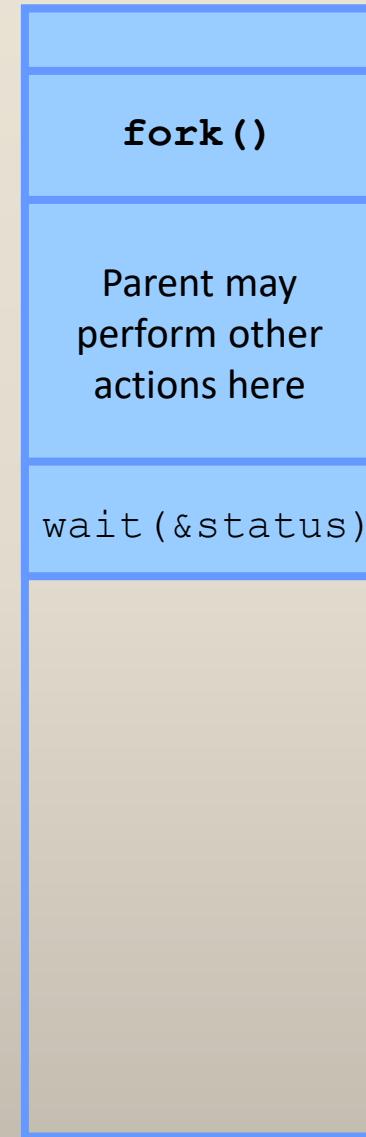
*Parent process
running program A*



time

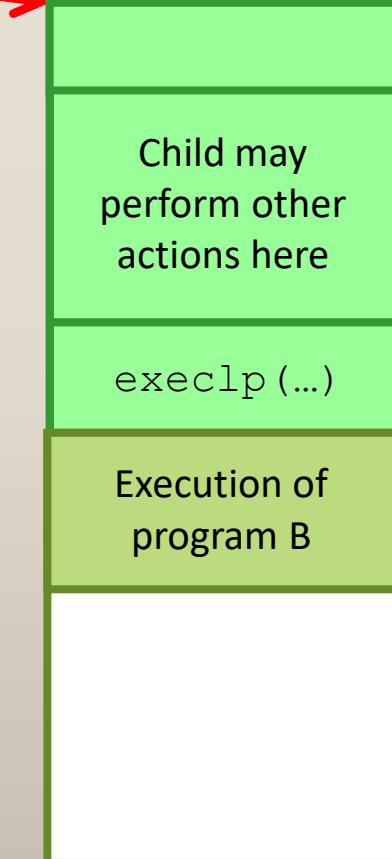


*Parent process
running program A*



Memory of parent
copied to child

*Child process (no
longer running A)*



time

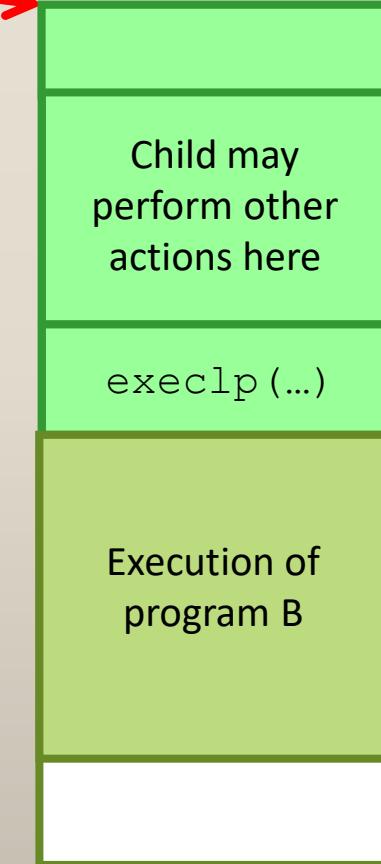


*Parent process
running program A*



Memory of parent copied to child

Child process



time



*Parent process
running program A*

`fork()`

Parent may
perform other
actions here

`wait(&status)`

Execution of
parent suspended

Kernel “resumes”
parent and
delivers SIGCHLD

Memory of parent
copied to child

Child process

Child may
perform other
actions here

`execvp(...)`

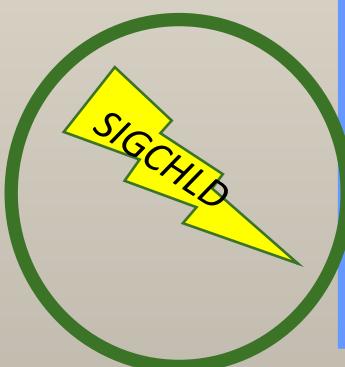
Execution of
program B

`exit(status)`

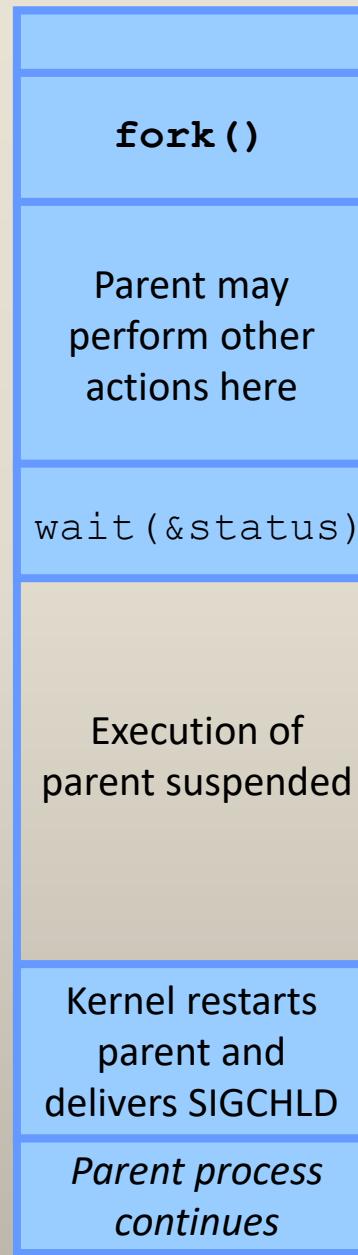
time



Child status
passed to parent

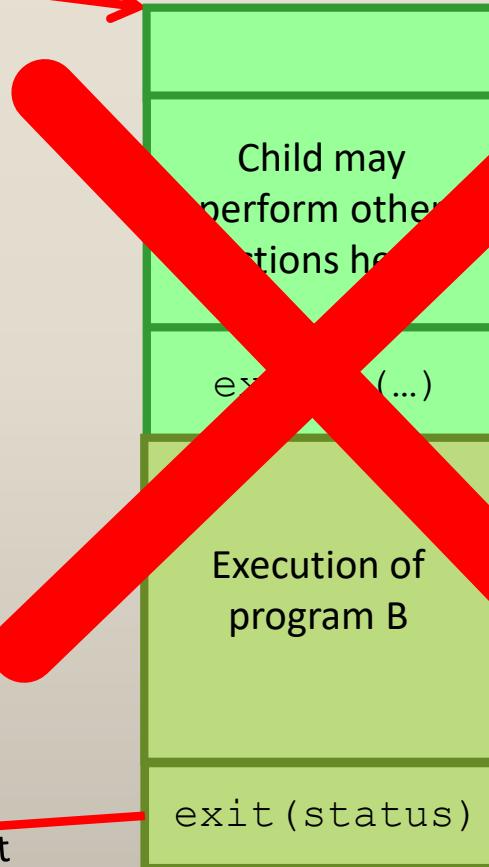


*Parent process
running program A*



Memory of parent copied to child

Child process



Child status passed to parent

time



Most Simple Example

```
int main( void ) {
    pid_t pid;
    printf("The original process with PID %d and PPID %d.\n"
           , getpid(), getppid());
    pid = fork();
    if (0 == pid) { /* The child process */
        printf("The child process with PID %d and PPID %d.\n"
               , getpid(), getppid());
    }
    else { /* Parent process */
        printf("The parent process with PID %d and PPID %d.\n"
               , getpid(), getppid());
        printf("Child PID is %d\n", pid);
    }
    /* Both Processes run this */
    printf("PID %d terminates.\n", getpid());
}
```

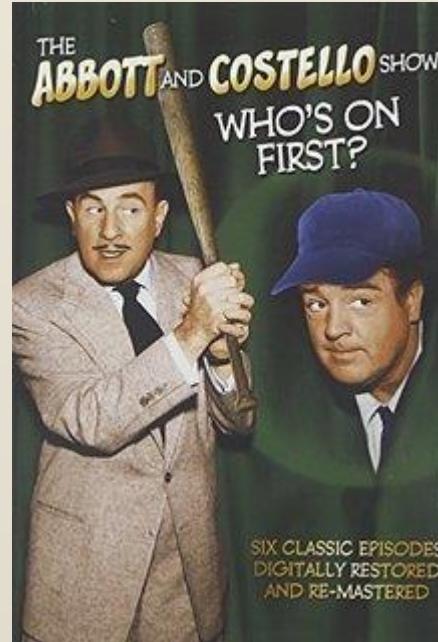
Returned pid is 0 for the child process.

Returned pid is not 0 for the parent process.

simple is beautiful.

Who's on First?

Q: When a parent process creates a child process using a call to `fork()`, **which process runs first**, the parent or the child?



After a successful call to `fork()`, it is **indeterminate** which process – the parent or the child – next has access to the CPU.



The Memory Semantics of `fork()`

- In theory, we consider `fork()` as creating copies of the parent's text, data, heap, and stack segments.
- Actually performing a complete copy of the parent's virtual memory pages into the new child process would be wasteful.
- Most current UNIX implementations, including Linux, use a couple techniques to avoid this wasteful copying.



The `vfork()` System Call

- Early BSD implementations were among those in which `fork()` performed **a literal duplication of the parent's data, heap, and stack.**
- This is wasteful, especially if the `fork()` is followed by an immediate `exec()`.
- The modern UNIX implementations employing copy-on-write for implementing `fork()` are much more efficient than older `fork()` implementations, thus largely eliminating the need for `vfork()`.

The `vfork()` System Call

Two features distinguish the `vfork()` system call from `fork()` to make it more efficient:

1. **No duplication of virtual memory pages** or page tables is done for the child process.
 - The **child shares** the parent's memory until it either performs a successful `exec()` or calls `_exit()` to terminate.
2. **Execution of the parent process is suspended** until the child has performed an `exec()` or `_exit()`.

The `vfork()` System Call

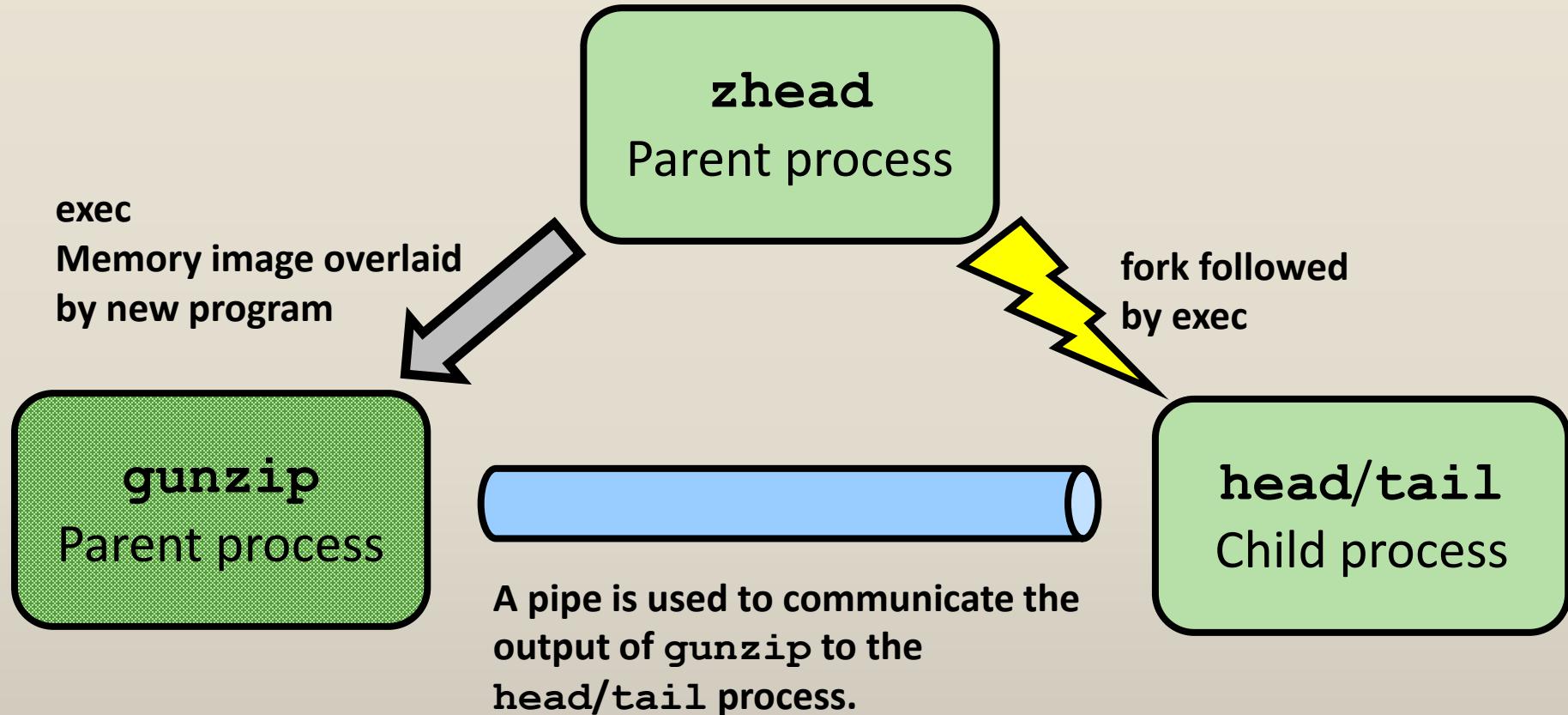
- The semantics of `vfork()` mean that after the call, the child is **guaranteed** to be scheduled for the CPU before the parent.
- Where it is used, `vfork()` should generally be immediately followed by a call to `exec()`.



Race Conditions After `fork()`

- After a `fork()`, **it is indeterminate which process – the parent or the child – will be scheduled next on the CPU.**
- Applications that **neeeeeed** a particular sequence of execution in order to achieve correct results are open to failure due to race conditions.
- **If you must guarantee a particular order, you must use some kind of synchronization technique.**





When the **gunzip** process completes, it closes the pipe and exits. The closed pipe is detected by the **head/tail** process and it exits.



VIKINGS™

CS 333

Intro to Operating Systems

TLPI 25: Process Termination



PORTLAND STATE

Terminating a Process

In general, a process may terminate in either of two ways

1. **Abnormal termination**, caused by the delivery of a signal whose default action is to terminate the process.
2. A process can **terminate normally**, using the `_exit()` or `exit()` system calls.



Exit Stage Right

```
void exit(int status);
```

- Exits the current process with status.
- All functions registered with `atexit()` and `on_exit()` are called, in the reverse order of their registration.

The exit status of the program.

- **All open stdio streams are flushed and closed.**
- Files created by `tmpfile()` are removed.



Exit Value

By convention, a termination status of 0 indicates that a process completed successfully, and a nonzero status value indicates that the process terminated unsuccessfully.

Although any exit value in the range 0 to 255 can be passed to the parent via the status argument to `exit()`, specifying a value greater than 128 can cause confusion in shell scripts.

When a command is terminated by a signal, the shell indicates this fact by setting the value of the variable `$?` to 128 **plus** the signal number.



Exit vs Return



Performing an **explicit `return (n)` in `main()`** is generally equivalent to calling `exit(n)`, since the run-time function that invokes `main()` uses the return value from `main()` in a call to `exit()`.

Performing a return without specifying a value, or falling off the end of the `main()` function, also results in the caller of `main()` invoking `exit()`. However ...

- In **C89**, the behavior in these circumstances is undefined; the program can terminate with an arbitrary status value.
- The **C99** standard requires that falling off the end of the main program should be equivalent to calling `exit(0)`.

The Other `_exit`

```
void _exit(int status);
```

The function `_exit()` terminates the calling process "*immediately*".

- The function `_exit()` is like `exit()`, but does **not** call any functions registered with `atexit()` or `on_exit()`.
- **Streams might NOT be flushed.**
- Temporary files **might** not be removed.
- File descriptors are closed.



`exit()` vs `_exit()`

Typically only one process of a family of processes (most often the parent) should terminate via `exit()`.

The child processes should terminate via `_exit()`.

29

Um

The element of
CONFUSION

Processes Playing Nice

```
int nice(int inc);
```

- Adds inc to the nice value for the calling process.
- **Changes the priority level of a process**
- Only super user and kernel process can have a negative priority.
- A higher nice value means a low priority.
- Priority's range from -20 to 19

A higher nice value means the process is less likely to be scheduled to run by the kernel.

More means less and less is more.

I try to be a nice person But sometimes my mouth doesn't want to cooperate!



Exit Handlers

- Sometimes, an application needs to **automatically perform some operations on process termination.**
- An exit handler is a programmer-supplied function that is registered during the life of the process and is then **automatically called during normal process termination via `exit()`.**

Sort of like a class destructor from C++.



Exit Handlers

```
#include <stdlib.h>

int atexit(void (*function) (void)) ;

atexit(func);

void func( void )
{
    /* Perform some actions */
}
```

After a normal termination
only (ie `exit()`)

The function `func()` will be called
after the call to `exit()` is made,
before the application terminates.



More Exit Handlers

```
#include <stdlib.h>

int on_exit(void (*function) (int , void *)
, void *arg);

void func( int status, void *arg )
{
    /* Perform cleanup actions */
}
```



The **non-standard `on_exit()`** function allows the registered `exit()` handler to take 2 arguments.

1. The exit status.
2. A pointer.

Portable applications should avoid this function, and use the standard `atexit()` instead.

Exit Handlers and `exec()`

A child process created via `fork()` inherits a copy of its parent's exit handler registrations.

When a process performs an `exec()`, all exit handler registrations are removed.

I often ask a question about this on an exam. Something like:

When a process calls one of the `exec()` functions, all exit handlers are retained.

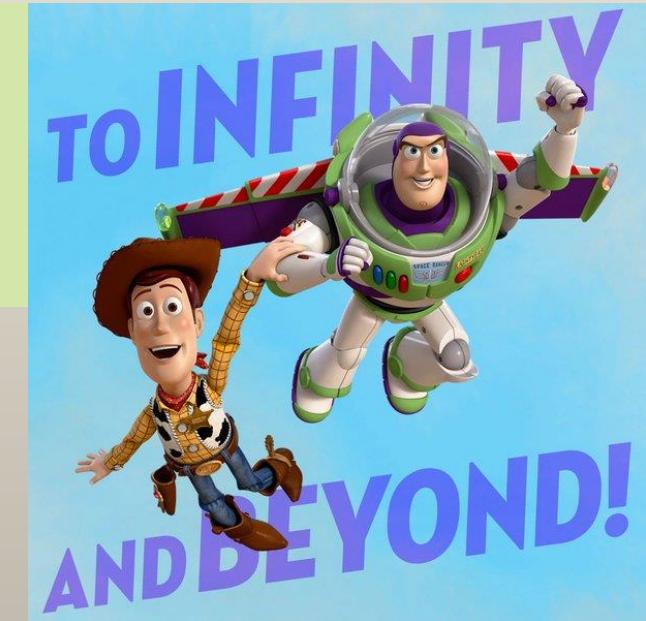


Exit Handlers Calling `exit()`

If an exit handler itself calls `exit()`, the results are undefined.

- On Linux, the remaining exit handlers are invoked as normal.

On some systems, this causes all of the exit handlers to be invoked, which can result in an **infinite recursion**.



Interactions Between `fork()`, stdio Buffers, and `_exit()`

When the parent and the child both call `exit()`, they both flush their copies of the stdio buffers.

This can occasionally result in duplicate output.



Instead of calling `exit()`, each child should call `_exit()`, so that it doesn't flush stdio buffers.

For an application that creates child processes, typically only one of the processes should terminate via `exit()` (typically the parent), while the other processes should terminate using `_exit()`.

The Return Value from `exit()`

What is the return value from a **successful** call to `exit()`/`_exit()`?

There is no return value, neither `exit()` nor `_exit()` return.

They, you know, **exit**...





The Waiting Game

Often it is useful for the parent process to be able to **monitor its child processes**, to find out when and how they may terminate.



The `wait()` Function

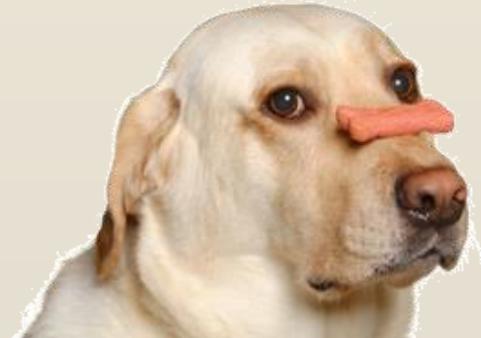
```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait(int *status);
```

Blocks execution of the calling process until one of its child processes changes status (typically terminate).

If the calling process has a child process, the call to `wait()` will return the child process' pid.

If there are no child processes for the parent process, then `wait()` immediately returns a -1.



The `wait()` Function

1. If no (previously unwaited-for) child of the calling process has yet terminated, the call **blocks** until one of the children terminates.
If a child has already terminated by the time of the call, `wait()` returns immediately.
2. If `status` is not `NULL`, information about how the child terminated is returned in the integer **to which `status` points**.
3. The kernel adds the process CPU times and resource usage statistics to running totals for all children of this parent process.
4. A call to `wait()` **returns the process ID** of the child that has terminated.

The `waitpid()` Function

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int *status, int options);
```



Suspends execution of the calling process until a child specified by `pid` argument has changed state (typically terminated).

The `waitpid()` Function

The `pid` argument enables the selection of the child to be waited for, as follows:

1. **If `pid` is greater than 0**, wait for the child whose process ID equals `pid`.
2. **If `pid` equals 0**, wait for any child in the same process group as the caller (parent).
3. **If `pid` is less than -1**, wait for any child whose process group identifier equals the absolute value of `pid`.
4. **If `pid` equals -1**, wait for any child. The call `wait(&status)` is equivalent to the call `waitpid(-1, &status, 0)`.

The `waitpid()` Function

The **options** argument is a bit mask that can include zero or more of the following flags in a bitwise or:

- **WUNTRACED**: In addition to returning information about terminated children, also return information when a child is stopped by a signal.
- **WCONTINUED**: Also return status information about stopped children that have been resumed by delivery of a `SIGCONT` signal.
- **WNOHANG**: If no child specified by `pid` has yet changed state, then **return immediately, instead of blocking.**

Other wait Functions



```
#include <sys/resource.h>
```

```
#include <sys/wait.h>
```

```
pid_t wait3(int *status, int options,  
           struct rusage *rusage);
```

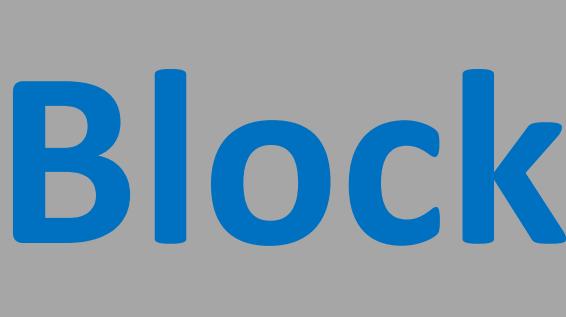
**WAIT WAIT...
DON'T TELL ME!®**

```
pid_t wait4(pid_t pid, int *status, int options,  
           struct rusage *rusage);
```

The `wait3()` and `wait4()` system calls perform a similar task to `waitpid()`.

The differences are that `wait3()` and `wait4()` return **resource usage information** about the terminated child in the structure pointed to by the **rusage argument**.

Q: When a parent process has one or more child processes, which have not exited, what does a call to `wait()` do for the parent process?



Block



Other Process States

- What happens to a child process if its **parent process terminates before the child?**
- What happens when a **parent process creates child processes and neglects to call `wait()` on them** as the child processes complete?



Orphaned Process

- What happens when a **parent process exits before a child** process terminates.
- The child process becomes an *orphan*.
- The orphaned process is **adopted by the init** process, PID 1.
 - The init process will perform a `wait()` and reap the child process.



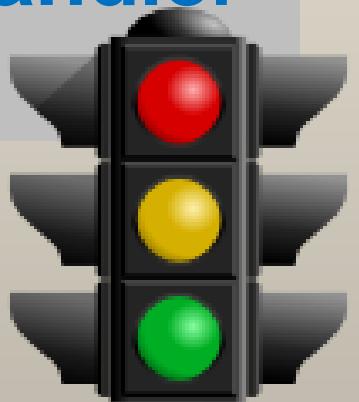
Zombie Process

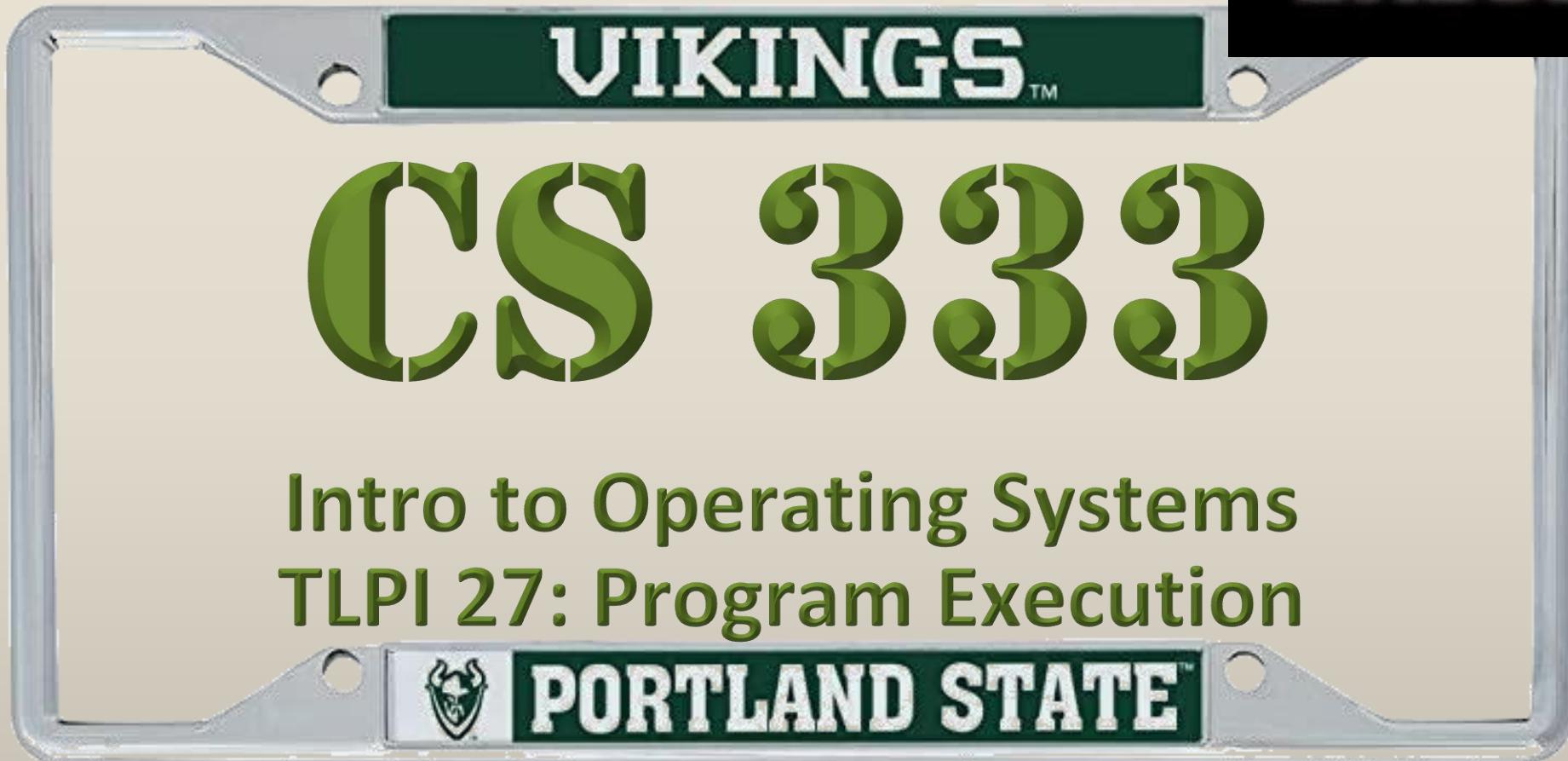
- A process that terminates does not fully leave the system until its parent accepts its return value (calls `wait()`).
- A zombie process doesn't have any code, data or stack.
- It continues to take up space the system's process table.
- A zombie process cannot be killed.
 - When the parent exits, the child (zombie process) gets adopted by `init` and cleared.



The SIGCHLD Signal

- How can a parent process keep track of when each child process may exit, calling `wait()` to prevent zombie processes?
- When a child process terminates, the parent process automatically receives the **SIGCHLD** signal.
- The parent process can **establish a signal handler** and reap the child process in the handler.





Proc-Creation Steps (review)

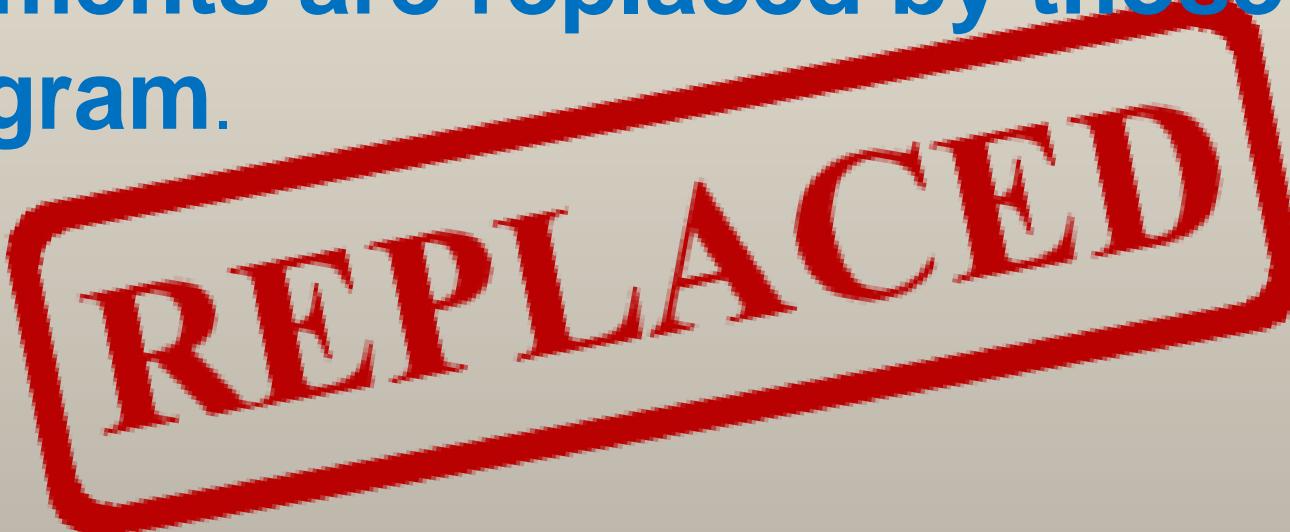
- A process calls `fork()` replicating its own process image.
- Child may **exec()** the image of another program.
The call to **exec()** overwrites the memory of the calling process.
- The parent process may,
 - `wait()` for child to complete execution (foreground execution)
 - continue with other tasks (background execution).
- The child process terminates, parent picks up exit status of child.
- The kernel removes entry for exited process from process table.



Executive Replacement

The `exec` family of functions **replaces** the current process image with a new process image.

The `exec` system calls load a **new** program into a process' memory. During this operation, the **old program is discarded**, and the old process' **text, stack, data, and heap segments are replaced by those of the new program**.



Executive Replacement

After a program calls one of the exec () functions, **the process ID of the process remains the same**, because the same entry in the kernel process table continues to exist.

A few other process attributes also remain unchanged.

The exec functions return only if an error has occurred.

Failure of Executive Branch

- **EACCES**: The pathname argument doesn't refer to a regular file, the file **doesn't have execute permission enabled**, or one of the directory components of pathname is not searchable.
- **ENOENT**: The file referred to by **pathname doesn't exist**.
- **ENOEXEC**: The file referred to by pathname is marked as being executable, but it is not in a recognizable executable format.
- **ETXTBSY**: The file referred to by pathname is open for writing by another process.
- **E2BIG**: The total space required by the argument list and environment list exceeds the allowed maximum.

Make use of perror() !

Exec and Friends

```
#include <unistd.h>

int exec1(const char *path, const char *arg, ...);

int exec1p(const char *file, const char *arg, ...);

int exec1e(const char *path, const char *arg, ...

, char * const envp[]);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);

int execvpe(const char *file, char *const argv[]
, char *const envp[]);
```



The `exec()` family of functions **replaces** the current process image with a new process image.

- a **p** indicates that the `PATH` will be searched for the executable.
- the **l** indicates that a list of command line arguments is given.
- a **v** means that command line arguments are listed in a vector (aka `argv`).
- the **e** indicates that a new environment is passed to the replacement process.

The PATH to Enlightenment

```
int exec1(const char *path, const char *arg, ...);  
int execlp(const char *file, const char *arg, ...);
```



Q: Why is it that the `exec1()` call has a path argument, but does not search the `PATH` environment variable and the `execlp()` call has a file argument and does search the `PATH`?

A: The path in the `exec1()` means **fully qualified path to the executable file** (eg `/usr/bin/who`). The path to the executable must begin with a slash (/).

The file in the `execlp()` call means the call will search through the `PATH` environment variable to find a matching file name.

Environmentalism

```
int execle(const char *path, const char *arg, ...  
          , char * const envp[]);  
  
int execvpe(const char *file, char *const argv[]  
            , char *const envp[]);
```

The `execve()` and `execle()` functions allow the programmer to explicitly specify a new environment for the program using `envp`, a NULL-terminated array of pointers to character strings.

Q: Why the big deal with the environment?

A: The exec-ing process may want to add, remove, or change certain items in the environment for the newly exec'd process.



Exec-ample 1

The program to run and the first argument are (almost always) the same string.

```
status = execlp( [ "ls", "ls", "-l", "-F", "-Bht", (char *) NULL ] );
```

execlp() : a list of arguments is given, it does search the PATH environment variable for the program to be loaded.

A **NULL terminated** list of parameters is used to pass the command line to the new process.

A **NULL terminated** list of parameters that will turn into **argv** for the exec'd program.

EiXEC
Executive Performance

Exec-ample 2



```
char *ls_argv[] = {  
    "ls"  
    , "-l"  
    , "-FB"  
    , "-htr"  
    , "-a"  
    , "--group-directories-first"  
    , (char *) NULL  
};
```

An array strings representing argv for the exec-ed program.

Of what does this remind you?

```
status = execvp( ls_argv[0], ls_argv );
```

The program to run and the first argument are (almost always) the same string.

execvp(): a vector of arguments is given, it does search the PATH environment variable.

Exec and Friends

```
#include <unistd.h>

int execve(const char *pathname,
            char *const argv[], char *const envp[]);
```

The `exec()` functions on the previous slides are really just built on top of the `execve()` function.



Executive Return

Q: What is the return value from a **successful** call to one of the `exec()` functions?

A: There is no return from a successful call to an `exec()` function.



Complete Exec-ample with an exec Call



```
#include <sys/wait.h>
#include <unistd.h>

int main(void) {
    pid_t pid;
    pid = fork();
    if (0 == pid) { /* The child process */
        int status = execlp(
            "ls"
            , "ls", "-l", "-F", "-Bht", (char *) NULL );
    }
    else { /* Parent process */
        int stat_loc;
        pid = wait(&stat_loc);
    }
    return 0;
}
```

The parent process calls `fork()` to create a child process.

The call to `execlp()`

The NULL terminated list of arguments to the program to be exec-ed.

The parent process calls `wait()` and will **block** until the child process terminates.

Executive Failure

Note misspelling

```
status = execlp("Ls", "ls", "-l", "-F"  
, "-Bht", (char *) NULL);
```

```
perror("execlp failed");  
fprintf(stderr, "    CHILD: pid = %d  line: %d  %d\n"  
        , mypid, __LINE__, status);  
fflush(stderr);  
_exit(EXIT_FAILURE);
```

This code is **executed ONLY** if the call to the exec () function fails.

Use the **perror()**
function. It will help you.

Interpreter Scripts

UNIX kernels allow interpreter scripts to be run in the same way as a binary program file, as long as two requirements are met.

1. **Execute permission must be enabled for the script file.**
2. **The file must contain an initial line that specifies the pathname of the interpreter** to be used to run the script.

The first line must have the following form:

```
#!interpreter-path [ optional-arg ]
```

Interpreter Scripts

Q: Can you use the `exec()` calls to execute a shell script or other scripting language program?

A: Yes!

As long as the initial line in the script file contains the correctly formatted string, `exec()` is fine with it.



Interpreter Scripts

For example, the following are valid interpreter script first lines.

```
#!/bin/bash
```

```
#!/bin/bash -v
```

```
#!/bin/python3
```

The path to the interpreter program must be **a fully qualified path name**.

Open Files When `exec` is Called

By default, all file descriptors opened by a program that calls an `exec` function **remain open across the `exec` and are available for use by the new program.**

You can affect this by using the `O_CLOEXEC` flag when you call `open()` for a file descriptor.

```
int open(const char *pathname, int flags );
```



The PATH Environment Variable

- The `execvp()` and `execlp()` functions allow us to specify **just** the name of the file to be executed, without the full path.
- These functions make use of the PATH environment variable to search for the file.
- The value of PATH is a string consisting of colon-separated directory names called path prefixes.
- The `execvp()` and `execlp()` functions search for the filename in each of the directories named in PATH, starting from the beginning of the list and continuing until a file with the given name is successfully found.

`PATH=$ { HOME } /bin:$ { PATH }`

Changing your PATH
environment variable.



Q: Which of the exec functions search the **PATH** environment variable when locating the program to be executed?

A: All the ones that don't have "path" in the function prototype.



The `system()` Function

The `system()` executes a command specified in command by calling `/bin/sh -c command`, and returns after the command has been completed.

The `system()` function creates a child process that invokes a shell to execute command.

An example of a call to `system()`:

```
system("ls | wc");
```

Notice that it runs under the **Bourne shell**,
`/bin/sh`, not `bash`.

You won't be using `system()` in this class.



The `system()` Function

The principal advantages of `system()` are simplicity and convenience:

- You don't need to handle the details of calling `fork()`, `exec()`, `wait()`, and `exit()`.
- Error and signal handling are performed by `system()` on our behalf.
- Because `system()` uses the shell to execute command, all of the usual shell processing, substitutions, and redirections are performed on command before it is executed.
 - This makes it easy to add an “execute a shell command” feature to an application.



The `system()` Function

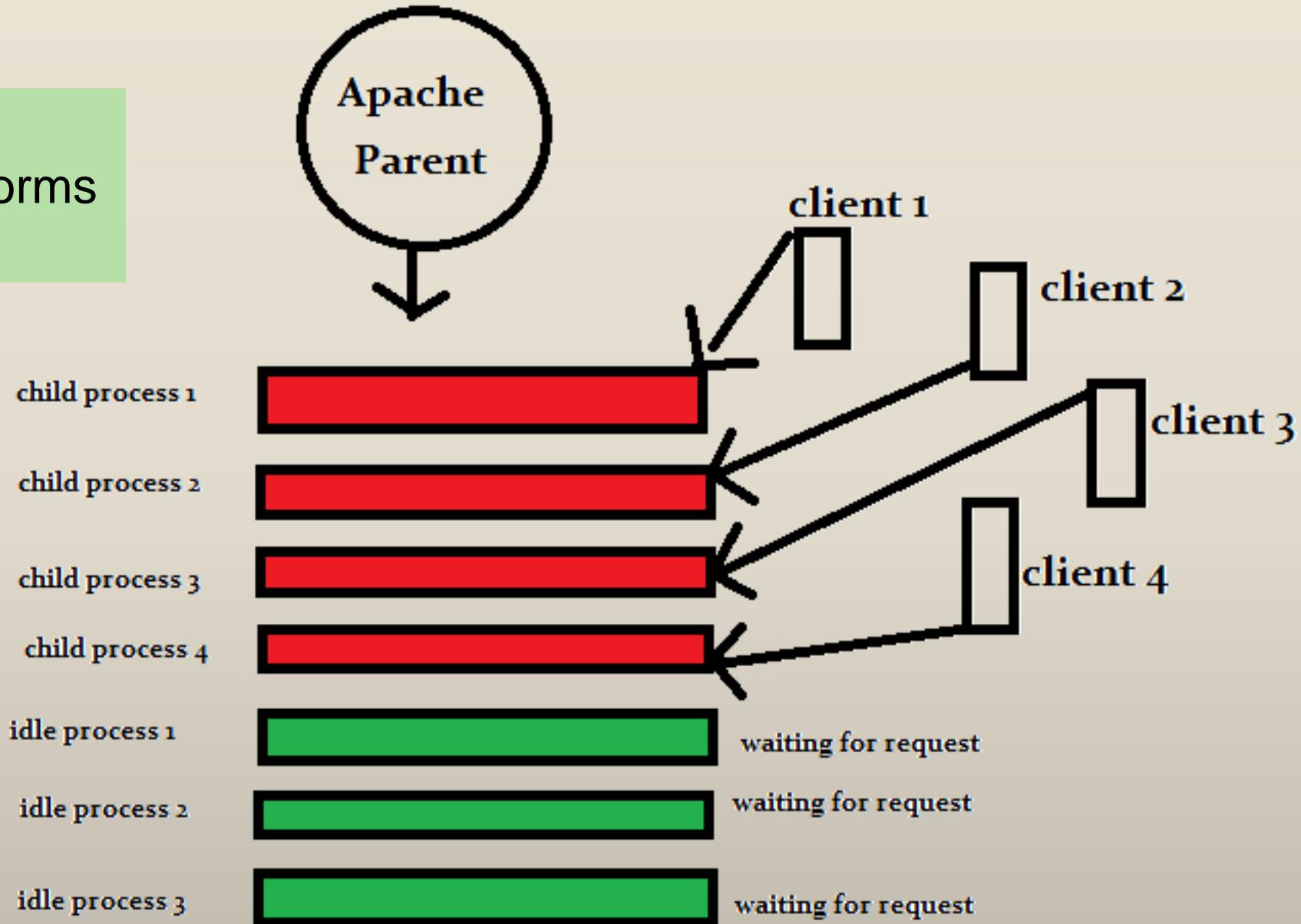
The main cost of `system()` is **inefficiency**.

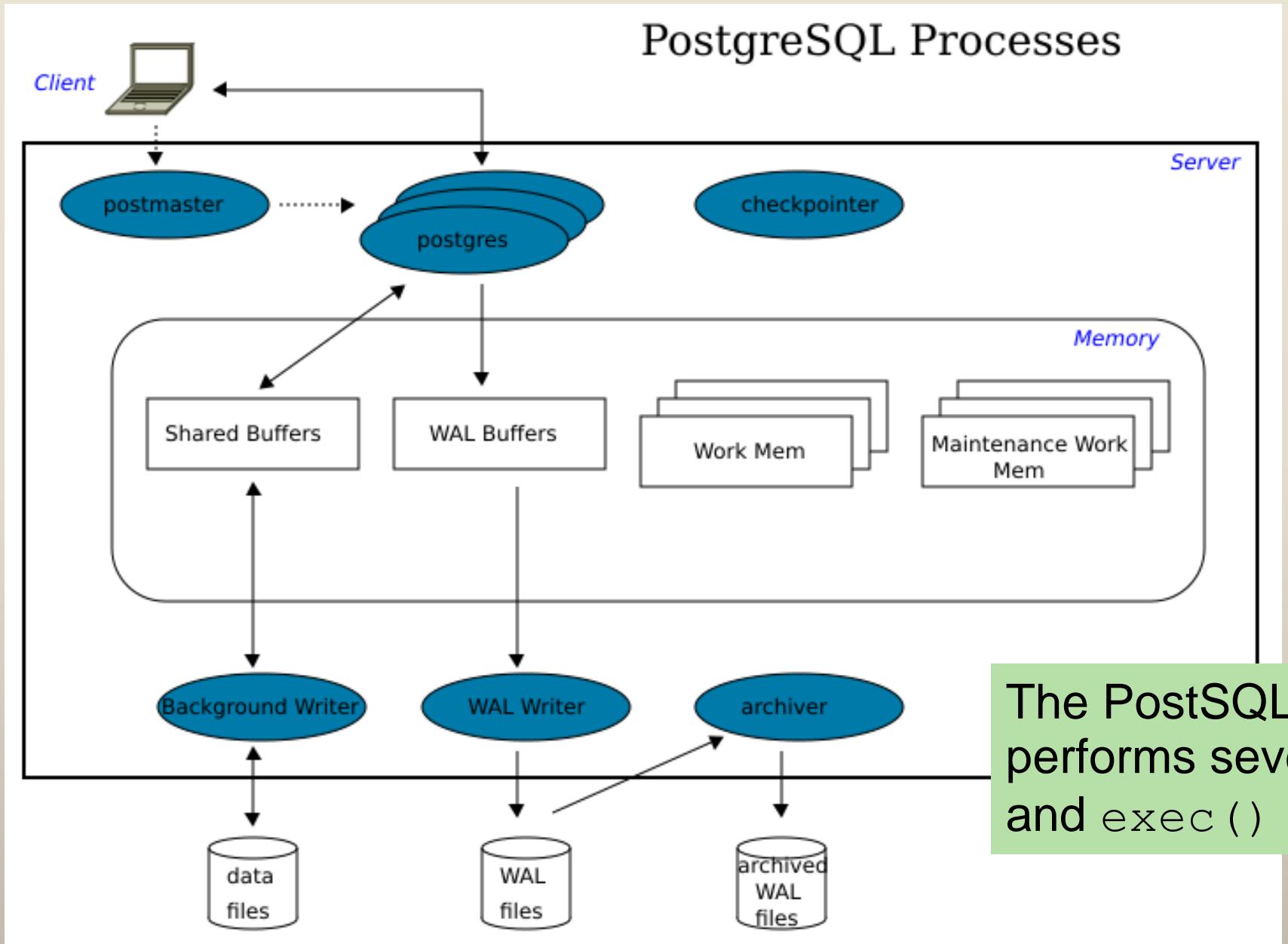
Executing a command using `system()` requires the creation of **at least two processes**.

You won't be using the `system()` in your labs for this class.



An example of a process that performs calls to `fork()`.

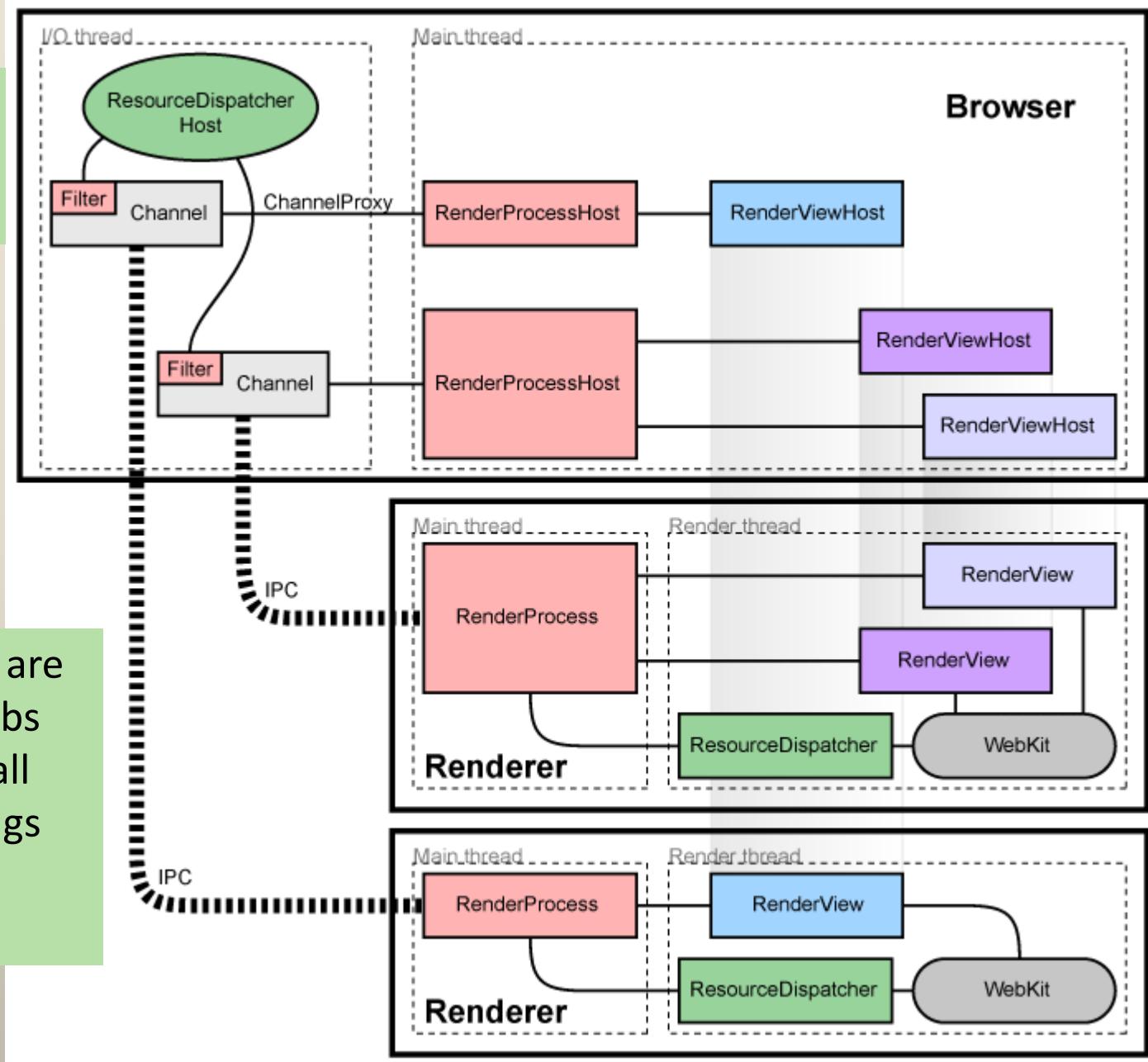




The PostgreSQL server performs several `fork()` and `exec()` calls.

The Chrome browser Multi-process Architecture.

Separate processes are used for browser tabs to protect the overall application from bugs and glitches in the rendering engine.



The main inter-process communication primitive is the named pipe. On Linux & OS X, they use a `socketpair()`.

<https://www.chromium.org/developers/design-documents/multi-process-architecture>

<https://blog.chromium.org/2008/09/multi-process-architecture.html>



Read the PThreads tutorial



Reading is the **KEY** to learning

Threads History



- Historically, hardware vendors have implemented their own proprietary versions of thread libraries.
- These implementations differed substantially from each other.
- Made it difficult for programmers to develop portable threaded applications.
- For UNIX systems, **a standardized programming interface has been specified by a POSIX standard.**
- Implementations which **adhere to this standard are referred to as POSIX threads, or PThreads.**
- Most hardware vendors now offer PThreads in addition to their proprietary API's.

Threads Overview

- Like processes, **threads are an independent stream of execution.**
- Threads are a mechanism that **permits an application to perform multiple tasks concurrently.**
- A **single process can contain multiple threads**, each executing a separate stream of instructions or executing the same stream of instructions.
- A standard UNIX process is simply a **special case of a multithreaded processes that contains a single thread.**

Threads Overview

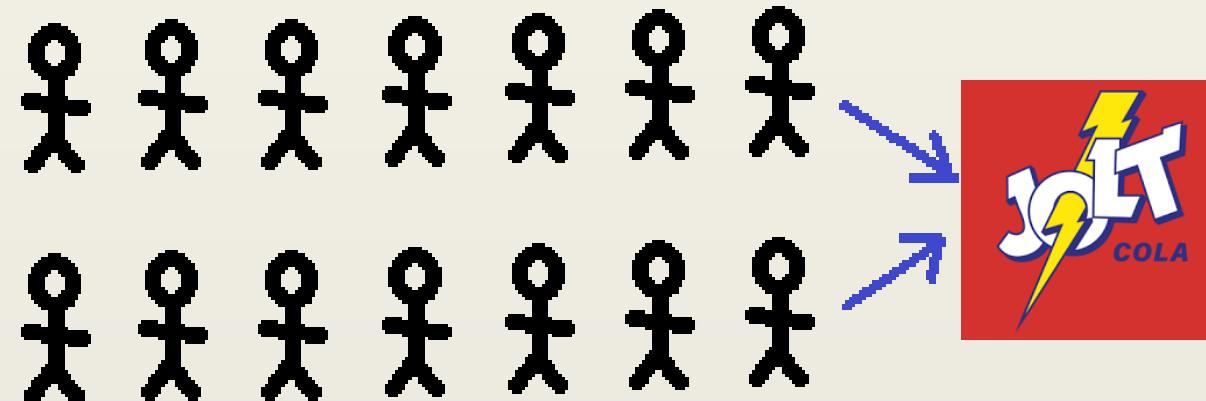
- The threads in a process can execute concurrently.
- **On a multiprocessor system, multiple threads can execute in parallel.**
- If one thread is blocked on I/O, other threads in the process are still able to execute.



Parallelism vs. Concurrency

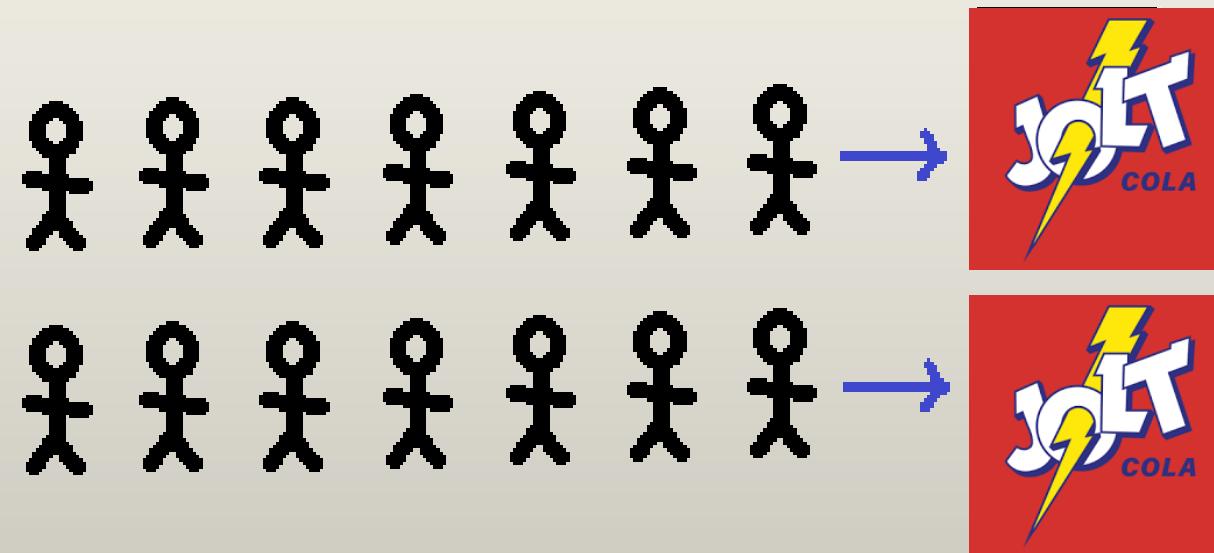
- Parallelism – **perform many tasks simultaneously.**
Purpose – improves throughput
- Concurrency – **mediates multi-party access to shared resources.**
Purpose – decrease response time

Concurrency



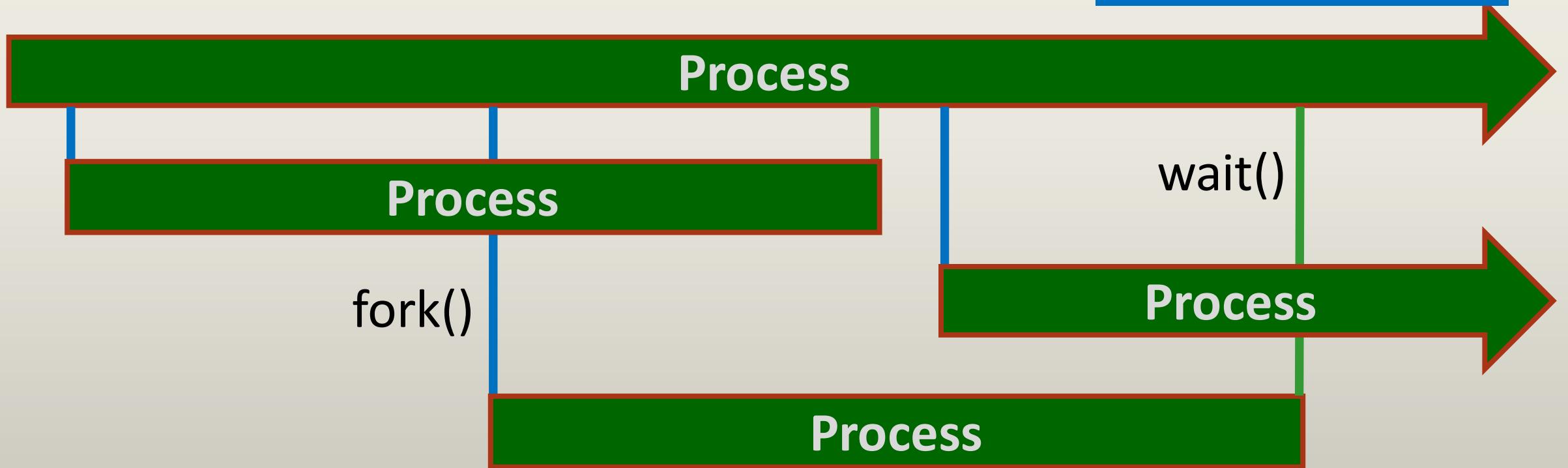
Concurrent: 2 queues, 1 vending machine

Parallelism

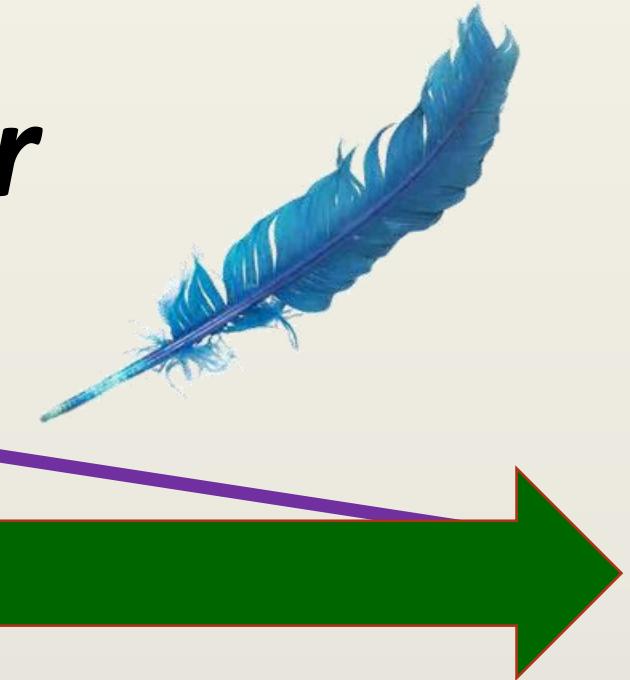


Parallel: 2 queues, 2 vending machines

Processes are *Heavy*



Threads are *Lighter*



Processes vs Threads



Processes vs Threads

Processes

Multiple independent processes

Independent memory space

Independent open file
descriptors

Threads

Multiple independent functions

Shared memory space

Shared open file descriptors

Threads like to Share

You are going to like this.

- **global memory**
- **process ID and parent process ID**
- process group ID and session ID
- controlling terminal
- process credentials (user and group IDs)
- open file descriptors
- record locks created using `fcntl()`
- signal dispositions
- file system-related information: umask, **current working directory**, and root directory

- interval timers (`setitimer()`) and POSIX timers (`timer_create()`)
- System V semaphore undo (`semadj`) values
- resource limits
- CPU time consumed (as returned by `times()`)
- resources consumed (as returned by `getrusage()`)
- **nice value** (set by `setpriority()` and `nice()`)



Share

Threads don't Share Everything

- **thread ID**
- signal mask
- thread-specific data
- alternate signal stack (`sigaltstack()`);
- **the errno variable**
- floating-point environment
- Real-time scheduling policy and priority
- **CPU affinity**  What is affinity?
- capabilities
- stack (local variables and function call linkage information).



Threads and `errno`

- In the *traditional* UNIX API, `errno` is a **global integer variable**.
- However, this doesn't suffice for threaded programs.
- If a thread made a function call that returned an error in a global `errno` variable, then this would confuse other threads that might also be making function calls and checking `errno`.
- **In threaded programs, each thread has a private `errno` value.**



Threads vs Processes

- Sharing data between threads is easy.
 - Sharing data between processes requires work and effort (creating a shared memory segment, using a pipe, ...).
- **Thread creation is faster** than process creation.
- Context-switch time may be lower for threads than for processes.
- Multiple threads can be run from within a single debugger.



Threads vs Processes

- When programming with threads, you need to **ensure that the functions you call are thread-safe** or are called in a thread-safe manner.
 - Multi-process applications don't need to be concerned with this (or less concerned).
- A bug in one thread (modifying memory via an incorrect pointer) can damage all of the threads in the process, since they share the same address space and other attributes.
- **Multiple processes are isolated** from one another. One process can crash and burn and not take down other processes.

A piece of code is thread-safe if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time.



CS333 Intro Op Sys

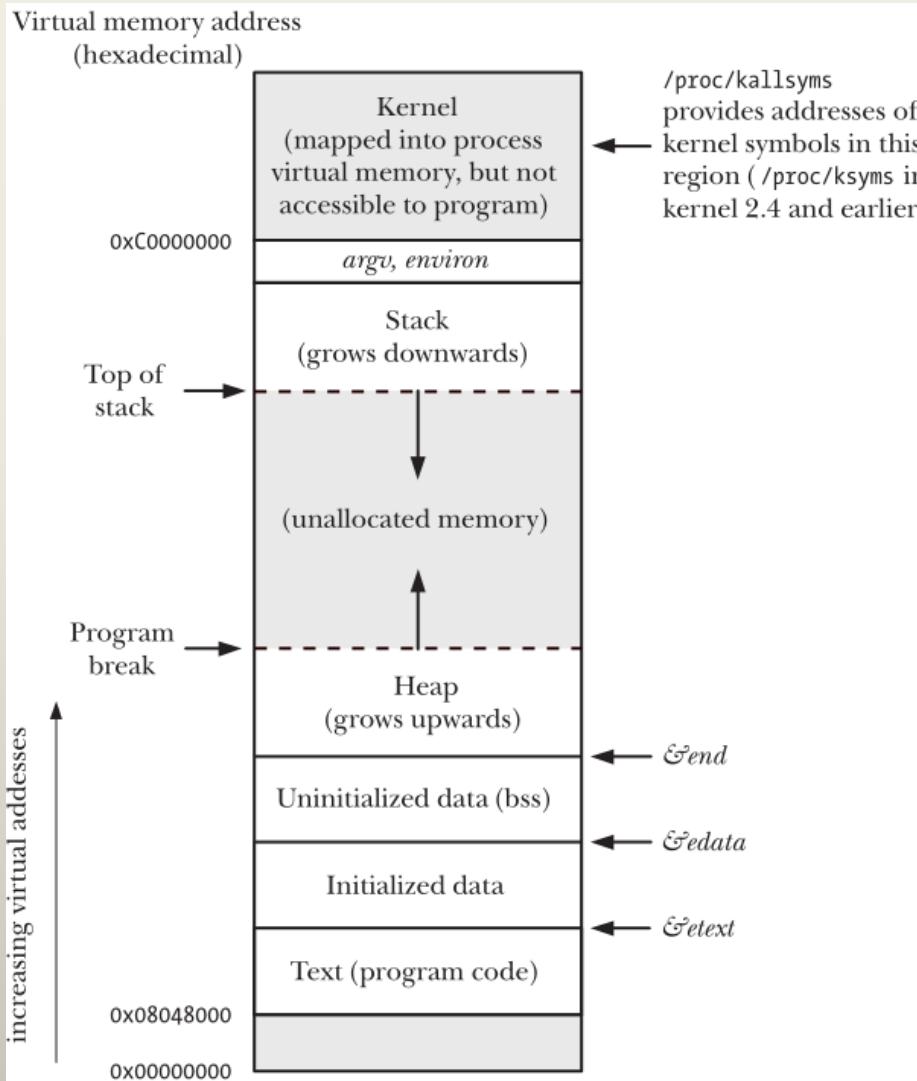
Threads vs Processes

- Each thread is competing for use of the **finite virtual address** space of the host process.
- **Threads can't span systems.** A multi-process application can be spread across several different machines (such as a load-balancing web server).
- Unlike processes, in which there is a parent child relationship between processes, **all threads are peers.**

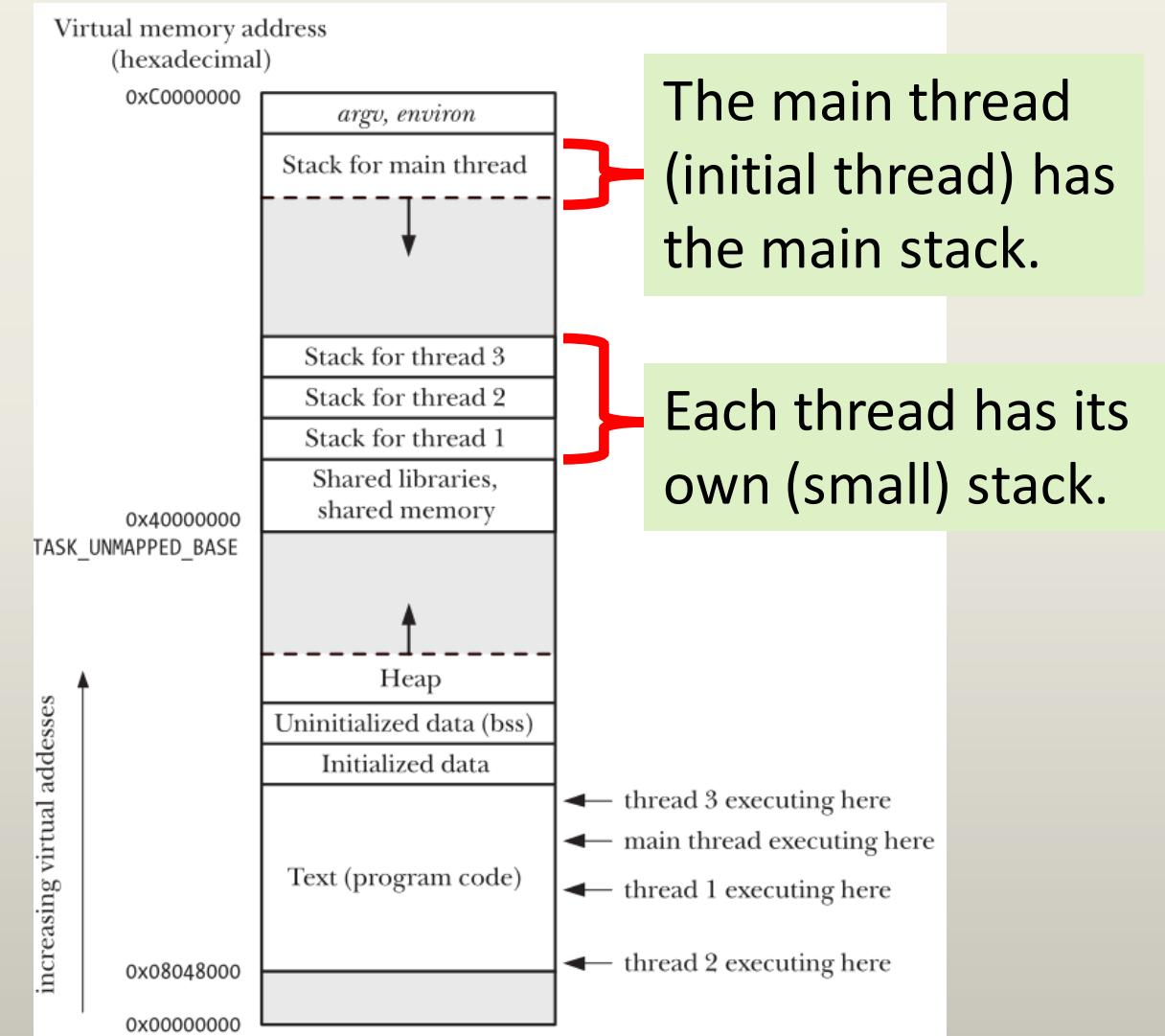


Threads vs Processes

UNIX Process



Threads within a UNIX Process



Threads vs Processes Performance

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Timings reflect **50,000** process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags.



What are PThreads?

- PThreads are defined as a set of **C language programming types and procedure calls**.
- Implemented with a **pthread.h header/include** file and a thread library.
- This makes it easier for programmers to develop portable threaded applications.
- This library may be part of another library, such as libc.



PThreads API

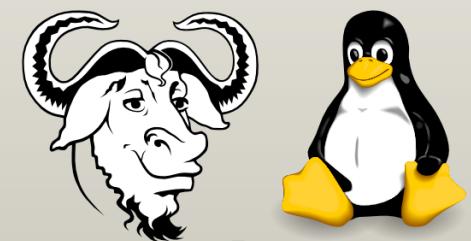
To compile using GNU cc on Linux:

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

```
gcc -pthread
```

Don't forget to use this with
gcc when you compile.

You need this include file
in your source code.



GNU/Linux

PThreads API

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

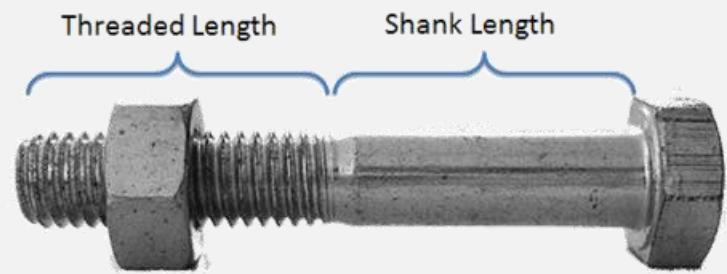


Creating Threads

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

`pthread_create` arguments:

- **thread**: An **opaque**, unique identifier for the new thread returned by the subroutine
- **attr**: An **opaque** attribute object that may be used to set thread attributes
You can specify a thread attributes object, or NULL for the default values
- **start**: the C function that the thread will execute once created
- **arg**: A single argument that may be passed to *start*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.



```
pthread_t tid[2];  
int main(void)  
{  
    int i = 0;  
    int err;  
  
    while(i < 2) {  
        err = pthread_create(&(tid[i]), NULL, doSomeThing, NULL);  
        if (err != 0)  
            printf("\ncan't create thread :[%s]", strerror(err));  
        else  
            printf("\n Thread created successfully\n");  
        i++;  
    }  
    sleep(5);  
    return 0;  
}
```

A pthread_t type is opaque. It may simply be an int, or it could be a structure.

The call to create new threads.

Defined on the following slide.

```
void *doSomeThing(void *arg)
{
    unsigned long i = 0;
    pthread_t id = pthread_self();

    if (pthread_equal(id, tid[0])) {
        printf("\n First thread processing\n");
    }
    else {
        printf("\n Second thread processing\n");
    }

    for(i = 0; i < (0xFFFF); i++)
    ;

    pthread_exit(NULL);
}
```

If you want to compare 2 pthread_t variables, use the pthread_equal() function.

Thread Termination

```
void pthread_exit(void * retval);
```



The execution of a thread terminates in one of the following ways:

- The thread's start function performs a `return` specifying a return value for the thread.
- The thread calls `pthread_exit()`.
- The thread is canceled using `pthread_cancel()`.

Any of the process threads calls `exit()`, or the main thread performs a return (in the `main()` function), causes all threads in the process to terminate immediately.

```
void pthread_exit(void * retval);
```

- The `retval` argument specifies the return value for the thread.
- **The value pointed to by `retval` should not be located on the thread's stack.**
 - The contents of the thread's stack become undefined on thread termination.
 - The same statement applies to the value given to a `return` statement in the thread's start function.



More `pthread_exit()`

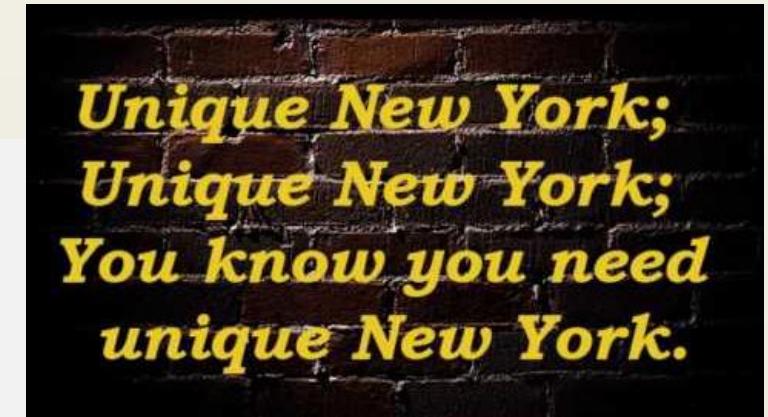
- Performing `return` from the start function of any thread other than `main()` results in an implicit call to `pthread_exit()`, using the function return value as the thread's exit status.
- In order to allow other threads to continue execution, the `main()` thread should terminate by calling `pthread_exit()` rather than `exit(3)`.

The main thread is **more equal** than the other threads. If it calls `pthread_exit()`, it will wait for all the other threads to complete before exiting the application.



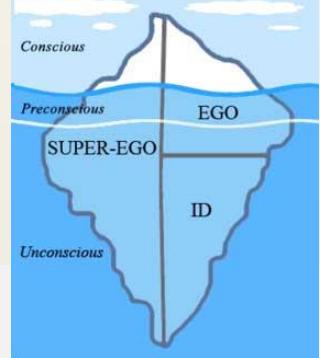
Thread IDs

```
#include <pthread.h>  
  
pthread_t pthread_self(void);
```



- Each thread **within a process** is uniquely identified by a thread ID (tid). However, **tids may be duplicated across processes**.
- The ID is returned to the caller of `pthread_create()`, and a thread can obtain its own ID using `pthread_self()`.
- The `pthread_t` type is opaque and you should not treat it as an integer for comparisons.

Thread IDs



```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

- If the 2 thread ids compare equal, then a non-zero value is returned.
 - If they compare as non-equal, 0 is returned.
- If you need to do comparisons of the thread ids of 2 threads, you should use the `pthread_equal()` function.
- Like a pid can be recycled within a system, a tid may be recycled.

Joining with a Terminated Thread

```
int pthread_join(  
    pthread_t thread  
, void ** retval);
```



- The `pthread_join()` function **waits for the thread identified by `thread` to terminate.**
- Notice that the `pthread_join()` **joins with a specific thread**, not just any 'ole thread.

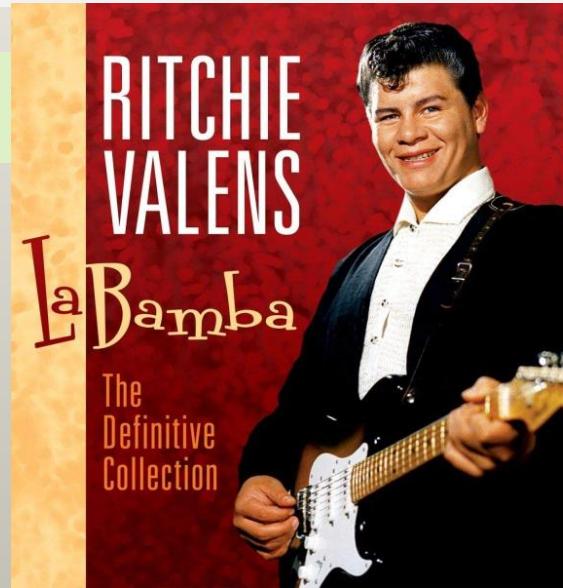
This is the thread equivalent of the `waitpid()` function for processes.

Joining with an Already Joined Thread

Calling `pthread_join()` for a thread ID that has been previously joined can lead to **unpredictable behavior**.

For example, it may instead join with a thread created later that happens to reuse the same thread ID.

Or, something worse...



Zombie Threads

- When a thread completes, **you must join with it, or it will turn into a zombie thread.**
- If enough zombie threads are created, then you will not be able to create more threads and your process will be *unstable*.
- However, the effects will be contained to your process, not the entire system.
- **However, threads can also be detached...**



Becoming Detached

- **By default, threads are created as joinable**, meaning that when it terminates, another thread can obtain its return status using `pthread_join()`.
- **Sometimes, we don't care about the thread's return status**; we simply want the system to automatically clean up and remove the thread when it terminates.
- Once a thread has been detached, it is no longer possible to use `pthread_join()` to obtain its return status, and the thread can't be made joinable again.



Detaching a Thread

```
int pthread_detach(pthread_t thread);
```

As an example of the use of `pthread_detach()`, a thread can detach itself using the following call:

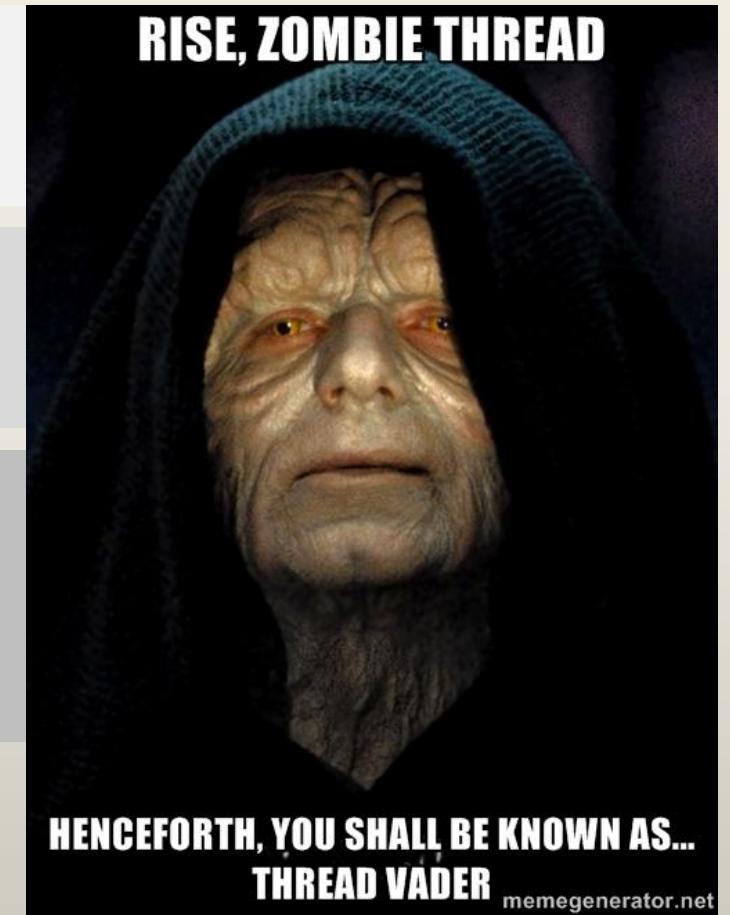
```
pthread_detach(pthread_self());
```

These are like **dæmon threads** which are begun, but will have their own life that you cannot join.



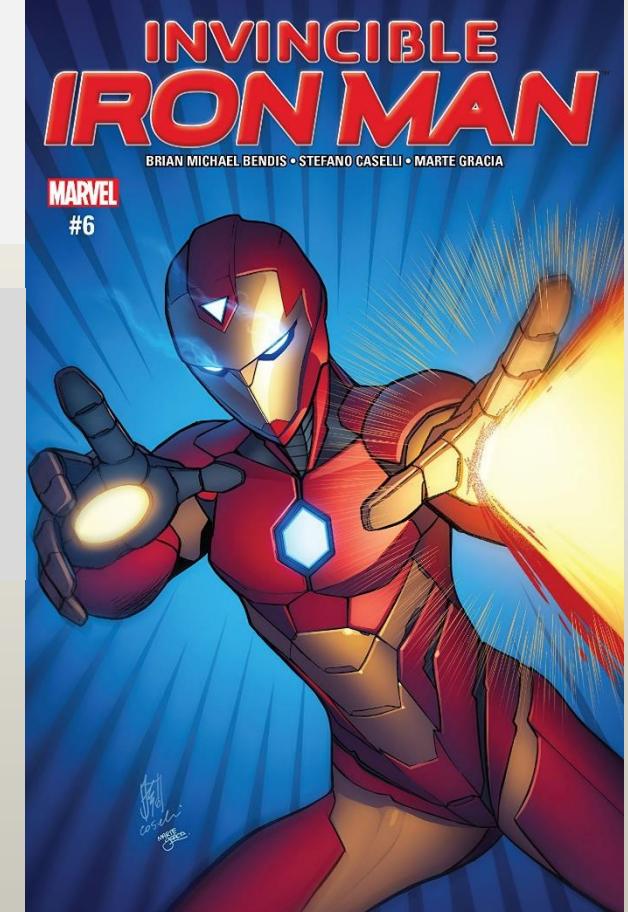
Detached Threads

- Detached threads run independently of the thread that created it.
- The creating thread will not wait for a detached thread to complete to join back with it.
- **Detached threads will be cleaned up when they terminate** without calling a join on them, **they do not turn into Zombies.**



Detached Threads

- **Detaching a thread doesn't make it invincible** to a call to `exit()` from another thread or a return in the main thread.
- If one thread calls `exit()`, all threads in the process are terminated immediately, regardless of whether they are joinable or detached.

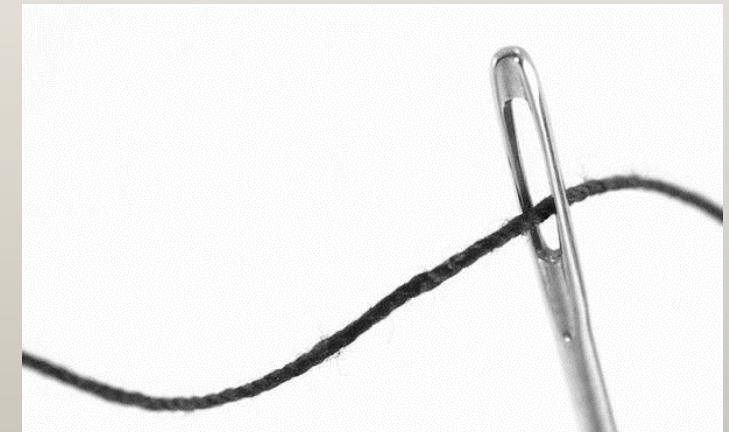


Further Detachment

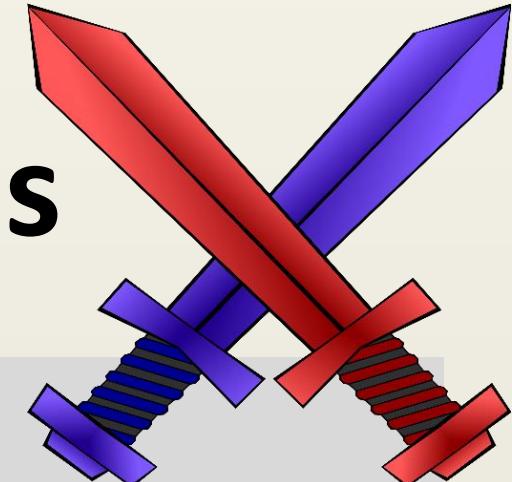
```
int pthread_create(pthread_t * thread
    , const pthread_attr_t * attr
    , void *(* start ) (void *)
    , void * arg);
```

Remember the attributes that you could pass to a thread upon creation?

- You can create threads as detached by setting the attributes.



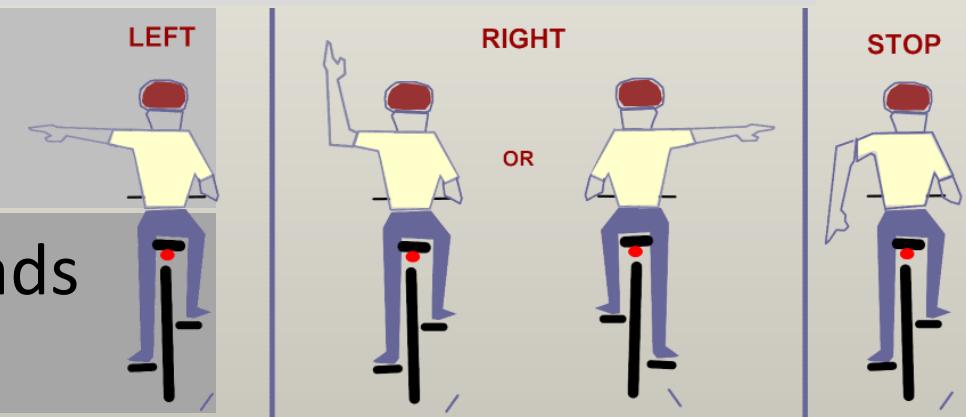
Threads vs Processes



- Sharing data between threads is easy.
 - Sharing data between processes requires more work.
-
- Thread creation is faster than process creation.
 - Context-switch time **may** be lower for threads than for processes.

Threads vs Processes

- When programming with threads, we need to ensure that the functions we call are thread-safe or are called in a thread-safe manner.
 - A flaw in one thread can damage all of the threads in the process, since they share the same address space.
- Each thread is competing for use of the **finite virtual address space** of the host process.
 - Dealing with signals in a multithreaded application requires ... **careful design**.
 - In a multithreaded application, all threads must be running the same program.





Protecting Accesses to Shared Resources

- One of the advantages of threads is that they can share information via **global variables**.
- This **easy sharing comes with costs**;
 - we must take care that multiple threads do not attempt to modify the same variable at the same time, or
 - that one thread doesn't try to read the value of a variable while another thread is modifying it.



Critical Section

- The term **critical section** is used to refer to a section of code that accesses a shared resource and whose execution should be “**atomic**.”
- The execution of a critical should not be interrupted by another thread that simultaneously accesses the same shared resource.

A large, red, rectangular stamp with the word "CRITICAL" written in bold, capital letters. The stamp has a slightly distressed, ink-stamped appearance with some texture and slight variations in color.

Protecting Accesses to Shared Resources

- If we fail to protect shared resources, it could cause the program to crash or display **non-deterministic behavior**.
- Non-deterministic behavior is very difficult to debug.



A nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm.

Enter the Mutex

Mutex (short for **MUTual EXclusion**) can be used to ensure that only one thread at a time can access a resource (such as a variable).

1. A mutex has two states: *locked* and *unlocked*.
2. At any moment, at most one thread may hold the lock on a mutex.
3. Attempting to lock a mutex that is already locked either blocks or fails.



The Mutex

- When a thread locks a mutex, it becomes the **owner of that mutex.**
- Only the mutex owner can unlock the mutex.
- The terms **acquire** and **release** are sometimes used for lock and unlock.



The Mutex

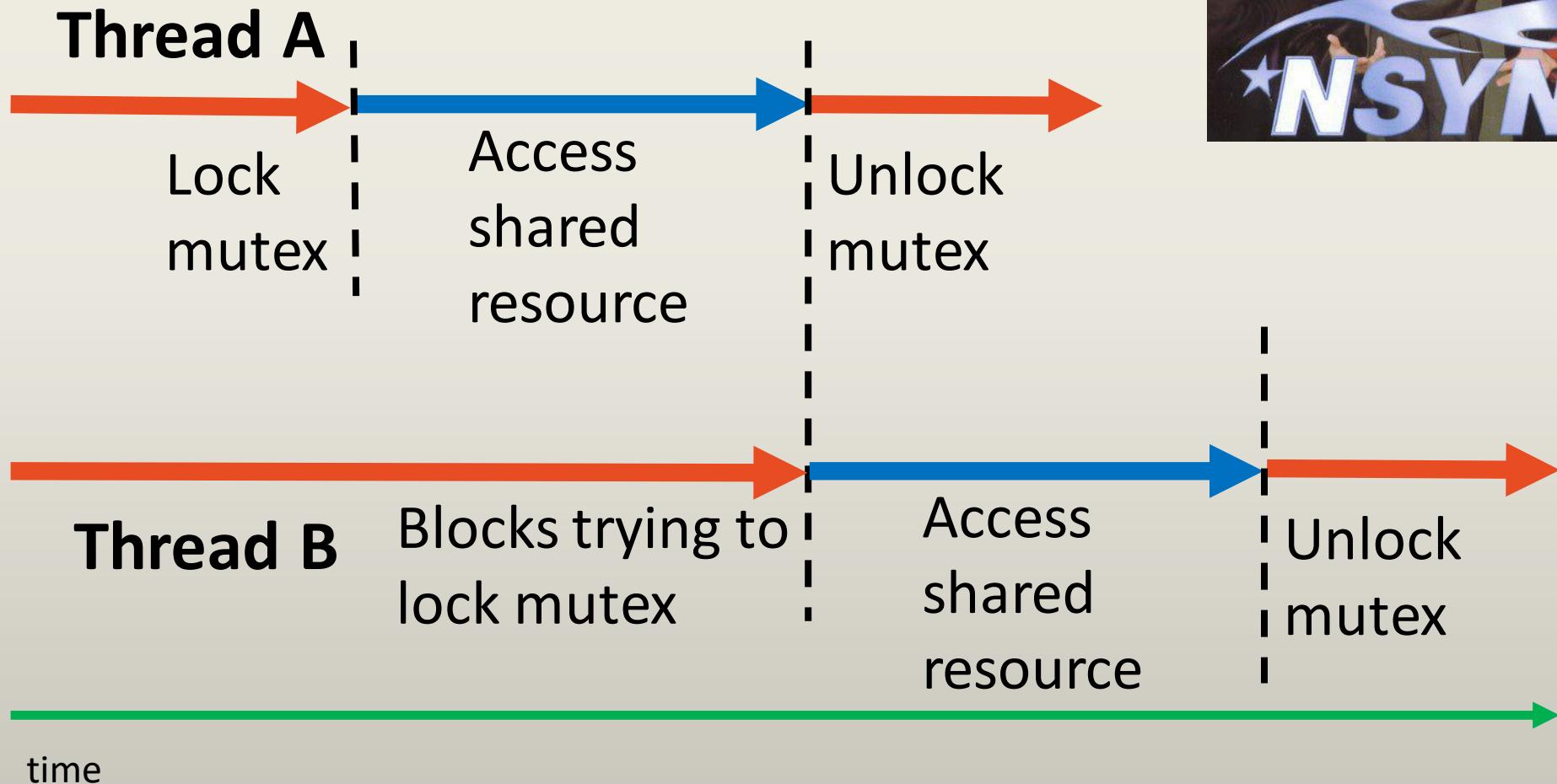
We use a different mutex for each *shared resource* (which may be composed of multiple related variables).

Each thread employs the following protocol for accessing a shared resource:

1. Lock the mutex for the shared resource
2. Access the shared resource
3. Unlock the mutex.



Mutex Protocol



Mutex Rules

- A single thread may not lock the same mutex twice.
- A thread may not unlock a mutex that it doesn't currently own (i.e., that it did not lock).
- A thread may not unlock a mutex that is not currently locked.



Initializing a Mutex

Before you make use of a mutex, you must initialize it.

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

- A **statically created** mutex can be initialized with the PTHREAD_MUTEX_INITIALIZER value.

```
pthread_mutex_init(&mtx  
, const pthread_mutexattr_t *attr);
```

- An **allocated variable** (returned from a call to malloc()) must have the function pthread_mutex_init() called on it.



Locking and Unlocking a Mutex

```
#include <thread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_timedlock(pthread_mutex_t *mutex
    , struct timespec *abs_timeout);
struct timespec {
    __time_t tv_sec;                      /* Seconds.      */
    long int tv_nsec;                     /* Nanoseconds. */
};
```



Deadlocks and Mutexes

Thread A

1. `pthread_mutex_lock(mutex1);`

2. `pthread_mutex_lock(mutex2);`

Blocks indefinitely

Thread B

1. `pthread_mutex_lock(mutex2);`

2. `pthread_mutex_lock(mutex1);`

Blocks indefinitely

- The simplest way to avoid such deadlocks is to define a mutex hierarchy.
- When threads can lock the same set of mutexes, they should always lock them in the same order.



Thread Safeness

- A function is said to be **thread-safe** if it can safely be invoked by multiple threads at the same time.
- On the other hand, if a function is not thread-safe, then it cannot be called from one thread while it is being executed in another thread.



Reentrant Functions

A **reentrant** function accomplishes *thread safety* **without the use of mutexes**.

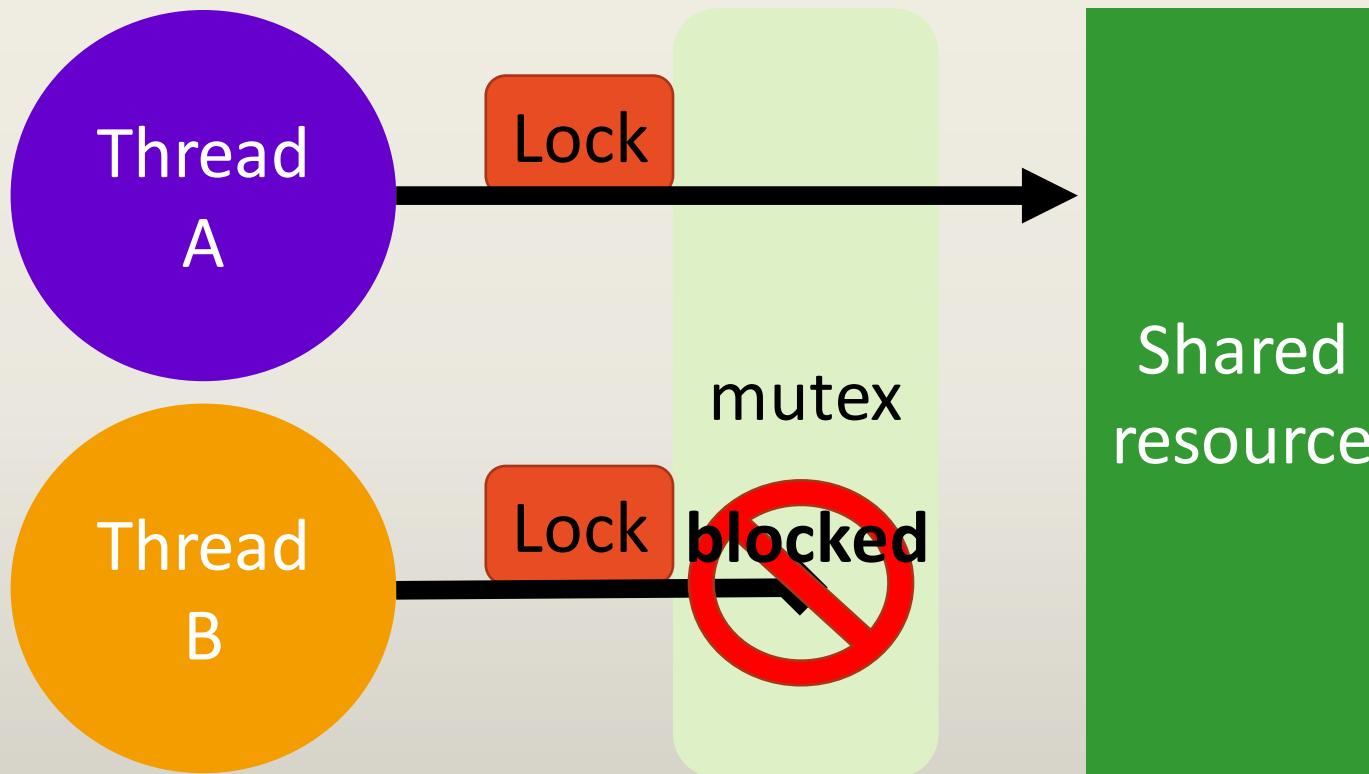
- It does so by **avoiding the use of global and static variables.**
- It uses **only automatic variables** or buffers allocated within the function.
- It should **not call a non-reentrant function.**



PThreads Mutex

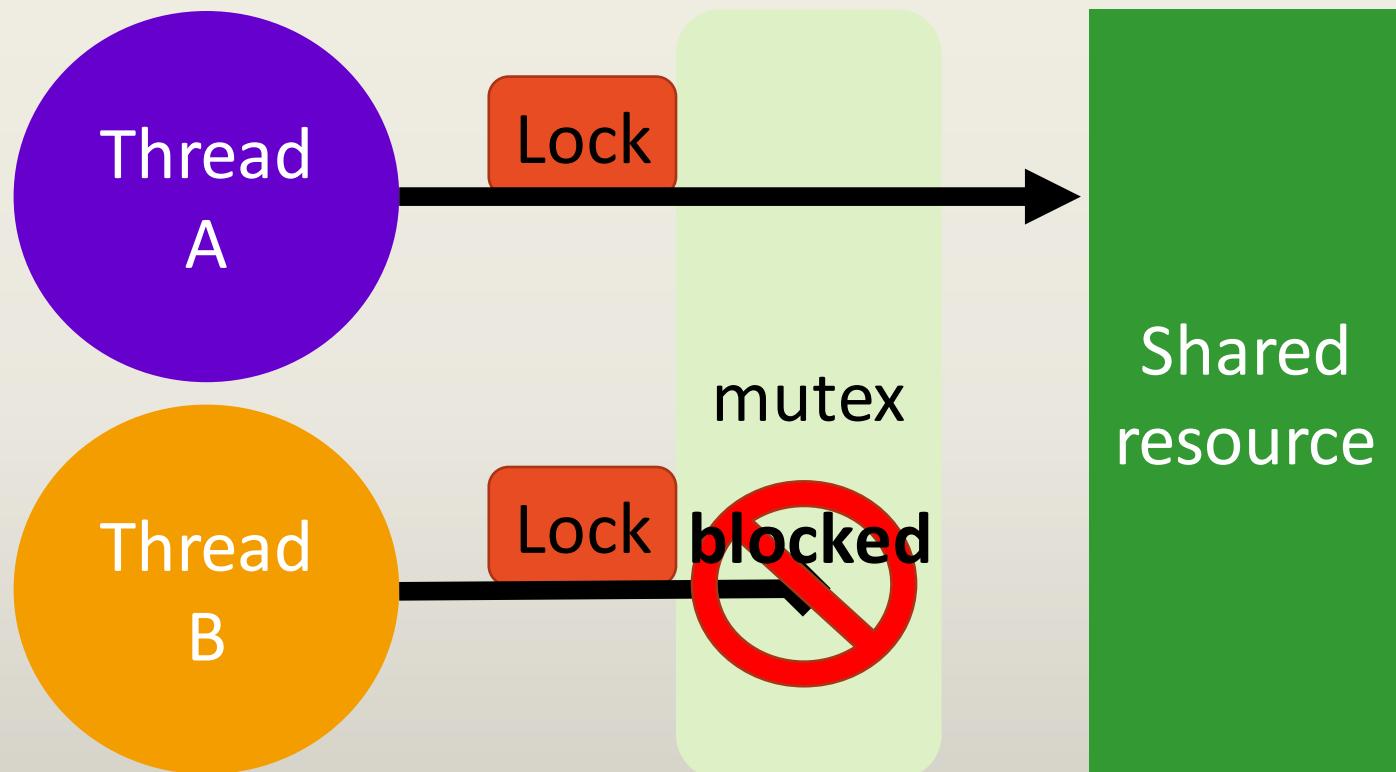
Thread A	Thread B	Balance
Read balance: \$1000		\$1000
	Read balance: \$1000	\$1000
	Deposit \$200	\$1000
Deposit PowerBall winnings		\$1000
Update balance \$1000 + \$gaZillion		\$\$\$\$\$\$\$\$\$
	Update balance \$1000+\$200 <i>oopsie</i>	\$1200

PThreads Mutex



- Multiple threads need to access a shared resource.
- The mutex is used to assure that only one thread at a time has access to the shared resource.

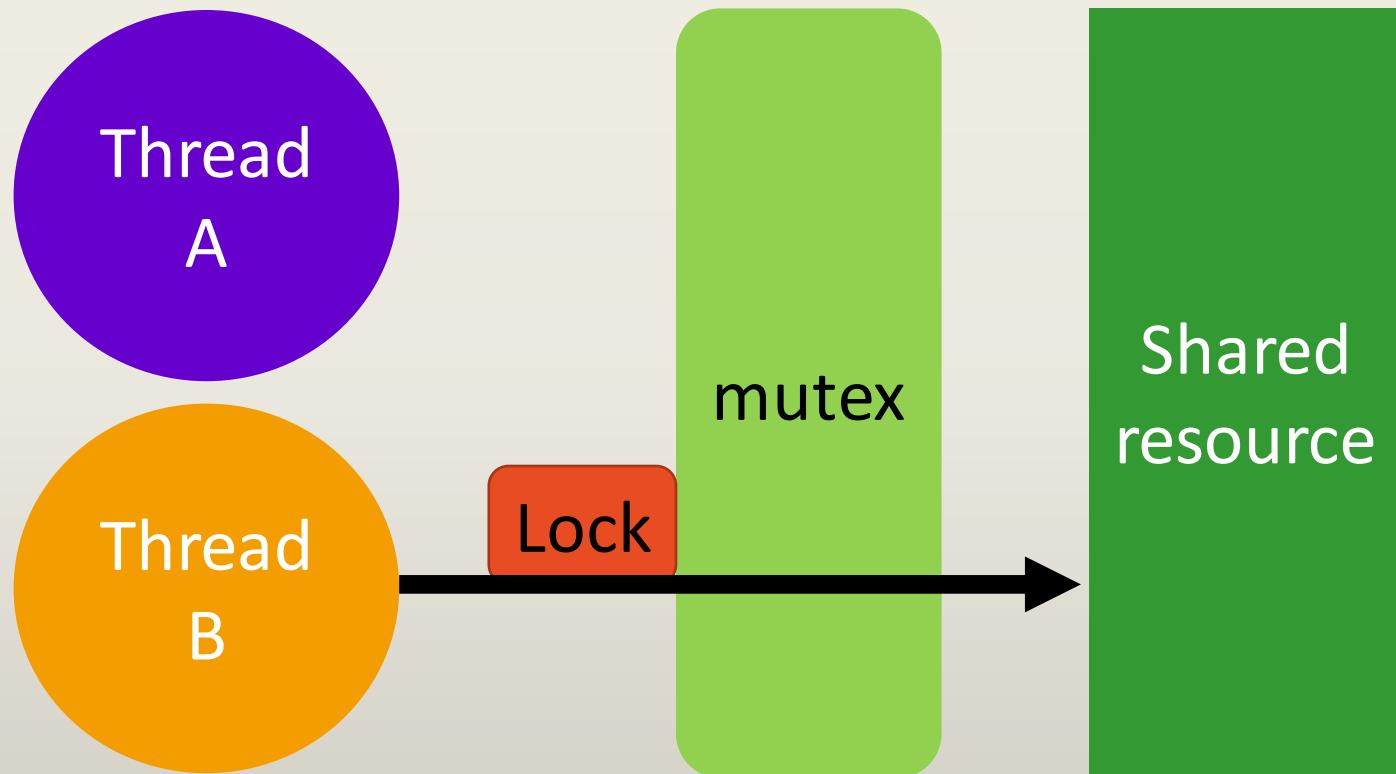
PThreads Mutex



- Thread A made the request to lock the mutex first and received the lock.
- Thread A now owns the lock.
- Thread B made has requested a lock on the mutex and is **blocked** until Thread A unlocks it.

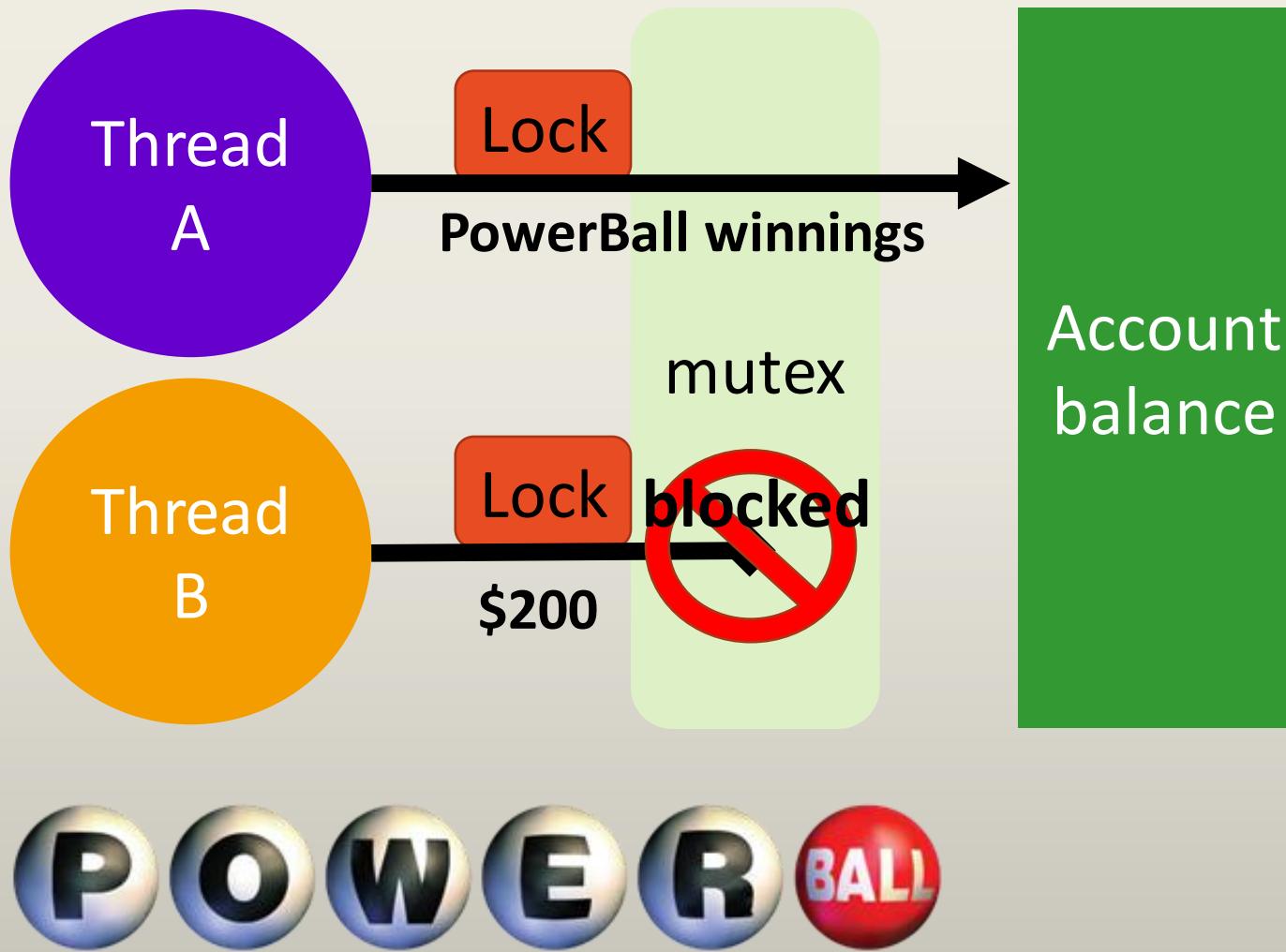
Only the thread that owns the lock can unlock it.

PThreads Mutex



- Thread A released its lock on the mutex.
- Thread B has acquired the lock and now owns it.
- Thread B can access the shared resource.

PThreads Mutex



- Going back to the original example, Thread A receives a lock on the account balance and keeps it throughout its transaction.
- Thread B can only access the account after Thread A has completed its transaction.

```
int main(int argc, char *argv[]) {
    pthread_t *thread;
    long i;
    int ret;
    // Initialize the mutex. You MUST do this before you use
    // the mutex.
    pthread_mutex_init(&mutex_sum, NULL);

    // Create the array that will hold the thread ids.
    thread = (pthread_t *) malloc(num_threads * sizeof(pthread_t));
    for (i = 0; i < num_threads; i++) {
        ret = pthread_create(&thread[i], NULL, do_work, (void *) i);
        if (ret != 0) {
            perror("Cannot create the mutex threads.");
            exit(EXIT_FAILURE);
        }
    }
    pthread_mutex_destroy(&mutex_sum);
    pthread_exit(EXIT_SUCCESS);
}
```



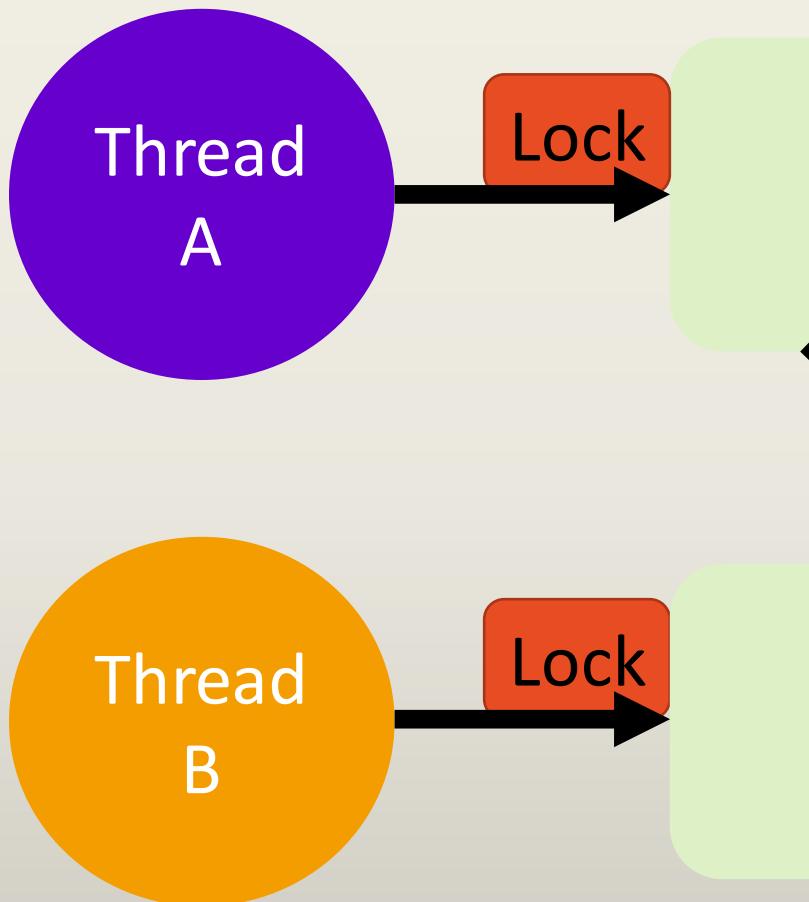
The *main* thread calls `pthread_exit()` and will “hang around” until all the other threads are complete.

Lock and unlock
the mutex.

```
void *do_work(void *arg) {  
    long id = (long) arg;  
    int i;  
  
    for (i = 0; i < 3; i++) {  
        printf("Thread %ld requesting the lock\n", id);  
        pthread_mutex_lock(&mutex_sum);  
        printf(" Thread %ld now has the lock\n", id);  
  
        sleep(2 * i);  
  
        pthread_mutex_unlock(&mutex_sum);  
    }  
    printf(" Thread %ld EXITING\n", id);  
    pthread_exit(EXIT_SUCCESS);  
}
```



Deadlock



mutex 1

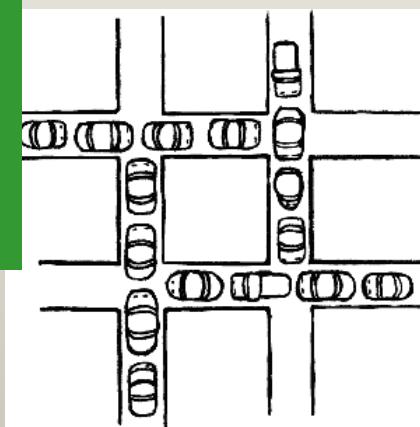
Blocked

Blocked

mutex 2

Shared
resource

Another word used for
this is **starvation**.



Deadlock

- Let's say your employer wants to deposit your paycheck.
 - You have your employer split up your pay so that some of it goes into checking and some goes into savings.
- At the same time, you notice that your checking account balance is a little low and decide to transfer some money from savings to checking.

When protecting shared data, **it is the programmer's responsibility** to make sure every thread that needs to use a mutex does so.



Deadlock

Transfer from savings to checking	Deposit from employer
Lock the savings account	Lock the checking account
Lock the checking account	Lock the savings account
<i>Each thread is blocked waiting for the other to complete.</i>	<i>Each thread is blocked waiting for the other to complete.</i>

Also called “The Deadly Embrace”
Edsger Dijkstra



```
void *thread_func1 (void *temp) {
    pthread_mutex_lock( &mutex1 );
    sleep(1);
    pthread_mutex_lock( &mutex2 );
    pthread_mutex_unlock( &mutex2 );
    pthread_mutex_unlock( &mutex1 );
    pthread_exit(EXIT_SUCCESS);
}

void *thread_func2 (void *temp) {
    pthread_mutex_lock( &mutex2 );
    sleep(1);
    pthread_mutex_lock( &mutex1 );
    pthread_mutex_unlock( &mutex1 );
    pthread_mutex_unlock( &mutex2 );
    pthread_exit(EXIT_SUCCESS);
}
```

Notice that the threads request the mutex-es in opposite order.

You release the lock first
Once I have finished
my task, you can continue.

Why should I?
You release the lock first
and wait until
I complete my task.



Mutual Exclusion vs Synchronization

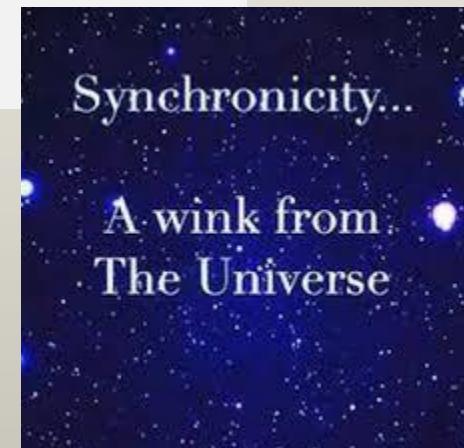
- A mutex allows us to control access to shared resources.
- We can specify a critical region of code that touches a shared resource and limit access to a single thread.
- We exclude other threads from access while one thread does have access.
- That section of code, or access to that resource, becomes **serialized**.
- We've controlled concurrency. Where we need serial access, mutexes provide it. In other sections of our program, we allow the streams of execution to go untethered.

Serialization gives us predictability.
And, we like predictability.

Mutual Exclusion vs Synchronization

Synchronization is related, but different.

- **Synchronization** controls where threads are in the their streams of execution.
- If we need threads A and B to begin something at the same time, we need them to **synchronize**.
- If we need thread A to pause until some other condition to occur, we need thread A to synchronize to that condition.

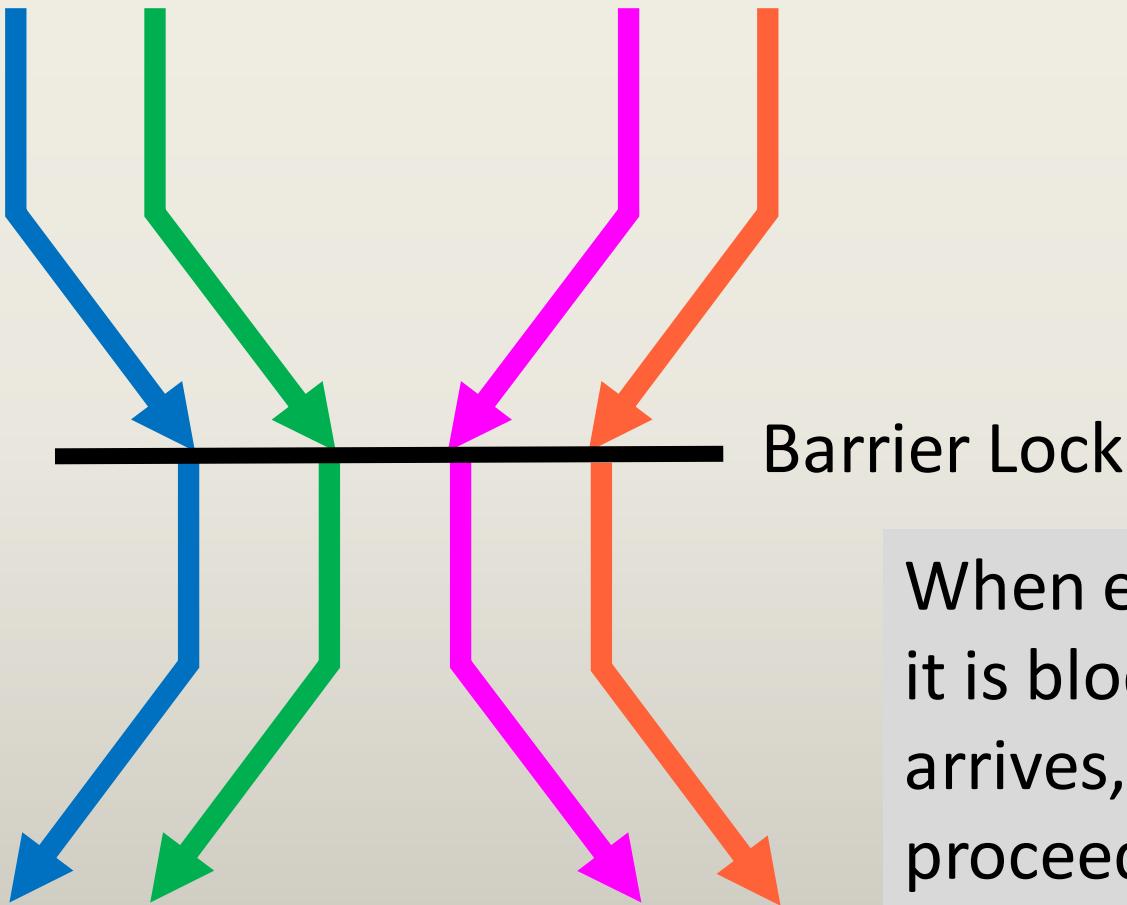


Barrier Locks

- In cases where you must wait for a number of tasks to be completed before an overall task can proceed, **barrier synchronization** can be used.
- POSIX threads specifies a **synchronization object called a barrier**, along with functions to utilize the barriers.
- The functions create the barrier, specifying the number of threads that are synchronizing on the barrier, and set up threads to perform tasks and wait at the barrier until all the threads reach the barrier.
- **When the last thread arrives at the barrier, all the threads resume execution.**



Barrier Locks



When each thread reaches the barrier, it is blocked there, until the last thread arrives, at which time, all threads can proceed.

```
#define NUM_THREADS 6
pthread_barrier_t barrier; → Create the barrier variable.

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int i, ids[NUM_THREADS] = {0,1,2,3,4,5}; → Data to pass to the threads.
    pthread_attr_t attr;

    pthread_barrier_init(&barrier, NULL, NUM_THREADS); → Initialize the barrier.

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED); → Create the threads as detached.

    for (i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], &attr, threadFunction, &ids[i]);
    } → Create and start the threads.

    pthread_exit(EXIT_SUCCESS);
}
```

```
void *threadFunction(void *a)
{
    int id = *(int *) a;

    printf("thread %d starting\n", id); First, thing, take a little gnap.
    sleep(id); // Simulation of work being done.
    printf(" thread %d to barrier\n", id);

    pthread_barrier_wait(&barrier); Check into the barrier.

    printf(" thread %d going again\n", id);

    pthread_exit(EXIT_SUCCESS);
}
```

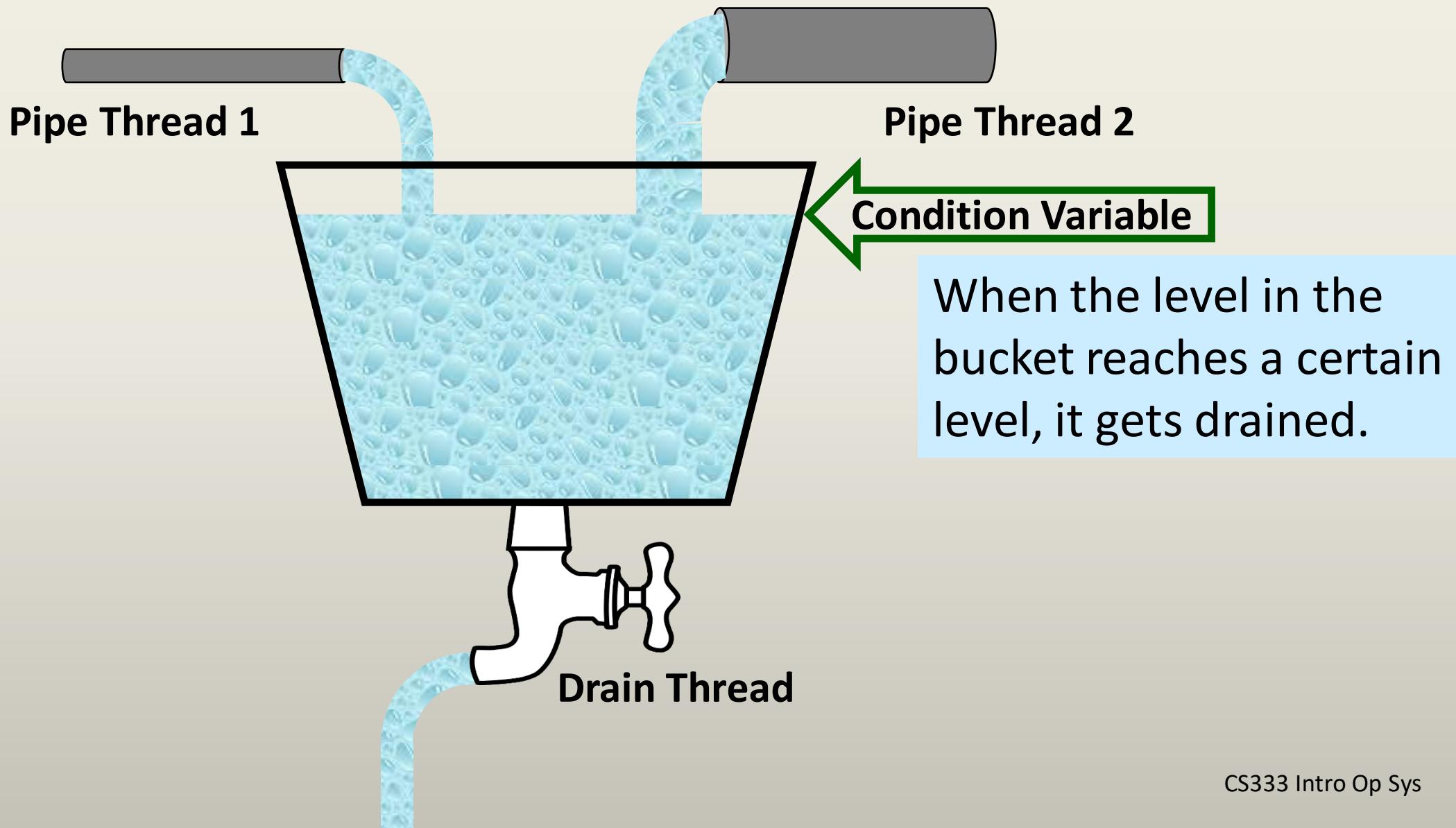
When all the necessary threads have checked into the barrier, they can all proceed.

Condition Variables

- Condition variables allow threads to synchronize based upon the actual value of data.
- A condition variable is always used in conjunction with a mutex lock.
- If PThreads did not provide condition variables, only providing mutexes, threads would need to **poll** a variable to determine when it reaches a certain state.



Condition Variables



Condition Variables

```
// For statically allocated mutex and condition variables.  
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cond_var = PTHREAD_COND_INITIALIZER;  
  
// For non-statically allocated mutex and condition variables.  
pthread_mutex_init(& mutex, NULL);  
pthread_cond_init(&cond_var, NULL);  
pthread_cond_destroy(&cond_var); // When you are done with it.  
  
// A thread calls this to wait until the condition is signaled.  
pthread_cond_wait(&cond_var, & mutex);  
  
// A different thread calls this to wake ONE of the threads  
// waiting on the condition variable.  
pthread_cond_signal(&cond_var);  
  
// A different thread calls this to wake ALL of the threads  
// waiting on the condition variable.  
pthread_cond_broadcast(&cond_var);
```

```
#define TCOUNT 10
#define WATCH_COUNT 12
int count = 0;
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  count_threshold_cv = PTHREAD_COND_INITIALIZER;

main(void)
{
    int i;
    pthread_t threads[3];

    pthread_create(&threads[0],NULL, inc_count, &thread_ids[0]);
    pthread_create(&threads[1],NULL, inc_count, &thread_ids[1]);
    pthread_create(&threads[2],NULL, watch_count, &thread_ids[2]);

    for (i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }
    pthread_exit(EXIT_SUCCESS);
}
```

A mutex and a condition variable.

Notice 2 different work functions.

```

void inc_count(int *idp) {
    for (i = 0; i < TCOUNT; i++) {
        pthread_mutex_lock( &count_mutex );
        count++;
        printf("inc_count(): Thread %d, old count %d, new count %d\n"
               , *idp, count - 1, count );
        if (count == WATCH_COUNT) { ←
            pthread_cond_signal( &count_threshold_cv );
        }
        pthread_mutex_unlock( &count_mutex );
    }
}

void watch_count(int *idp) {
    pthread_mutex_lock( &count_mutex );
    while (count <= WATCH_COUNT) { ←
        pthread_cond_wait( &count_threshold_cv , &count_mutex );
        printf("watch_count(): Thread %d, Count is %d\n",*idp, count);
    }
    pthread_mutex_unlock( &count_mutex );
}

```

When the condition is reached, signal a waiting thread.

Waiting on the condition variable.

Reader-Writer Locks

Sometimes there are 2 clear categories of uses for shared resources: **readers** and **writers**.

You may have called them setters and getters, or accessor and mutator in the OOP context.

In order to manage throughput and the resources well, you have the Reader Writer Problem.

- Allowing multiple readers is not a problem.
- Allowing multiple writers can cause corruption.
- Allowing readers and writers at the same time can lead to incorrect data being returned.



Meet *the*
Writers

Reader-Writer Locks

Assume you have a big linked list (or an array or a hash table, or tree structure, or ... something with a lot of elements in it).

It is a popular resource (like a list of books at a large library or an airline reservation system).

You have browsers and check-outs.

1. You also want to minimize wait time and maximize utilization of the resource (that's how you make money).
2. And, you don't want your resource corrupted (that's how you go out of business).



Reader-Writer Locks

You could put one mutex around the whole structure.

Only allow a single thread in at a time.

While this does prevent data corruption, it minimizes throughput and money.

Your competition loves it!



Reader-Writer Locks

You always allow readers in and only allow writers in when all the readers are done.

You probably make more money from writers than readers and you've just made writers wait an unbounded amount of time.

Your competition loves it!



Reader-Writer Locks

You always allow writers in and only allow readers in when all the writers are done.

You make more money from the writers, but without the hordes of readers, the writers are not interested in your product.

Your competition loves it!



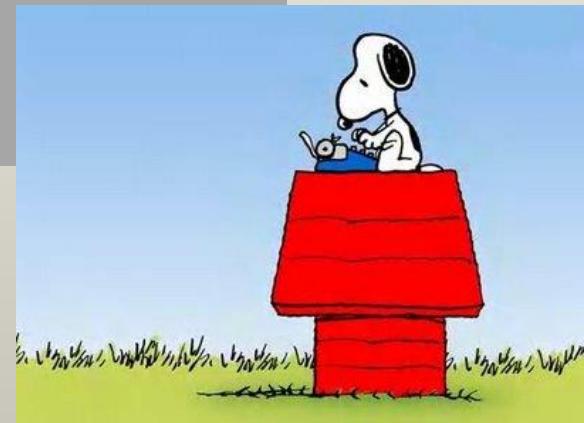
Reader-Writer Locks

See why it's called the Readers Writers **Problem**?

First, give up on the idea that having one mutex is an answer that will give you the performance you need.

Second, consider putting some form of mutex to control concurrency in each element of the resource.

Not all examples of this type need to be solved with the reader/writer mutex, but it is an option.



You probably feel like
you are about to
create something like
this.

Don't try and jump in
the deep end too
quickly.



Reader-Writer Locks



```
// For statically allocated rw mutex variables.  
pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;  
  
// For non-statically allocated mutex and condition variables.  
pthread_rwlock_t lock;  
pthread_rwlock_init(&lock, NULL);  
  
pthread_rwlock_destroy(&lock); // When you are done with the rw mutex.  
  
// Acquiring a read lock.  
pthread_rwlock_rdlock(&lock);  
pthread_rwlock_tryrdlock(&lock); pthread_rwlock_timedrdlock(&lock);  
  
// Acquiring a write lock.  
pthread_rwlock_trywrlock(&lock);  
pthread_rwlock_wrlock(&lock); pthread_rwlock_timedwrlock(&lock);  
  
// Releasing a read/write lock.  
pthread_rwlock_unlock(&lock);
```



Thread Safety

- A function is said to be **thread-safe** if it can safely be invoked by multiple threads at the same time **and yield predictable results**.
- A function is not thread-safe, then if cannot be called from one thread while it is being executed in another thread, **yielding predictable results**.

Is this function thread-safe?

```
static int glob = 0;  
  
static void incr(int loops)  
{  
    int loc, j;  
    for (j = 0; j < loops; j++)  
    {  
        loc = glob;  
        loc++;  
        glob = loc;  
    }  
}
```

NO!

Non-Thread-Safe Functions

<code>asctime()</code>	<code>fcvt()</code>	<code>getpwname()</code>	<code>nl_langinfo()</code>
<code>basename()</code>	<code>ftw()</code>	<code>getpwuid()</code>	<code>ptsname()</code>
<code>catgets()</code>	<code>gcvt()</code>	<code>getservbyname()</code>	<code>putc_unlocked()</code>
<code>crypt()</code>	<code>getc_unlocked()</code>	<code>getservbyport()</code>	<code>putchar_unlocked()</code>
<code>ctime()</code>	<code>getchar_unlocked()</code>	<code>getservent()</code>	<code>putenv()</code>
<code>dbm_clearerr()</code>	<code>getdate()</code>	<code>getutxent()</code>	<code>pututxline()</code>
<code>dbm_close()</code>	<code>getenv()</code>	<code>getutxid()</code>	<code>rand()</code>
<code>dbm_delete()</code>	<code>getgrent()</code>	<code>getutxline()</code>	<code>readdir()</code>
<code>dbm_error()</code>	<code>getgrgid()</code>	<code>gmtime()</code>	<code>setenv()</code>
<code>dbm_fetch()</code>	<code>getgrnam()</code>	<code>hcreate()</code>	<code>setgrent()</code>
<code>dbm_firstkey()</code>	<code>gethostbyaddr()</code>	<code>hdestroy()</code>	<code>setkey()</code>
<code>dbm_nextkey()</code>	<code>gethostbyname()</code>	<code>hsearch()</code>	<code>setpwent()</code>
<code>dbm_open()</code>	<code>gethostent()</code>	<code>inet_ntoa()</code>	<code>setutxent()</code>
<code>dbm_store()</code>	<code>getlogin()</code>	<code>l64a()</code>	<code>strerror()</code>
<code>dirname()</code>	<code>getnetbyaddr()</code>	<code>lgamma()</code>	<code>strtok()</code>
<code>derror()</code>	<code>getnetbyname()</code>	<code>lgammaf()</code>	<code>ttyname()</code>
<code>drand48()</code>	<code>getnetent()</code>	<code>lgammal()</code>	<code>unsetenv()</code>
<code>ec</code>	<code> getopt()</code>	<code>localeconv()</code>	<code>wcstombs()</code>
<code>encrypt()</code>	<code>getprotobynumber()</code>	<code>localtime()</code>	<code>wctomb()</code>
<code>endgrent()</code>	<code>getprotoent()</code>	<code>lrand48()</code>	
<code>endpwent()</code>	<code>getpwent()</code>	<code>mrand48()</code>	
<code>endutxent()</code>		<code>nftw()</code>	

Reentrant and Non-Reentrant

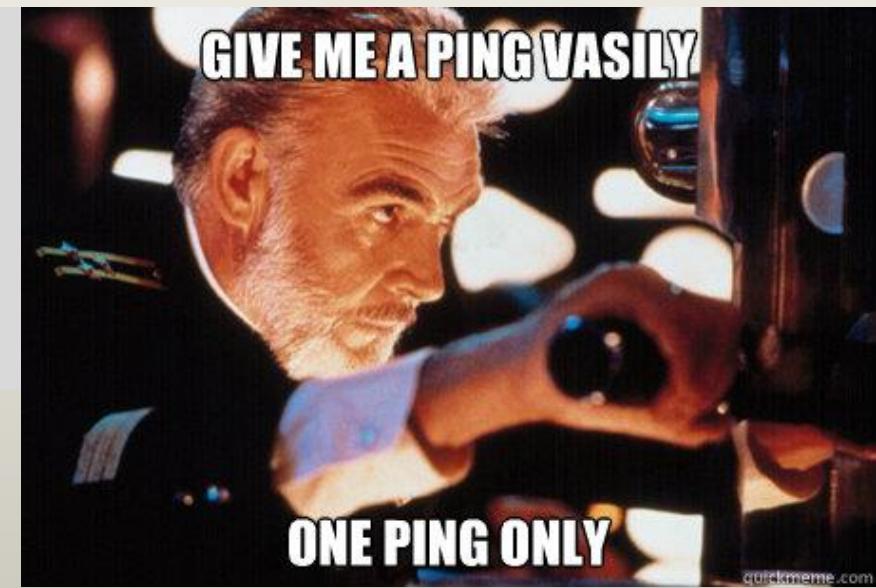
- The **use of critical sections** to implement thread safety is a significant improvement over the use of per-function mutexes, it is still somewhat inefficient because **there is a cost to locking and unlocking a mutex**.
- A reentrant function achieves thread safety without the use of mutexes.
 - All information that must be returned to the caller, or maintained between calls to the function, is **stored in buffers allocated by the caller**.

One-Time Initialization

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control
, void (*init) (void));
```

Sometimes, a threaded application needs to ensure that some initialization action occurs **just once**, regardless of how many threads are created.



One-Time Initialization

The `once_control` argument is a pointer to a variable that must be statically initialized with the value `PTHREAD_ONCE_INIT`:

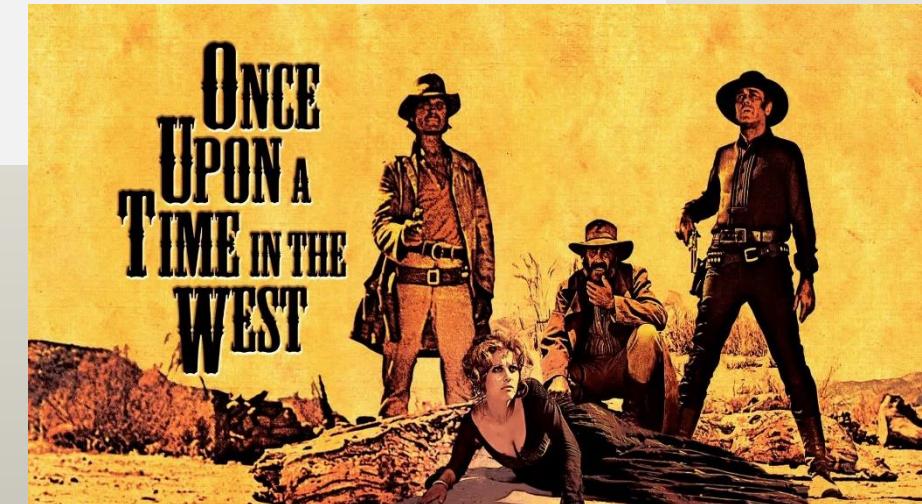
```
pthread_once_t once_var = PTHREAD_ONCE_INIT;
```



One-Time Initialization

The `init` function is called without any arguments, and thus has the following form:

```
void init(void)
{
    /* Function body */
}
```

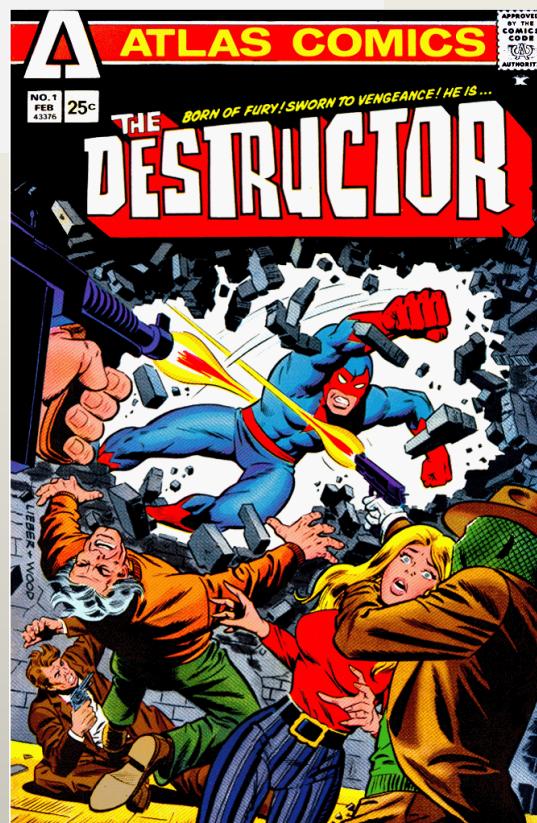


Thread-Specific Data

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key
, void (*destructor) (void *));
```

This function allows you to set up a destructor
(ante-constructor?) for a thread's local data.



Thread-Local Storage

To create a **thread-local variable**, you simply include the `__thread` specifier in the declaration of a **global or static variable**:

```
static __thread char buf[MAX_ERROR_LEN];
```

- Each thread has its own copy of the variables declared with this specifier.
- The variables in a thread's thread-local storage persist until the thread terminates, at which time the storage is **automatically deallocated**.



Local
Storage



Thread Cancellation

- Usually, multiple threads execute in parallel in the program, each thread performing a task until it needs to terminate by calling `pthread_exit()` or returning from the thread's start function.
- Sometimes, it can be useful to cancel a thread; send it a request asking it to terminate **now!**



Cancelling a Thread



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

```
int pthread_cancel(pthread_t thread);
```

Returns 0 on success, or a positive error number on error

`pthread_cancel()` **returns immediately**; it does not wait for the target thread to actually terminate.

Precisely what happens to the target thread and when it happens **depends on that thread's cancellation state and type**.

Cancellation State

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

- PTHREAD_CANCEL_DISABLE

The thread **is not cancelable**.

If a cancellation request is received, it remains pending until cancelability is enabled.

- PTHREAD_CANCEL_ENABLE

The thread **is cancelable**.

This is the default cancelability state in newly created threads.

Temporarily disabling cancellation **is useful if a thread is executing a section of code where all of the steps must be completed (a critical section)**.

Cancellation Type

```
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

- PTHREAD_CANCEL_ASYNCHRONOUS

The thread may be canceled at any time (perhaps, but not necessarily, immediately).
Typically, it will be canceled immediately upon receiving a cancellation request, but the system doesn't guarantee this.
- PTHREAD_CANCEL_DEFERRED

The cancellation remains pending until a **cancellation point** is reached.
This is the default cancelability type in newly created threads.

Cancellation Points

- When **cancelability is enabled and deferred**, a cancellation request is **acted upon only when a thread next reaches a cancellation point**.
- A cancellation point is a call to one of a set of functions defined by the implementation.

- A cancellation point, in general, is a point in the stream of execution where **control returns to the scheduler**.
- The meaning of "cancellation" is to not get scheduled again, so you cancel something when you can influence the scheduling decisions.
- System calls form a natural interaction with the scheduler.



Cancellation Points

<i>accept()</i>	<i>nanosleep()</i>	<i>sem_timedwait()</i>
<i>aio_suspend()</i>	<i>open()</i>	<i>sem_wait()</i>
<i>clock_nanosleep()</i>	<i>pause()</i>	<i>send()</i>
<i>close()</i>	<i>poll()</i>	<i>sendmsg()</i>
<i>connect()</i>	<i>pread()</i>	<i>sendto()</i>
<i>creat()</i>	<i>pselect()</i>	<i>sigpause()</i>
<i>fentl(F_SETLKW)</i>	<i>pthread_cond_timedwait()</i>	<i>sigsuspend()</i>
<i>fsync()</i>	<i>pthread_cond_wait()</i>	<i>sigtimedwait()</i>
<i>fdatasync()</i>	<i>pthread_join()</i>	<i>sigwait()</i>
<i>getmsg()</i>	<i>pthread_testcancel()</i>	<i>sigwaitinfo()</i>
<i>getpmsg()</i>	<i>putmsg()</i>	<i>sleep()</i>
<i>lockf(F_LOCK)</i>	<i>putpmsg()</i>	<i>system()</i>
<i>mq_receive()</i>	<i>pwrite()</i>	<i>tcdrain()</i>
<i>mq_send()</i>	<i>read()</i>	<i>usleep()</i>
<i>mq_timedreceive()</i>	<i>readv()</i>	<i>wait()</i>
<i>mq_timedsend()</i>	<i>recv()</i>	<i>waitid()</i>
<i>msgrcv()</i>	<i>recvfrom()</i>	<i>waitpid()</i>
<i>msgsnd()</i>	<i>recvmsg()</i>	<i>write()</i>
<i>msync()</i>	<i>select()</i>	<i>writev()</i>

Testing for Thread Cancellation

```
#include <pthread.h>

void pthread_testcancel(void);
```

Calling `pthread_testcancel()` creates a cancellation point within the calling thread.

If a cancellation is pending when this function is called, then the calling thread is terminated.



Cleanup Handlers

- If a thread with a pending cancellation were simply terminated when it reached a cancellation point, then shared variables and Pthreads objects (for example, **mutexes**) **might be left in an inconsistent state**, perhaps causing the remaining threads in the process to produce **incorrect results, deadlock, or crash**.
- A **cleanup handler** can perform tasks such as modifying the values of global variables and unlocking mutexes before the thread is terminated.



Cleanup Handlers

```
#include <pthread.h>

void pthread_cleanup_push(void (*cleanup_routine) (void*)
    , void *arg);
void pthread_cleanup_pop(int execute);
```

- Each thread can have a stack of cleanup handlers.
- When a thread is canceled, the cleanup handlers are executed working down from the top of the stack; the most recently established handler is called first.
- After all of the cleanup handlers have been executed, the thread terminates.



Cleanup Handlers

The routine argument is a pointer to a function that has the following form:

```
void cleanup_routine( void *arg )  
{  
    /* Code to perform cleanup */  
}
```



Cleanup Handlers

- If a thread reaches the end of a section without being canceled and the cleanup action is no longer required...
- Each call to `pthread_cleanup_push()` can have a matching call to `pthread_cleanup_pop()`.
 - If the execute argument is nonzero, the handler is also executed.
- As a **convenience**, any cleanup handlers that have not been popped are executed automatically if a thread terminates by calling `pthread_exit()`
 - **They are not run if it does a simple return.**



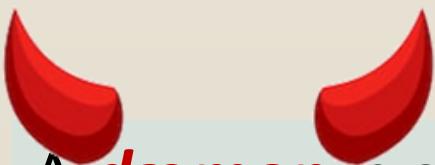


Dæmon's Origin

- The word **dæmon** was first used in a computer context at the pioneering Project MAC (which later became the MIT Laboratory for Computer Science) using the IBM 7094 in 1963.
- This usage was inspired by Maxwell's daemon of physics and thermodynamics, which was an imaginary agent that helped sort molecules of different speeds and worked tirelessly in the background.
- The term was then used to describe background processes which worked tirelessly to perform system chores.
- The first computer **dæmon** was a program that automatically made tape backups.
- After the term was adopted for computer use, it was rationalized as a **backronym** for Disk And Execution MONitor (DAEMON -> DÆMON)

<http://www.linfo.org/daemon.html>

What's a Dæmon?



A **dæmon** is a process with the following characteristics:

- It is **long-lived**. Often, a **dæmon** is created at system startup and runs until the system is shut down.
- It **runs in the background and has no controlling terminal**.
The lack of a controlling terminal ensures that the kernel never automatically generates any job-control or terminal-related signals (such as SIGINT, SIGTSTP, and SIGHUP) for a **dæmon**.

Some “other” operating systems call **dæmon** processes “services.”

Dæmon Examples

Dæmons are w

- *cron*: a **dæmon** that runs commands at a scheduled time.
- *sshd*: the secure shell daemon, which permits logins from remote hosts using a public key infrastructure protocol.
- *httpd*: the HTTP daemon.
- *inetd*: the Internet daemon, which listens for incoming network connections on specified TCP/IP ports and launches appropriate server programs to handle these connections.



ut specific tasks. Examples include:

commands at a scheduled time.
, which permits logins from remote applications protocol.

Dæmon processes often have a ‘d’ as the last letter in the name.

Creating a Daemon



There are several steps to go through when created a **dæmon** process.

- The 2 most important ones are for the process to **disassociate itself with any controlling terminal.**
 1. A controlling terminal would allow the process to be stopped (via a control-Z) or killed when the controlling terminal is closed (receiving the SIGHUP signal). This is done with a call to the setsid() function.
 2. Redirect file descriptors 0, 1, and 2 (stdin, stdout, and stderr) to /dev/null. This prevents the process from hanging waiting for input in stdin or filling buffers sending output to stdout or stderr and hanging from that.

A **dæmon** will often be a child of the `init` process (`pid == 0`).

The `/dev/null` Device

Data written to `/dev/null` is always discarded. In effect, it is a device that **can never be filled up**. You just cannot get back any of the data you write there.



Reads from `/dev/null` always return end of file. So, a read will always return 0.

Use of this device allows a **daemon** to not hang when reading from or writing to the stdio file descriptors.

System Shutdown



- When a (graceful) system shutdown is initiated, the `init` process will send a `SIGTERM` to all its child processes.
- A **dæmon** process can establish a handler for `SIGTERM` and quickly perform any cleanup necessary for exiting.
- The **dæmon** process will need to be fairly quick about, because the `init` process will send a `SIGKILL` signal to any remaining child processes 5 seconds after the `SIGTERM`. The `SIGKILL` cannot be caught or ignored.
- This allows for a graceful shutdown of all **dæmon** processes during a system shutdown.

Reinitialize a Daemon

- Since most **dæmon** processes are intended to run continuously, it can become a problem if you want change a configuration parameter of a daemon process.
- Killing it and restarting it means that any existing connections are terminated and lost.
- A common solution to this is to establish a handler for the **SIGHUP** signal and have receipt of that signal cause the daemon to reread any configuration files.





Logging Messages

- When writing a **dæmon**, one problem we encounter is **where to send diagnostics and error messages**.
- Since a **dæmon** runs in the background (without an associated terminal), we can't display messages on a terminal, as we would typically do with other programs.
- One possible alternative is to write messages to an **application-specific log file**.
 - This makes it difficult for a system administrator to manage the multiple application log files and monitor them all for error messages or even know where they are all located on the file system.

Logging Messages

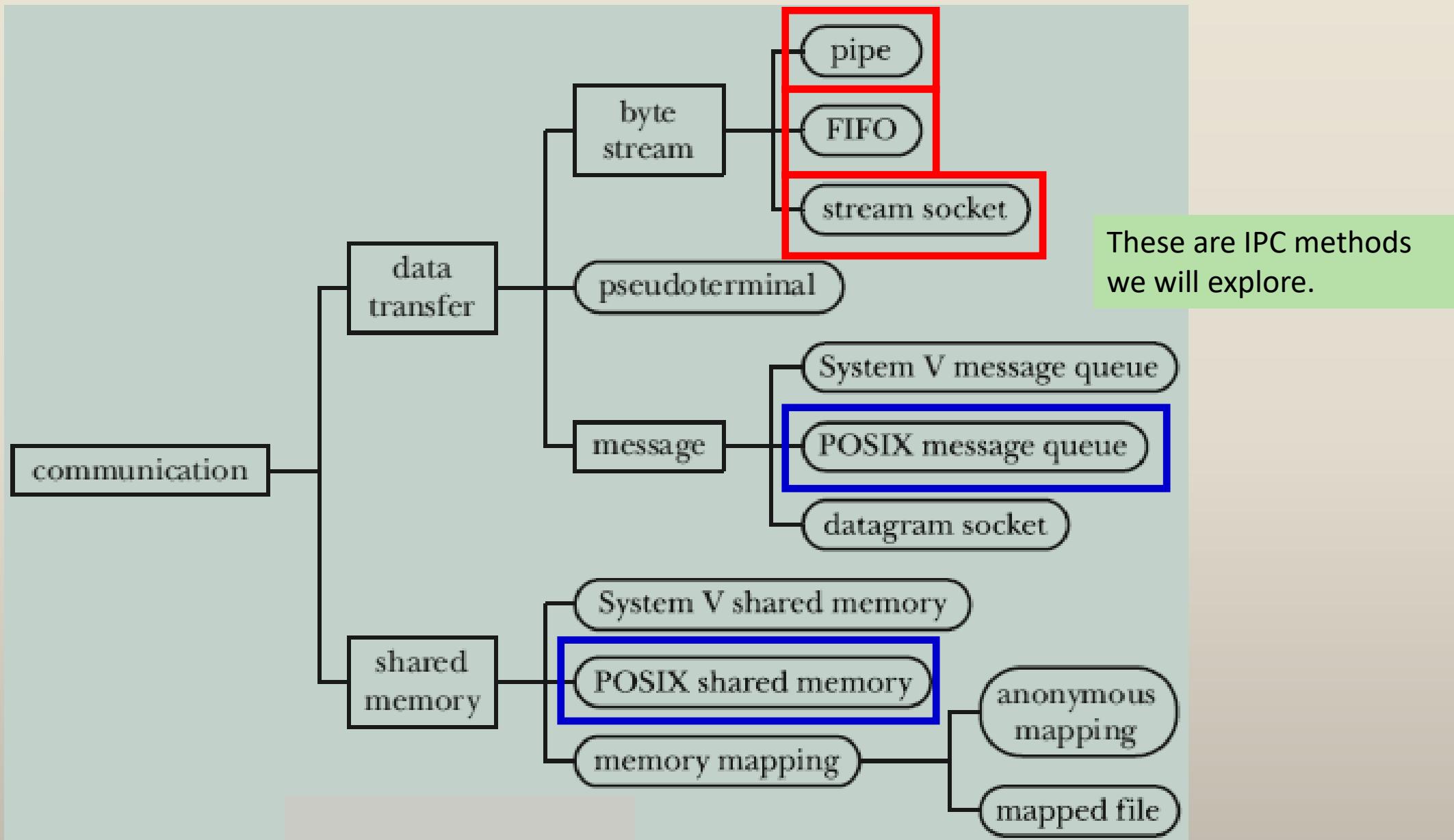


- A better solution for storing diagnostic and error messages is use of the **syslog** facility.
- The syslog facility provides a single, **centralized logging facility that can be used to log messages by all applications on the system**.
- The **syslog facility has two principal components**:
 1. the **syslogd daemon** and
 2. the **syslog(3) library function**.
- The syslog facility is not limited to **dæmons**, it can be used by any process to log messages to a standard location.





InterProcess Communication (**IPC**) in UNIX is how processes exchange data or state information.



Communication Facilities

- **Data-transfer facilities:** The key factor distinguishing these facilities is the notion of writing and reading.
 - To communicate, one process writes data to the IPC facility, and another process reads the data.
- **Shared memory:** Shared memory allows processes to exchange information by placing it in a region of memory that is **shared between the processes**.



Data Transfer

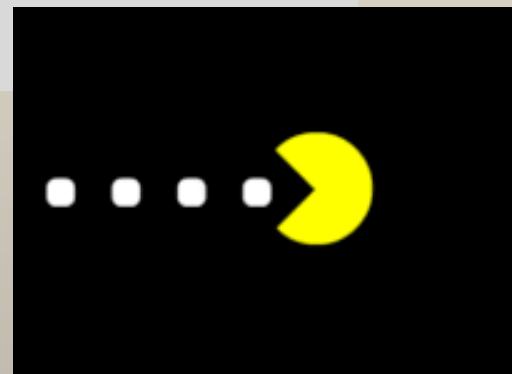
- **Byte stream:** The data exchanged via pipes, FIFOs, and stream sockets is an **un-delimited** stream of bytes.
- **Message:** The data exchanged via message queues and datagram sockets takes the form of **delimited messages**.



Data Transfer

A few general features distinguish data-transfer facilities from shared memory.

- Although a data-transfer facility may have multiple readers,
reads are destructive.
 - A **read operation consumes data**, and that data are not available to any other process.

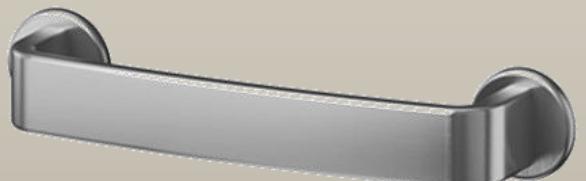


Data Transfer

- Synchronization between the reader and writer processes is automatic.
 - If a reader attempts to fetch data from a data-transfer facility that currently has no data, then the read operation will **block** until some process writes data to the facility.



IPC Facility	Name used to identify object	Handle used to refer to object in programs
Pipe FIFO	no name pathname	File descriptor File descriptor
UNIX domain socket Internet domain socket	pathname IP address + port number	File descriptor File descriptor
POSIX message queue POSIX named semaphore POSIX unnamed semaphore POSIX shared memory	POSIX IPC pathname POSIX IPC pathname no name POSIX IPC pathname	message queue descriptor named semaphore unnamed semaphore File descriptor

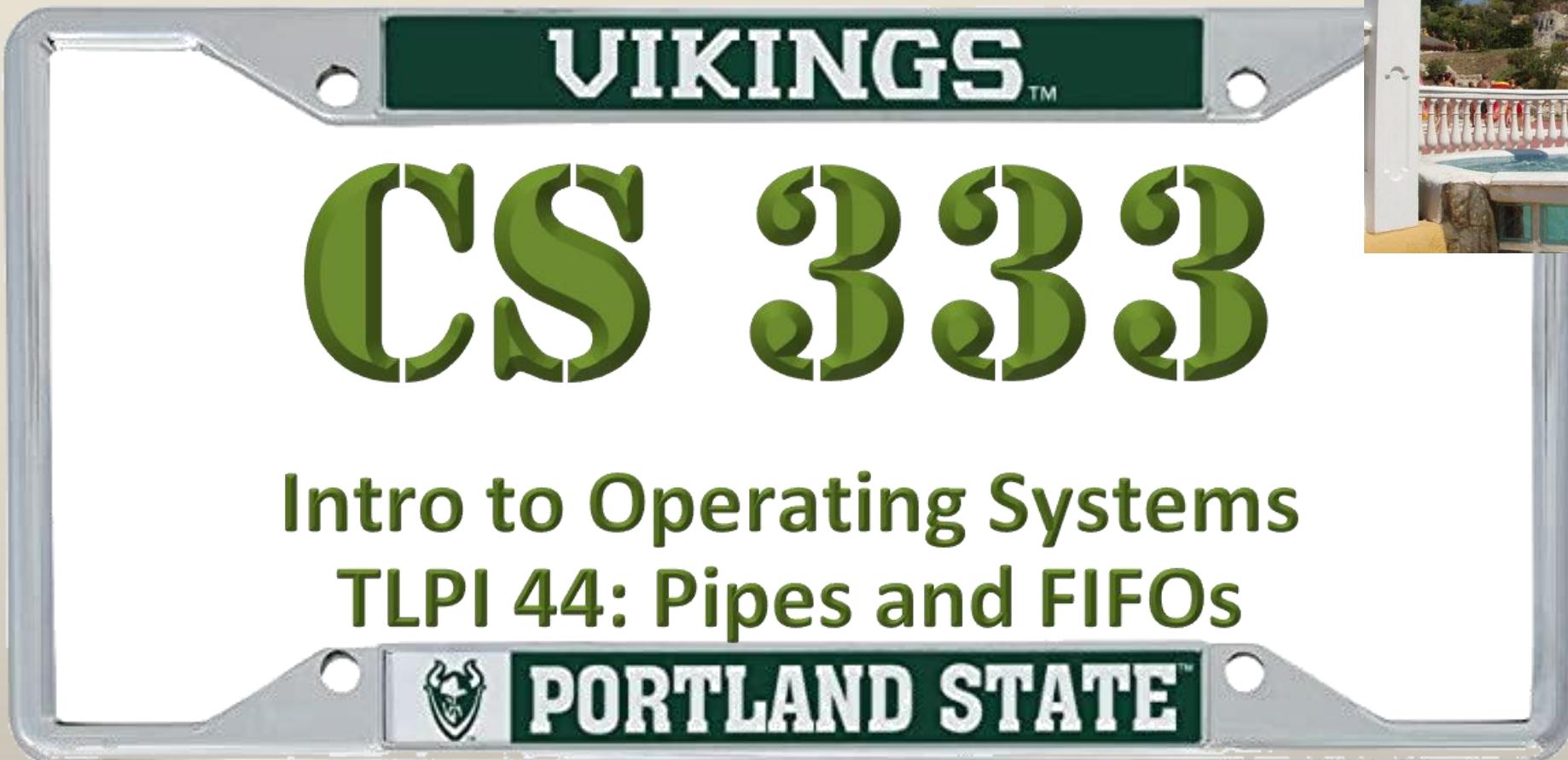


Persistence

- **Process persistence:** A process-persistent IPC object remains in existence only as long as it is held open by at least one process.
- **Kernel persistence:** A kernel-persistent IPC object exists until either it is explicitly deleted or the system is shut down.
- **File-system persistence:** An IPC object with file-system persistence retains its information even when the system is rebooted.

I do like to ask about this on exams.





Pipes and FIFOs



- **Pipes are the oldest** method of IPC on the UNIX system, having appeared in Third Edition UNIX in the early 1970s.
 - **Pipes provide a solution to the common requirement, create two processes running different commands, allow the output produced by one process to be used as the input to the other process (aka a pipeline).**
- **FIFOs are also called *named pipes*.**
- Both are **byte-streams**. There are **no message boundaries** between data written to a pipe or FIFO. Bytes go in and bytes come out.

Pipes

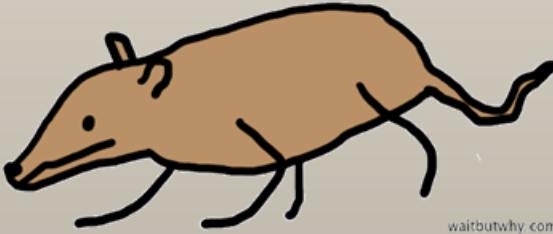
- Pipes are used between **related** processes. Sometimes called **anonymous pipes** or **unnamed pipes**.
- Processes using pipes to communicate will have a **common ancestor process**.
- Pipes are created by a parent process then child processes are created (using **fork()**) to make use of the pipes.



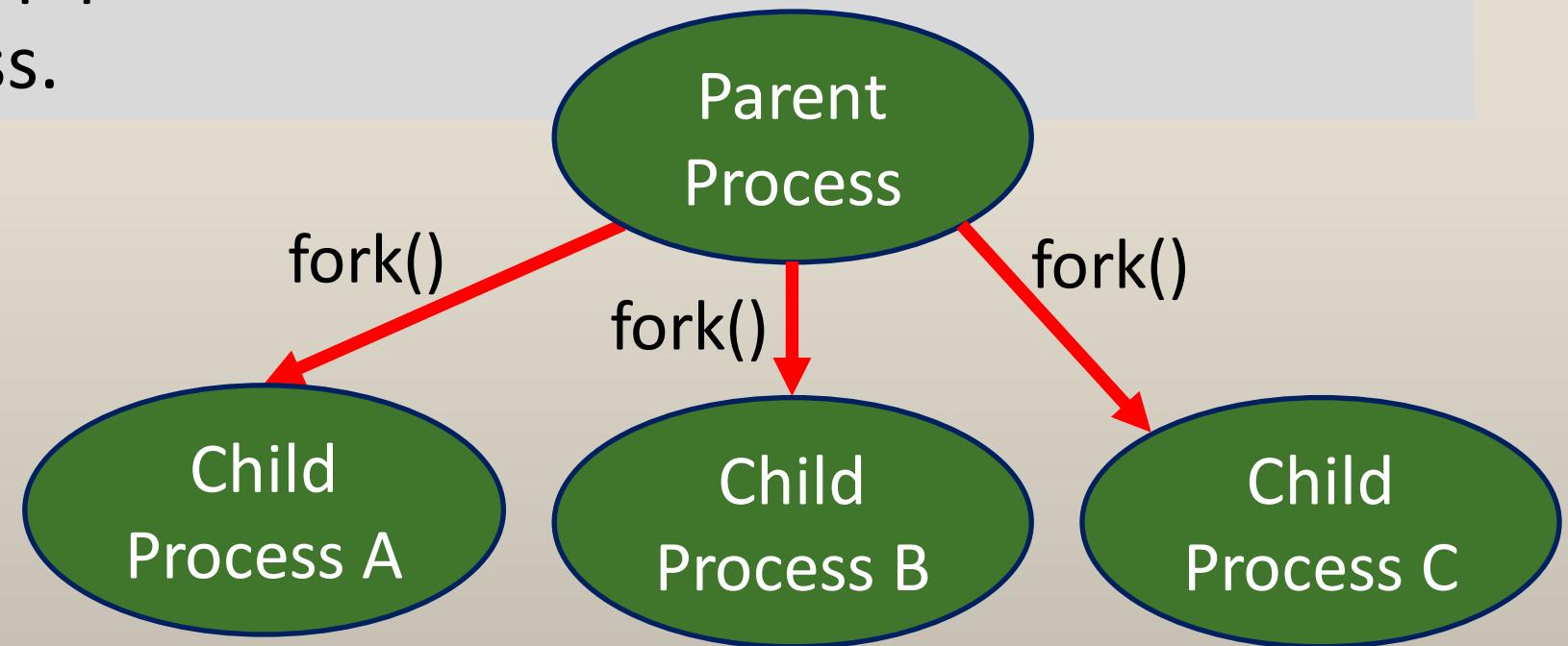
Relations Only

- Pipes are **used between related** processes.
Sometimes called anonymous or unnamed pipes.
- Processes using pipes to communicate will have a **common ancestor** process.

– 160 Million Years Ago –
Your Great^{55,000,000} Grandfather



waitbutwhy.com



Pipes

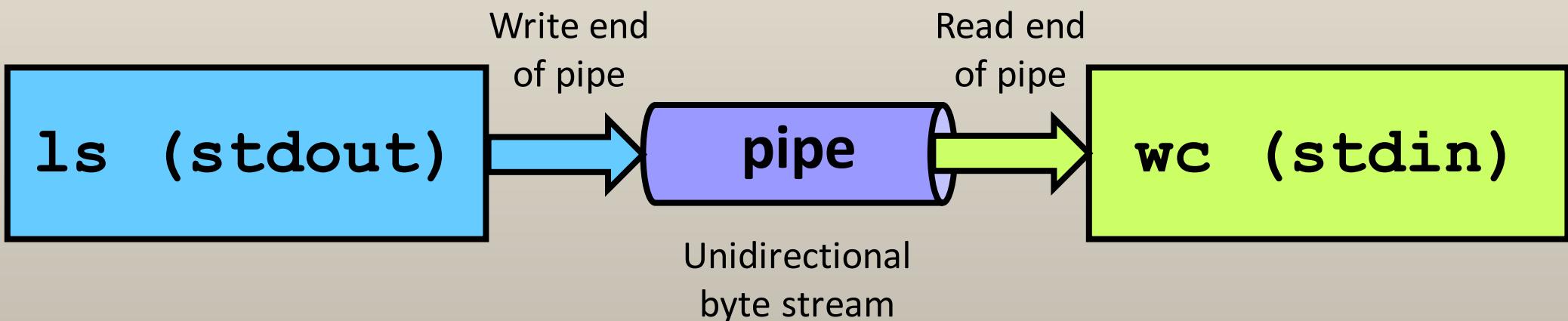
You've used pipes on the command line between processes.

The shell connects the processes via a pipe.

```
$ ls | wc -l
```



In order to execute the above command, **the shell creates two processes**, executing `ls` and `wc`, respectively.



A Pipe is a Byte Stream

BYTE



- A pipe is a byte stream, **there is no concept of messages or message boundaries** when using a pipe.

The process reading from a pipe can read chunks of data of any size, regardless of the size of chunks written by the writing process.
- **Data passes through the pipe sequentially** – bytes are read from a pipe in exactly the order they were written.

It is not possible to randomly access the data in a pipe using `lseek()`.

Reading from a Pipe

- Attempts to **read from a pipe that is currently empty will block** until at least one byte has been written to the pipe.
- **If the write end of a pipe is closed, then a process reading from the pipe will see end-of-file** (a `read()` will return 0) once it has read all remaining data in the pipe.

I use this as a test question.



Pipes are Unidirectional

- **Data can travel only in one direction** through a pipe.
- One end of the pipe is used for writing, and the other end is used for reading.
- You can have multiple readers/writers, but it is difficult.
- **Pipes are anonymous** (there is no entry in the file system for a pipe).



Writes of up to PIPE_BUF Bytes are Guaranteed to be Atomic

- If multiple processes are writing to a single pipe, then it is guaranteed that their data will not be comingled if they write no more than PIPE_BUF bytes at a time.



Pipes have a Limited Buffer Capacity

- Pipes are kernel structures. A pipe is simply a buffer maintained in kernel memory.
- The buffer has a maximum capacity.
- Once a pipe is full, further writes to the pipe **block** until the reader removes some data from the pipe.



This is not about how much total data can move through the pipe, but about how much data can be **held** in the pipe at any one time.

Creating Pipes



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

```
int pipe(int filedes[2]);
```

Returns 0 on success, or -1 on error.

Notice the array of size 2.

A successful call to `pipe()` results **in two open file descriptors** in the array `filedes`:

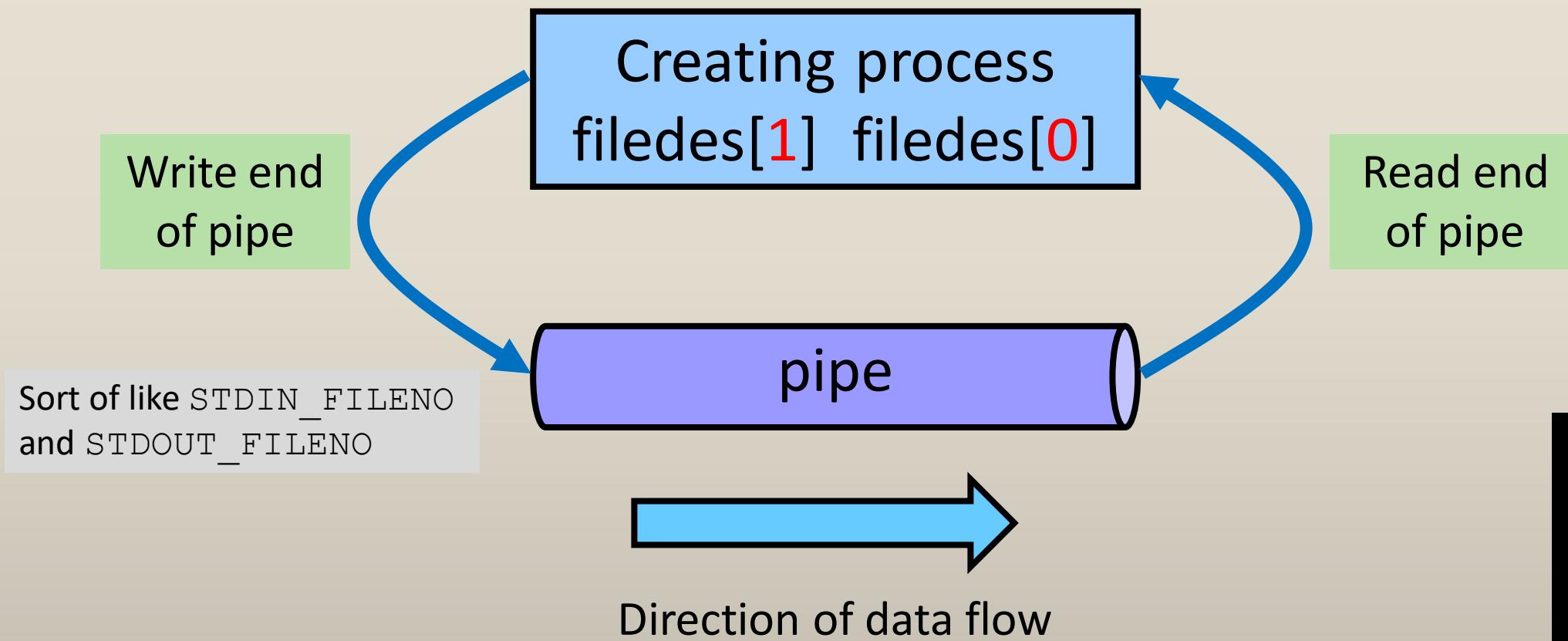
- one for the read end of the pipe (`filedes[0]`) and
- one for the write end (`filedes[1]`).

Using Pipes



- We use the `read()` and `write()` system calls to perform I/O on the pipe.
- Once data are written to the write end of a pipe, the data are immediately available from the read end.
- A `read()` from a pipe obtains the lesser of the number of bytes requested and the number of bytes currently available in the pipe (but **blocks** if the pipe is empty).

File Descriptors after Creating a Pipe



Creating Pipes



- Created using *pipe()*:

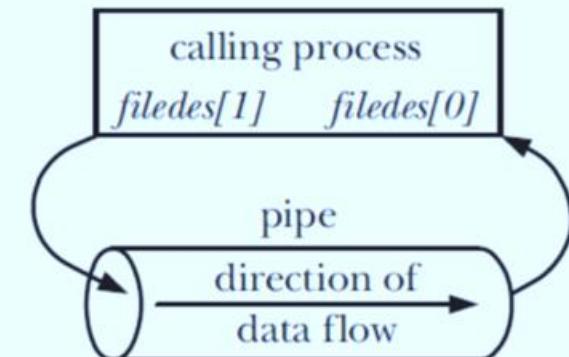
```
int filedes[ 2 ];
pipe( filedes );
```

...

```
write( filedes[1], buf, count );
read( filedes[0], buf, count );
```

The `pipe()` call creates the pipes. The pipes are file descriptors, just like `open()` returns.

You use `write()` and `read()` on a pipe to send or receive data.



An example of the **Universality of I/O**, sending data to another process using a pipe uses the same calls as writing to a file.

The parent process creates the pipe. A single pipe is represented as 2 file descriptors in the array `filedes`.

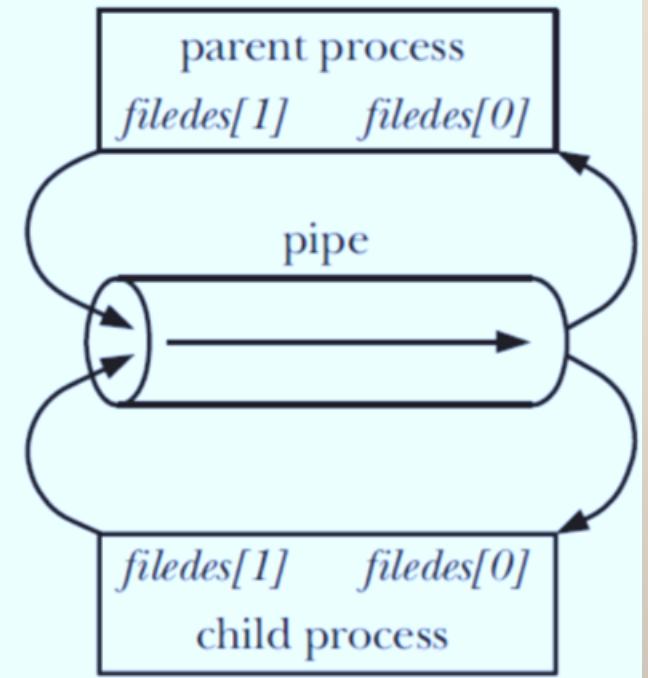
```
int filedes[2];  
pipe(filedes);  
child_pid = fork();
```

***fork()* duplicates parent's file descriptors**

The parent/child relationship means both get the newly created pipe.

Sharing a Pipe

Sharing is caring



Sharing a Pipe

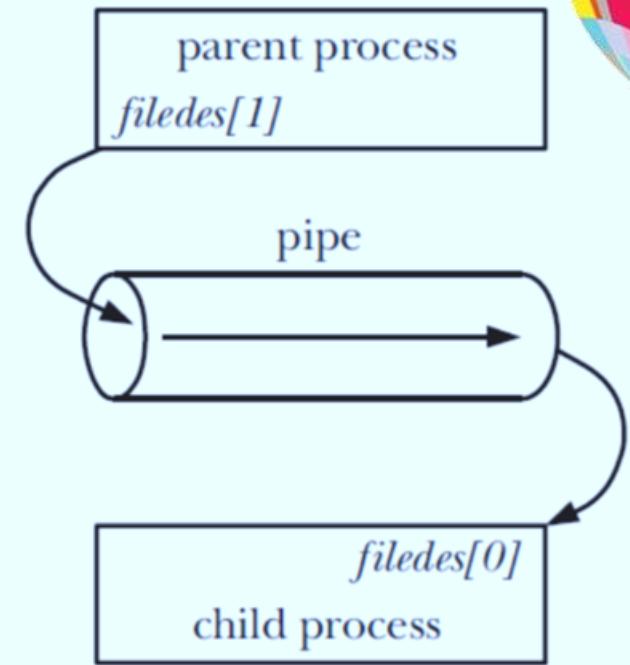
Sharing the Love



```
int filedes[2];
pipe(filedes);

child_pid = fork();
if (child_pid == 0) {
    close(filedes[1]);
    /* Child now reads */
} else {
    close(filedes[0]);
    /* Parent now writes */
}
```

Close the unused side of the pipe in each process.



In this image, the parent process will write data into the pipe for the child process to read.

Close Unused Sides of the Pipe



- Parent and child processes **must close** the unused file descriptors from a pipe.
 - This is necessary for the correct use of pipes.
 - `close()` write end
 - `read()` returns 0 (EOF)
 - `close()` read end
 - `write()` fails with EPIPE error and SIGPIPE signal



I/O on Pipes



- `read()` **blocks if pipe is empty**
- `write()` **blocks if pipe is full**
- Writes $\leq \text{PIPE_BUF}$ guaranteed to be atomic
 - Multiple writers $> \text{PIPE_BUF}$ may be interleaved
 - POSIX: `PIPE_BUF` at least 512 Bytes
 - Linux: `PIPE_BUF` is 4096 Bytes
- Can use `dup2()` to connect filters via a pipe.
- You don't have to open pipes, only close unused side.

Details to soon follow.

Using popen ()



- A frequent use for pipes is to execute a shell command and either read its output or send it some input.
- The `popen()` and `pclose()` functions are provided to simplify this task.

A call to `popen()` returns a file stream.

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

```
FILE *popen(const char *command, const char *mode);
```

```
int pclose(FILE *stream);
```

You can either read from **OR** write to the file stream returned from `popen()`

Returns file stream, or **NULL** on error

Returns termination status of child process, or **-1** on error

popen ()

- The popen () function creates a pipe, and then forks a child process that **execs a shell**, which in turn creates a child process to execute the string given in command.
- The mode argument is a string that determines whether the calling process will read from the pipe (mode is “r”) or write to it (mode is “w”).
- Since pipes are **unidirectional**, **two-way communication with the executed command is not possible**.



popen ()

While there are several similarities between `system()` and `popen()` plus `pclose()`, there are also some differences.

- The return value from `system()` is an `int`, indicating either success or failure.
- The `popen()` command **returns a file stream** that can either be read from or written to (using any of the stream functions, `fprintf()`, `fputs()`, `fscanf()`, or `fgets()`).
- There are some differences in how signals are handled between `system()` and `popen()`.

Don't forget the call
to `pclose()`



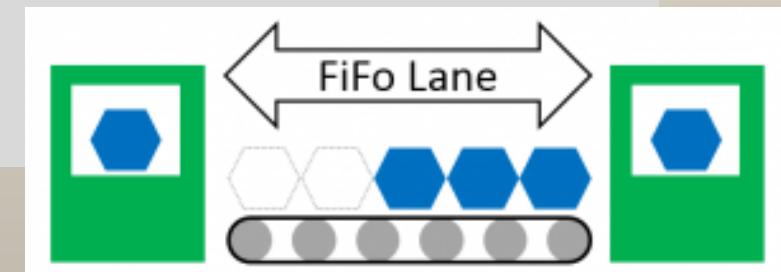
FIFOs aka *Named Pipes*

- Anonymous pipes can **only** be used between **related processes**.
- A FIFO is a pipe with a name in the file system.
- Creation:
 - **`mkfifo(path, permissions);`**
 - There is also a command to create a fifo.
- Unlike pipes, you **do have to open FIFOs before use.**
- Any process can open and use a FIFO (based on permissions).



FIFOs aka *Named Pipes*

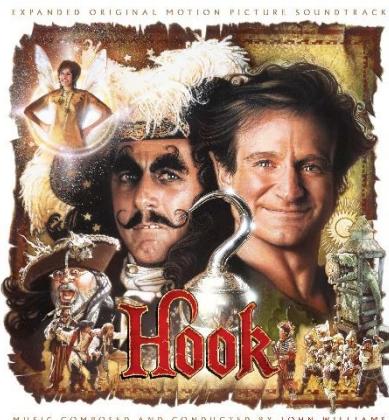
- I/O is the same as for pipes (read and write).
- A FIFO has a write end and a read end, and data are read from the pipe in the same order as it is written.
- FIFOs are kernel data structures, exactly like pipe.
- Unlike pipes, **you must explicitly delete a fifo from the file system when you are done with it.**
 - A **fifo** has file-system persistence.



Opening a FIFO

- `open(path, O_RDONLY);`
 - Open read end of a FIFO
- `open(path, O_WRONLY);`
 - Open the write side of a FIFO
- **Calls to `open()` will block until the other end of the FIFO is opened.**
- Opens are synchronized.
- You can also
 - `open(path, O_RDONLY | O_NONBLOCK);`

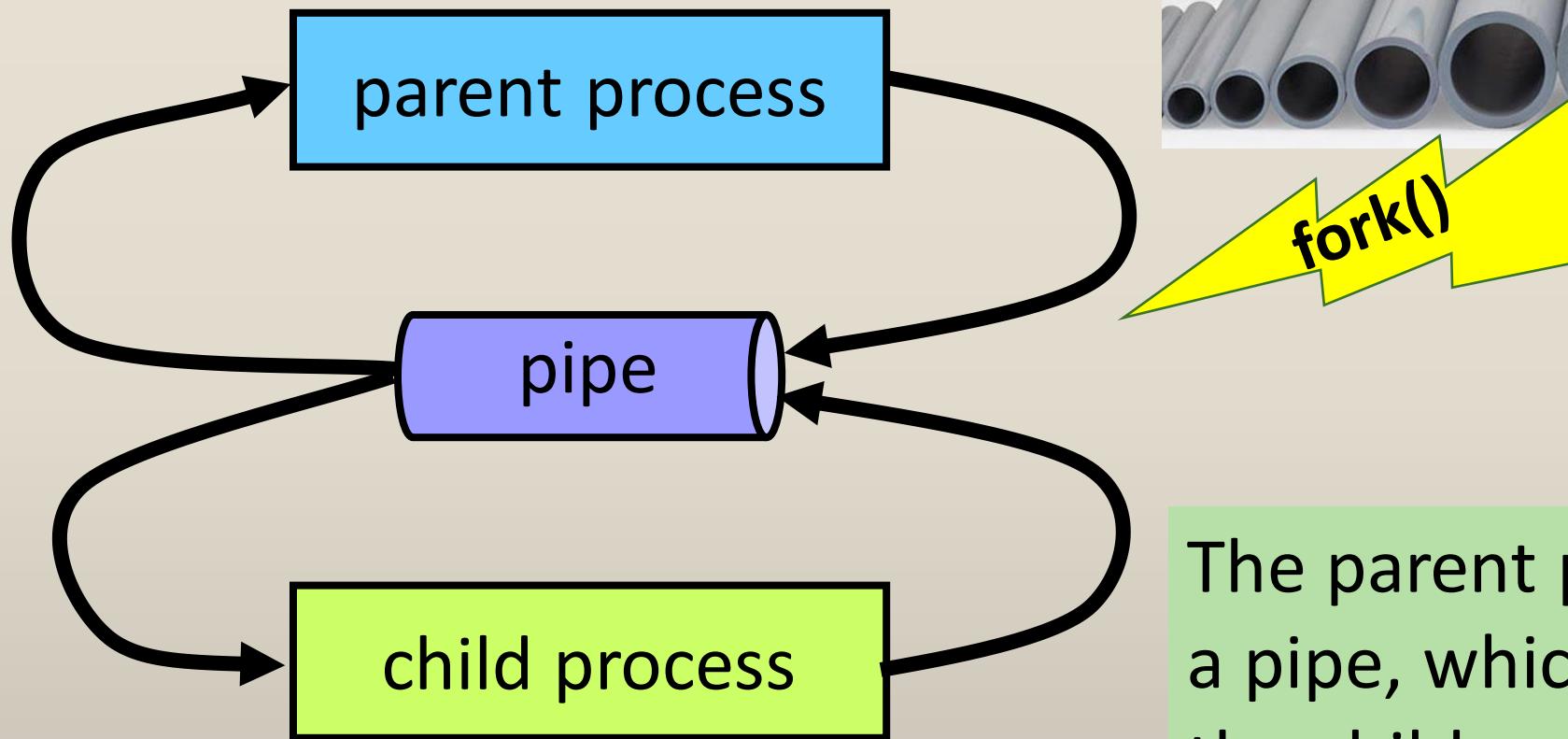




How Do We *Hook* Together the Standard Out from the Parent to Standard In of the Child?

Bring on the pipe

1. The parent process creates a pipe, which is inherited by the child process on the call to `fork()`.
2. The parent process **resets** its `stdout` to one side of the pipe.
3. The child process **resets** its `stdin` to the other side of the pipe.
4. Parent and child processes each close **both** sides of the pipe.



The parent process creates a pipe, which is inherited by the child process on the call to `fork()`.



stdin of the parent process is connected to the keyboard.

parent process

stdout of the parent process is redirected to the pipe, using `dup2()`.

stdin of the child process is redirected to the pipe, using `dup2()`.

pipe

child process

The parent process **resets** its stdout to one side of the pipe.
The child process **resets** its stdin to the other side of the pipe.
Then, `exec()` happens.

stdout of the child process remains connected to the terminal.

The dup2 () Call



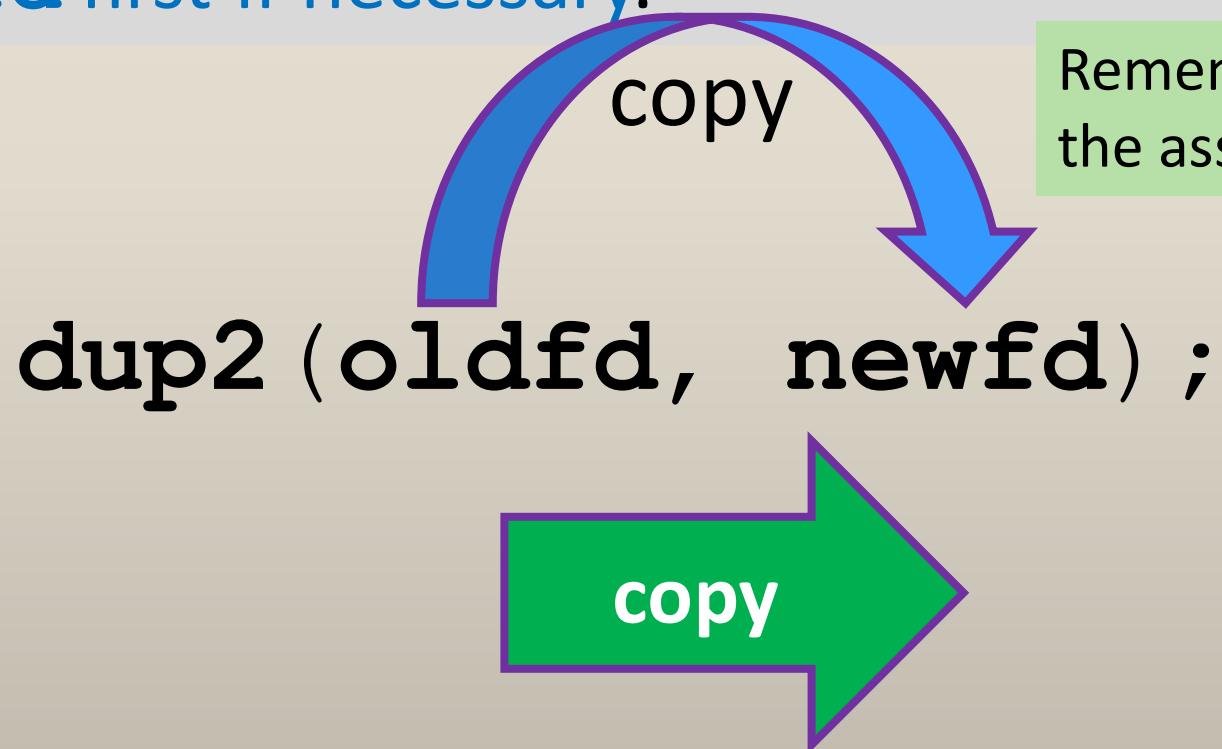
```
int dup2(int oldfd, int newfd);
```

dup2 () **makes newfd be a copy of oldfd**, closing **newfd** first if necessary:

- If **oldfd** is not a valid file descriptor, then the call fails, and **newfd** is not closed.
- If **oldfd** is a valid file descriptor, and **newfd** has the same value as **oldfd**, then **dup2 ()** does nothing, and returns **newfd**.

The dup2 () Call

Makes **newfd** be the copy of **oldfd**, closing **newfd** first if necessary.



Remember the direction
the assignment goes.



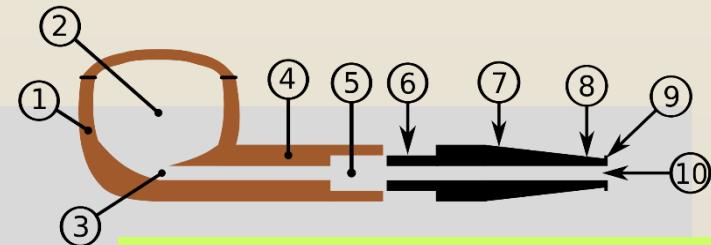
```
int main(void) {
    int filedes[2];
    pipe(filedes);
    pid_t pid = fork();
    if (0 == pid) { /* The child process */
        dup2(filedes[STDIN_FILENO], STDIN_FILENO);
        close(filedes[STDIN_FILENO]);
        close(filedes[STDOUT_FILENO]);
        int status = execvp( ... ... ... );
    }
}
```

The parent creates the pipe to be used for IPC.

```
else { /* Parent process */
    dup2(filedes[STDOUT_FILENO], STDOUT_FILENO);
    close(filedes[STDIN_FILENO]);
    close(filedes[STDOUT_FILENO]);
}
```

The call to `fork()`.

```
/* Parent process continues */
}
```



The call to `dup2()` copies the input side of the pipe to `stdin` for the child process.

Since the fd from the pipe has been duplicated onto `stdin`, you can close **both** sides of the pipe.

Then, `exec()` happens.

The `stdout` in the parent is copied from the pipe.

Since the fd from the pipe has been duplicated onto `stdout`, you can close **both** sides of the pipe.



Pipes and FIFOs



- **Pipes are the oldest** method of IPC on the UNIX system, having appeared in Third Edition UNIX in the early 1970s.
 - **Pipes provide a solution to the common requirement, create two processes running different commands, allow the output produced by one process to be used as the input to the other process (aka a pipeline).**
- **FIFOs are also called *named pipes*.**
- Both are **byte-streams**. There are **no message boundaries** between data written to a pipe or FIFO. Bytes go in and bytes come out.

Pipes

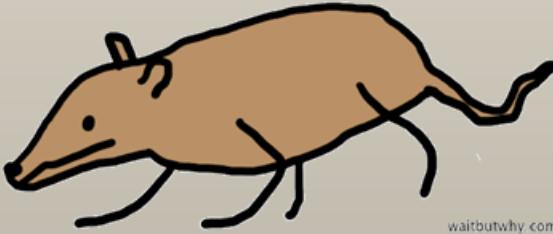
- Pipes are used between **related** processes. Sometimes called **anonymous pipes** or **unnamed pipes**.
- Processes using pipes to communicate will have a **common ancestor process**.
- Pipes are created by a parent process then child processes are created (using **fork()**) to make use of the pipes.



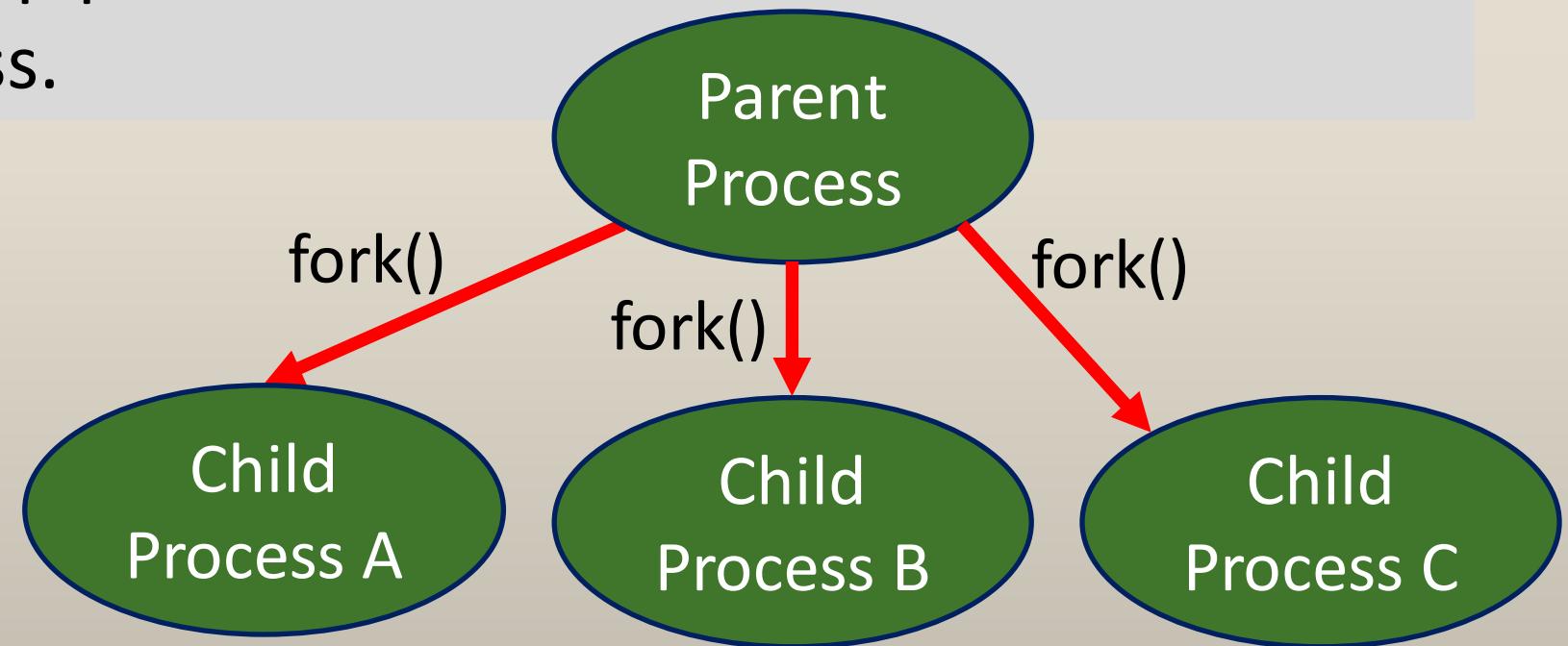
Relations Only

- Pipes are **used between related** processes.
Sometimes called anonymous or unnamed pipes.
- Processes using pipes to communicate will have a **common ancestor** process.

– 160 Million Years Ago –
Your Great^{55,000,000} Grandfather



waitbutwhy.com



Pipes

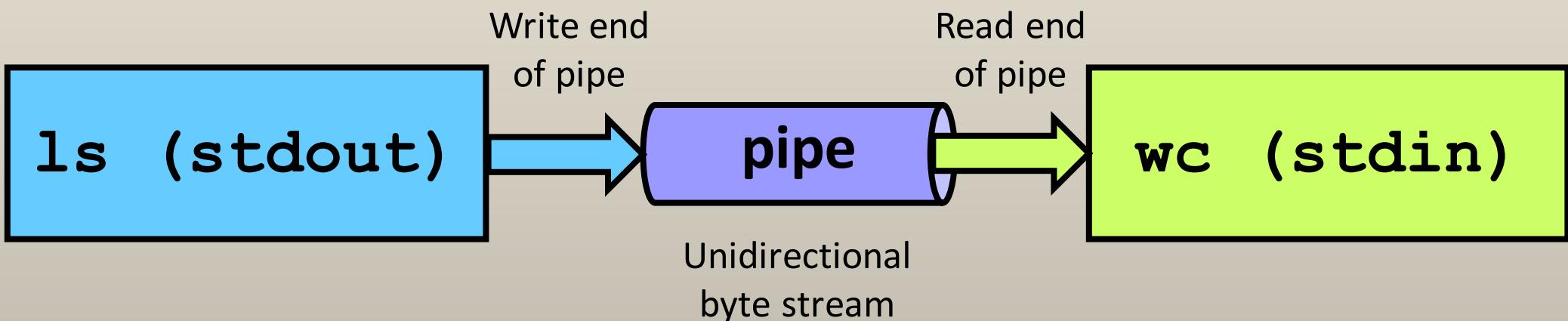
You've used pipes on the command line between processes.

The shell connects the processes via a pipe.

```
$ ls | wc -l
```



In order to execute the above command, **the shell creates two processes**, executing `ls` and `wc`, respectively.



A Pipe is a Byte Stream

BYTE



- A pipe is a byte stream, **there is no concept of messages or message boundaries** when using a pipe.

The process reading from a pipe can read chunks of data of any size, regardless of the size of chunks written by the writing process.
- **Data passes through the pipe sequentially** – bytes are read from a pipe in exactly the order they were written.

It is not possible to randomly access the data in a pipe using `lseek()`.

Reading from a Pipe

- Attempts to **read from a pipe that is currently empty will block** until at least one byte has been written to the pipe.
- **If the write end of a pipe is closed, then a process reading from the pipe will see end-of-file** (a `read()` will return 0) once it has read all remaining data in the pipe.

I use this as a test question.



Pipes are Unidirectional

- **Data can travel only in one direction** through a pipe.
- One end of the pipe is used for writing, and the other end is used for reading.
- You can have multiple readers/writers, but it is difficult.
- **Pipes are anonymous** (there is no entry in the file system for a pipe).



Writes of up to PIPE_BUF Bytes are Guaranteed to be Atomic

- If multiple processes are writing to a single pipe, then it is guaranteed that their data will not be comingled if they write no more than PIPE_BUF bytes at a time.



Pipes have a Limited Buffer Capacity

- **Pipes are kernel structures.** A pipe is simply a buffer maintained in kernel memory.
- **The buffer has a maximum capacity.**
- **Once a pipe is full, further writes to the pipe **block**** until the reader removes some data from the pipe.

.limited

This is not about how much total data can move through the pipe, but about how much data can be **held** in the pipe at any one time.

Creating Pipes



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

```
int pipe(int filedes[2]);
```

Returns 0 on success, or -1 on error.

Notice the array of size 2.

A successful call to `pipe()` results **in two open file descriptors** in the array `filedes`:

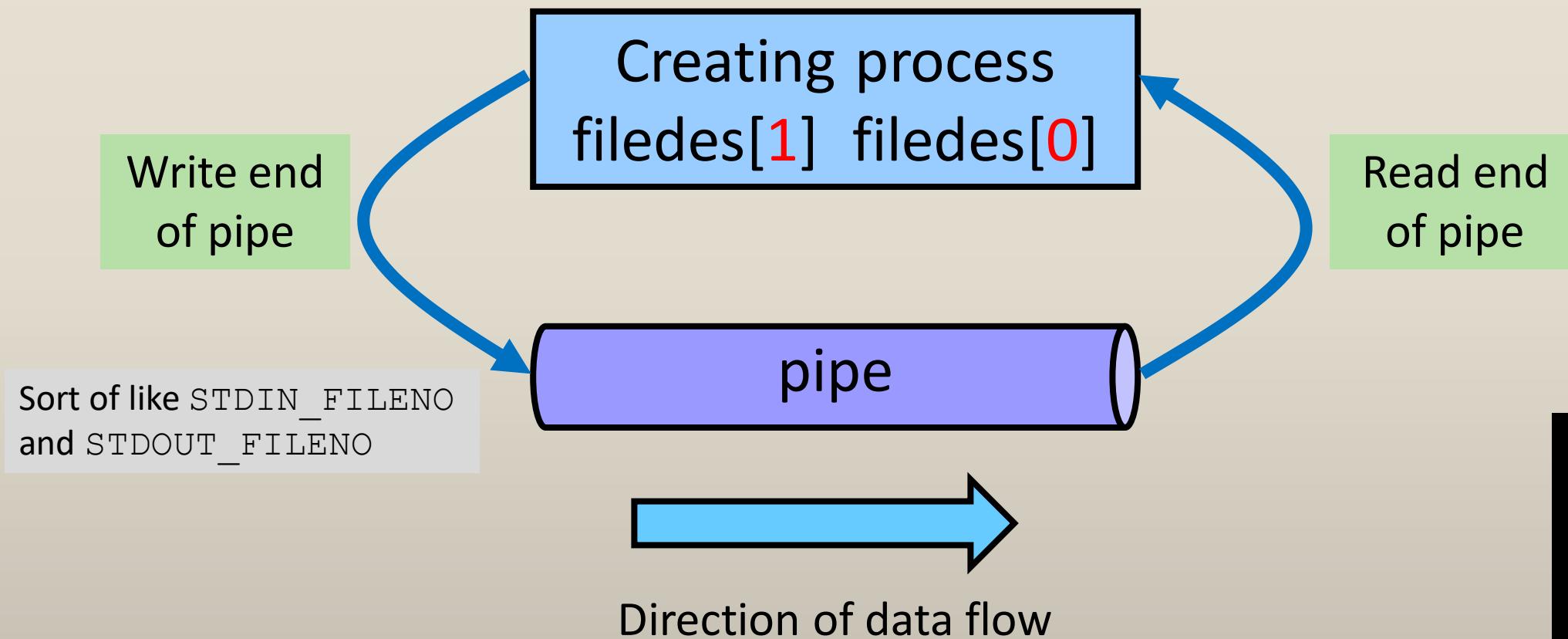
- one for the read end of the pipe (`filedes[0]`) and
- one for the write end (`filedes[1]`).

Using Pipes



- We use the `read()` and `write()` system calls to perform I/O on the pipe.
- Once data are written to the write end of a pipe, the data are immediately available from the read end.
- A `read()` from a pipe obtains the lesser of the number of bytes requested and the number of bytes currently available in the pipe (but **blocks** if the pipe is empty).

File Descriptors after Creating a Pipe



Creating Pipes



- Created using *pipe()*:

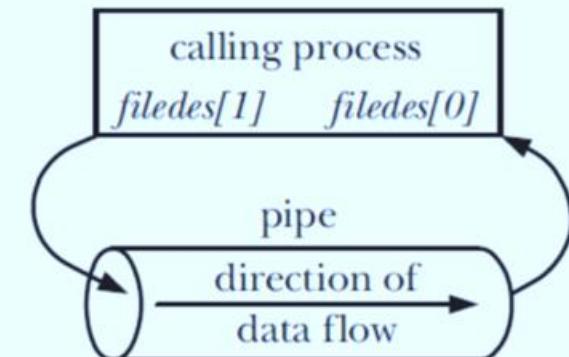
```
int filedes[2];  
pipe(filedes);
```

...

```
write(filedes[1], buf, count);  
read(filedes[0], buf, count);
```

The `pipe()` call creates the pipes. The pipes are file descriptors, just like `open()` returns.

You use `write()` and `read()` on a pipe to send or receive data.



An example of the **Universality of I/O**, sending data to another process using a pipe uses the same calls as writing to a file.

The parent process creates the pipe. A single pipe is represented as 2 file descriptors in the array `filedes`.

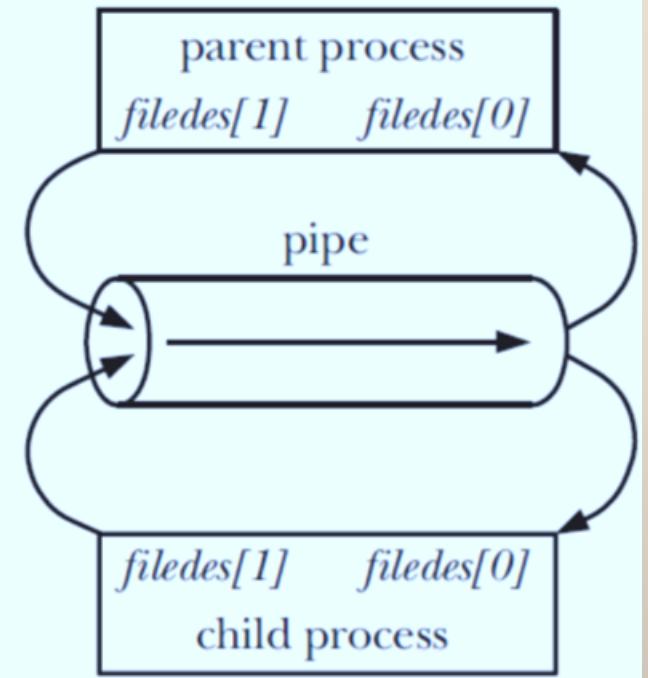
```
int filedes[2];  
pipe(filedes);  
child_pid = fork();
```

***fork()* duplicates parent's file descriptors**

The parent/child relationship means both get the newly created pipe.

Sharing a Pipe

Sharing is caring



Sharing a Pipe

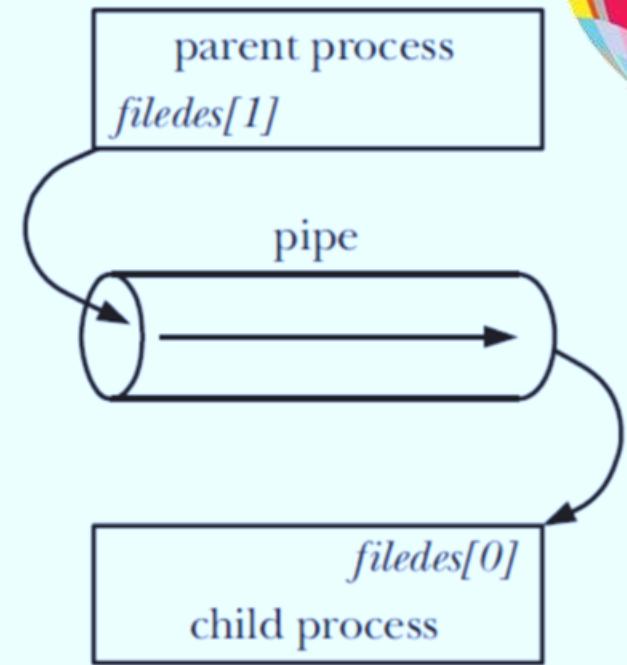
Sharing the Love



```
int filedes[2];
pipe(filedes);

child_pid = fork();
if (child_pid == 0) {
    close(filedes[1]);
    /* Child now reads */
} else {
    close(filedes[0]);
    /* Parent now writes */
}
```

Close the unused side of the pipe in each process.



In this image, the parent process will write data into the pipe for the child process to read.

Close Unused Sides of the Pipe



- Parent and child processes **must close** the unused file descriptors from a pipe.
 - This is necessary for the correct use of pipes.
 - `close()` write end
 - `read()` returns 0 (EOF)
 - `close()` read end
 - `write()` fails with EPIPE error and SIGPIPE signal



I/O on Pipes



- `read()` **blocks if pipe is empty**
- `write()` **blocks if pipe is full**
- Writes $\leq \text{PIPE_BUF}$ guaranteed to be atomic
 - Multiple writers $> \text{PIPE_BUF}$ may be interleaved
 - POSIX: `PIPE_BUF` at least 512 Bytes
 - Linux: `PIPE_BUF` is 4096 Bytes
- Can use `dup2()` to connect filters via a pipe.
- You don't have to open pipes, only close unused side.

Details to soon follow.

Using popen ()



- A frequent use for pipes is to execute a shell command and either read its output or send it some input.
- The `popen()` and `pclose()` functions are provided to simplify this task.

A call to `popen()` returns a file stream.

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

```
FILE *popen(const char *command, const char *mode);
```

```
int pclose(FILE *stream);
```

You can either read from **OR** write to the file stream returned from `popen()`

Returns file stream, or **NULL** on error

Returns termination status of child process, or **-1** on error

popen ()

- The popen () function creates a pipe, and then forks a child process that **execs a shell**, which in turn creates a child process to execute the string given in command.
- The mode argument is a string that determines whether the calling process will read from the pipe (mode is “r”) or write to it (mode is “w”).
- Since pipes are **unidirectional**, **two-way communication with the executed command is not possible**.



popen ()

While there are several similarities between `system()` and `popen()` plus `pclose()`, there are also some differences.

- The return value from `system()` is an `int`, indicating either success or failure.
- The `popen()` command **returns a file stream** that can either be read from or written to (using any of the stream functions, `fprintf()`, `fputs()`, `fscanf()`, or `fgets()`).
- There are some differences in how signals are handled between `system()` and `popen()`.

Don't forget the call
to `pclose()`



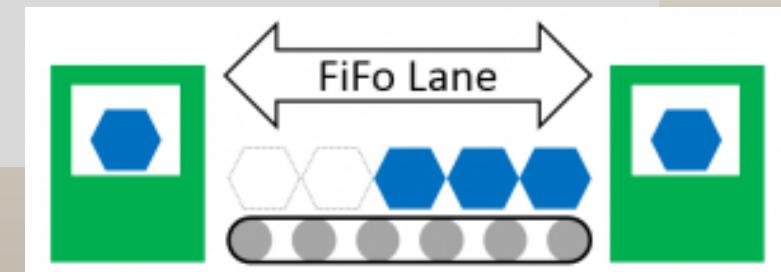
FIFOs aka *Named Pipes*

- Anonymous pipes can **only** be used between **related processes**.
- A FIFO is a pipe with a name in the file system.
- Creation:
 - **`mkfifo(path, permissions);`**
 - There is also a command to create a fifo.
- Unlike pipes, you **do have to open FIFOs before use.**
- Any process can open and use a FIFO (based on permissions).



FIFOs aka *Named Pipes*

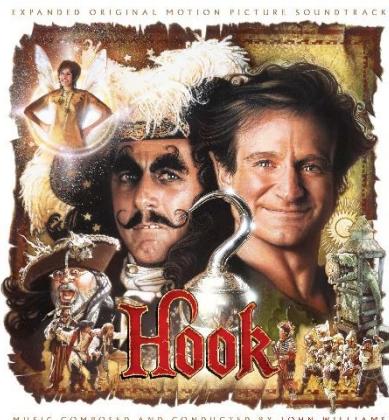
- I/O is the same as for pipes (read and write).
- A FIFO has a write end and a read end, and data are read from the pipe in the same order as it is written.
- FIFOs are kernel data structures, exactly like pipe.
- Unlike pipes, **you must explicitly delete a fifo from the file system when you are done with it.**
 - A **fifo** has file-system persistence.



Opening a FIFO

- `open(path, O_RDONLY);`
 - Open read end of a FIFO
- `open(path, O_WRONLY);`
 - Open the write side of a FIFO
- **Calls to `open()` will block until the other end of the FIFO is opened.**
- Opens are synchronized.
- You can also
 - `open(path, O_RDONLY | O_NONBLOCK);`

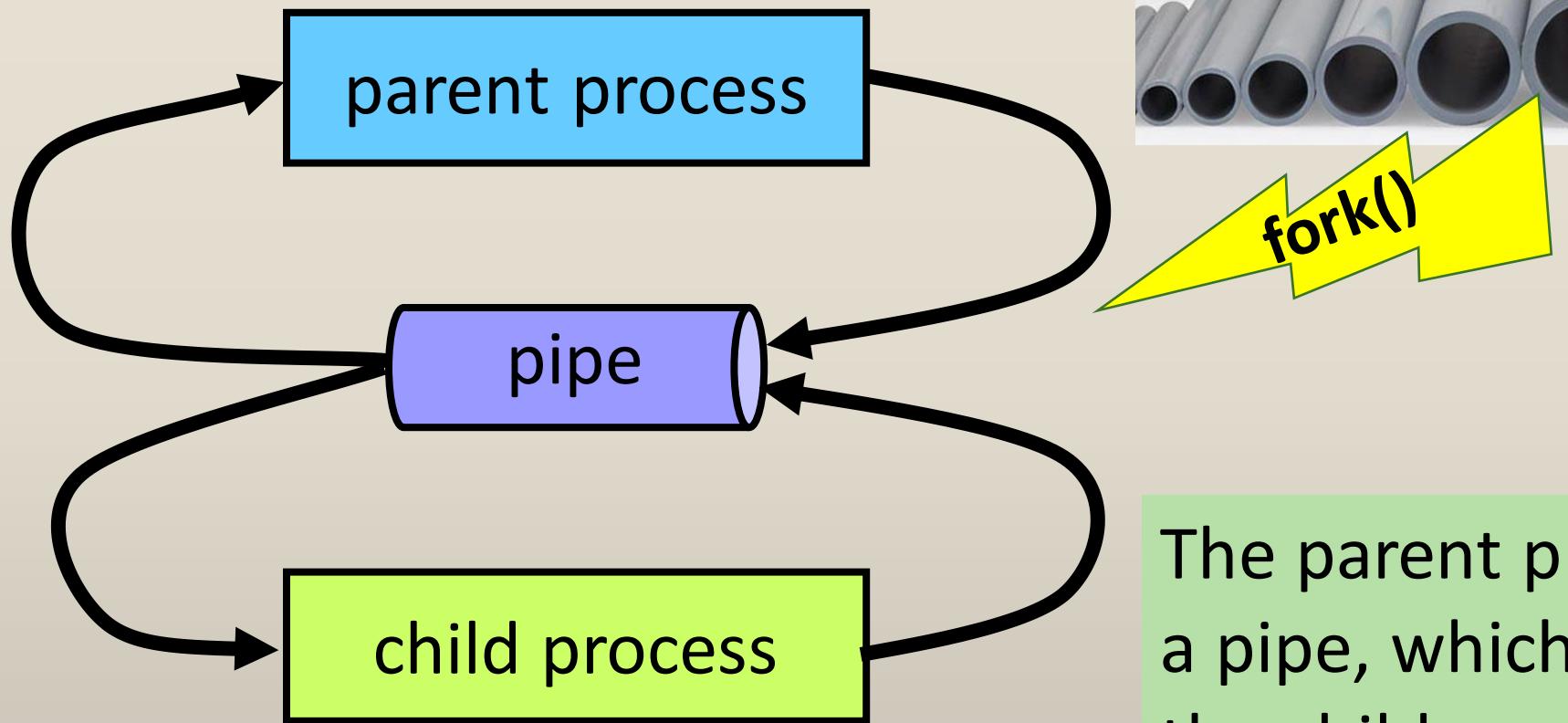




How Do We *Hook* Together the Standard Out from the Parent to Standard In of the Child?

Bring on the pipe

1. The parent process creates a pipe, which is inherited by the child process on the call to `fork()`.
2. The parent process **resets** its `stdout` to one side of the pipe.
3. The child process **resets** its `stdin` to the other side of the pipe.
4. Parent and child processes each close **both** sides of the pipe.



The parent process creates a pipe, which is inherited by the child process on the call to `fork()`.



stdin of the parent process is connected to the keyboard.

parent process

stdout of the parent process is redirected to the pipe, using `dup2()`.

stdin of the child process is redirected to the pipe, using `dup2()`.

pipe

child process

The parent process **resets** its stdout to one side of the pipe.
The child process **resets** its stdin to the other side of the pipe.
Then, `exec()` happens.

stdout of the child process remains connected to the terminal.

The dup2 () Call



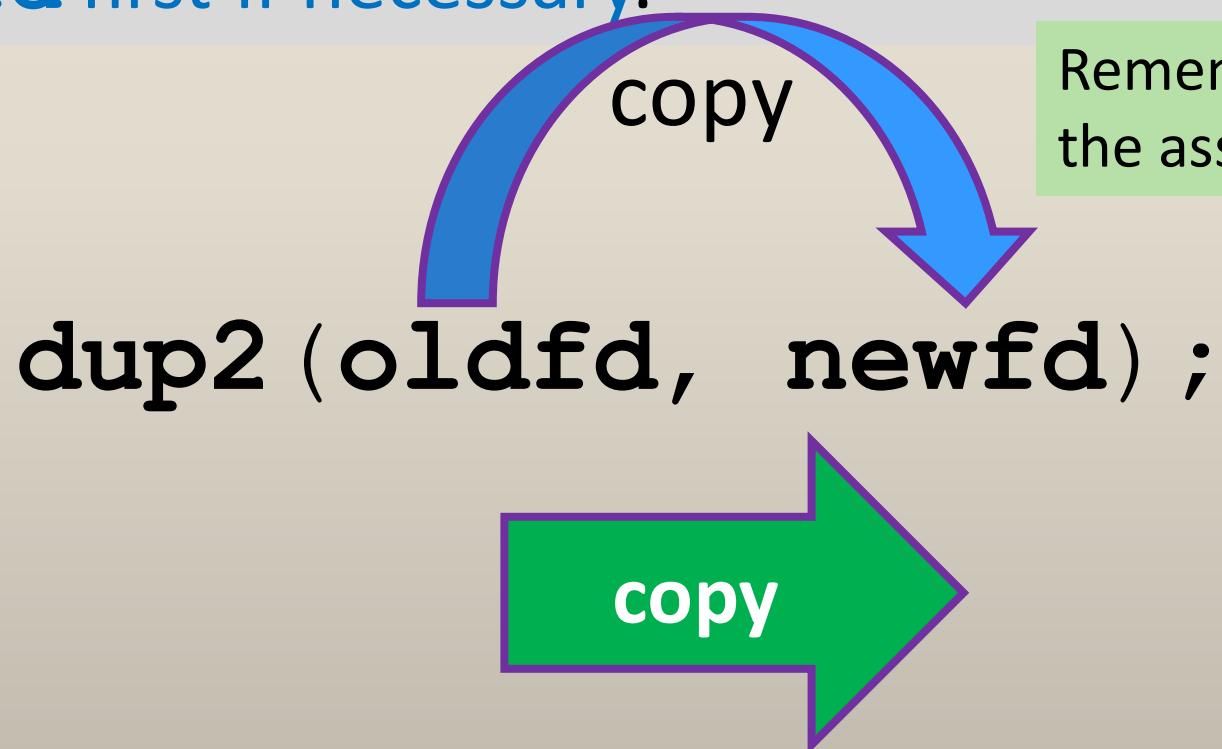
```
int dup2(int oldfd, int newfd);
```

dup2 () makes newfd be a copy of oldfd, closing newfd first if necessary:

- If `oldfd` is not a valid file descriptor, then the call fails, and `newfd` is not closed.
- If `oldfd` is a valid file descriptor, and `newfd` has the same value as `oldfd`, then `dup2 ()` does nothing, and returns `newfd`.

The dup2 () Call

Makes **newfd** be the copy of **oldfd**, closing **newfd** first if necessary.



Remember the direction
the assignment goes.



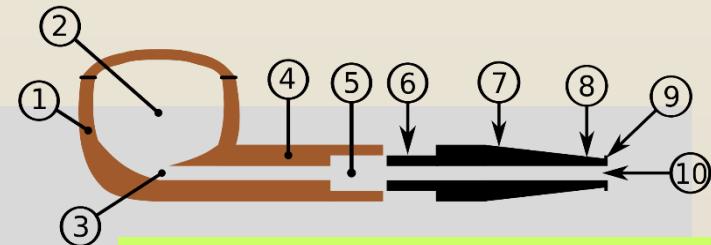
```
int main(void) {
    int filedes[2];
    pipe(filedes);
    pid_t pid = fork();
    if (0 == pid) { /* The child process */
        dup2(filedes[STDIN_FILENO], STDIN_FILENO);
        close(filedes[STDIN_FILENO]);
        close(filedes[STDOUT_FILENO]);
        int status = execvp( ... ... ... );
    }
}
```

The parent creates the pipe to be used for IPC.

```
else { /* Parent process */
    dup2(filedes[STDOUT_FILENO], STDOUT_FILENO);
    close(filedes[STDIN_FILENO]);
    close(filedes[STDOUT_FILENO]);
}
```

The call to `fork()`.

```
/* Parent process continues */
}
```



The call to `dup2()` copies the input side of the pipe to `stdin` for the child process.

Since the fd from the pipe has been duplicated onto `stdin`, you can close **both** sides of the pipe.

Then, `exec()` happens.

The `stdout` in the parent is copied from the pipe.

Since the fd from the pipe has been duplicated onto `stdout`, you can close **both** sides of the pipe.



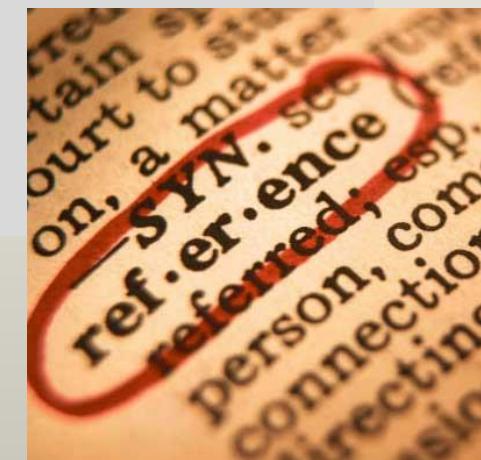
POSIX Message Queues

- On UNIX we have both System V and POSIX message queues.
- The book describes both.
- Today, we will cover **POSIX message queues**.

You may also want to view the POSIX Message Queue man page

```
man mq_overview
```

- POSIX message queues **are reference counted.**
 - A queue that is **marked for deletion is removed only after it is closed by all processes that are currently using it.**
- POSIX message queues provide a feature that **allows a process to be asynchronously notified when a message is available on a queue.**
 - I mention, but we don't use this feature.



- POSIX message queues allow an **efficient, priority-driven IPC** mechanism with **multiple readers and writers.**
- For an experienced Unix developer, this description may sound a lot like named pipes.
- There are some important differences between named pipes and message queues.



- **Message queues have structure.**
 - With a named pipe, a writer is just pushing bits.
 - For a reader, there's no distinction between different calls to `write()` from different writers. It is up to the programmer to ensure the correct number of bits are put into or taken out of the pipe.
- **With named pipes, variable-sized messages are a ... challenge.**



- POSIX Message queues are priority-driven.
 - Whenever a writer sends a message to a queue, **a priority is specified for that message.**
 - The queue will remain sorted such that the **oldest message of the highest priority will always be at the front.**
 - A message with **priority 0 is the lowest priority.**
- The programmer has control over the size of a message queue.
 - When a message queue is created, the programmer may set the number of messages that can be in the queue and the size of each message.



- A process can determine the status of a message queue.
 - One major drawback of pipes is that their state is unknown.
 - With POSIX message queues, a process can determine how many messages are in the queue, the various sizes and flags that have been set for the queue, and the number of processes that are blocking to either send or receive.

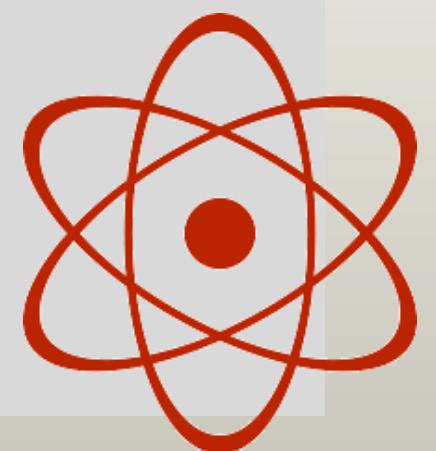
- **POSIX Message queues use special identifiers, rather than basic file descriptors.**
 - Therefore, message queues **require special functions for sending and receiving data**, rather than the standard `read()` and `write()` file interface functions.



SPECIAL

The word "SPECIAL" is written in a bold, sans-serif font. The letters are filled with a gradient color transitioning from yellow at the top to red at the bottom, giving them a flaming appearance. The letters are slightly irregular and overlap each other, creating a dynamic and intense visual effect.

- When a message is added to a queue, it is either added in its entirety or not at all.
 - There are no partial message writes into a queue.
- When a message is read from a queue, it is either consumed in its entirety, or not at all (and left on the queue).
 - There are no partial messages or partial reads.
 - Data from messages cannot be intermingled.
- **Reads and writes to a message queue are atomic.**



It is important to understand the **difference between messages and byte streams.**

1. Assume that one process has made 100 calls to a function to transmit a single byte at a time.
2. If pipes or FIFOs are used, all of this data can be retrieved with a single call to `read()` that requests 100 bytes.
3. In the case of message queues, each byte is a distinct message that must be retrieved individually.
4. There is no short-cut to retrieve all of the messages at once.

Functions for setting up and removing POSIX message queues.

```
#include <mqueue.h>
```

```
mqd_t mq_open (const char *name, int oflag, ...  
                  /* mode_t mode, struct mq_attr *attr */);
```

Open (and possibly create) a POSIX message queue

```
int mq_close (mqd_t mqdes);
```

Close a message queue

```
int mq_unlink (const char *name);
```

Initiate deletion of a message queue



Functions for sending and receiving on POSIX message queues.

```
int mq_send (mqd_t mqdes, const char *msg_ptr  
, size_t msg_len, unsigned int msg_prio);
```

Send the message pointed to by `msg_ptr` with priority `msg_prio`

```
ssize_t mq_receive (mqd_t mqdes, char *msg_ptr  
, size_t msg_len, unsigned int *msg_prio);
```

Receive a message into a buffer pointed to by `msg_ptr` and get its priority `msg_prio`.

Removes the oldest message with the highest priority from the message queue.



RECEIVED

```
queue = mq_open(argv[1], O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR, NULL);
if (queue == (mqd_t) -1) {
    perror("mq_open");
    return 1;
}

if (mq_send(queue, argv[2], strlen(argv[2]), 0) == -1) {
    perror("mq_send");
    mq_close(queue);
    return 1;
}
```

```
queue = mq_open(argv[1], O_RDONLY | O_CREAT, S_IRUSR | S_IWUSR, NULL);
```

```
msg_ptr = calloc(1, attrs.mq_msgsize);
if (msg_ptr == NULL) {
    perror("calloc for msg_ptr");
    mq_close(queue);
    return 1;
}
```

```
recv = mq_receive(queue, msg_ptr, attrs.mq_msgsize, NULL);
if (recv == -1){
    perror("mq_receive");
    return 1;
}
```

Unlike pipes, **message queues can be used to send structure instances.**

Consider the following trivial structure declaration:

```
struct message {  
    int x;  
    char y;  
    long z;  
};
```

```
struct message msg;  
msg.x = 0;  
msg.y = 'q';  
msg.z = -1;
```

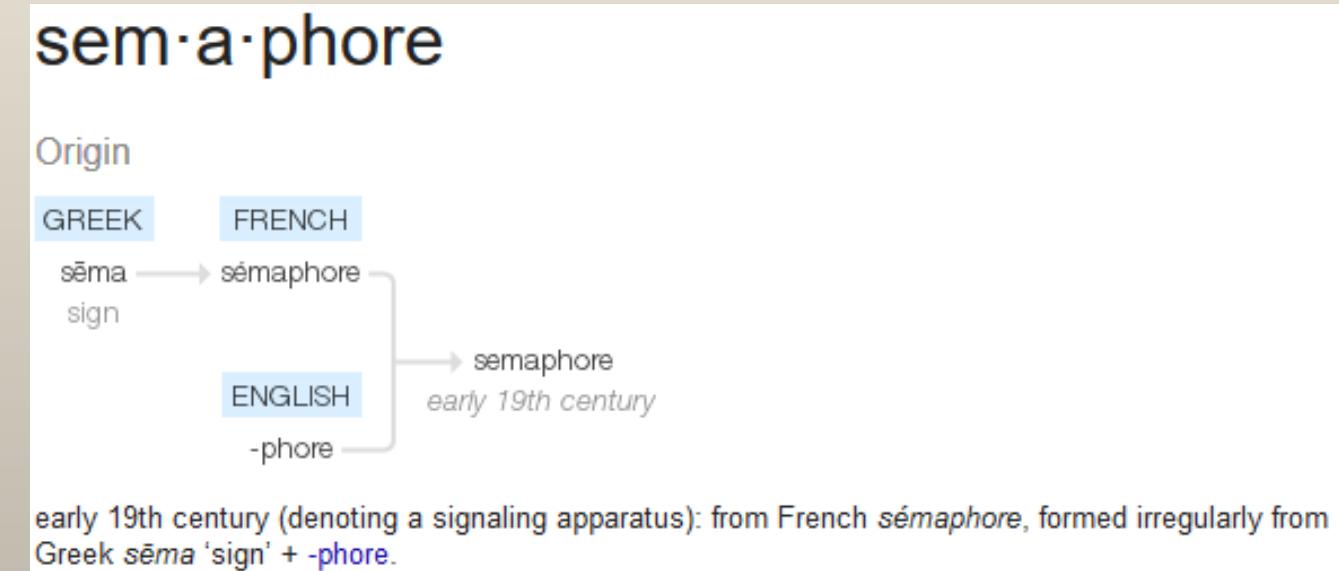
```
/* Sending a struct works identically to a char array */  
mq_send (mqd, (const char *)&msg, sizeof (struct message), 10);
```





POSIX Semaphores

- On UNIX we have both System V and POSIX semaphores.
- The book describes both.
- Today, we will cover POSIX semaphores.

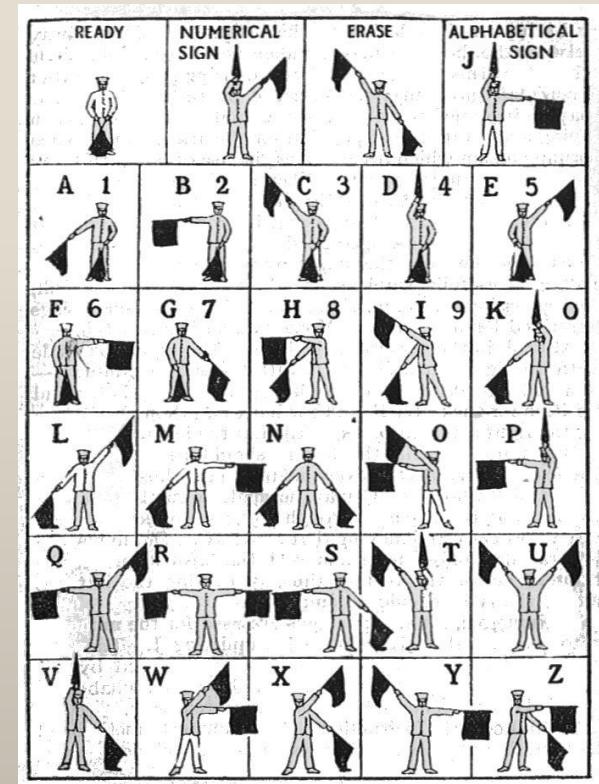


POSIX Semaphores

semaphore (*plural* semaphores)

1. Any visual signaling system with flags, lights, or mechanically moving arms.
2. A visual system for transmitting information by means of two flags that are held one in each hand, using an alphabetic and numeric code based on the position of the signaler's arms.
3. (**computing**) A bit, token, fragment of code, or some other mechanism which is used to restrict access to a shared function or device to a single process at a time, or to synchronize and coordinate events in different processes.

The word semaphore goes back to a Greek word, used for signaling.



Semaphores as Communication

A **semaphore telegraph**, **optical telegraph**, **shutter telegraph chain**, **Chappe telegraph**, or **Napoleonic semaphore** is a system of conveying information by means of visual signals, using towers with pivoting shutters, also known as blades or paddles.

Information is encoded by the position of the mechanical elements; it is read when the shutter is in a fixed position.

Semaphore lines were a precursor of the electrical telegraph.



Semaphores as Communication



Edsger Dijkstra

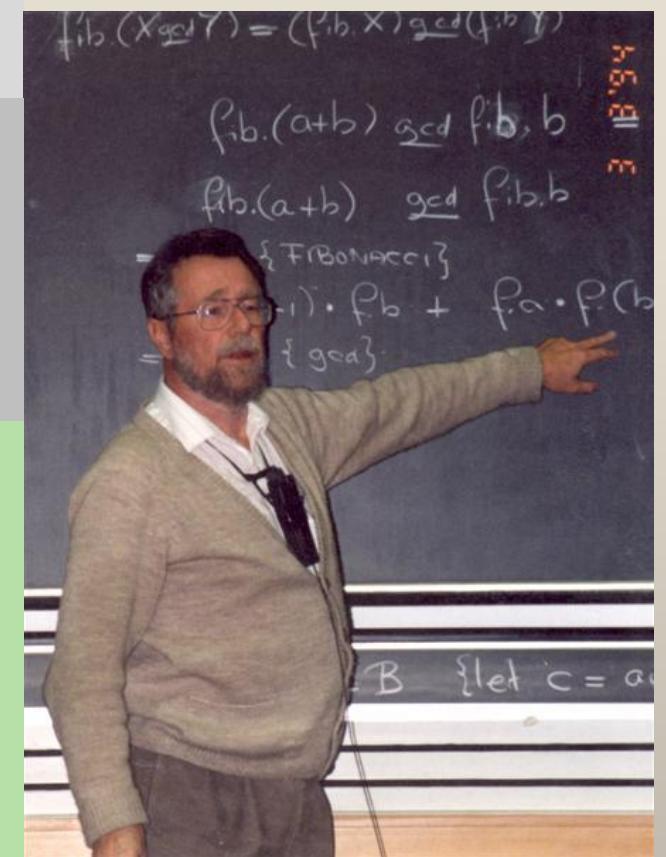
The (computing) semaphore concept was invented by Dutch computer scientist **Edsger Dijkstra** in 1962 or 1963.

Semaphores allow processes and threads to synchronize access to shared resources.

Sound familiar?

Dijkstra introduced a pair of synchronization primitives called P() and V().

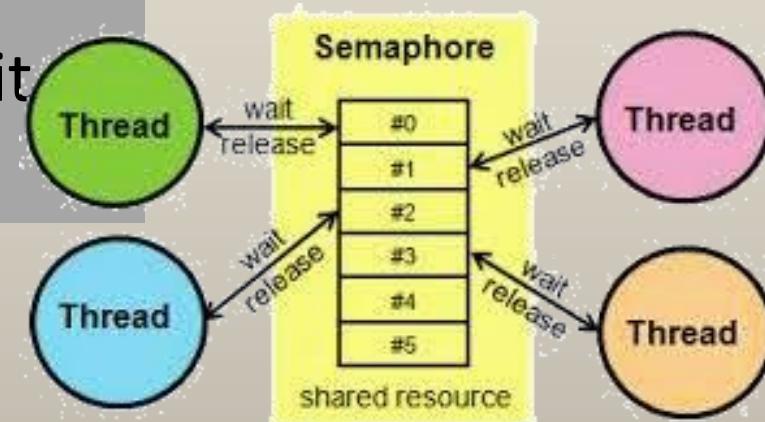
- P = Plantinga, V = Vegter.
- P = Probeer ('Try') and
- V = Verhoog ('Increment', 'Increase by one').



POSIX Semaphores

POSIX Semaphores come in 2 types:

- **Named Semaphores**: This type of semaphore **has a name on the file system**. By calling `sem_open()` with the same name, unrelated processes can access the same semaphore.
- **Unnamed Semaphores**: This type of semaphore **doesn't have a name in the file system**; instead, it resides at an agreed-upon location in memory.



Library Analogy



- Suppose a library has 10 identical quiet study rooms, **to be used by one student at a time**.
- To prevent disputes, **students must request a room from the front desk** if they wish to make use of a study room.
- **If no rooms are free**, students wait at the desk until someone relinquishes a room.
- When a student has finished using a room, the **student must return to the desk and indicate that one room has become free**.
- The front desk count-holder represents a ***counting semaphore***, the rooms are the ***resources***, and the students represent ***processes***.

POSIX Semaphores

- A POSIX semaphore is an **integer** whose value is **not permitted to fall below 0**.
- If a process **attempts to decrease the value of a semaphore below 0**, then, depending on the function used, the call either **blocks or fails with an error indicating that the operation was not currently possible**.
- Two operations can be performed on semaphores:
 1. **increment** the semaphore value by one
 2. **decrement** the semaphore value by one
- If the value of a semaphore is currently zero, then a **sem_wait()** operation will **block** until the value becomes greater than zero.





Semaphores

Some Observations

When used to control access to a pool of resources, a semaphore tracks only **how many resources are free**.

It does not keep track of which of the resources are free.

Some other mechanism (possibly involving more semaphores) may be required to select a particular free resource.

Semaphores

The success of using semaphores to control access requires applications **follow the process correctly**.

Fairness and safety are likely to be lost if even a single process acts incorrectly. This includes:

- Requesting a resource and forgetting to release it;
- Releasing a resource that was never requested;
- Holding a resource for a long time without needing it;
- Using a resource without requesting it first (or after releasing it).



Named Semaphores



To work with a named semaphore, we employ the following functions:

- The `sem_open()` function **opens or creates** a semaphore, initializes the semaphore if it is created by the call, and returns a handle for use in later calls.
- The `sem_post(sem)` and `sem_wait(sem)` functions respectively **increment** and **decrement** a semaphore's value.
- The `sem_getvalue()` function retrieves a semaphore's current value.
- The `sem_close()` function removes the calling process' association with a semaphore that it previously opened.
- The `sem_unlink()` function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.

Opening a Named Semaphore

```
#include <fcntl.h>          /* Defines O_* constants */  
#include <sys/stat.h>        /* Defines mode constants */  
#include <semaphore.h>
```

Link with -pthread.

```
sem_t *sem_open(const char *name, int oflag);
```

```
sem_t *sem_open(const char *name, int oflag,  
                mode_t mode, unsigned int value);
```

If `O_CREAT` is specified in `oflag`, then two additional arguments must be supplied. The `mode` argument specifies the permissions to be placed on the new semaphore, as for `open(2)`.



Opening a Named Semaphore

```
#include <fcntl.h>          /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>
#include <stdio.h>

int main(void)
{
    sem_t * my_semaphore = NULL;
    my_semaphore = sem_open ("/mysemaphore"
                           , O_CREAT | O_EXCL ←
                           , S_IRUSR | S_IWUSR, 10);
    return EXIT_SUCCESS;
}
```

Creates a semaphore called **sem.mysemaphore** in **/dev/shm**

If you want to make sure this **creates** the semaphore, add **O_EXCL** to the call.

The initial value of this semaphore is 10.

Closing and Removing a Named Semaphore

```
#include <semaphore.h>

int sem_close(sem_t *sem) ;

int sem_unlink(const char *name) ;
```

Removes the semaphore name and marks the semaphore to be destroyed, once all processes have stopped using it.



```
#define SEM_NAME "/mysemaphore"
int main(int argc, char *argv[])
{
    sem_t *my_semaphore = NULL;
```



```
    my_semaphore = sem_open(SEM_NAME
                           , O_CREAT, S_IRUSR | S_IWUSR, 10);
```

Open the semaphore.

Close the semaphore.

```
if (argc > 1) {
    sem_close(my_semaphore);
    sem_unlink(SEM_NAME);
}
```

When the last process is done with the semaphore,
remove it from the file system.

```
return EXIT_SUCCESS;
}
```

Semaphore Operations

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

```
int sem_wait(sem_t *sem);
```

Block until acquiring
the semaphore.

```
int sem_post(sem_t *sem);
```

Release a semaphore.

```
int sem_trywait(sem_t *sem);
```

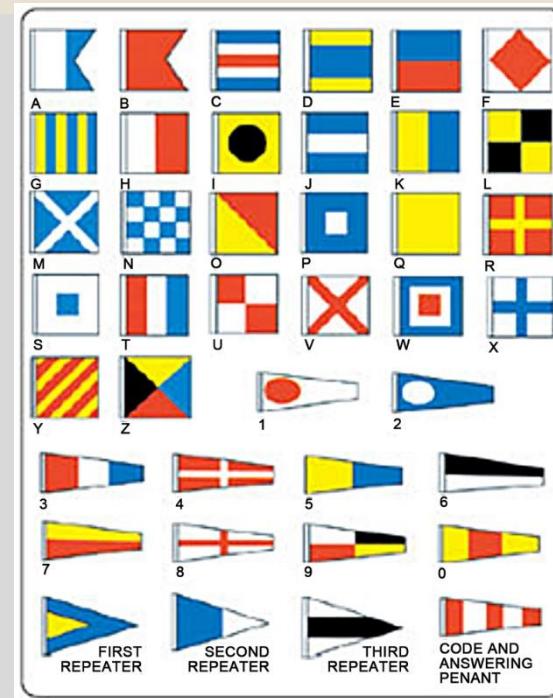
Try to acquire a semaphore.

```
int sem_timedwait(sem_t *sem
```

```
, const struct timespec *abs_timeout);
```

```
int sem_getvalue(sem_t * sem, int *sval);
```

Fetch the value without acquiring, but
it is of limited value.



Get a semaphore, but
only wait so long.

Unnamed Semaphores

- Unnamed semaphores (also known as *memory-based* semaphores) are variables of type `sem_t` that are **stored in memory allocated by the application.**
- The semaphore is made available to the processes or threads that use it by placing it in an area of memory that they **share**.



Unnamed Semaphore Ops

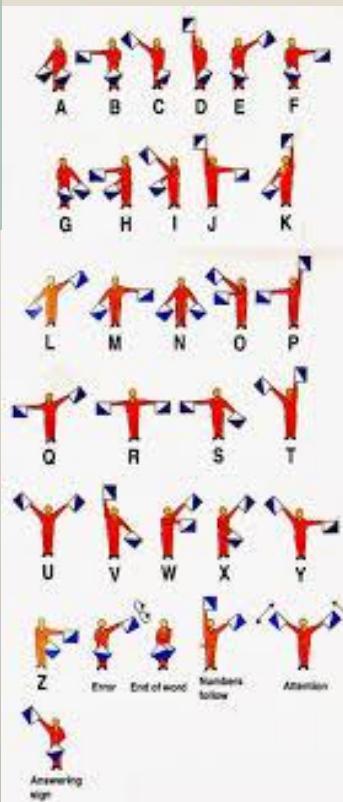
```
#include <semaphore.h>

int sem_init(sem_t *sem
             , int pshared
             , unsigned int value);

int sem_destroy(sem_t *sem );
```

These functions should not be used with named semaphores.

The `sem_init()` function initializes a semaphore and informs the system of whether the semaphore will be shared between processes or between the threads of a single process



Semaphores versus Mutexes

- Mutexes are generally preferable.
- Semaphores are async-signal-safe.
- Mutexes generally are faster than semaphores.

Calls to the semaphore functions require a kernel call.

- A mutex can be handled within user mode.

The **ownership** property of mutexes enforces good structuring of code.

- One thread can increment a semaphore that was decremented by another thread.



Semaphores versus Mutexes

- A mutex is similar to a binary semaphore, but not the same.
- The differences between them are in how they are used.
- A binary semaphore may be used like a mutex, a mutex is a more specific use-case, in that only the process that locked the mutex is allowed to unlock it.
- A **mutex is locking mechanism** used to **synchronize access to a resource**. Only one task can acquire the mutex. There is **ownership** associated with mutex, and only the owner can release the lock (mutex).
- **Semaphore is signaling mechanism**. For example, if you are listening to songs on your mobile device and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.





The Vulcan Mind Meld of IPC



POSIX Shared Memory

- On UNIX we have both System V and POSIX shared memory
- The book describes both.
- Today, we will talk about **POSIX** shared memory.
- If you are looking for code examples of Unix shared memory, most of the ones you'll find are for System V shared memory.



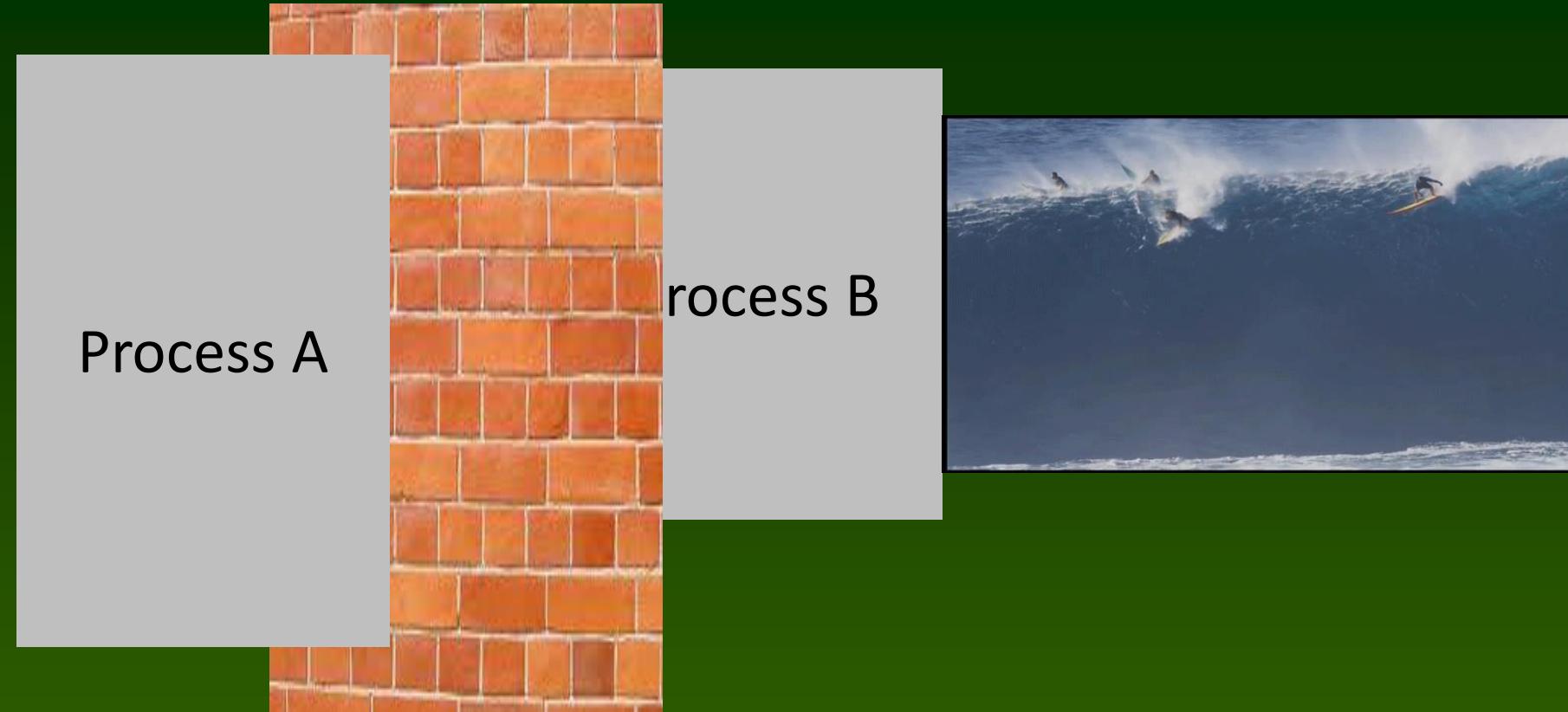
POSIX Shared Memory

Shared Memory is a form of IPC that allows processes to exchange large amounts of information very quickly.

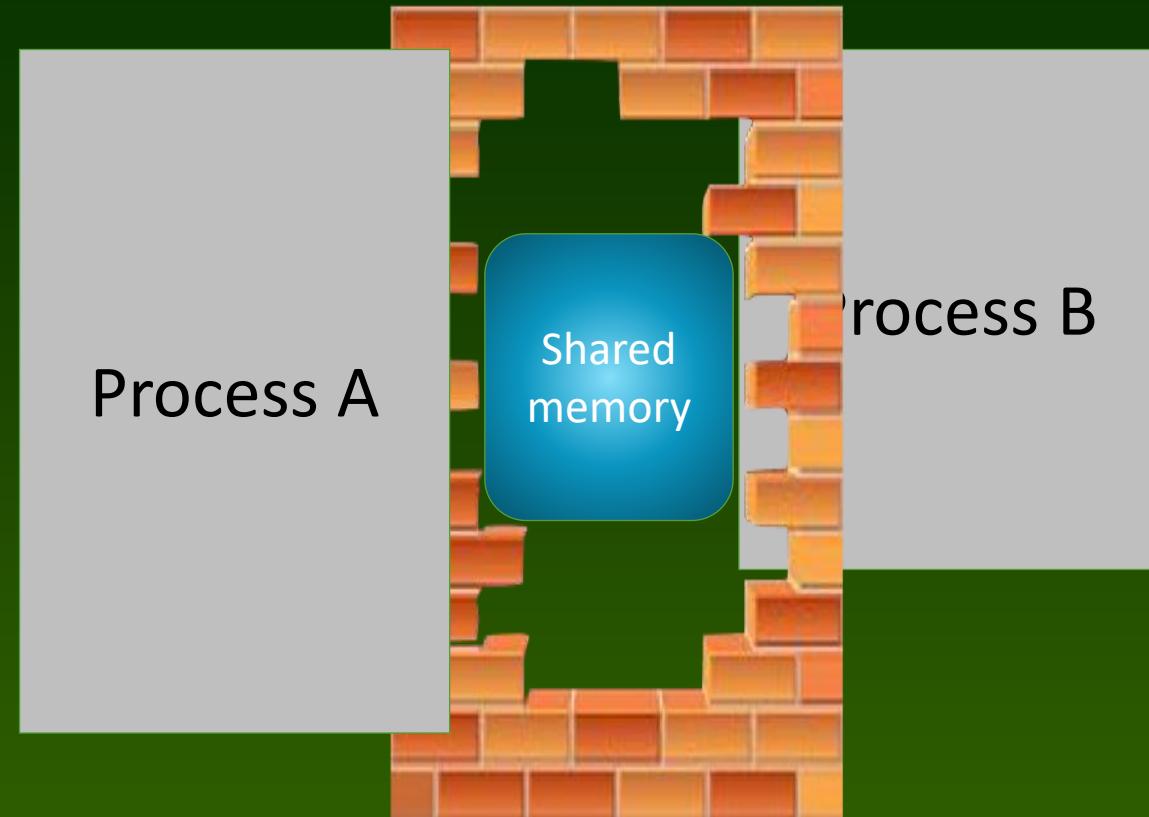
Once processes maps a **Shared Memory** segment into the process address space, there are no system calls necessary to share data between the attached processes.

Shared Memory does not provide any means of synchronization or mutual-exclusion for multiple or concurrent access.

- You need to build that into your application.
- A common method for providing synchronization of access to Shared Memory is using **semaphores** (i.e. – the previous lecture).



- There's **normally a brick wall** between the memory of different processes. Process A cannot access or modify the memory of Process B. The brick wall is enforced by the kernel.
- Normally, the brick wall is a good thing. It protects applications from accidental corruption from other applications.



Shared Memory allows you to **punch a hole in the brick wall** and allow processes to share regions of memory.
The processes inform the kernel (through system calls) that the processes are **cooperating** and that they want to share some memory resources.

Process A

Process B

- Two processes A and B are running on a system and have been coded to coordinate and share data using shared memory.
- The processes do not need to be related.



Create a shared
memory segment
called shm-demo.
Truncate the
segment to be
1024 bytes.

Process A

```
int shmfds;
shmfds = shm_open(
    "/shm-demo"
    , O_RDWR | O_CREAT
    , S_IRUSR | S_IWUSR
);
ftruncate(
    shmfds, 1024);
```

Process B

Shared memory



- Process A requests a segment of shared memory be created and opened, using the **shm_open()** call.
- Process A calls **ftruncate()** on the shared memory segment, setting its size to 1024 bytes.

Process A

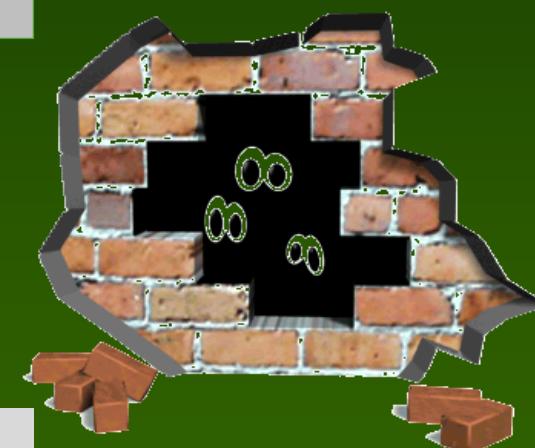
```
int shmfds;
void *shmaddr;
shmaddr = mmap(
    NULL, 1024
    , PROT_READ | PROT_WRITE
    , MAP_SHARED
    , shmfds, 0);
```

map

Shared memory

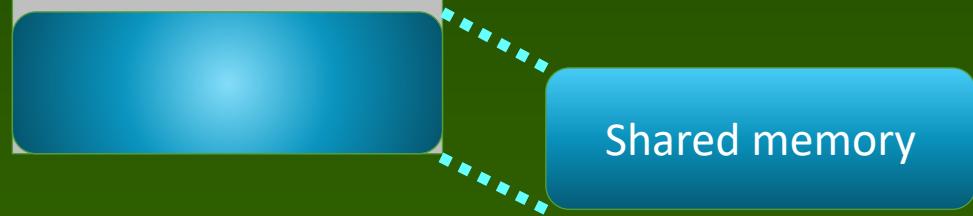
Map the shared
memory segment
into the memory
of the process.

Process B



- Process A maps the shared memory segment into its own memory space, using [mmap \(\)](#).
- The `shmaddr` pointer points to the beginning of the shared memory segment, **it can be cast** to another pointer type (such as a `struct`) for use in the code.

Process A



Shared memory

Process B

```
int shmfds;
shmfds = shm_open(
    "/shm-demo"
    , O_RDWR
    , S_IRUSR | S_IWUSR
);
```

Open a shared
memory segment
called shm-demo.
Don't truncate the
segment again.



- Process B now opens the existing shared memory segment, using `shm_open()`.
- Since Process A has already sized the segment, **Process B does not need to resize it (`ftruncate()`)**.

Process A



Shared memory

Process B

```
int shmfds;
void *shmaddr;
shmaddr = mmap(
    NULL, 1024
    , PROT_READ |
        PROT_WRITE
    , MAP_SHARED
    , shmfds, 0);
```

map

Map the shared memory segment into the memory of the process.

Process B maps the shared memory segment into its own memory space, using [mmap \(\)](#). The pointer `shmaddr` pointer points to the beginning of the shared memory segment, **it can be cast** to another pointer type (such as a struct) for use in the code.

Process A

Process B

The mind meld is active.



Shared memory

Each process can
read from and
write to the
shared memory
segment.

- Process A and Process B can now freely read from and write to the shared memory segment.
- The shared memory segment is treated the same as local memory in Process A and Process B.

Process A

```
int shmfds;
void *shmaddr;

munmap(shmaddr);
close(shmfds);
shm_unlink(
    "/shm-demo");
```

Unmap the shared memory segment from the process, close the file descriptor, and unlink the file from the file system.

Process B

```
int shmfds;
void *shmaddr;

munmap(shmaddr);
close(shmfds);
```

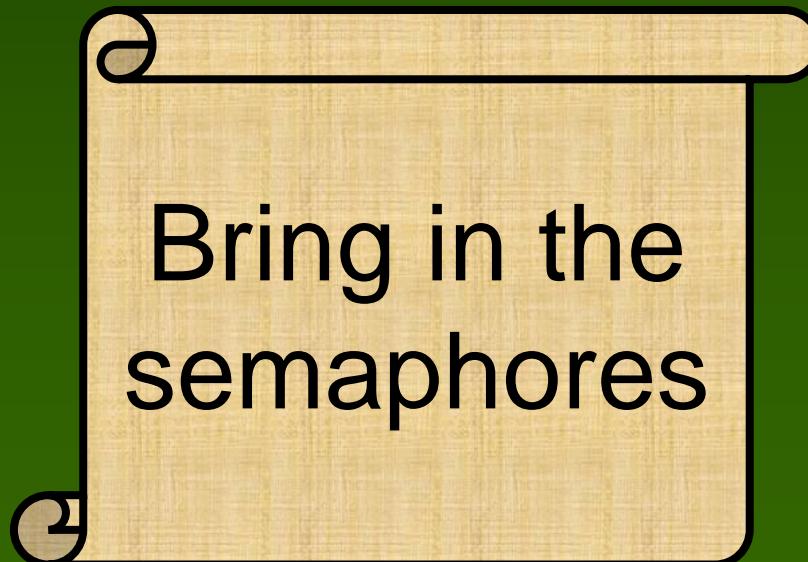
Unmap the shared memory segment from the process and close the file descriptor.



- Once Process A and Process B have finished the use of the shared memory segment, **each process should un-map** it and **one process should unlink** the shared memory name.
- A POSIX shared memory segment will exist until explicitly removed (with the `shm_unlink()` call) or the system is rebooted.

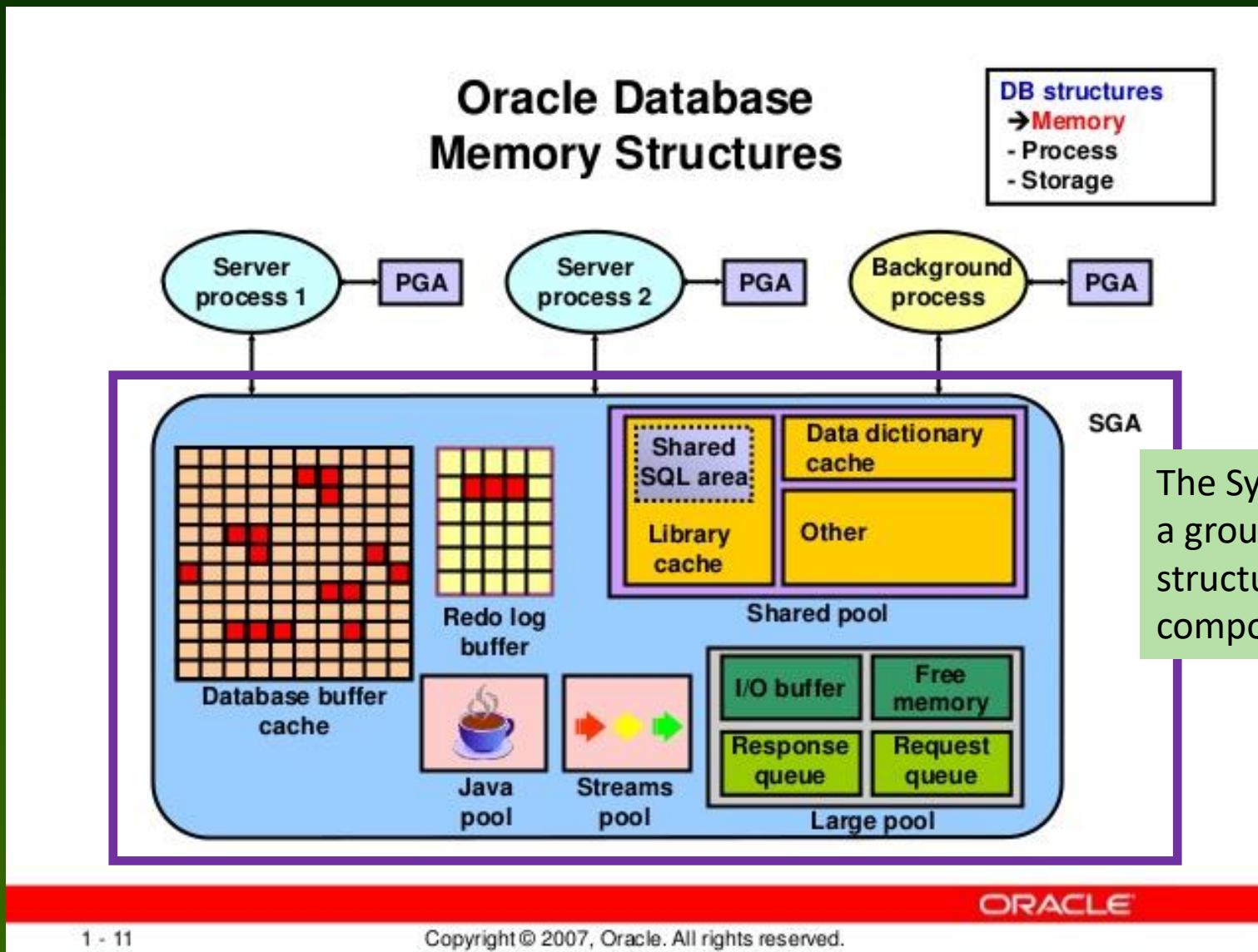
Using Shared Memory

Establishing shared memory between processes does not do anything about protecting that shared resource from corruption.

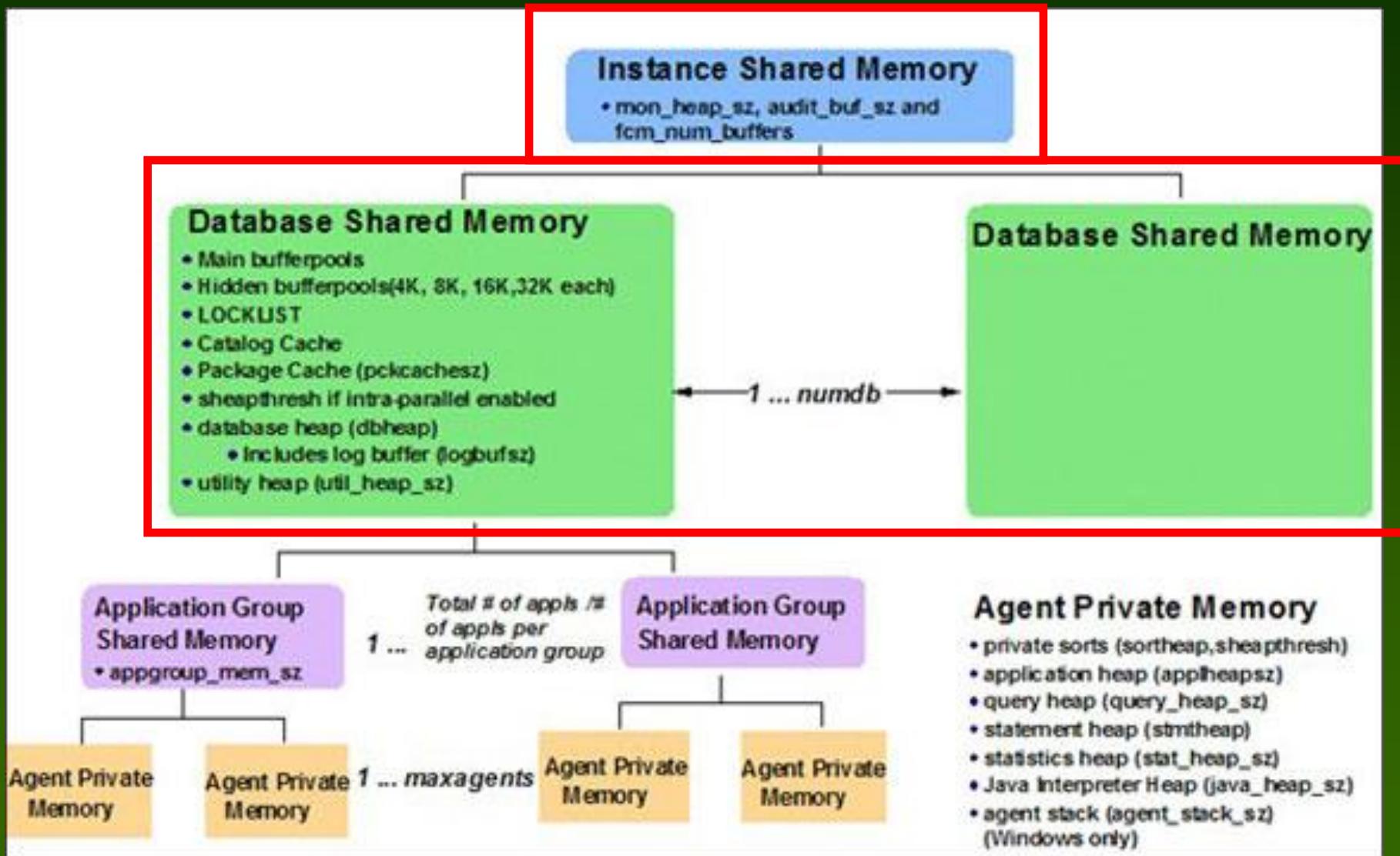


Bring in the
semaphores

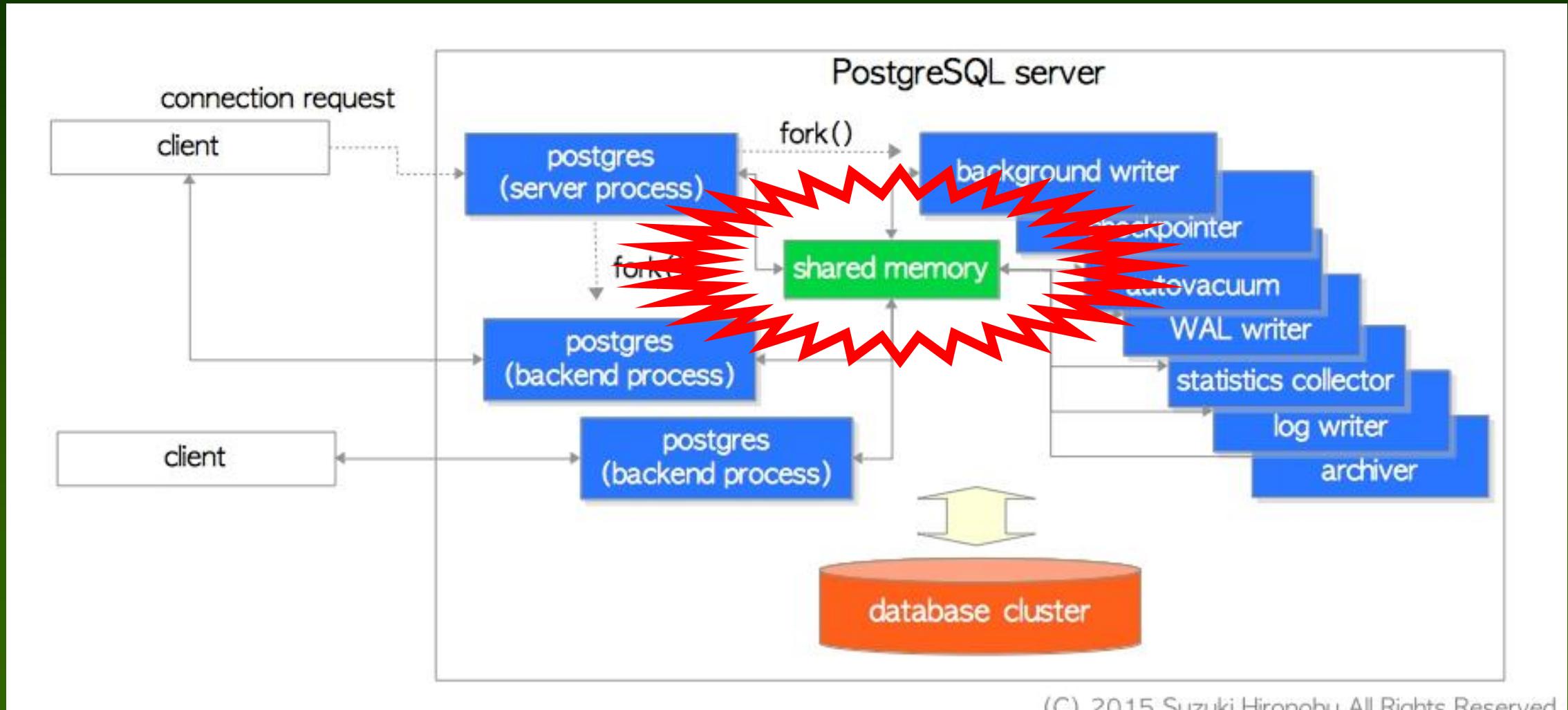
Oracle Shared Memory



DB2 Shared Memory

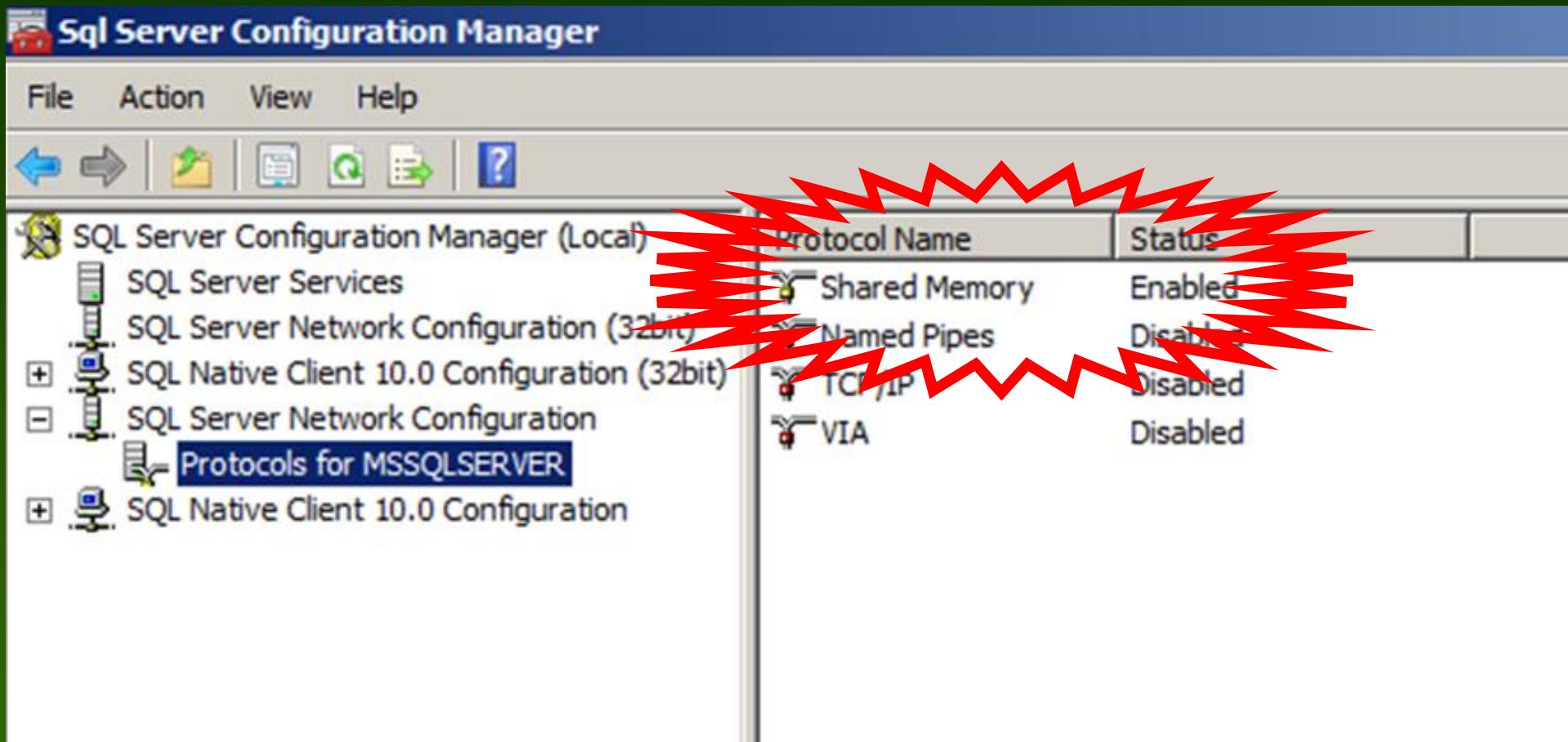


PostgreSQL Shared Memory

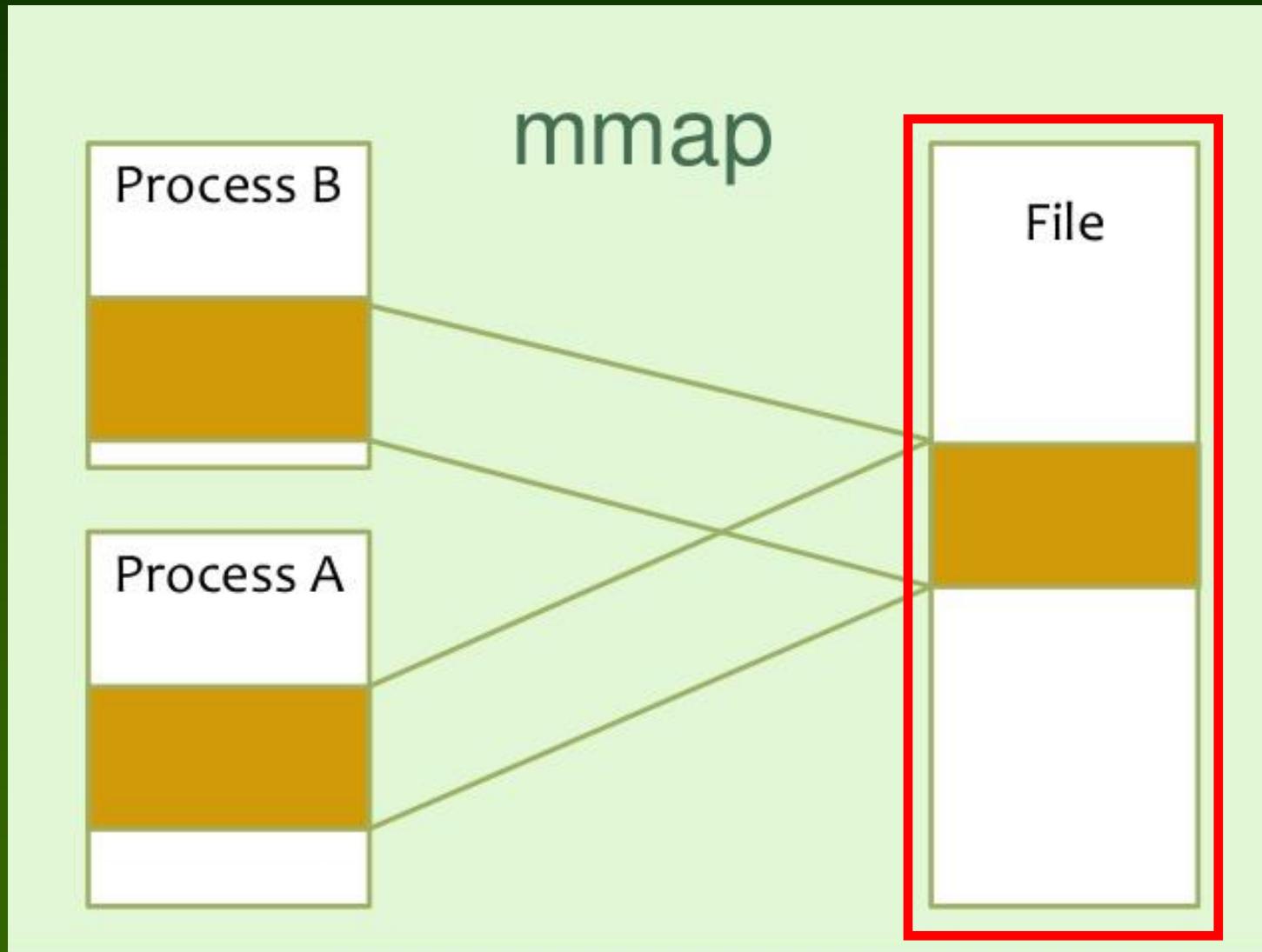


(C) 2015 Suzuki Hironobu All Rights Reserved.

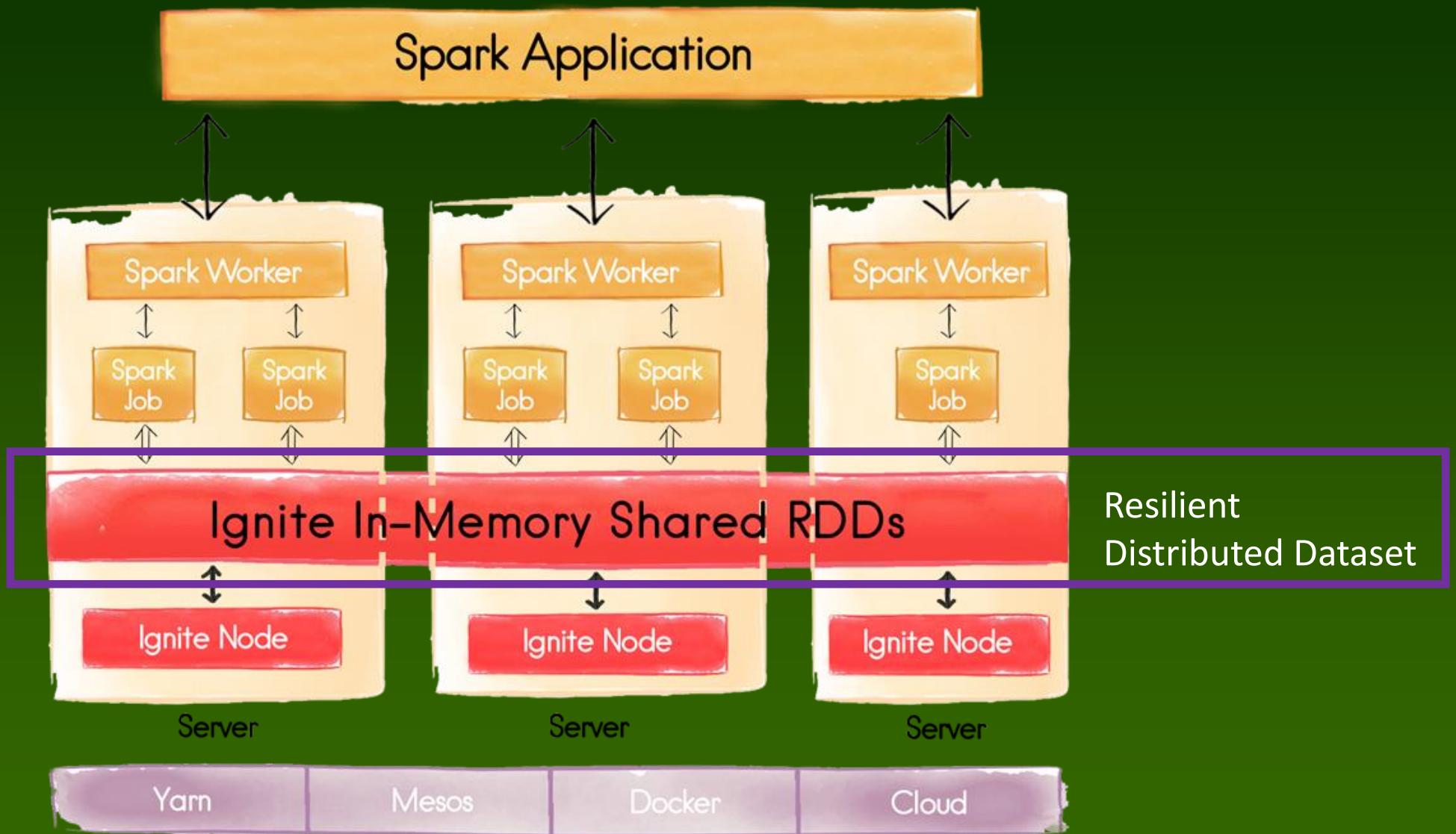
MS SQL Shared Memory



MongoDB Memory Mapped Files



Apache Spark





Intro to Operating Systems

TLPI 56: Sockets



Internet History

- The US Department of Defense awarded contracts as early as the 1960s for packet network systems, including the development of the ARPANET.
- The first recorded description of the social interactions that could be enabled through networking was a series of memos written by J.C.R. Licklider of MIT in August **1962** discussing his "Galactic Network" concept. **He envisioned a globally interconnected set of computers** through which everyone could quickly access data and programs from any site.

Though Vennevar Bush's "As We May Think" came out in 1945 (as an expanded version of an essay from 1939), communication of information over connected memex systems was never mentioned. Bush's vision has been instrumental in the creation of the internet and the information society.



Internet History

- The first message was sent over the ARPANET from computer science Professor Kleinrock's laboratory at University of UCLA to the second network node at Stanford Research Institute, at **22:30 hours on October 29, 1969.**
 - The computer at SRI was connected to Doug Engelbart's project on "Augmentation of Human Intellect."
 - The first message sent of the internet was "lo."
 - It was supposed to me "login," but the SRI computer crashed when the "g" was received.
- By **December 5, 1969**, a 4-node network was connected by adding the University of Utah and the University of California, Santa Barbara.

Grew up in Portland

The first internet router, YouTube by Kleinrock himself.

29 Oct 69	100	LOADED	OP. PROGRAM	(SK)
		EOR BEN BAIRER		
		BRK		
22:30		<u>Talked to SRF</u> <u>Host to Host</u>		(CSIC)

Left op. program (SK)
running after sending
a host dead message
to imp.

These are Charley Kline's actual notes
from the first internet connection.

Internet History

- By **June 1970**, MIT, Harvard, BBN, and Systems Development Corp in Santa Monica, Cal. were added. By January 1971, Stanford, MIT's Lincoln Labs, Carnegie-Mellon, and Case-Western Reserve U were added. In months to come, NASA/Ames, Mitre, Burroughs, RAND, and the Univ of Illinois were plugged in.
- In **1972** the initial "**hot**" application, **electronic mail**, was introduced, along with telnet and ftp. Sendmail (as delivermail) came out with BSD in 1979.
- In the early **1980s** the **NSF funded the establishment for national supercomputing centers at several universities**, and provided interconnectivity in **1986** with the NSFNET project, which also created network access to the supercomputer sites in the United States from research and education organizations. Commercial Internet service providers (ISPs) began to emerge in the very late 1980s.

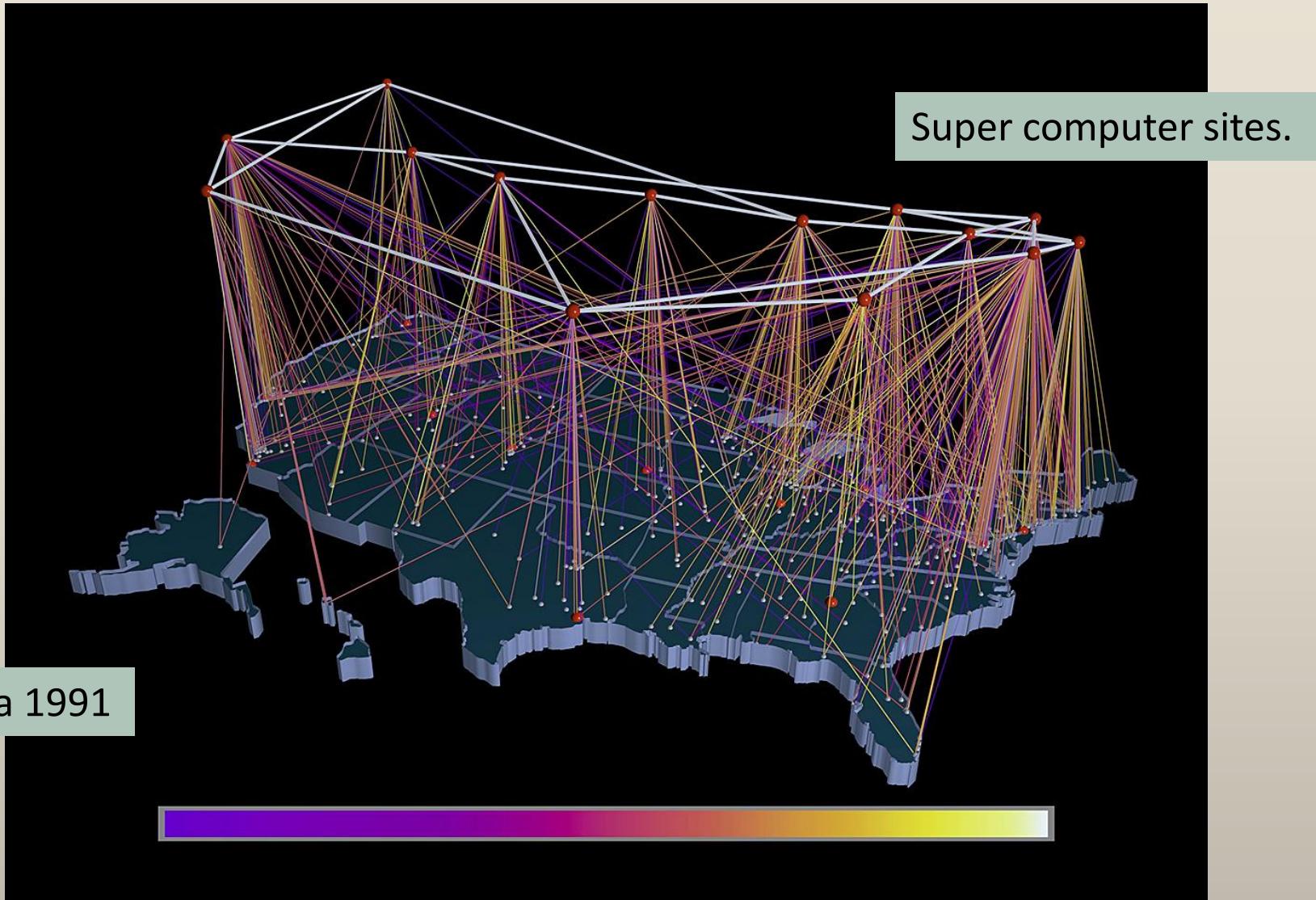
Internet History

- The **ARPANET was decommissioned** in 1990.
 - Limited private connections to parts of the Internet by officially commercial entities emerged in several American cities by late 1989 and 1990, and the **NSFNET was decommissioned** in 1995, **removing the last restrictions on the use of the Internet to carry commercial traffic.**

Advanced **R**esearch **P**rojects **A**gency **N**etwork (**ARPANET**)

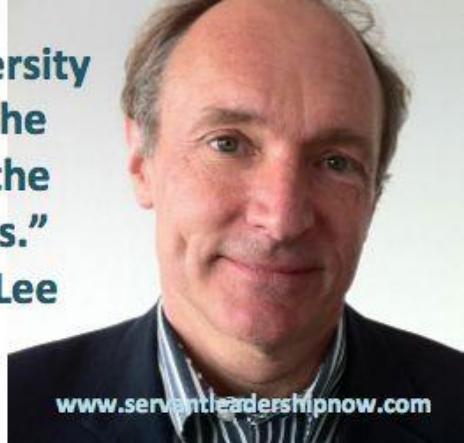
The ARPANET was established by the Advanced Research Projects Agency (ARPA) of the United States Department of Defense.

NSFNET Backbone Nodes



Internet History

"We need diversity of thought in the world to face the new challenges."
- Tim Berners-Lee



- There were other networks that started and contributed to the growth of networking. Specifically, there was **UUCP**, **FidoNet**, and **BITNET**, but they were eclipsed by the internet because of its speed and availability.
- In the **1980s**, the work of British computer scientist **Tim Berners-Lee** on the World Wide Web theorized protocols linking **hypertext documents** into a working system, marking the beginning of the modern Internet.
- The Internet's takeover of the global communication landscape was almost instant in historical terms: it only represented 1% of the information in the year **1993**, already 51% by **2000**, and more than 97% of the telecommunicated information by **2007**.

Sockets

Sockets are a method of IPC that allow data to be exchanged between applications, either on the **same host** (server/computer) or on **different hosts** connected over a network.

- Sockets were first introduced in **2.1BSD** in 1979 and subsequently refined into their current form with **4.2BSD** in 1983.
- The sockets feature is now available with most current UNIX system releases as well as most other computer operating systems.



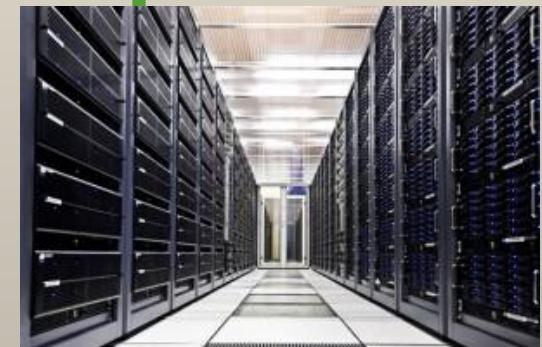
The Internet



Your computer



Server somewhere



Communication Domains

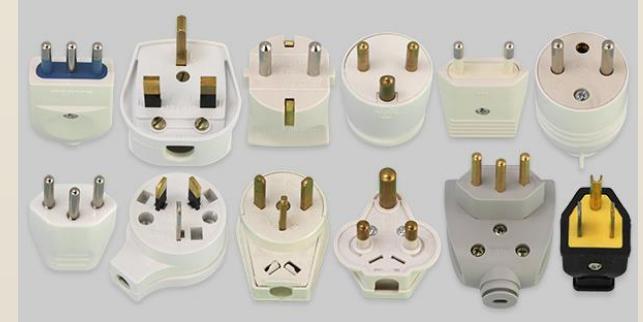


- The **UNIX** (**AF_UNIX**) domain allows communication between applications on the **same host**. (POSIX.1g used the name AF_LOCAL as a synonym for AF_UNIX, but this name is not used in SUSv3.). The address used for these is a pathname.
- The **IPv4** (**AF_INET**) domain allows communication between applications running on hosts **connected via an Internet Protocol version 4 (IPv4) network**. The address used for these is a **32-bit address** and a 16-bit port number.
- The **IPv6** (**AF_INET6**) domain allows communication between applications running on hosts **connected via an Internet Protocol version 6 (IPv6) network**. Although IPv6 is designed as the successor to IPv4, IPv4 is currently still the most widely used. The address used for IPv6 is a **128-bit address** and a 16-bit port number.

Communication Domains

Domain	Communication performed	Communication between applications	Address format	Address structure
AF_UNIX	Within kernel	on same host	pathname	<code>sockaddr_un</code>
AF_INET	Via IPv4	on hosts connected via an IPv4 network	32-bit IPv4 address + 16-bit port number	<code>sockaddr_in</code>
AF_INET6	Via IPv6	on hosts connected via an IPv6 network	128-bit IPv6 address + 16-bit port number	<code>sockaddr_in6</code>

Socket Types



Property	Socket Type	
	Stream	Datagram
Reliable delivery?	Y	N
Message boundaries preserved?	N	Y
Connection oriented?	Y	N

Socket Types



Stream sockets provide a **reliable, bidirectional, byte-stream** communication channel.

- **Reliable** means that it is **guaranteed** that either the transmitted data will arrive intact at the receiving application, exactly as it was transmitted by the sender (assuming that neither the network link nor the receiver crashes), **or** that you receive notification of a probable failure in transmission.
- **Bidirectional** means that data may be transmitted in **either direction** between two sockets.
- **Byte-stream** means that, as with pipes, there is **no concept of message boundaries**.

Socket Types



Datagram sockets allow data to be exchanged in the form of messages called **datagrams**.

- With datagram sockets, **message boundaries are preserved**, but **data transmission is not reliable**.
- Messages may arrive **out of order**, be **duplicated**, or **not arrive at all**.
- Datagram sockets are an example of the more generic concept of a **connectionless socket**.

Datagram vs. Stream

Use **Stream** when every packet matters.

- File transfers Financial data.
- User I/O
- Having every bit is important.
- Also known as TCP sockets and Connection Oriented



Use **Datagram** when the loss of a few packets is okay.

- Streaming quotes
- Real time video Weather data. Fortune cookies.
- It's more important to be current than to have everything.
- Also known as UDP sockets and Connectionless

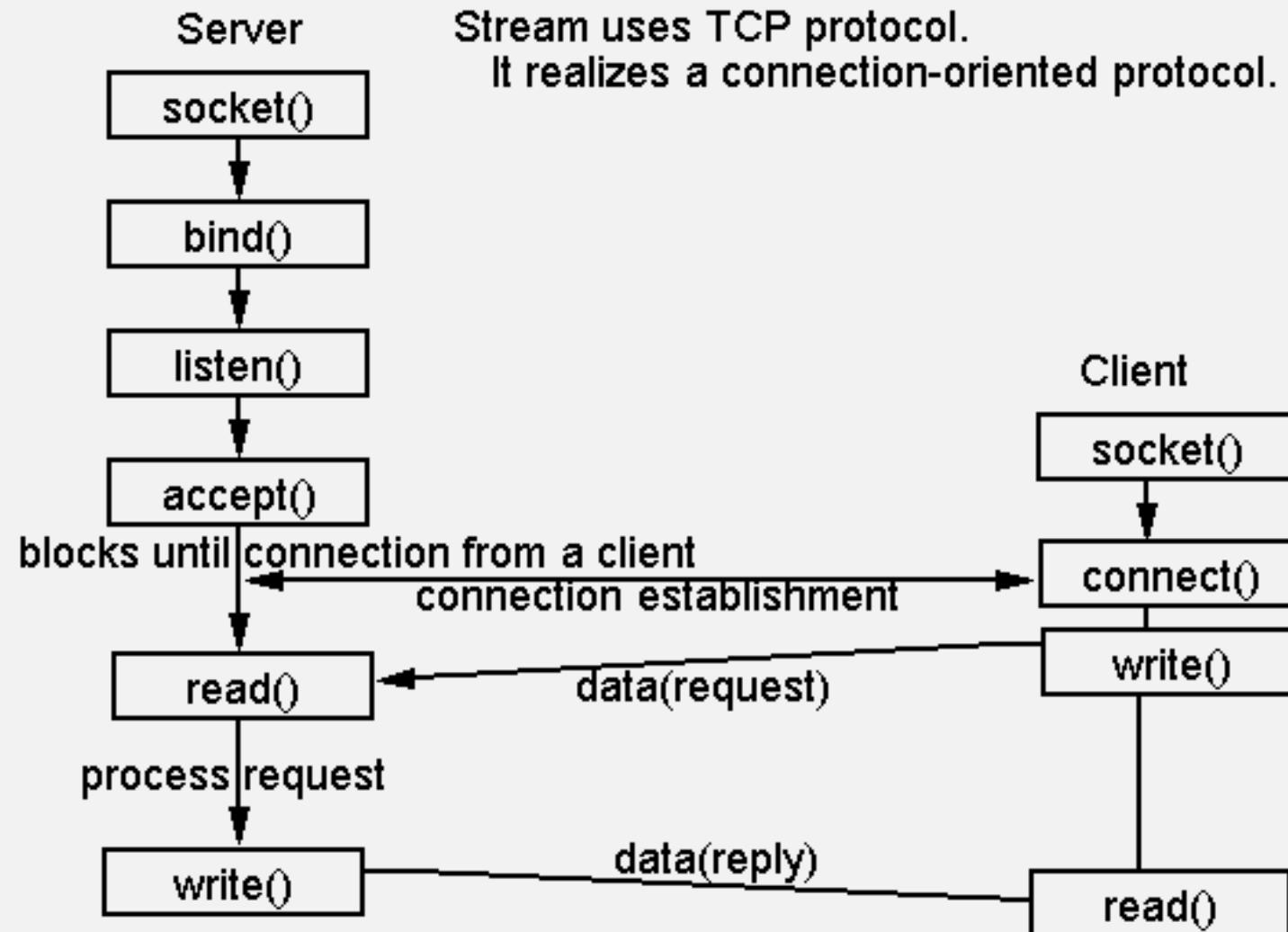
Socket System Calls



The key socket system calls are the following:

- The **socket()** system call creates a new socket.
- The **bind()** system call binds a socket to an address. Usually, a **server executes this call to bind its socket** to a well-known address so that clients can locate the socket.
- The **listen()** system call allows a stream socket to **accept incoming connections** from other sockets.
- The **accept()** system call accepts a **connection from a peer** application on a listening stream socket, and optionally returns the address of the peer socket.
- The **connect()** system call **establishes a connection** with another socket.

Stream Sockets in an Application



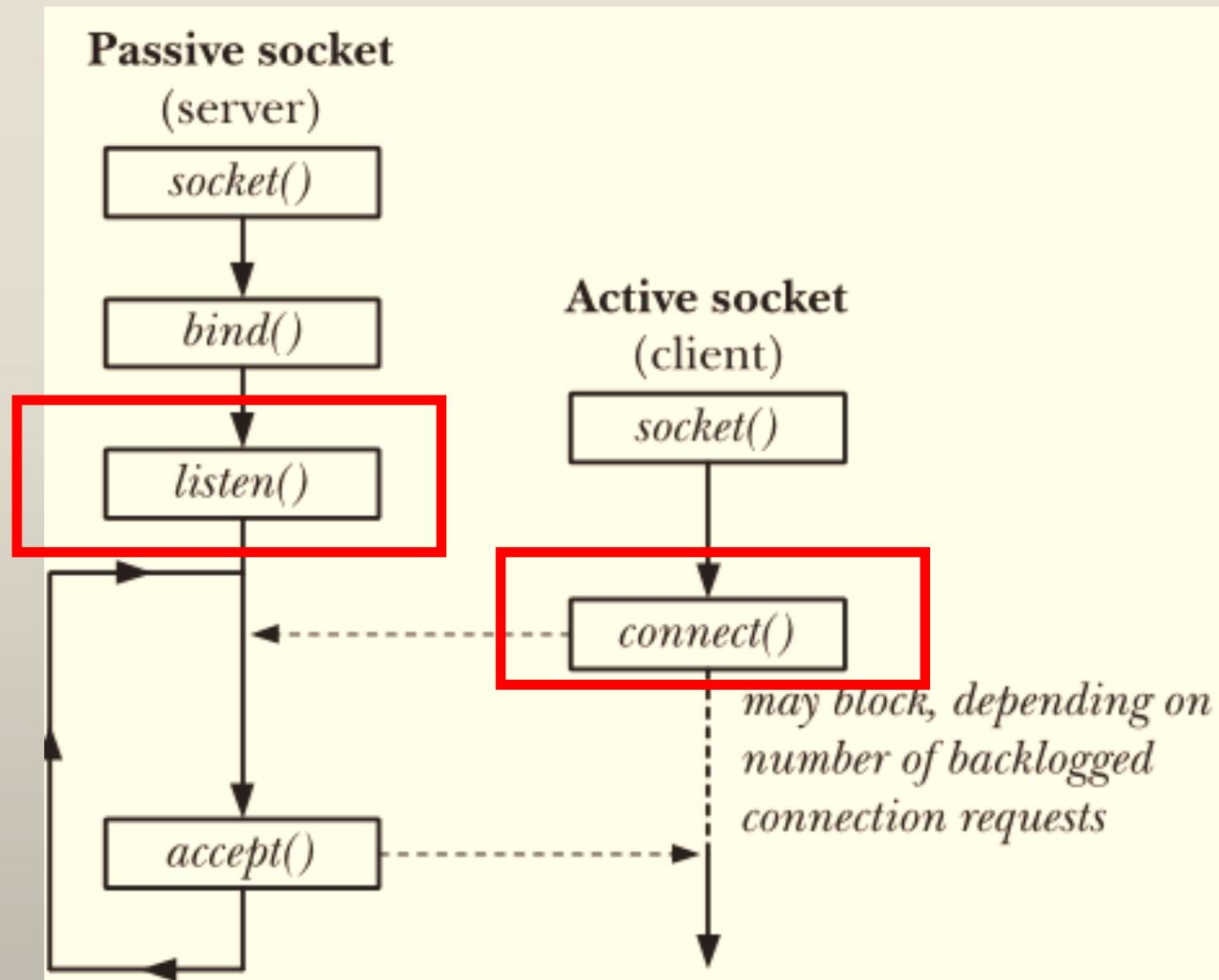
Active vs Passive Sockets

Stream sockets are often distinguished as being either **active** or **passive**:

- A socket that has been **created using `socket()`** is by default active. An active socket can be used in a **`connect()`** call to establish a connection to a passive socket.
 - This is referred to as performing an **active open**.
- A **passive socket** (or a **listening socket**) is one that has been marked to **allow incoming connections** by calling **`listen()`**.
 - Accepting an incoming connection is referred to as performing a **passive open**.



Active Client and Passive Server



Listening Sockets

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);

int bind(int sockfd
         , const struct sockaddr *addr , socklen_t addrlen);

int listen(int sockfd, int backlog);

int accept(int sockfd, struct sockaddr *addr
           , socklen_t * addrlen );
```



Connecting to a Peer Socket

```
#include <sys/socket.h>

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



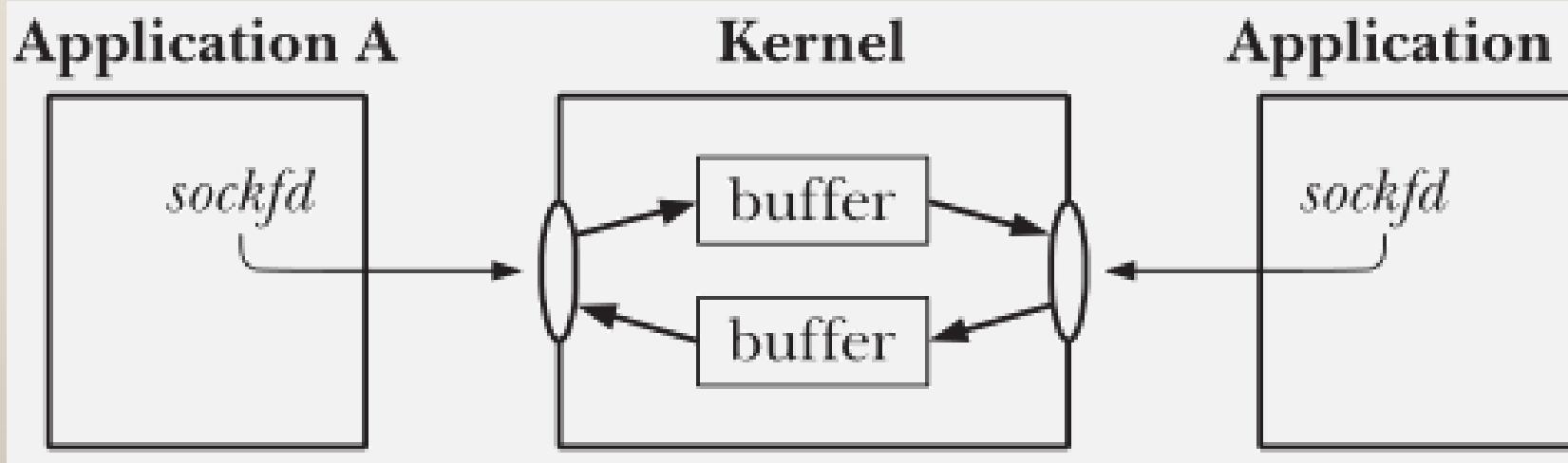
Stream Sockets

- Connection based. The connection must be established **before** data can be exchanged.
- Delivery in a networked environment is **guaranteed**.
 - If delivery is impossible, the sender receives an error indicator.
- If you send bytes through the stream socket three items "A, B, C", they will **arrive in the same order** – "A, B, C".
- These sockets use TCP (Transmission Control Protocol) for data transmission.
- Data records do **not have any boundaries**.



The common analogy for stream sockets is that of the telephone system.

I/O on Stream Sockets



read() and **write()** or
send() and **recv()**

A single socket file descriptor can be used to both send and receive data through the socket.

Server Code

```
int main(int argc, char *argv[]) {
    int listenfd, sockfd, n;
    char buf[4096];
    socklen_t clilen;
    struct sockaddr_in cliaddr, servaddr;
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&servaddr, 0, sizeof(servaddr));
```

Create the
socket

The protocol argument is always specified as 0 for the socket types we describe in this class.

```
servaddr.sin_family = AF_INET;      htonl() is Host to Network Long
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
```

Bind the
socket

```
if (bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) != 0) {
    perror("Something is not working");
    exit(1);
}
```

Max backlog length.

Liston on
the socket

```
listen(listenfd, LISTENQ);
printf("server listening on %s\n", SERV_PORT_STR);
```

Server Code (contd)

Accept connections
on the socket

```
clilen = sizeof(cliaddr);
sockfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen);
for ( ; ; ) {
```

Read data from
the client

```
    memset(buf, 0, sizeof(buf));
    if ((n = read(sockfd, buf, sizeof(buf))) == 0) {
        printf("EOF found on client connection socket, exiting\n");
        close(sockfd);
        break;
    }
```

```
    else {
        fprintf(stdout, "message from client: <%s>\n", buf);
        write(sockfd, buf, n);
    }
}
```

Write data back to
the client

```
printf("Closing listen socket\n");
close(listenfd);
return(EXIT_SUCCESS);
```

Close the socket

Client Code

```
int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in servaddr;
    char sendline[MAXLINE];
    char recvline[MAXLINE];
    char ip_addr[50] = "10.217.113.250";
    if (argc > 1)
        strcpy(ip_addr, argv[1]);
    }
    sockfd = socket(AF_INET, SOCK_STREAM, 0);          Create the socket
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    inet_pton(AF_INET, ip_addr, &servaddr.sin_addr.s_addr);      Connect to the server
    if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) != 0) {
        perror("could not connect");
        exit(EXIT_FAILURE);
    }
```

Client Code (contd)

Send data to
the server

Receive data back
from the server

```
fputs("ready >> ", stdout);
while (fgets(sendline, sizeof(sendline), stdin) != NULL) {
    sendline[strlen(sendline) - 1] = 0;
    memset(recvline, 0, sizeof(recvline));
    write(sockfd, sendline, strlen(sendline));

    if (read(sockfd, recvline, sizeof(recvline)) == 0) {
        perror("socket is closed");
        break;
    }

    fprintf(stdout, "back from server: <%s>\n", recvline);
    fputs("ready >> ", stdout);
}

close(sockfd);
return(EXIT_SUCCESS);
}
```

Close the socket

Datagram Sockets

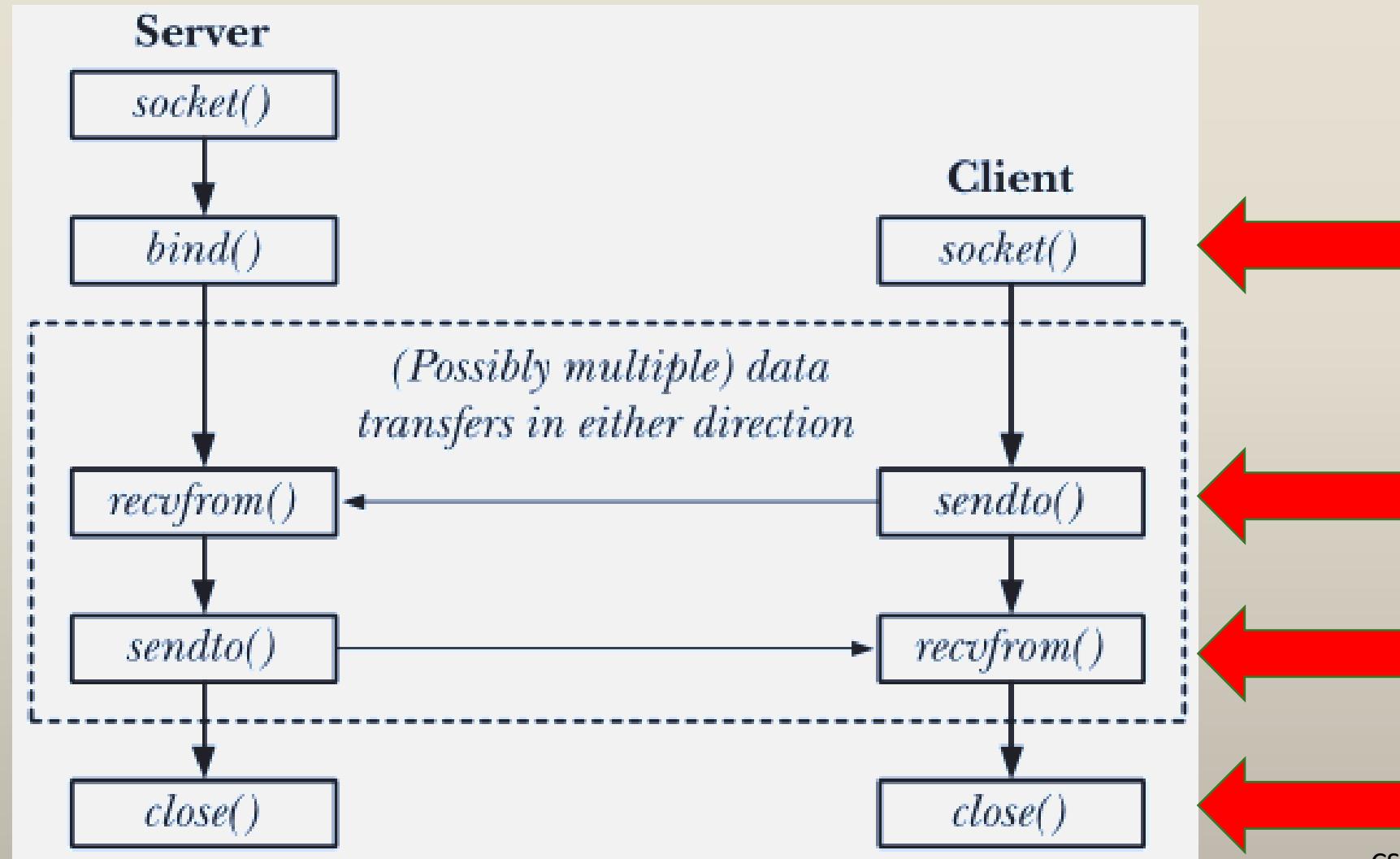
- Delivery in a networked environment is **not guaranteed**.
- There is **no guarantee that they will arrive in the order they were sent**.
- They're **connectionless** because you don't need to have an open connection as in Stream Sockets – you build a packet with the destination information and send it out.
- They use **UDP (User Datagram Protocol)**.

The common analogy for stream sockets is that of the postal system.

Put a stamp on it, drop it in the box, and **trust** it gets there and that someone reads it.



Datagram Sockets in an Application



Datagram Sockets

```
#include <sys/socket.h>

ssize_t recvfrom(int sockfd
                 , void *buffer
                 , size_t length int flags
                 , struct sockaddr *src_addr
                 , socklen_t *addrlen);

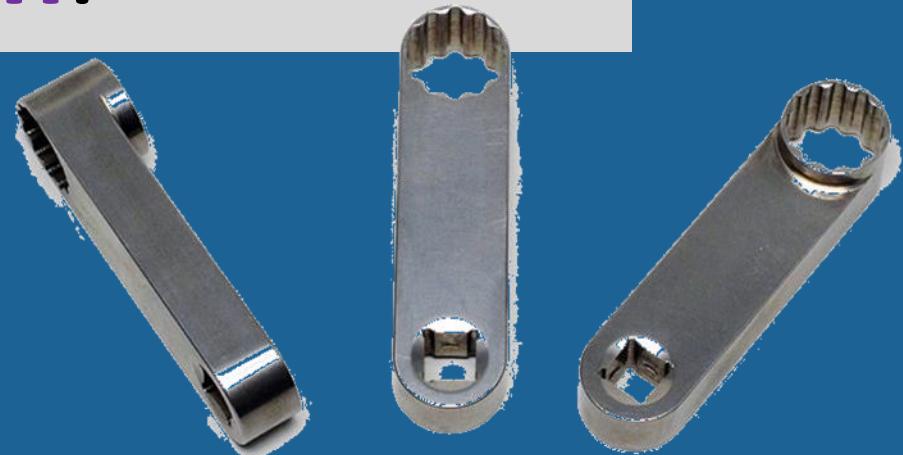
ssize_t sendto(int sockfd
               , const void *buffer
               , size_t length, int flags
               , const struct sockaddr *dest_addr
               , socklen_t adrlen);
```





UNIX Domain Sockets

UNIX domain sockets only allow communication **between processes** on the **same host system**.



UNIX Domain Sockets

- Unix domain sockets are **secure in the network protocol** sense of the word, because:
 - they **cannot be eavesdropped on by a untrusted network**
 - **remote computers cannot connect to them** without some sort of forwarding mechanism.
- They do not require a properly configured network, or even network support at all.
- They are **full duplex** (send and receive on a single socket).
- Many clients can be connect to the same server using the same named socket (client/server).

UNIX Domain Sockets

- Both **datagram, and stream sockets** are supported
- Unix domain sockets are secure in the IPC sense of the world
 - File permissions can be configured on the socket to limit access to certain users or groups.
 - Everything that is going on takes place on the same computer controlled by a single kernel, the kernel knows everything about the socket and the parties on both sides.
- **Open file descriptors from one process can be sent to another totally unrelated process.**

Simplex vs Duplex

- A **duplex communication** system is a point-to-point system composed of two connected parties or devices that can communicate with one another in both directions.
 - In a **full-duplex** system, both parties can communicate with each other simultaneously. An example of a full-duplex device is a telephone.
 - In a **half-duplex** system, each party can communicate with the other but not simultaneously; the communication is one direction at a time. An example of a half-duplex device is a walkie-talkie two-way radio that has a "push-to-talk" button.
- **Simplex communication** is a communication channel that sends information in one direction only.

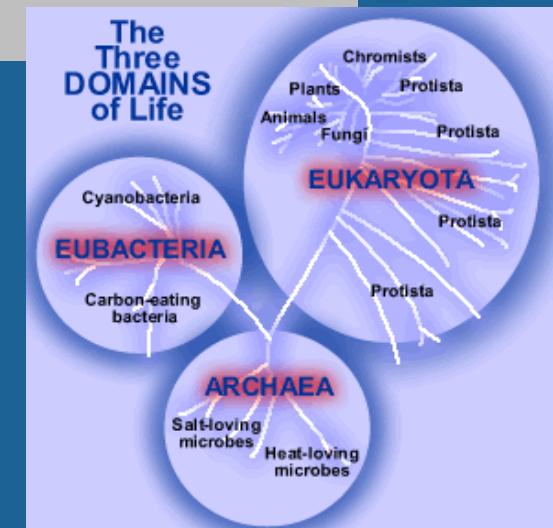


UNIX Domain Socket Addresses

UNIX domain, a socket address takes the form of a **pathname**

```
struct sockaddr_un {  
    sa_family_t sun_family; /* Always AF_UNIX */  
    char sun_path[108];  
    /* Null-terminated socket pathname */  
};
```

The client and server have to agree on the pathname for them to find each other.



UNIX Domain Socket Addresses

Creates an entry in the file system

```
bind(sfd  
      , (struct sockaddr *) &addr  
      , sizeof(struct sockaddr_un));
```

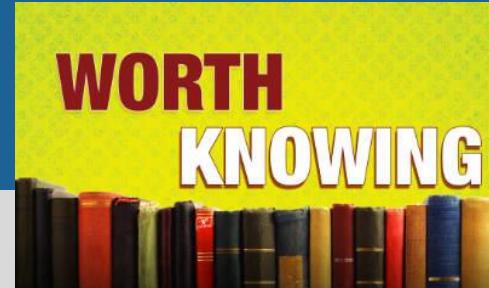
When used to bind a UNIX domain socket,
bind() creates an entry in the file system.



UNIX Domain Socket Points Worth Knowing

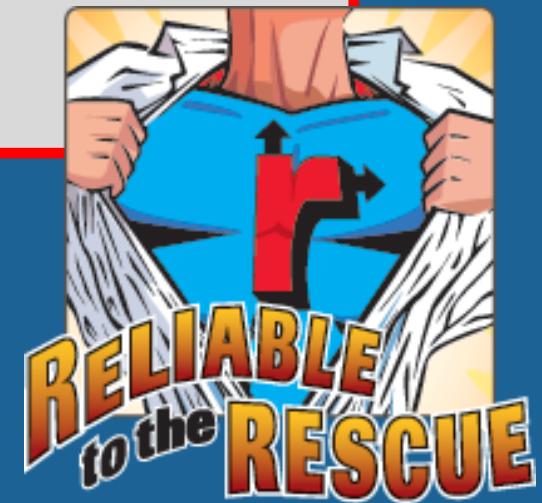
- You **cannot bind a socket to an existing pathname**
- It is standard to bind a socket to an **absolute pathname**, so that the socket resides at a fixed address in the file system.
- A socket may be bound to only one pathname; conversely, a pathname can be bound to only one socket.
- You **cannot use open () to open AF_UNIX socket**.
- When the socket is no longer required, **its pathname entry should be removed using unlink ()**

A great case for exit handlers.



Datagram Sockets in the UNIX Domain

- The generic description of datagram sockets is that communication is unreliable.
- UNIX domain sockets, datagram transmission is carried out within the kernel and **is reliable**.
- All messages are delivered in order and unduplicated.



Socket Pair

- What if you wanted a `pipe()`, but you wanted to use a single pipe to send and receive data from both sides?
- But, pipes are **unidirectional**.
- There is a solution: use a **Unix domain socket**, since they can handle bi-directional data.
- Setting up the code with `listen()` and `connect()` is a chore!
- Check out system call known as **`socketpair()`**
- This is nice enough to return to you a pair of **already connected sockets**!
- No extra work is needed on your part; you can **immediately** use these socket descriptors for inter-process communication.



Creating a Connected Socket Pair

```
#include <sys/socket.h>

int socketpair(int domain // must be AF_UNIX
               , int type
               , int protocol           // must be 0
               , int sockfd[2]
);
```

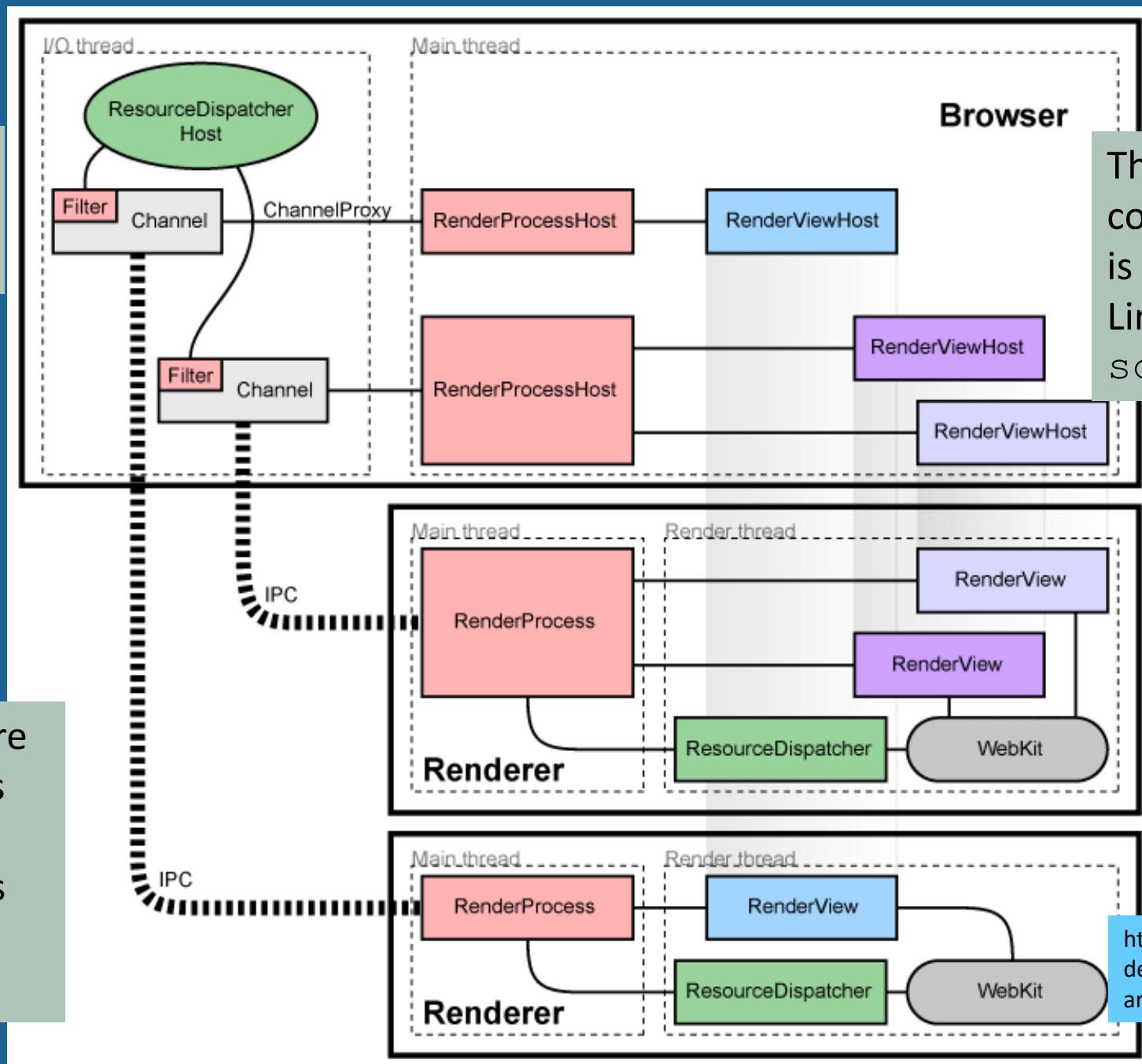
Returns 0 on success, or -1 on error

Each side of the socket can be used for both reading and writing.



The Chrome browser Multi-process Architecture.

Separate processes are used for browser tabs to protect the overall application from bugs and glitches in the rendering engine.

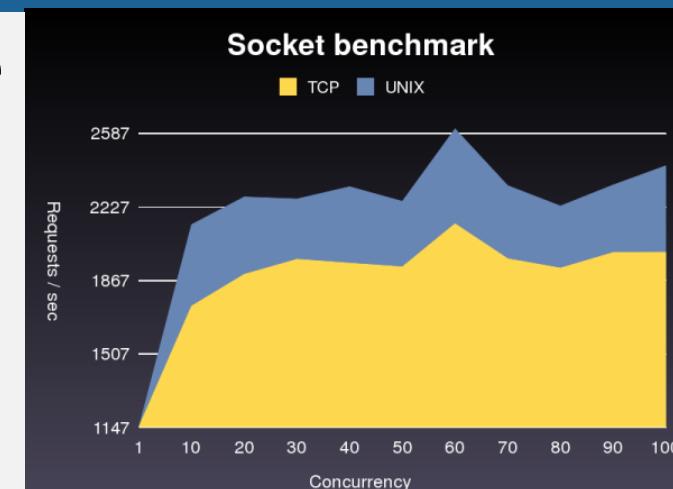


The main inter-process communication primitive is the named pipe. On Linux & OS X, they use a `socketpair()`.

<https://www.chromium.org/developers/design-documents/multi-process-architecture>

UNIX Versus Internet Domain Sockets

- On some implementations, **UNIX domain sockets are faster than Internet domain sockets.**
- We can use **directory** (and, on Linux, file) **permissions to control access** to UNIX domain sockets. With Internet domain sockets, we need to do rather more work (password validation) if we wish to authenticate clients.
- Using UNIX domain sockets, we can pass **open file descriptors** and sender credentials.



Comparison with Named Pipes

- Duplex: Stream sockets provide bi-directional communication while named pipes are uni-directional.
- Distinct clients: Clients using sockets each have an independent connection to the server.
- With named pipes, many clients may write to the pipe, but the server cannot distinguish the clients from each other – the server has only one descriptor to read from the named pipe.



Comparison with Named Pipes

- Method of creating and opening: **Sockets are created using `socket` and assigned their identity via `bind`.**
- Named pipes are created using **`mkfifo`**.
 - To connect to a UNIX domain socket the normal `socket`/`connect` calls are used, but a named pipe is written using regular file `open` and `write`.
 - This can make **named pipes easier to use from a shell script.**

Linux Specific Features

Linux Abstract Socket Namespace

If the pathname for a UNIX domain socket **begins with a null byte**, `\0`, its name is **not mapped** into the file system.

The name **cannot collide** with other names in the filesystem.

When a server closes its UNIX domain listening socket in the abstract namespace, its file is **automatically deleted**; with regular UNIX domain sockets, the file persists after the server closes it.



Socket Sundry

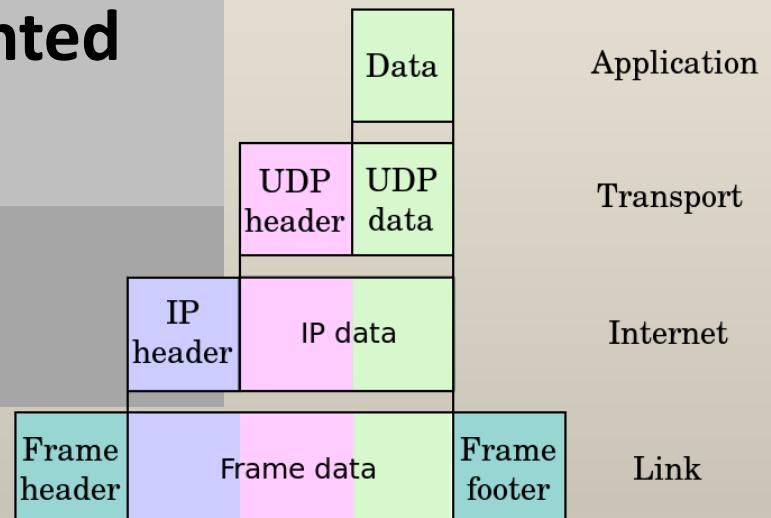


- Sockets are, in general, bi-directional.
 - It is possible to close one side of a socket, allowing one-way communication. This is done using the **shutdown()** call.
- The **read()** and **write()** functions are not the only way to work with sockets.
 - The **original** calls are **recv()** and **send()**. They provide some socket specific functionality beyond **read()** and **write()** (such as peek in to a socket buffer).
 - The **getsockname()** and **getpeername()** system calls return the local address to which a socket is bound and the address of the peer socket to which the local socket is connected.



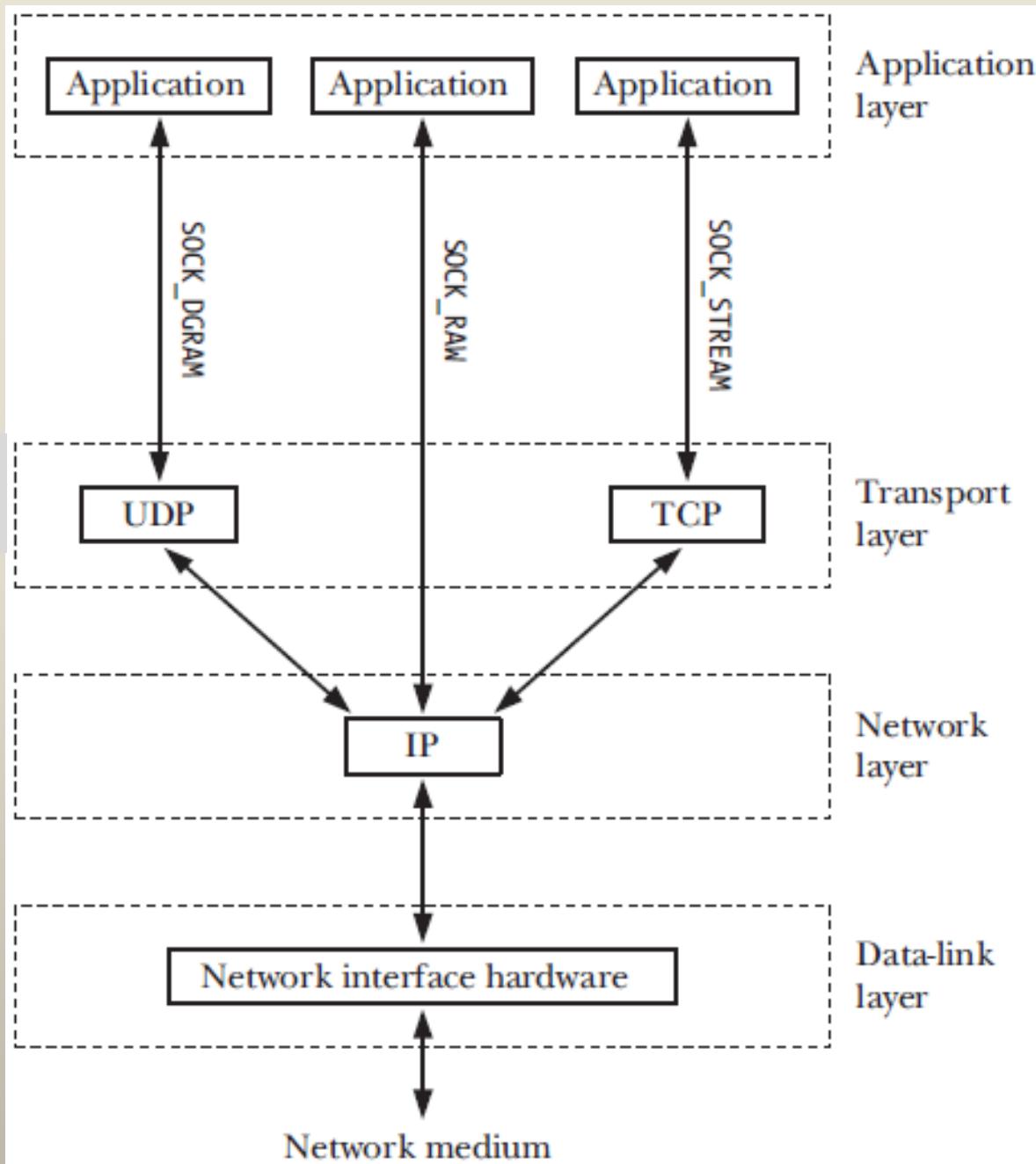
Internet Protocol

- Historically, **IP** was the **connectionless** datagram service in the original Transmission Control Program introduced by **Vint Cerf and Bob Kahn in 1974**.
- The *other* protocol being the **connection-oriented** Transmission Control Protocol (**TCP**).
- The Internet protocol suite is therefore often referred to as TCP/IP.**



Sample encapsulation of application data from UDP to a Link protocol frame.

User Datagram Protocol



Transmission Control Protocol

Protocols in
the TCP/IP
suite

Internet Socket Addresses

There are two types of Internet domain socket addresses:

IPv4 and IPv6.

- IPv4 address has **32 bits**
- IPv6 address has **128 bits**
- With only 32 address bits, IPv4 faced address exhaustion



But but but, what ever happened to IPv5?

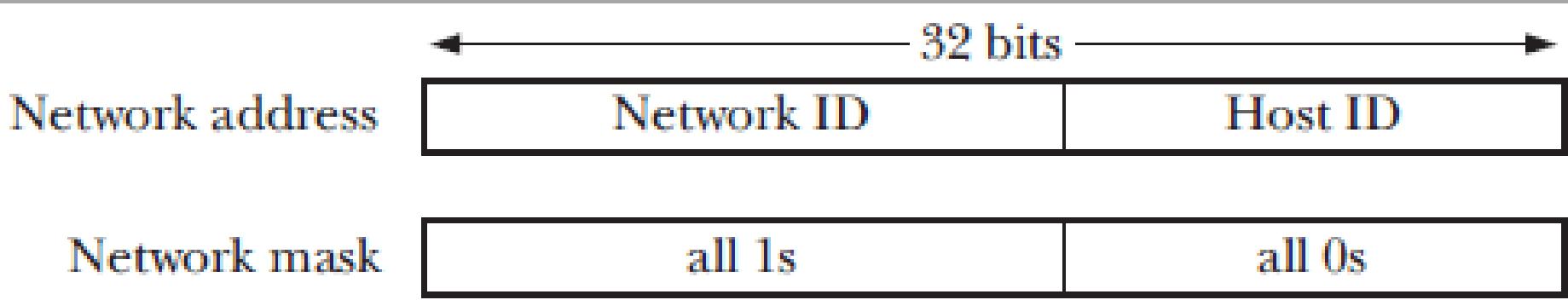
- In the late 1970's, a protocol named ST – The Internet Stream Protocol – was created for the **experimental transmission of voice, video, and distributed simulation**.
 - Two decades later, this protocol was revised to become ST2 and started to get implemented into commercial projects by groups like IBM, NeXT, Apple, and Sun.
 - ST and ST+ offered connections, instead of its connection-less IPv4 counterpart.
- **IP and ST packets can be distinguished by the IP Version Number field, i.e., the first four (4) bits of the packet; ST was assigned the value 5.**
 - There is no requirement for compatibility between IP and ST packet headers beyond the first four bits. (IPv4 uses value 4.)



IP Addresses

An IP address consists of two parts: **a network ID**, which specifies the network on which a host resides, and **a host ID**, which identifies the host within that network.

- An **IPv4** address consists of 32 bits.
- When expressed in **human-readable** form, these addresses are normally written in dotted-decimal notation, with the 4 bytes of the address being written as decimal numbers separated by dots, as in 131.252.208.23.



IPv4 Address Exhaustion

IPv4 **address exhaustion** is the depletion of the pool of unallocated Internet Protocol Version 4 (IPv4) addresses, which was been anticipated since the late 1980s.

This depletion is the reason for the development and deployment of its successor protocol, IPv6.

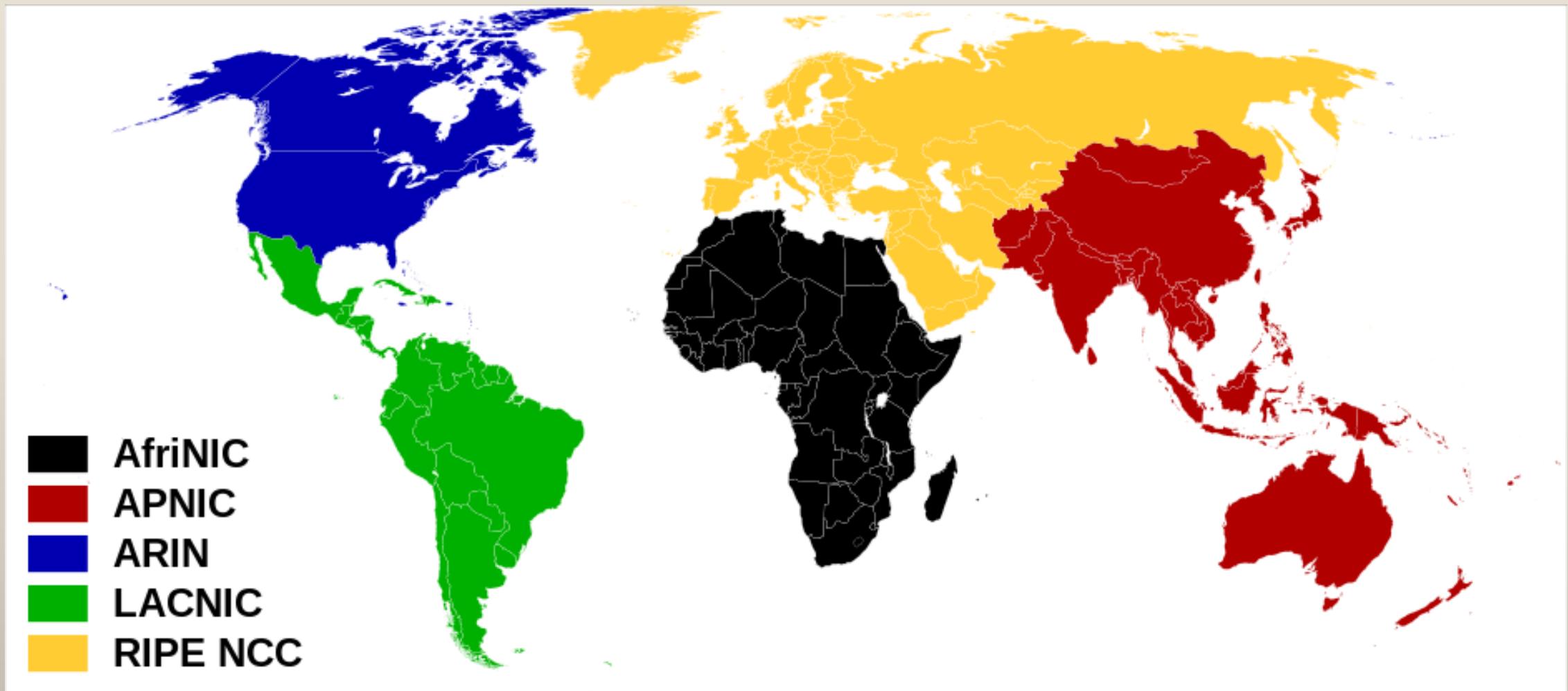
The top-level exhaustion occurred on 31 January 2011.

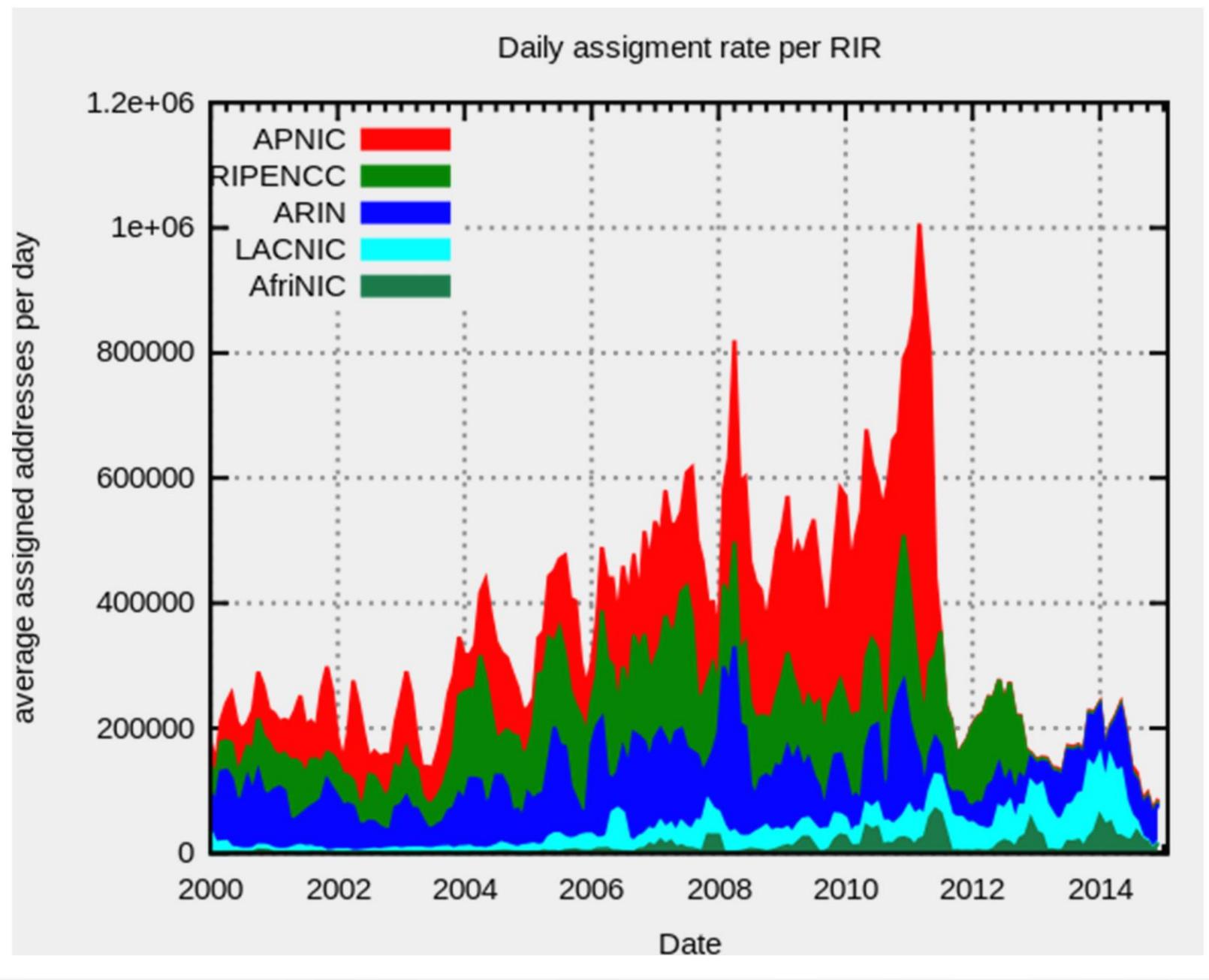
The five Regional Internet registries have exhausted allocation of all the blocks; this occurred on:

- 15 April 2011 for the Asia-Pacific
- 14 September 2012 for Europe
- 10 June 2014 for Latin America and the Caribbean
- 24 September 2015 for North America
- April 2017, AFRINIC



Map of Regional Internet Registries





IPv6 Addresses

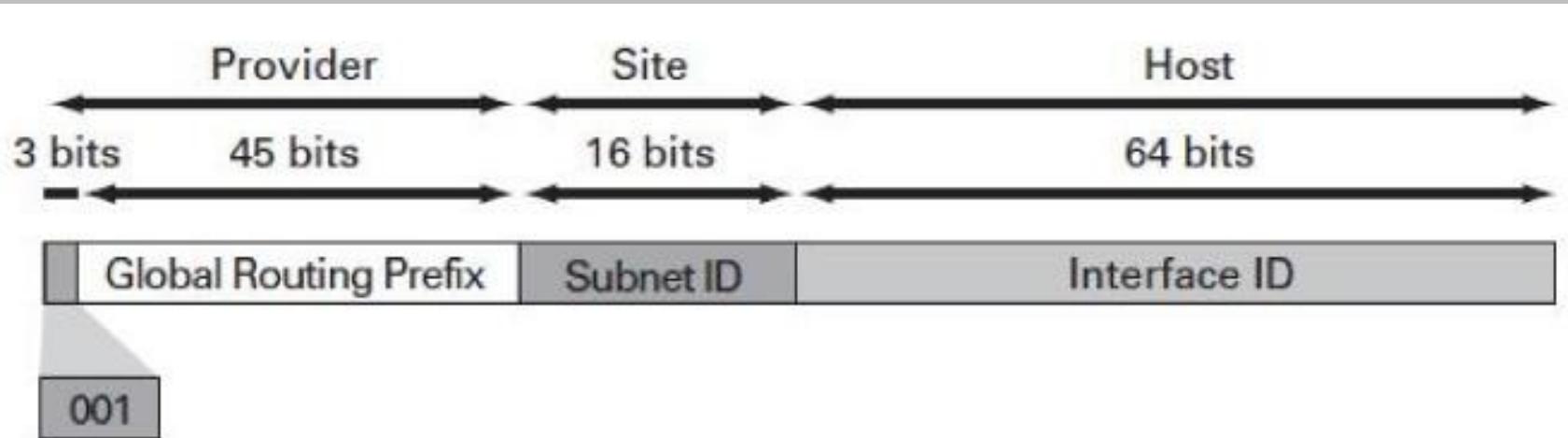
THERE'S NO PLACE LIKE
127.0.0.1

The principles of **IPv6** addresses are similar to IPv4 addresses.

- The key difference is that IPv6 addresses consist of **128 bits**, and the first few bits of the address are a format prefix, indicating the address type.
- IPv6 addresses are typically written as a series of 16-bit hexadecimal numbers separated by colons, as in the following:

2a03:2880:f101:83:**face:b00c**:0:25de

This happens to be facebook.com's IPv6 address



How Big is IPv6?

4.295 x 10⁹ IPv4 address space size (32-bit)

10^{22} Estimated number of stars in the universe.

4.339×10^{26} Nanoseconds that have passed since the Big Bang.

3.156×10^{31} Unique IPv6 addresses that would be assigned after 1 trillion years if a new IPv6 address was assigned every **pico**second (10^{-12} of a second).

3.403 x 10³⁸ IPv6 address space size (128-bit).

10^{81} Estimated total number of atoms in the universe



Sockets: Internet Domain

Internet domain **stream** sockets are implemented on top of Transmission Control Protocol (TCP).

- TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating over an IP network.
- The first widespread implementation of TCP/IP appeared with **4.2BSD in 1983**.



Internet domain **datagram** sockets are implemented on top of UDP.

Network Byte Order

2-byte integer

address N	address N + 1
1 (MSB)	0 (LSB)

Big-endian
byte order

4-byte integer

address N	address N + 1	address N + 2	address N + 3
3 (MSB)	2	1	0 (LSB)

Network
byte order
follows big
endian
order.

address N	address N + 1
0 (LSB)	1 (MSB)

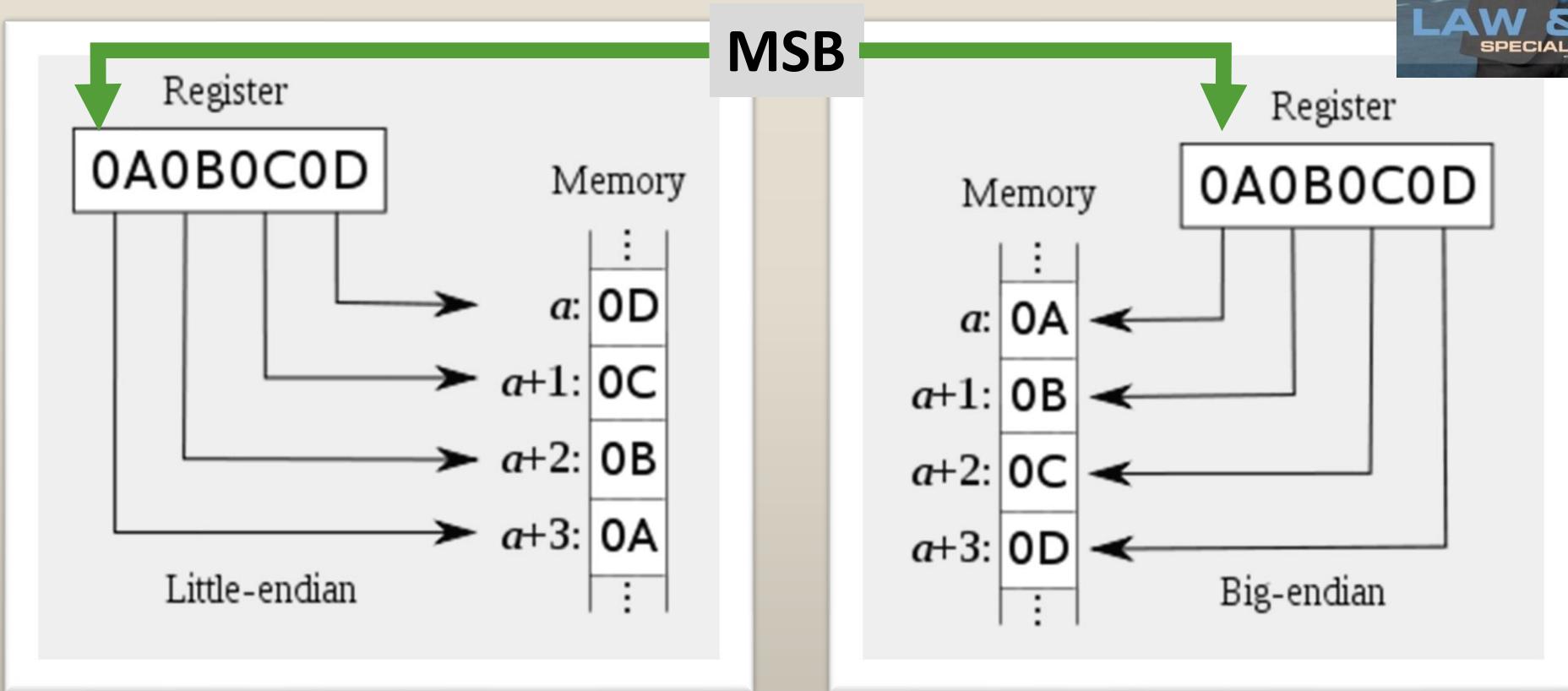
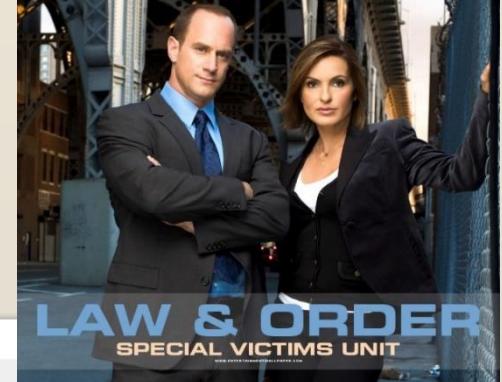
Little-endian
byte order

address N	address N + 1	address N + 2	address N + 3
0 (LSB)	1	2	3 (MSB)

MSB = Most Significant Byte, LSB = Least Significant Byte



Network Byte Order



Ordering Your Bytes



```
#include <arpa/inet.h>
```

```
uint16_t htons(uint16_t host_uint16); Host to network short
```

Returns host_uint16 converted to network byte order

```
uint32_t htonl(uint32_t host_uint32); Host to network long
```

Returns host_uint32 converted to network byte order

```
uint16_t  ntohs(uint16_t net_uint16);
```

Network to host short

Returns net_uint16 converted to host byte order

```
uint32_t ntohl(uint32_t net_uint32);
```

Network to host long

Returns net_uint32 converted to host byte order

Data Representation



The **C long** data type may be 32 bits on some systems and 64 bits on others. Yuck!

The issue with structures is further complicated by the fact that different implementations may employ different rules for **aligning** the fields of a structure to **address boundaries** on the host system, leaving different numbers of **padding** bytes between the fields.

Host and Service Conversion Functions



ASCII to network

The `inet_aton()` and `inet_ntoa()` functions convert an IPv4 address in dotted-decimal notation to binary and from binary to dotted-decimal.

Network to presentation

The `inet_pton()` and `inet_ntop()` functions are like `inet_aton()` and `inet_ntoa()`, but differ in that they **also handle IPv6 addresses**.

Use these!



Conversion Functions



```
#include <arpa/inet.h>
```

```
int inet_pton(int domain  
, const char *src_str ,void *addrptr );
```

Presentation to network

Returns 1 on successful conversion, 0 if src_str is not in presentation format,
or -1 on error

```
const char *inet_ntop(int domain  
, const void * addrptr  
, char *dst_str, size_t len );
```

Network to presentation

Returns pointer to dst_str on success, or NULL on error

Non-Conversion Functions

Another way to manage the big-endian vs little-endian for values passed over the wire...

Only send character data!!!

Host and Service Conversion

```
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char * host
                , const char * service
                , const struct addrinfo * hints
                , struct addrinfo ** result );
```

Returns 0 on success, or nonzero on error



The **getaddrinfo ()** function converts host and service names to IP addresses and port numbers.

It was defined in POSIX.1g as the replacement to the **obsolete** **gethostname ()** and **getservbyname ()** functions.

ifconfig

The netmask.

```
rchaney # ifconfig
```

```
ens3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
```

```
      inet 131.252.208.103 netmask 255.255.255.0 broadcast 131.252.208.255
```

This is probably what you are looking for. It is the IPv4 address.

```
      ether 52:54:00:13:a0:c6 txqueuelen 1000 (Ethernet)
```

```
        RX packets 1168309586 bytes 6247867744599 (6.2 TB)
```

```
        RX errors 0 dropped 0 overruns 0 frame 0
```

```
        TX packets 1152009609 bytes 830851730177 (830.8 GB)
```

```
        TX errors 0 dropped 0 carrier 0 collisions 0
```

This is for ada

```
rchaney # ifconfig
```

```
ens3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
```

```
      inet 131.252.208.23 netmask 255.255.255.0 broadcast 131.252.208.255
```

```
      ether 52:54:00:5c:6f:6e txqueuelen 1000 (Ethernet)
```

```
        RX packets 378754818 bytes 1174830218163 (1.1 TB)
```

```
        RX errors 0 dropped 0 overruns 0 frame 0
```

```
        TX packets 267751538 bytes 137928373834 (137.9 GB)
```

```
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

This is for babbage



Iterative and Concurrent

Two common designs for network servers using sockets are the following:

Iterative: The server handles a single client at a time, processing that client's requests completely, before closing the connection and proceeding to the next client.

Concurrent: The server is designed to handle multiple clients simultaneously.

We've looked at iterative. Let's do concurrent.

Caution: This is a complex form of implementation.
Read closely and ask questions.

I/O Multiplexing



I/O multiplexing allows us to simultaneously monitor **multiple file descriptors** to see if I/O is possible on any of them.

We can use **select()** to monitor multiple file descriptors for **regular files**, terminals, **pipes**, **FIFOs**, **sockets**, and some types of character devices.

The `select()` System Call

```
#include <sys/time.h> /* For portability */  
#include <sys/select.h>
```

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

Returns number of
ready file descriptors.

File descriptor sets.

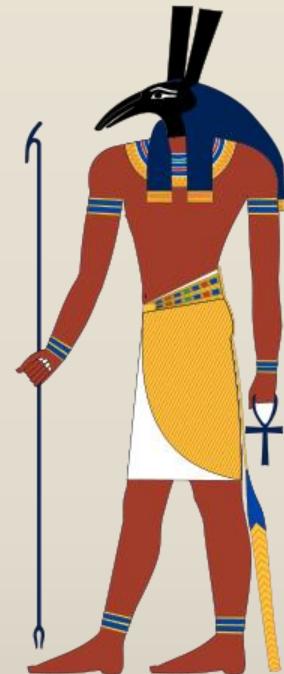


Returns number of **ready** file descriptors, 0 on timeout, or -1 on error

File Descriptor Set

The readfds, writefds, and exceptfds arguments are pointers to **file descriptor sets**, represented using the data type fd_set:

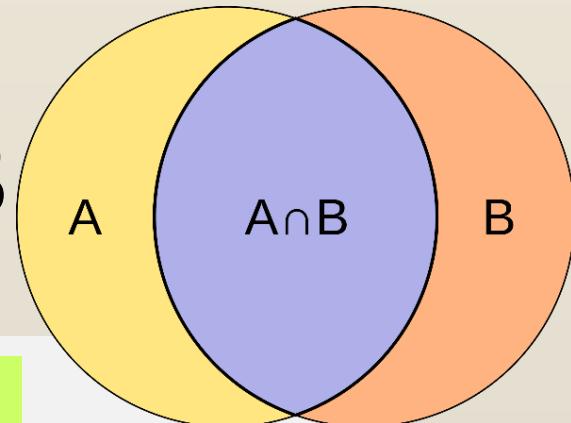
- **readfds** is the set of file descriptors to be tested to see if **input** is possible,
- **writefds** is the set of file descriptors to be tested to see if **output** is possible,
- **exceptfds** is the set of file descriptors to be tested to see if an **exception** condition has occurred.



An exception condition occurs in just two circumstances on Linux:

- A state change occurs on a pseudoterminal that is in packet mode
- Out-of-band data is received on a stream socket

File Descriptor Sets



```
#include <sys/select.h>
```

FD_ZERO() initializes the set pointed to by `fdset` to be **empty**.

```
void FD_ZERO(fd_set *fdset);
```

FD_SET() adds the file descriptor `fd` to the set pointed to by `fdset`.

```
void FD_SET(int fd, fd_set *fdset);
```

FD_CLR() removes the file descriptor `fd` from the set pointed to by `fdset`

```
void FD_CLR(int fd, fd_set *fdset);
```

```
int FD_ISSET(int fd, fd_set *fdset);
```

Returns true if `fd` is in `fdset`, or false (0) otherwise

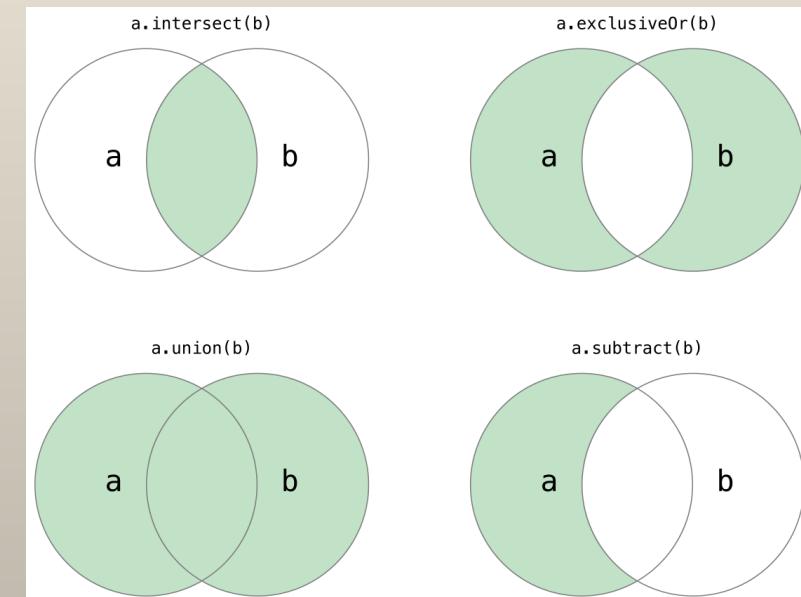
FD_ISSET() returns true if the file descriptor `fd` is a member of the set pointed to by `fdset`

File Descriptor Sets

- A file descriptor set has a **maximum size**, defined by the constant **FD_SETSIZE**.
- On Linux, this constant has the value **1024**.

You can monitor the status of 1024 file descriptors for ready input.

Think of this as a web server monitoring the status of 1000+ active connections.



Calling select()

Before the call to `select()`, the `fd_set` structures pointed to by the arguments **must be initialized** (using `FD_ZERO()` and `FD_SET()`) to contain the set of file **descriptors of interest**.

The `select()` call **modifies** each of these structures so that, on return, they contain the set of file descriptors that are *ready*.



Calling select()

```
select(int nfds, ...
```



The **nfds** argument must be set ***one greater than the highest file descriptor number*** included in any of the three file descriptor sets.

The **nfds** argument allows `select()` to be *more* efficient, since the kernel then knows not to check whether file descriptor numbers higher than this value are part of each file descriptor set.

The Timeout Argument

- The timeout argument to `select()` controls the blocking behavior of `select()`.
- It can be specified either as `NULL`, in which case `select()` blocks indefinitely, or as a pointer to a `timeval` structure (a duration to wait):

```
struct timeval {  
    time_t tv_sec;          /* Seconds */  
    suseconds_t tv_usec;    /* Microseconds (long int) */  
};
```



You're in Timeout

The Timeout Argument

When *timeout* is `NULL`, or points to a structure containing nonzero fields, `select()` blocks until one of the following occurs:

- **at least one** of the file descriptors specified in `readfds`, `writelfds`, or `exceptfds` becomes ready,
- the call is **interrupted** by a signal handler,
- the amount of time specified by `timeout` has passed.



Return Value From `select()`

As its function result, `select()` returns one of the following:

- A return value of -1 indicates that an **error** occurred.
- A return value of 0, the call **timed out** before any file descriptor became ready.
 - In this case, each of the returned file descriptor sets will be empty.
- A **positive return** value indicates that one or more file descriptors is ready.
 - The return value is the sum of the number of ready descriptors.



```
int main(void) {
    fd_set rfds;
    struct timeval tv;
    int retval;
    /* Watch stdin (fd 0) to see when it has input. */
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
    /* Wait up to five seconds
    tv.tv_sec = 5;
    tv.tv_usec = 0;

    retval = select(1, &rfds, NULL, NULL, &tv);
    if (retval == -1)
        perror("select() failure");
    else if (retval)
        printf("Data are available now.\n");
        /* FD_ISSET(0, &rfds) will be true. */
    else
        printf("Timeout: No data within five seconds.\n");

    exit(EXIT_SUCCESS);
}
```

Monitor only 1 file descriptor
in this case, stdin

Set the timeout to 5 seconds.

Check the return value
from select to see what
occurred.

Call select and await its
return.

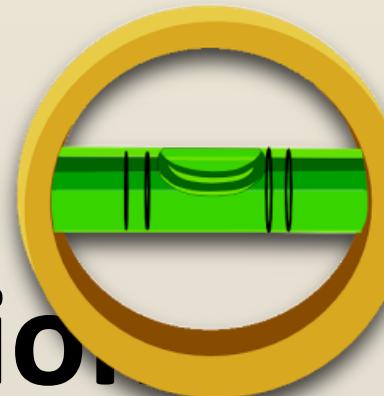
When Is a File Descriptor “*Ready*”?

Using `select()` requires an understanding of the conditions under which a file descriptor is considered *ready*.

A file descriptor is considered to be ready if a call to an I/O function **would not block**, *regardless of whether the function would actually transfer data*.



Level-Triggered and Edge-Triggered Notification



- **Level-triggered notification:** A file descriptor is considered to be ready if it is **possible to perform an I/O system call without blocking.**
- **Edge-triggered notification:** Notification is provided if **there is I/O activity on a file descriptor since it was last monitored.**

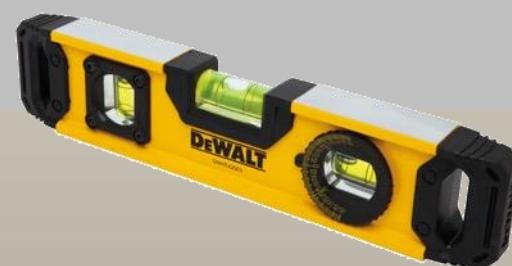


Level-triggered Notification

When we employ **level-triggered notification**, we can check the readiness of a file descriptor at any time.

- When we determine that a file descriptor is ready, we can perform some I/O on the descriptor, and then repeat the monitoring operation to check if the descriptor is still ready, in which case we can perform more I/O.

Level-triggered model allows us to repeat the I/O monitoring operation at any time, it is not necessary to perform as much I/O as possible on the file descriptor each time we are notified that a file descriptor is ready.



Edge-triggered Notification

Edge-triggered notification, we receive notification **only when an I/O event occurs**.

We won't receive any further notification until **another I/O event occurs**.

When an I/O event is notified for a file descriptor, we usually **don't know how much I/O is possible**.



The epoll API

- A common way to implement tcp servers is **one thread or process per connection.**
- On very high load servers this approach can be inefficient and we need to use another patterns of connection handling.
 - **We just run out of threads.**

The C10k Problem



The C10k problem is the problem of optimizing network sockets to handle a **large** number of clients at the same time.

The name C10k is name for concurrently handling **ten thousand connections at a time**.

The term was coined in 1999 by Dan Kegel, citing the Simtel FTP host, cdrom.com, serving 10,000 clients at once over 1 Gigabit Ethernet in that year.

By the early 2010s millions of connections on a single commodity server became possible.

The C10k Problem

`select()` calls are **O(n)**

`epoll` is an **O(1)** algorithm – this means that it scales better as the number of watched file descriptors increases.

`select()` uses a **linear search** through the list of watched file descriptors, which causes its $O(n)$ behavior.

`epoll()` uses **callbacks** in the kernel file structure.

`select()` → $O(n)$

`epoll()` → $O(1)$



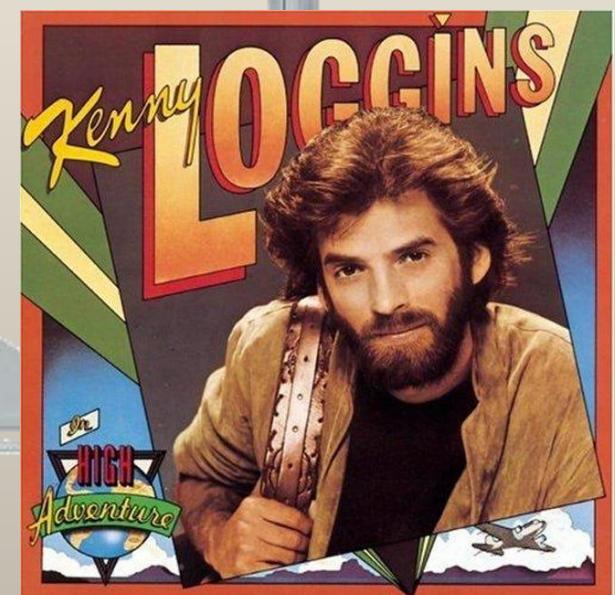
The epoll API

The `epoll()` API can be used **either** as an **edge-triggered or a level-triggered** interface and scales well to **large numbers** of watched file descriptors.

It is meant to replace the older POSIX `select()` and `poll()` system calls, to achieve **better performance** in more demanding applications, where the **number of watched file descriptors is large**.

A Linux only API.





Logging in and Duo

Frequently logging in or using an FTP client (as you will do) in the world with **Duo** can be exasperating.

Do yourself (and those around you) a favor and setup **passwordless** login with ssh.

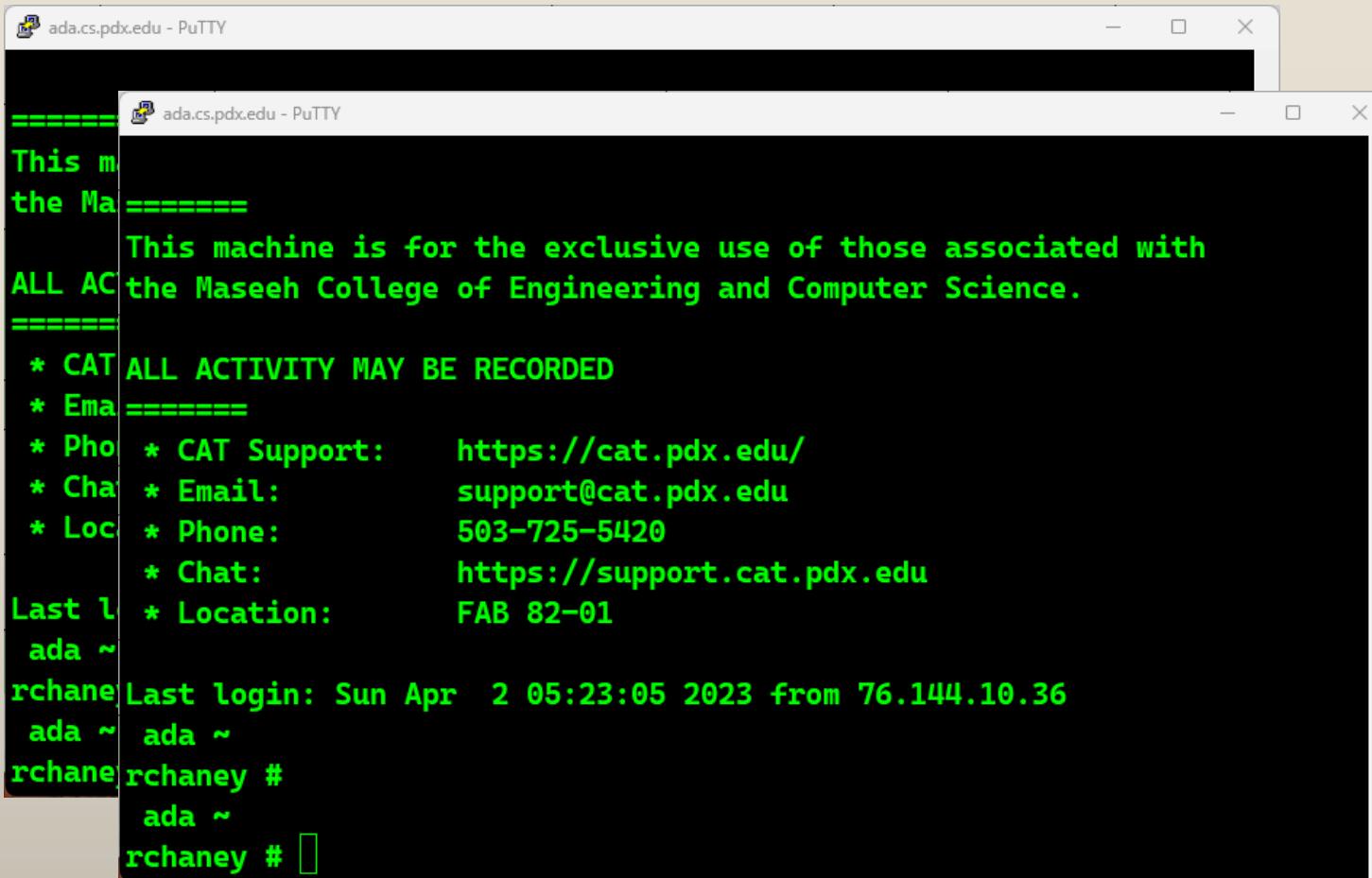
It will take you 20-30 minutes and save you 10 (maybe 100) times that through the term (to say nothing about the anguish).

  [SSH Passwordless Login Using SSH Keygen](#) 

  [How do I set up passwordless login in PuTTY?](#) 



Class Server



The screenshot shows a PuTTY terminal window titled "ada.cs.pdx.edu - PuTTY". The session is connected to the host "ada.cs.pdx.edu". The terminal displays a welcome message from the server:

```
This machine is for the exclusive use of those associated with
ALL AC the Maseeh College of Engineering and Computer Science.
```

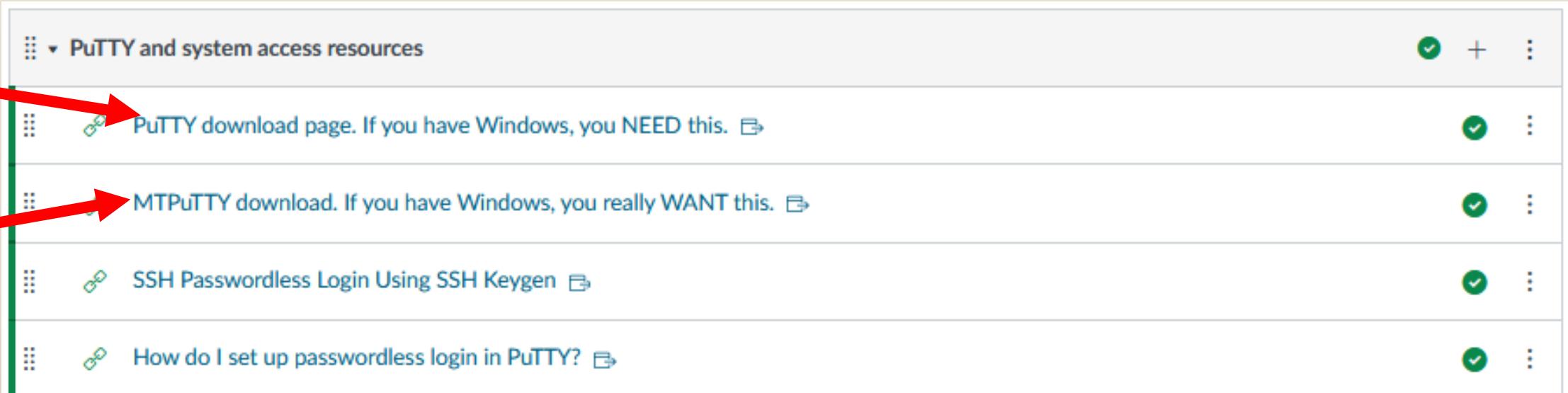
It also lists information about the server's activity monitoring and support resources:

- * CAT ALL ACTIVITY MAY BE RECORDED
- * Email =====
- * Photo * CAT Support: <https://cat.pdx.edu/>
- * Chat: * Email: support@cat.pdx.edu
- * Location: * Phone: 503-725-5420
- * Chat: <https://support.cat.pdx.edu>
- Last login: Sun Apr 2 05:23:05 2023 from 76.144.10.36

The user "ada" is currently logged in, indicated by the prompt "ada ~". The user "rchaney" has also logged in, indicated by the prompt "rchaney #".

You'll only get to use terminal login windows on our server.

PuTTY



A screenshot of a Canvas resource list titled "PuTTY and system access resources". The list contains four items:

- PuTTY download page. If you have Windows, you NEED this.
- MTPuTTY download. If you have Windows, you really WANT this.
- SSH Passwordless Login Using SSH Keygen
- How do I set up passwordless login in PuTTY?

Two red arrows point to the first two items in the list.

If you are a Windows user, you are going to want/need/crave these applications to make your life better.

Some folk are successful using Powershell, but I prefer PuTTY.

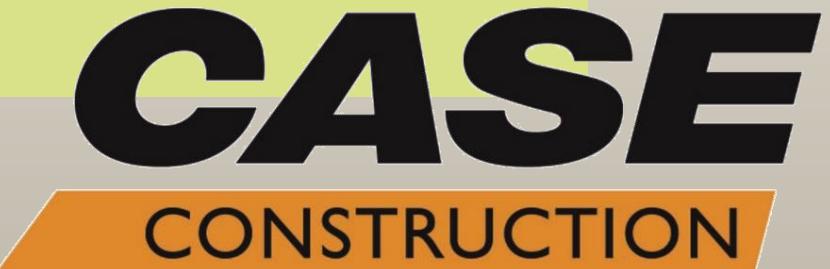
I have placed links to the download sites in the **PuTTY and system access resources** section on Canvas.

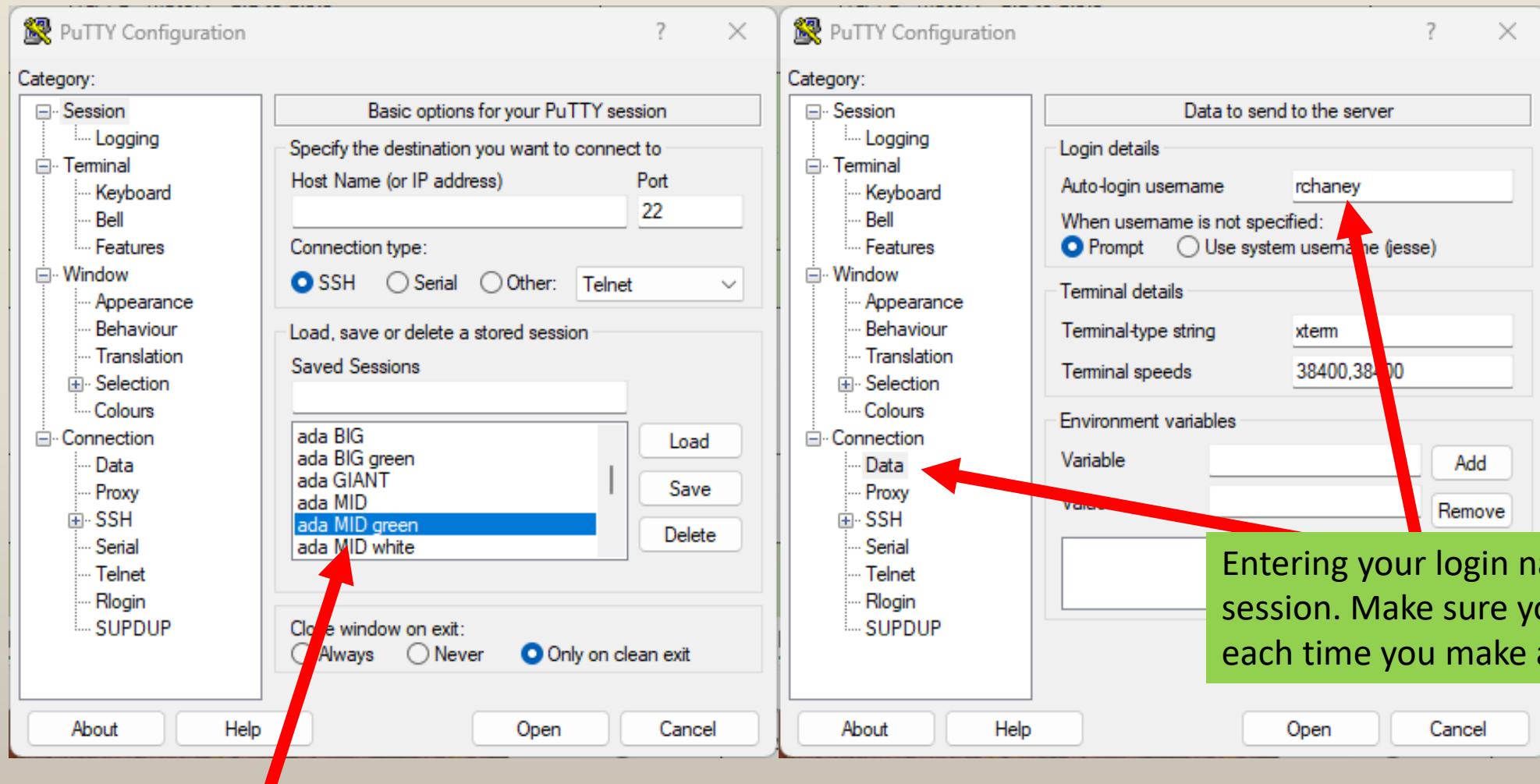
<https://www.putty.org/>
<http://ttyplus.com/multi-tabbed-putty/>

Logging In

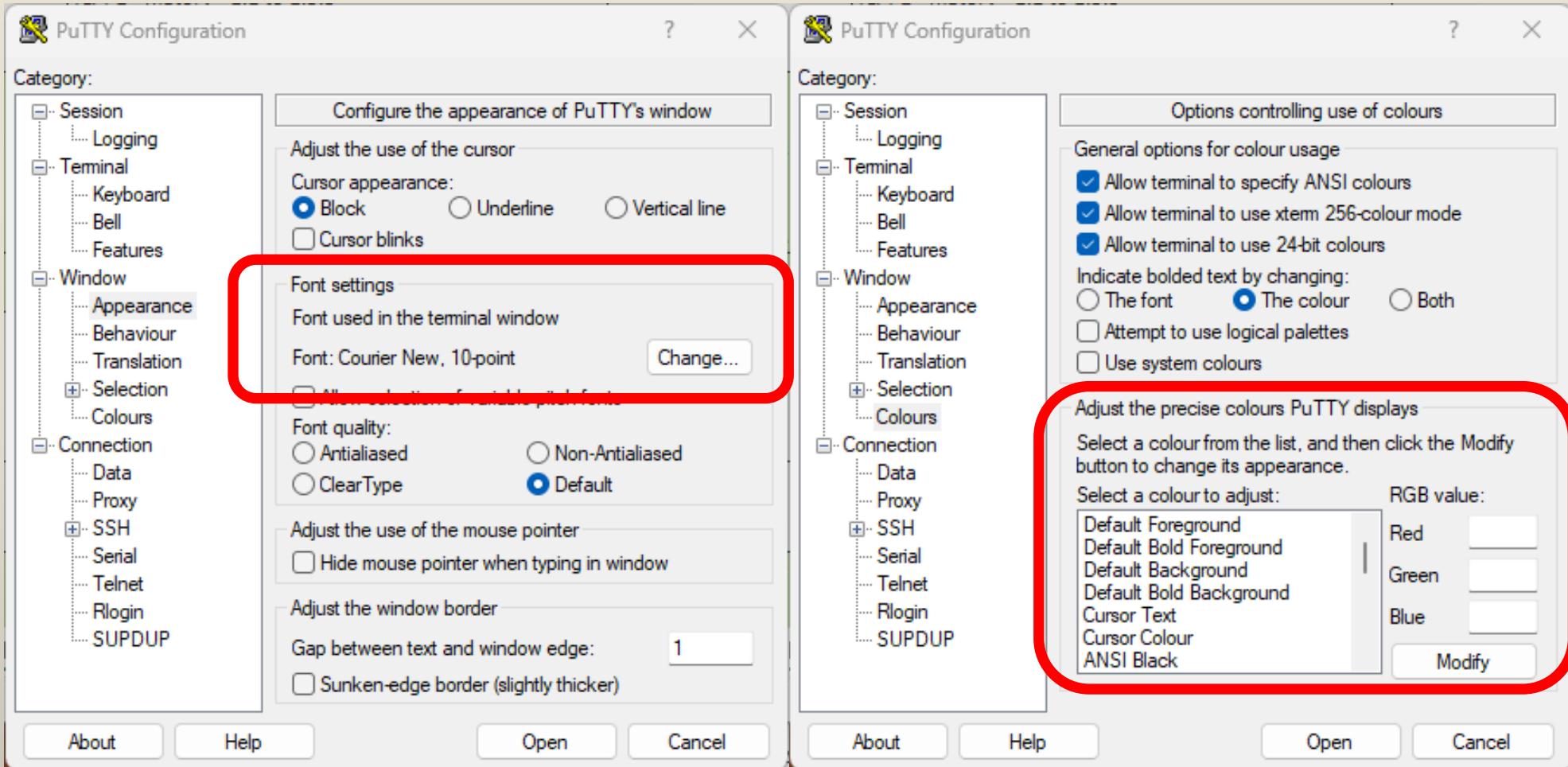
Everything in UNIX is **CaSe sEnSiTiVe**.

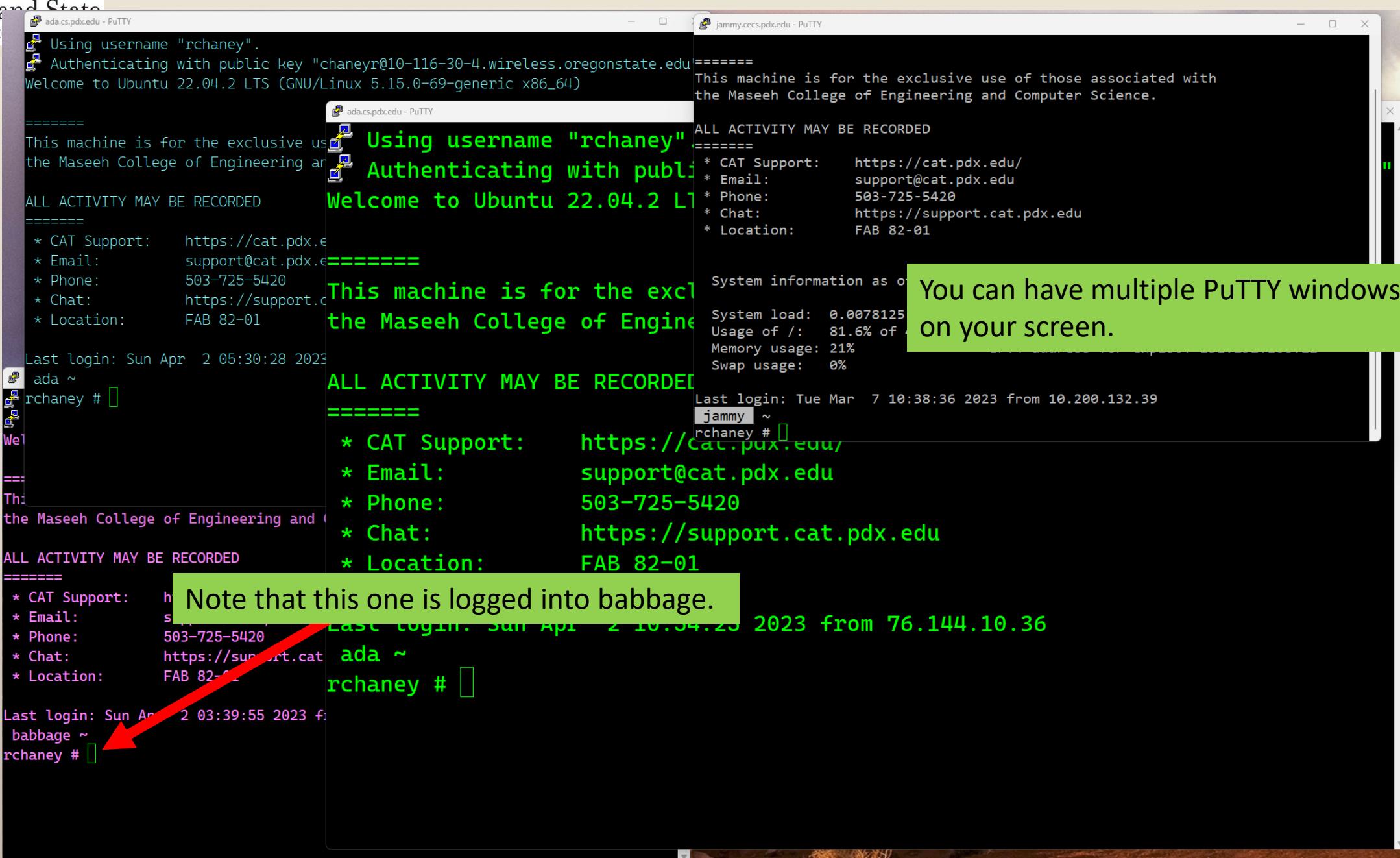
This includes user names, passwords, and commands.





You can easily create a custom login by saving the configuration under a name.

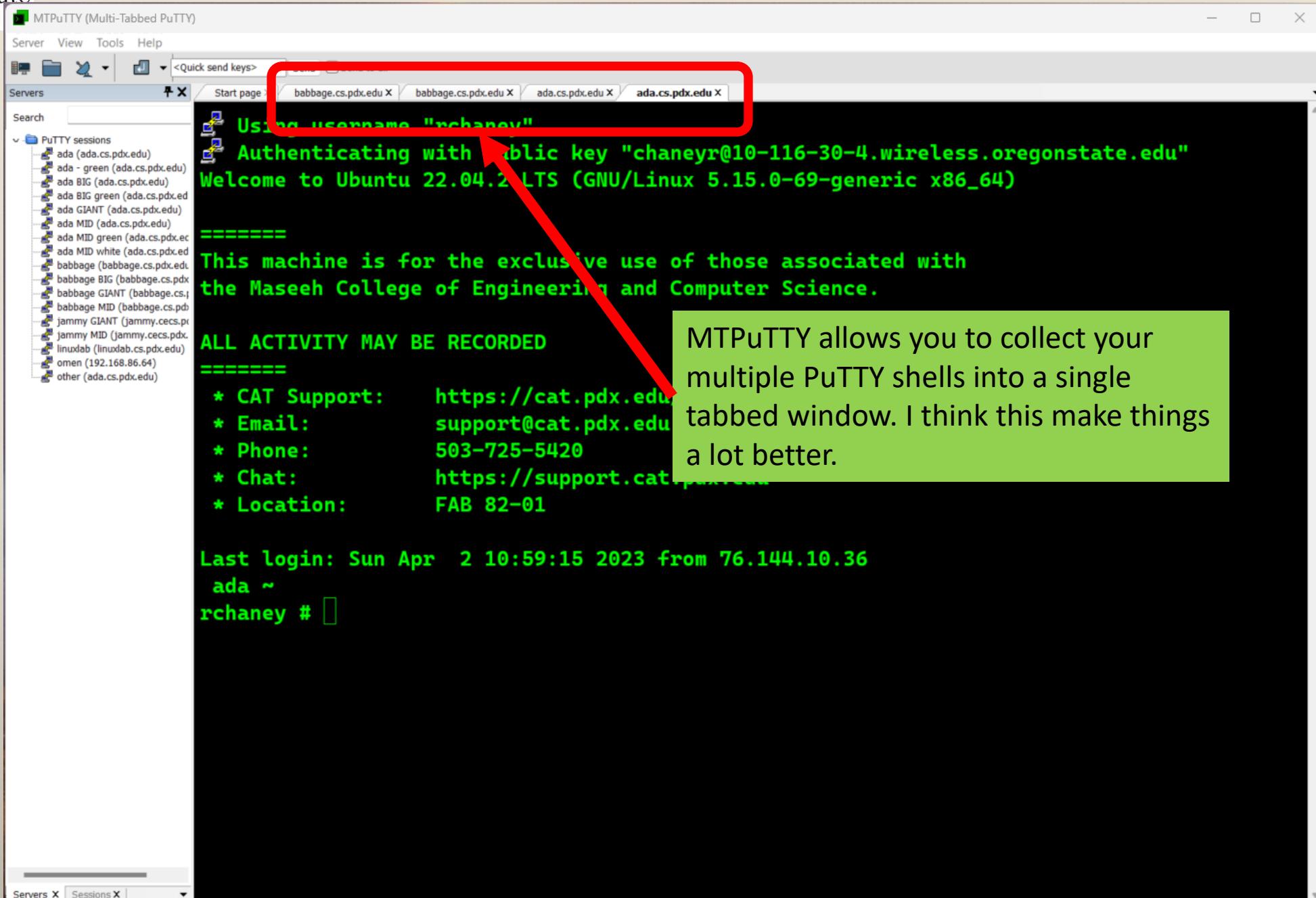


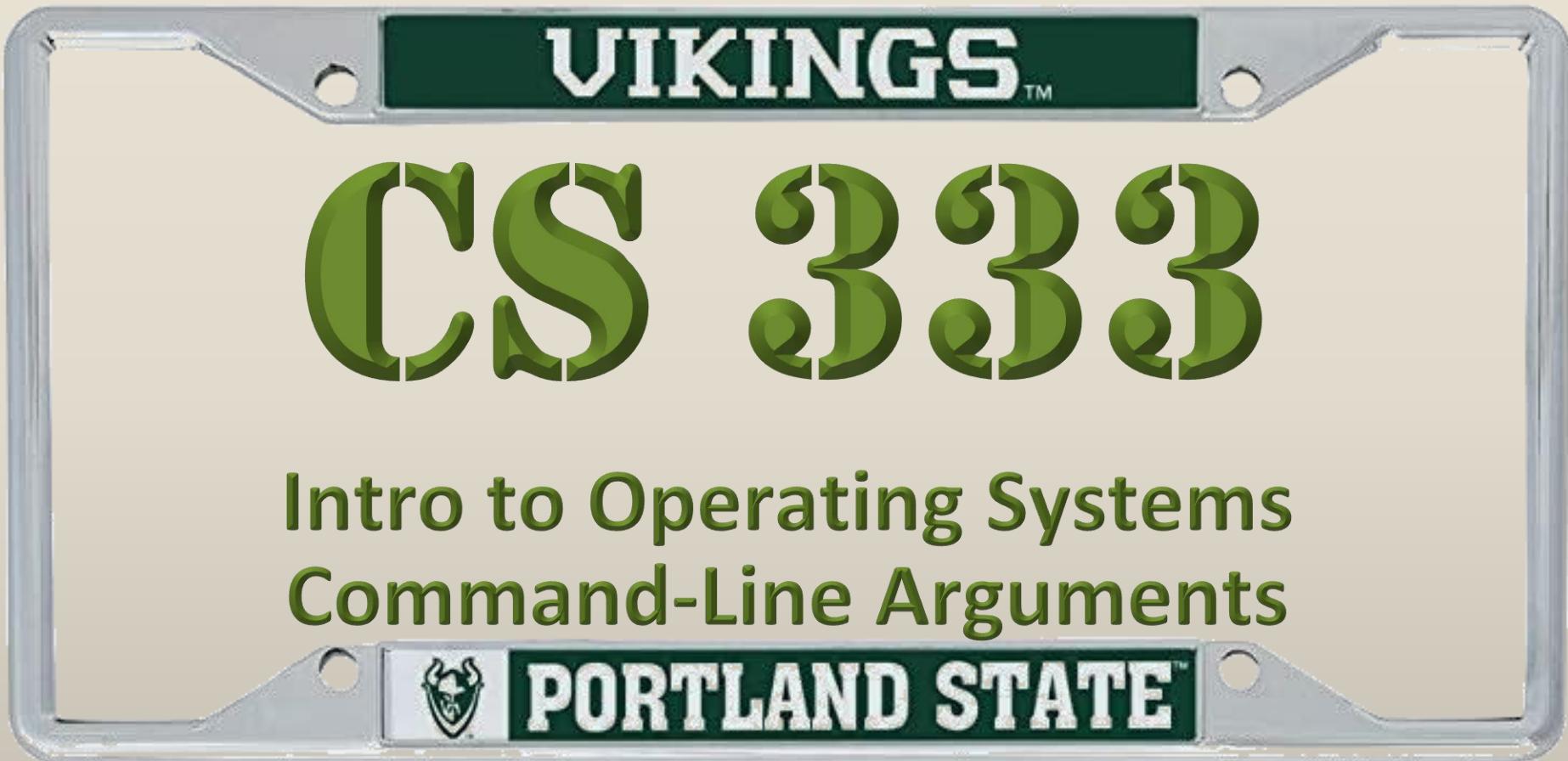


The screenshot shows four PuTTY windows running simultaneously on a Windows desktop. The windows are titled 'ada.cs.pdx.edu - PuTTY', 'jammy.cecs.pdx.edu - PuTTY', 'ada.cs.pdx.edu - PuTTY', and 'ada.cs.pdx.edu - PuTTY'. Each window displays a terminal session for the user 'rchaney'.

- Top Left Window:** Displays a login message for 'ada.cs.pdx.edu' with the IP '10.116.30.4'. It includes a 'CAT Support' link and a 'Last login' timestamp of Sun Apr 2 05:30:28 2023.
- Top Right Window:** Displays a message from 'jammy.cecs.pdx.edu' stating 'ALL ACTIVITY MAY BE RECORDED' and provides support contact information.
- Middle Left Window:** Displays a message from 'ada.cs.pdx.edu' with the IP '10.34.25.203'. It includes a 'CAT Support' link and a 'Last login' timestamp of Sun Apr 2 03:39:55 2023.
- Middle Right Window:** Displays a message from 'ada.cs.pdx.edu' with the IP '76.144.10.36'. It includes a 'CAT Support' link and a 'Last login' timestamp of Sun Apr 2 10:54:25 2023.

A red arrow points to the bottom-left terminal window, highlighting the fact that it is logged into the host 'babbage'. A green callout box with the text 'Note that this one is logged into babbage.' is positioned over this window. Another green callout box with the text 'You can have multiple PuTTY windows open on your screen.' is positioned over the top-right terminal window.

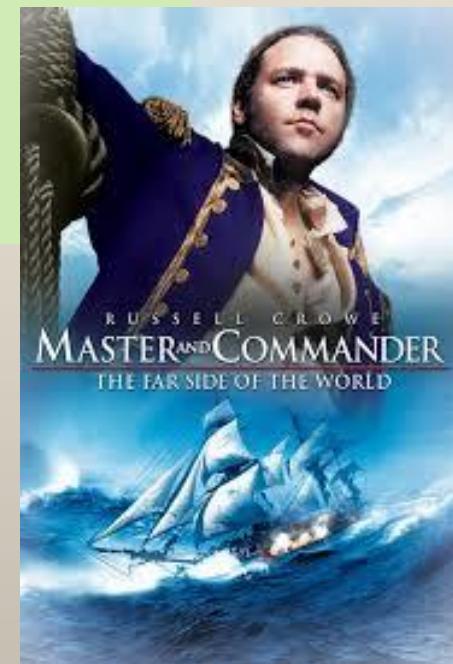




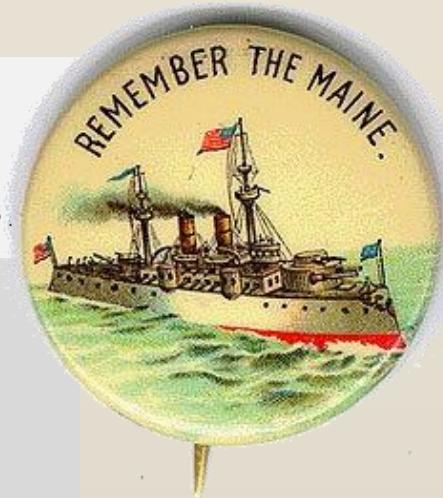
Intro to Operating Systems Command-Line Arguments

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

Examples can be found in
~rchaney/Classes/cs333/src/argc_argv



- Every C program must have a function called `main()`, which is the point where execution of the program starts.
- When the program is executed, the command line arguments (**the separate words parsed by the shell**) are made available via two arguments to the function `main()`.
- The shell is what does the meta-character evaluation. Your C program (**probably**) never sees the * or ? a user may place on the command line.
- Your shell, or login shell, is probably either bash or zsh.



Command Line Examples

A couple examples of command lines:

```
ls -l -a
```

Run the ls command, passing 2 command line arguments, -l and -a.

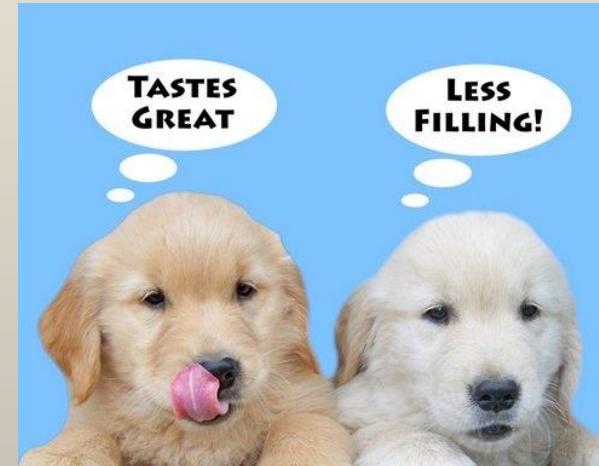
```
head -n 5 /etc/passwd
```

Run the head command, passing 2 command line arguments, -n. The -n command line argument has an option, the 5.

The second command line argument is the file name /etc/passwd.

It's really by convention that we continue call them argc and argv.
We could call them *yin* and *yang* or *Coke* and *Pepsi*. Or *TastesGreat* and *LessFilling*.

But, we **will** continue to call them argc and argv.



- The first argument, `int argc`, indicates **how many** command-line arguments are on the command-line.
- The second argument, `char *argv[]`, is an **array of pointers** to the command-line arguments, each of which is a **null-terminated character string**.



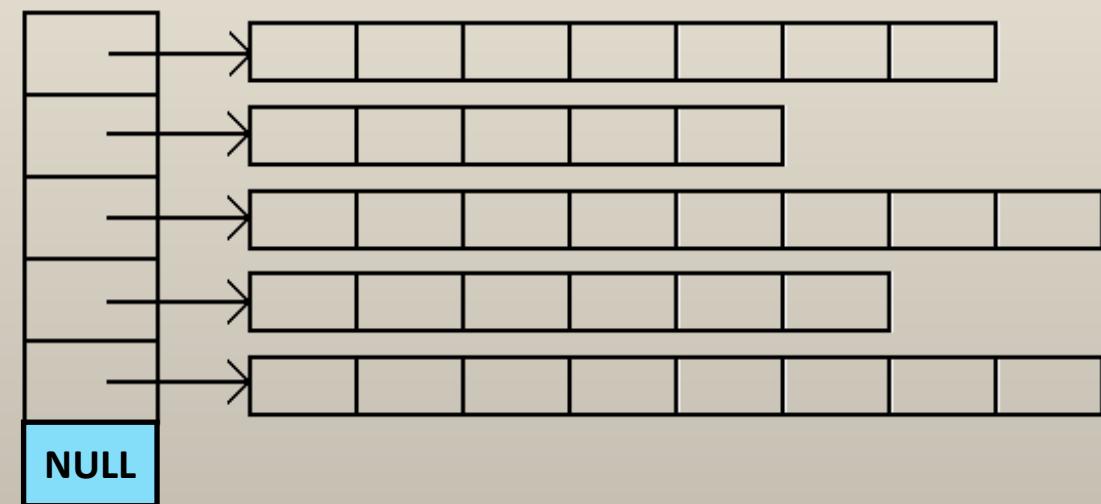
`argv[0]` is (almost) always the name of the program.

```
for (i = 0; i < argc; i++)
{
    printf("\tThe value of argv[%d] is: %s\n", i, argv[i]);
}
```

The **argv** parameter is exactly like a **ragged array**, except that it has an additional trailing **NULL** pointer at the end. Also known as a jagged array.

```
for (i = 1; NULL != argv[i]; i++)
{
    printf("\tThe value of argv[%d] is: %s\n", i, argv[i]);
}
```

The above code simply loops through `argv` and prints each item, except for `argv[0]`, the name of the program.



A regular array of C strings

a	b	c	NULL		
d	NULL				
e	f	g	h	i	NULL
j	k	l	NULL		
m	n	NULL			
o	p	q	r	NULL	
s	t	u	v	w	NULL

Every row has the same number of columns

A ragged array of C strings

a	b	c	NULL		
d	NULL				
e	f	g	h	i	NULL
j	k	l	NULL		
m	n	NULL			
o	p	q	r	NULL	
s	t	u	v	w	NULL

Each row has only the **necessary** number of columns

Environment Variables

Every UNIX process runs in a specific environment.

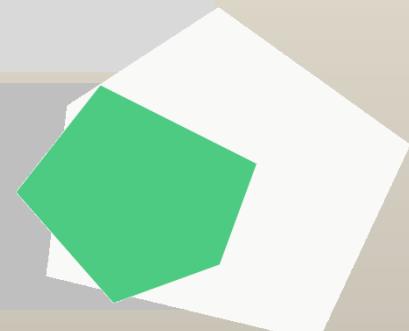
- An environment consists of a **table of environment variables**, each with an assigned value.
- When you log in certain login files are executed, which initialize the table holding the environment variables for the process.
- When this file passes the process to the shell, the table becomes accessible to the shell.
- When a parent process starts up a child process, the **child process is given a copy of the parent's environment table**.
- Environment variable names are generally given in **upper case**, by convention.

Environment Variables

The environment with which a process starts is **inherited** from the shell/process in which it was started.

You can easily see what your shell environment is by issuing the command `printenv` or `env` from your shell.

Your environment variables contain a large number **interesting** and **useful** information.



interesting

Environment Variables

```
// This is the newer, better, cooler way to handle environment
// variables
#include <unistd.h> // POSIX stuff.
extern char **environ;

for (i = 0; NULL != environ[i]; i++)
{
    printf("\tThe value of environ[%d] is: %s\n", i, environ[i]);
}
```

Environment Variables

```
// When using envp (from main), this will not be
// found in your environment. The envp data are static.
// If you use the environ external variable, you
// will find these.
```

```
putenv("ENVIRONMENT_TEST=test_value");
putenv("HOME=test_value");
```

```
new_env = getenv("ENVIRONMENT_TEST");
new_env = getenv("HOME");
```



SAVE ENVIRONMENT

Processing the Command Line

Processing the command line yourself can be challenging.

1. Are there command line options?
2. Do the options have arguments?
3. Can no-argument options be grouped?
4. Can the options be given in any order?
5. Are there things on the command line other than options with/without arguments?

Luckily, there's an app for that!



The getopt() Library Function

```
#include <unistd.h>

int getopt(int argc
           , char * const argv[]
           , const char * optstring);

extern char *optarg;
extern int optind, opterr, optopt;
```

The getopt() function makes your life better.

These are magic global getopt() variables.

The getopt() library function **parses** the command-line.

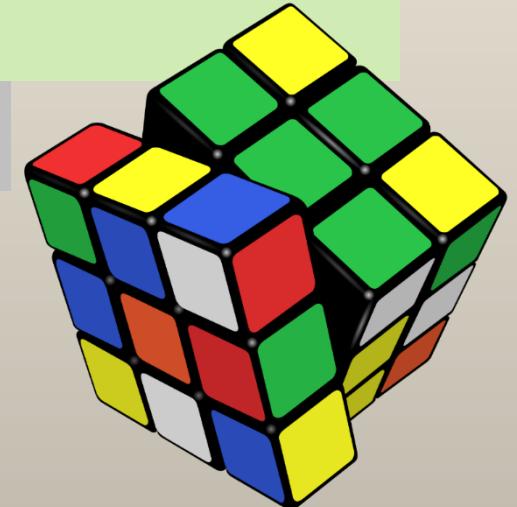
Its arguments `argc` and `argv` are the argument count and array as passed to the `main()` function on program invocation.

- An element of `argv` that starts with '`-`' (and is not exactly "`-`" or "`--`") is an option element.

- **optstring** is a string containing the legitimate option characters.
 - If such a character is **followed by a single colon**, the option **requires** an argument, so getopt () places a pointer to the following text in the same argv-element, or the text of the following argv-element, in optarg.
 - **Two colons** mean an option takes an **optional arg**; if there is text in the current argv-element (i.e., in the same word as the option name itself, for example, "-oarg"), then it is returned in optarg, otherwise optarg is set to zero.
- The variable **optind** is the index of the next element to be processed in argv. The system initializes this value to 1.

- By default, getopt () **permutes** the contents of argv as it scans, so that eventually **all the non-options are at the end.**
- If getopt () does not recognize an option character, it prints an error message to stderr, stores the character in optopt, and returns '?'.

Permute: to rearrange.



```

while ((opt = getopt(argc, argv, "os:i:")) != -1) {
    switch (opt) {
        case 'o':
            o_opt++; // Increment the variable each time the -o option is seen.
            printf("The -o option has been seen: %d\n", o_opt);
            break;
        case 's':
            strcpy(s_opt, optarg);
            printf("The -s option has been seen with argument %s\n", s_opt);
            break;
        case 'i':
            i_opt = (int) strtol(optarg, NULL, 10);
            printf("The -i option has been seen with argument %d\n", i_opt);
            break;
        default:
            printf("something strange has happened\n");
            break;
    }
}
  
```

The **colon** in the list of command line options means an argument is **required** for that option.

Magic variable that contains the string for the argument to a command line option.

Examples of **valid** command lines for this program are:

prog -s str -i17
 prog -i5 -o -sStr
 prog -osStr

Example of **invalid** command lines are:

prog -s
 prog -s str -i



What might remain on the command line after getopt() is done chewing on it?

```
if (optind < argc)
{
    int j;

    fprintf(stderr, "\nThis is what remains on the command line:\n");
    for(j = optind; j < argc; j++) {
        printf("\t%s\n", argv[j]);
    }
}
```

Magic variable that contains the index from argv that is just **past** the last command line option (and argument) that was processed by getopt().

Examples of other stuff on the command line are:

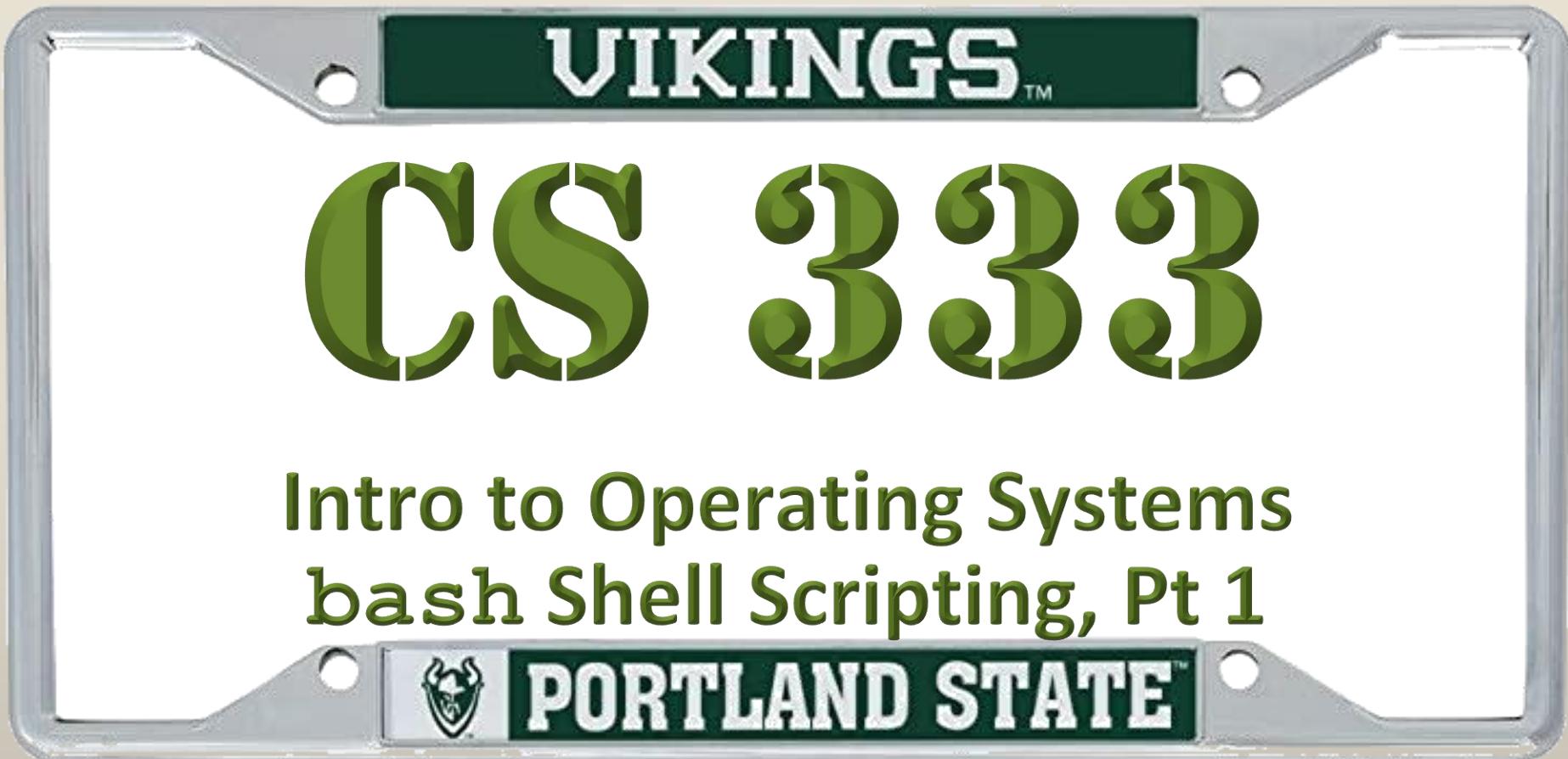
```
prog -osStr stuff1 stuff2 moreStuff
prog someStuff -s str -i17
prog -i5 oddStuff -o -sStr
```



Man Page for getopt

- The man page for getopt (`man 3 getopt`) not only contains an excellent description of how getopt works, but it also contains **a terrific example** of its use.
- I often start a new program by copying and pasting the example from the getopt man page into my code.
- Appendix B from TLPI also has a description for how getopt works.
- Learning to use getopt will make your life better and easier.



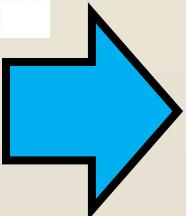




KEEP
CALM
AND
BASH
BASH BASH

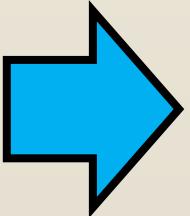
All/Many/Most/Several of the scripts
mentioned in this set of slides are
available in

```
~rchaney/Classes/cs333/src/bash
```



- The Hello World bash script
 - The first line of your script
 - Comments
 - Setting execute permissions on your script
- Calling Linux commands from within your script
- Using Variables
 - Quotes
 - Double quotes - "
 - Single quote - '
 - Back quote - ` (above the tab key, aka **grave** character)
 - Arithmetic on variables
- Conditionals
 - Use of if and case
- Looping
 - Regular Expressions
- Functions





Commands that we'll cover

- echo – might be a shell built-in
- test
- grep
- find
- tr
- and so much more...



bash built-ins that we'll cover

- exit
- getopt – covered in `hello_world_GETOPTS.bash`
- if
- for loops
- while loops
- Function calls

Shell Scripting

- All of the commands that are accessible from the shell prompt can be placed into a shell “script”.
- Shell scripts are executed line by line, as if they were being typed in line by line.

Shell Scripting

Shell scripts

- High level programming language
 - Variables, conditionals, loops, etc.
- Interpreted
 - No compilation required, no variable declarations, no memory management
- Powerful and efficient
 - You can do lot with very few lines of code
 - Can easily interface with any UNIX program that uses `stdin`, `stdout`, `stderr`
- String and file oriented

When Can You Use Shell Scripts?

- Automate frequent UNIX tasks
- Simplify complex commands
- For small programs that:
 - you want to create quickly
 - you will change frequently
 - you need portable between different *nix systems
 - you want others to see and understand easily
 - **testing**
- As a **glue** to connect together other programs

Environment Variables

- A set of variables that are always available in the shell (or other programs)
- Environment variables control options that change the operation of the shell or contain information
- Here are a few common ones:
 - **PATH** - the set of directories bash will search through to find a command
 - **HOME** – the path back to your home directory
 - **SHELL** - the full path to the default shell
 - **HOSTNAME** - the name of the system you're currently using

Special Environment Variables

Some Special bash variables (there are others)

- **\$?** - Expands to the exit status of the most recently executed foreground process. You'll use this one a lot when you are writing testing scripts.
 - An exit status of 0 typically indicates that the previous command completed successfully.
 - An exit status other than zero indicates some type of error.
- **\$\$** - Expands to the process ID of the shell.
- **\$0** - Expands to the name of the shell or shell script.
- **\$#** - the number of arguments (positional parameters) given when a script is executed .

Using \$#

The \$# environment variable represents the number of parameters passed to the script on the command line (a lot like argc in C programming).

```
#!/bin/bash
# if there are no cmd line params, exit with a 1
if [ $# -eq 0 ]
then
    exit 1
elif [ $# -eq 1 ]
then
    # if the num of cmd line params is 1, exit with 0
    exit 0
else
    # exit with the number of command line parameters
    exit $#
fi
```

This example jumps ahead a number of slides, you may wish to return to it later, after reviewing the entire slide deck.

exitGen.bash

Using \$?

The \$? environment variable represents the exit value for the previously completed command.

```
#!/bin/bash

exitGen.bash
echo $?

exitGen.bash 1
echo $?

exitGen.bash 2 3 4 5
echo $?
```

This script does not exist, but a better more complete one does exist:

exitStatus.bash

The script does includes features covered later in the slide set.

View Your Environment Variables

If you want to see a list of all of your environment variables, use the command `printenv`.

Using `printenv` by itself will show all of your environment variables.

You can also follow `printenv` with specific variable name(s) to show only the value for that variable(s):

```
printenv HOME
```

You can also use the `echo` built-in to show the value of an environment variable:

```
echo $HOME
```

You must use the `$` character in front of the variable name with `echo`.

The Hello World bash Script

```
#!/bin/bash
echo "Hello World"
```

All bash scripts start with this first line. The path given should be the location of the bash executable. This can vary from system to system. Sometimes, it is /usr/bin/bash.

The echo command simply displays the following text to the terminal (stdout).

hello_world.bash



The Hello World bash Script

```
#!/bin/bash

# This is a comment.
# Comments begin with the # character
# and continue to end of line.

echo "Hello World" # Comments do not
# have to begin at the start of a line.
```

Comments begin with the # character and continue to the end of the line.

There is no form for block comments like C has with /* */.

hello_world-wComments.bash

R. Jesse Chaney

Hello
World

- The # character is called
- Pound (this is what I in a C program)
 - Hash or hashtag
 - Sharp (if you are musically inclined)
 - Number sign
 - Octothorpe (what I sometimes call it)

The Hello World bash Script

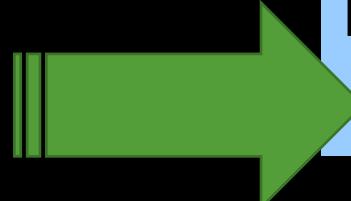
Before you can run your script, you must set the execute permissions on the script.

I like to give the `.bash` file name extension to my bash scripts. Some people use `.sh`. I use `.sh` with older style Bourne Shell scripts (`/bin/sh`).

```
jesse.chaney@brynhildr-new: ~/bin
(~/.bin)
jesse.chaney@brynhildr-new 12:11 AM $ chmod a+x hello_world.bash
( /bin )
jesse.chaney@brynhildr-new 12:11 AM $ ls -la hello_world.bash
-rwxr-xr-x 1 jesse.chaney faculty 32 Jan 13 00:03 hello_world.bash
( /bin )
jesse.chaney@brynhildr-new 12:11 AM $
```

I'm calling my script `hello_world.bash`.

Run:



`chmod a+x *.bash`

The Hello World bash Script

```
jesse.chaney@brynhildr-new: ~/bin  
(~/bin)  
jesse.chaney@brynhildr-new 12:18 AM $ ./hello_world.bash  
Hello World  
(~/bin)  
jesse.chaney@brynhildr-new 12:18 AM $ cat hello_world.bash  
#!/bin/bash  
  
# This is a comment.  
# Comments begin with the # character  
# and continue to end of line.  

```

Notice how I start the command with a ./ That means “Look in the current directory for an executable program called hello_world.bash and execute it.”

Calling Linux Commands from within Your Shell Script

```
#!/bin/bash

last | head -5
echo
date
```

The way you run Linux commands from within your shell script is to just put them in the text.

If the command requires user input, you'll still need to supply that (somehow).

last - show listing of last logged in users

hello_world_COMMANDS.bash



```
jesse.chaney@brynhildr-new: ~/bin

(~/.bin)
jesse.chaney@brynhildr-new 01:18 AM $ ./hello_world_COMMANDS.bash
jesse.ch pts/3          68-186-9-194.dhc Wed Jan 13 00:27    still logged in
jesse.ch pts/0          68-186-9-194.dhc Wed Jan 13 00:10    still logged in
jesse.ch pts/1          68-186-9-194.dhc Tue Jan 12 23:45    still logged in
tristan. pts/0          75-142-136-118.d Tue Jan 12 23:35 - 23:59  (00:23)
thomas.m pts/4          75-142-136-118.d Tue Jan 12 22:56 - 23:34  (00:38)

Wed Jan 13 01:18:29 PST 2016
(~/.bin)
jesse.chaney@brynhildr-new 01:18 AM $ █
```

Using Variables

Let's count how many spaces there are around the assignment to the variable.

```
#!/bin/bash  
  
STR="Hello World"  
  
echo $STR  
echo ${STR} again
```

Creating and assigning a value to the variable **STR**

Using the **STR** variable requires we put a \$ in front of it.

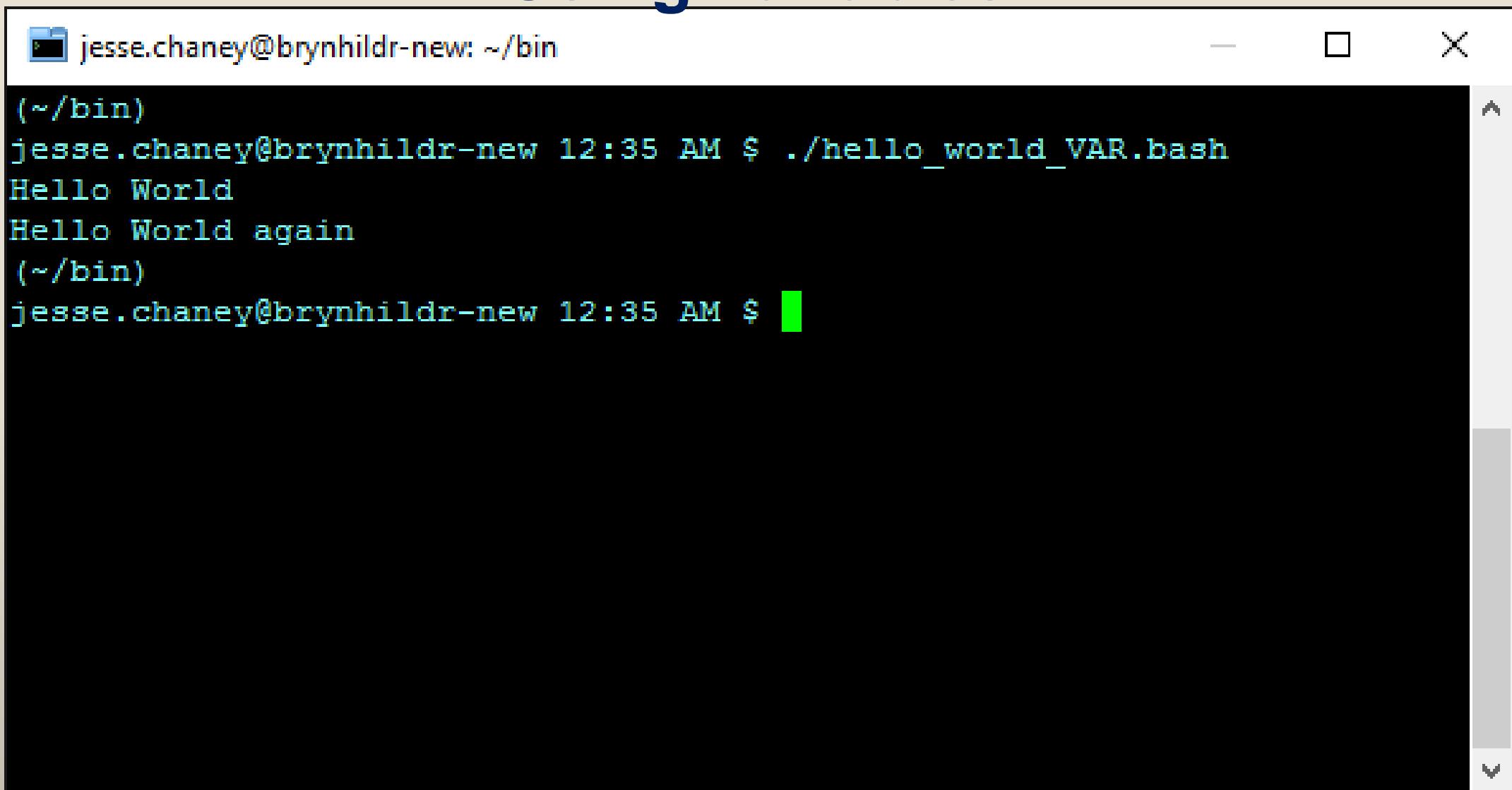
You can also put curly braces around a variable name. I tend to use the braces a lot.

hello_world_VAR.bash



VARIABLE

Using Variables



The screenshot shows a terminal window with the following session:

```
jesse.chaney@brynhildr-new: ~/bin
(~/.bin)
jesse.chaney@brynhildr-new 12:35 AM $ ./hello_world_VAR.bash
Hello World
Hello World again
(~/.bin)
jesse.chaney@brynhildr-new 12:35 AM $ █
```

The terminal window has a dark background and light-colored text. It includes standard window controls (minimize, maximize, close) and scroll bars on the right side.

Quotes

```
#!/bin/bash
```

Variables within double quotes
will be expanded.

```
STR1="Hello World"
```

```
STR2="This is within double ${STR1} quotes"
```

```
STR3='This is within single ${STR1} quotes'
```

```
echo $STR1
```

```
echo $STR2
```

```
echo $STR3
```

Variables within single quotes will
NOT be expanded.

hello_world_QUOTES.bash



Quotes

```
jesse.chaney@brynhildr-new: ~/bin  
  
(~/.bin)  
jesse.chaney@brynhildr-new 12:39 AM $ ./hello_world_QUOTES.bash  
Hello World  
This is within double Hello World quotes  
This is within single ${STR1} quotes  
  
(~/.bin)  
jesse.chaney@brynhildr-new 12:39 AM $
```

Variables within double quotes will be expanded.

This is within double Hello World quotes

This is within single \${STR1} quotes

Variables within single quotes will **NOT** be expanded.

Back Quotes



```
#!/bin/bash
STR1=`date`
STR2=`last | head`
echo "Today's date is \$STR1"
echo -e "The 10 most recent users to log in are\n\$STR2"
```

Back quotes (sometimes called back-ticks) mean:
take whatever is within the back quotes, run it,
and return the result into the variable.

Back quotes are an easy way to
run a command and capture the
result into a variable.



The -e tells echo to recognize the embedded
newline escape sequence.

The backquote character is actually called
a grave (short A, not a long A)

There is an alternative syntax for the
command evaluation. You can also
put a command within \$()

hello_world_BACKQUOTES.bash

Back Quotes

```
jesse.chaney@brynhildr-new: ~/bin

(~/.bin)
jesse.chaney@brynhildr-new 12:55 AM $ ./hello_world_BACKQUOTES.bash
Today's date is Wed Jan 13 00:55:28 PST 2016
The 10 most recent users to log in are
jesse.ch pts/3          68-186-9-194.dhc Wed Jan 13 00:27      still logged in
jesse.ch pts/0          68-186-9-194.dhc Wed Jan 13 00:10      still logged in
jesse.ch pts/1          68-186-9-194.dhc Tue Jan 12 23:45      still logged in
tristan. pts/0          75-142-136-118.d Tue Jan 12 23:35 - 23:59 (00:23)
thomas.m pts/4          75-142-136-118.d Tue Jan 12 22:56 - 23:34 (00:38)
tristan. pts/3          75-142-136-118.d Tue Jan 12 22:41 - 23:34 (00:53)
jesse.ch pts/4          68-186-9-194.dhc Tue Jan 12 21:13 - 22:00 (00:46)
jesse.ch pts/3          68-186-9-194.dhc Tue Jan 12 21:12 - 21:58 (00:46)
jessica. pts/1          96-41-175-237.dh Tue Jan 12 20:31 - 23:03 (02:32)
jessica. pts/0          96-41-175-237.dh Tue Jan 12 20:23 - 23:03 (02:40)
(~/.bin)
jesse.chaney@brynhildr-new 12:55 AM $
```

Arithmetic on Variables

```
#!/bin/bash

COUNT=1
echo "COUNT starts off as equal $COUNT"

COUNT=$((COUNT + 1)) ←
echo "COUNT plus 1 = $COUNT"

COUNT=$((COUNT * 2)) ←
echo "COUNT times 2 = $COUNT"

OTHER=6
COUNT=$((COUNT + OTHER)) ←
echo "COUNT with $OTHER added = $COUNT"
```

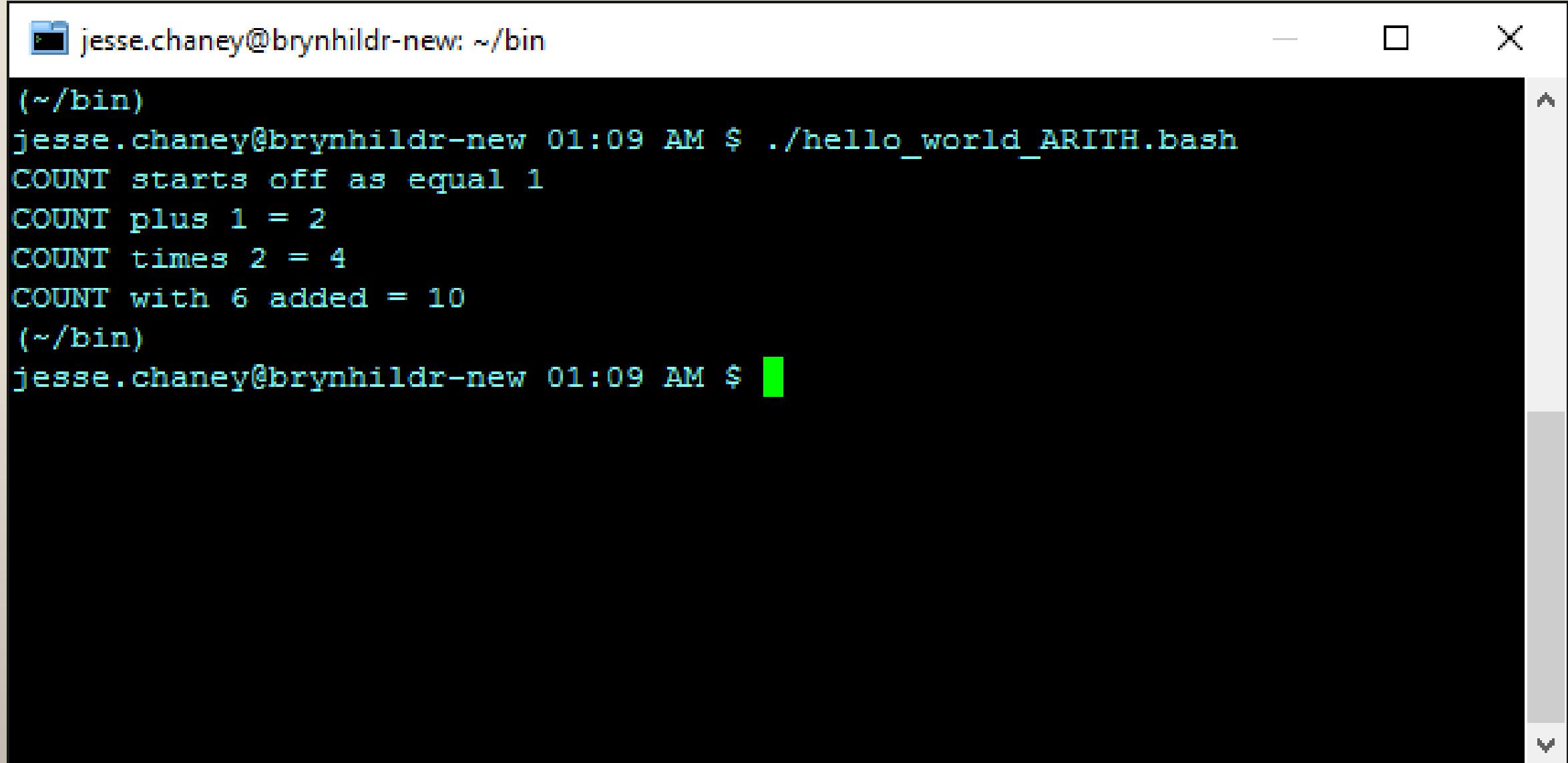
arithmetic mean =
$$\frac{\sum_{n=1}^k x_n}{k}$$

I have to admit that arithmetic on variables in bash is ... awkward.

All the extra parenthesis are required.

There are some alternative *syntaxes* for arithmetic evaluation.

Arithmetic on Variables



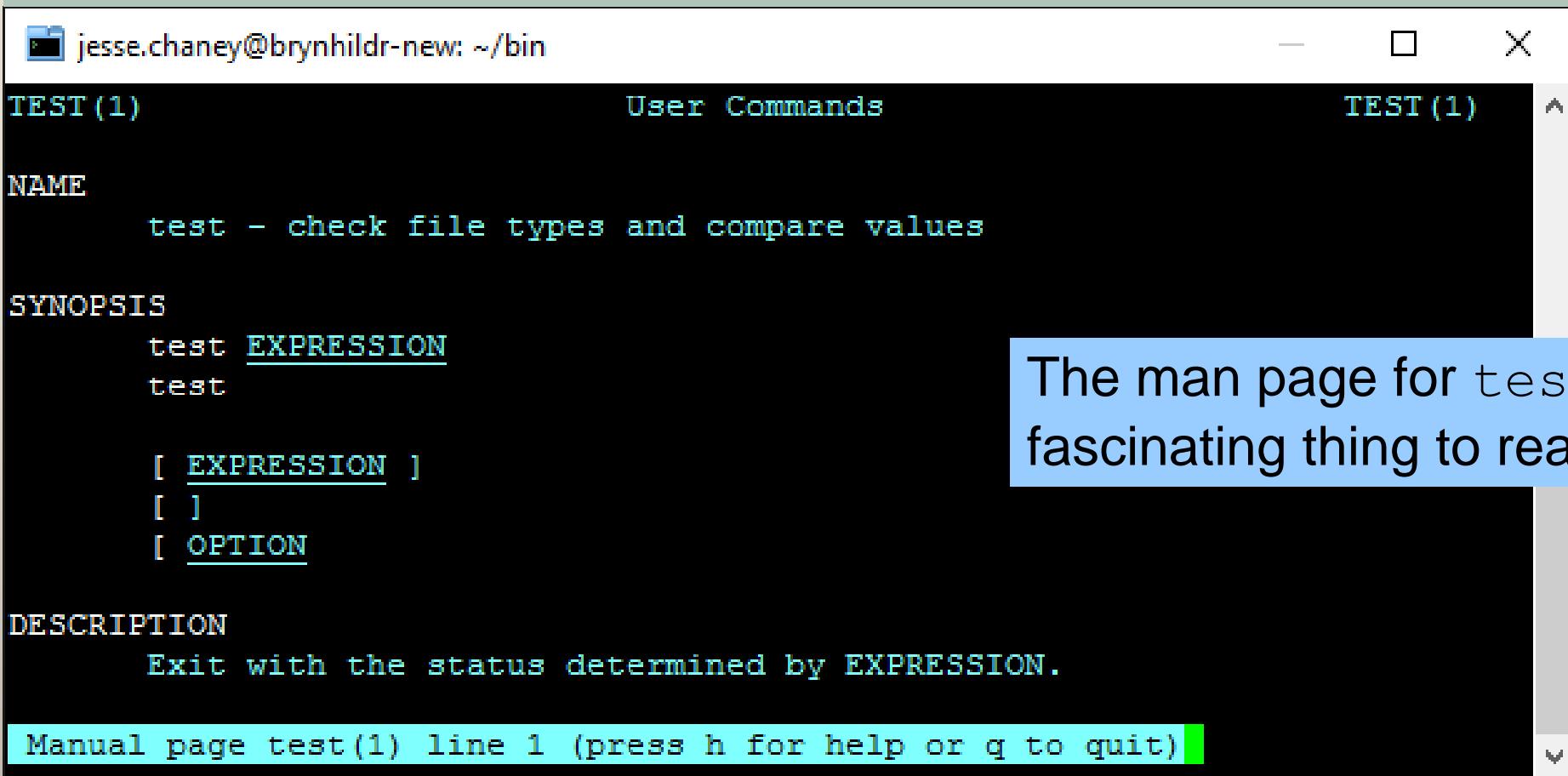
The screenshot shows a terminal window with the following session:

```
jesse.chaney@brynhildr-new: ~/bin
(~/.bin)
jesse.chaney@brynhildr-new 01:09 AM $ ./hello_world_ARITH.bash
COUNT starts off as equal 1
COUNT plus 1 = 2
COUNT times 2 = 4
COUNT with 6 added = 10
(~/.bin)
jesse.chaney@brynhildr-new 01:09 AM $ █
```

The terminal window has a dark background and light-colored text. It includes standard window controls (minimize, maximize, close) in the top right corner.

Conditionals - if

Conditionals in bash scripts are built using the syntax from the test command. The test command is almost never used by itself, but its syntax is used in conditionals.



The terminal window shows the man page for the test command. The title bar says "TEST (1)" and "User Commands". The page content includes:

NAME
test - check file types and compare values

SYNOPSIS
test EXPRESSION
test
[EXPRESSION]
[]
[OPTION]

DESCRIPTION
Exit with the status determined by EXPRESSION.

At the bottom of the terminal window, a blue box contains the text: "The man page for test is a fascinating thing to read."

```
jesse.chaney@brynhildr-new: ~/bin
TEST (1)                               User Commands                         TEST (1)
NAME
    test - check file types and compare values
SYNOPSIS
    test EXPRESSION
    test
    [ EXPRESSION ]
    [ ]
    [ OPTION ]
DESCRIPTION
    Exit with the status determined by EXPRESSION.
The man page for test is a
fascinating thing to read.
Manual page test(1) line 1 (press h for help or q to quit)
```

Conditionals - if

```
#!/bin/bash
if [ "foo" = "foo" ]
then
    echo "Expression is true"
fi
```

Compare to strings for equality

Do not omit spaces between operators and operands or brackets.

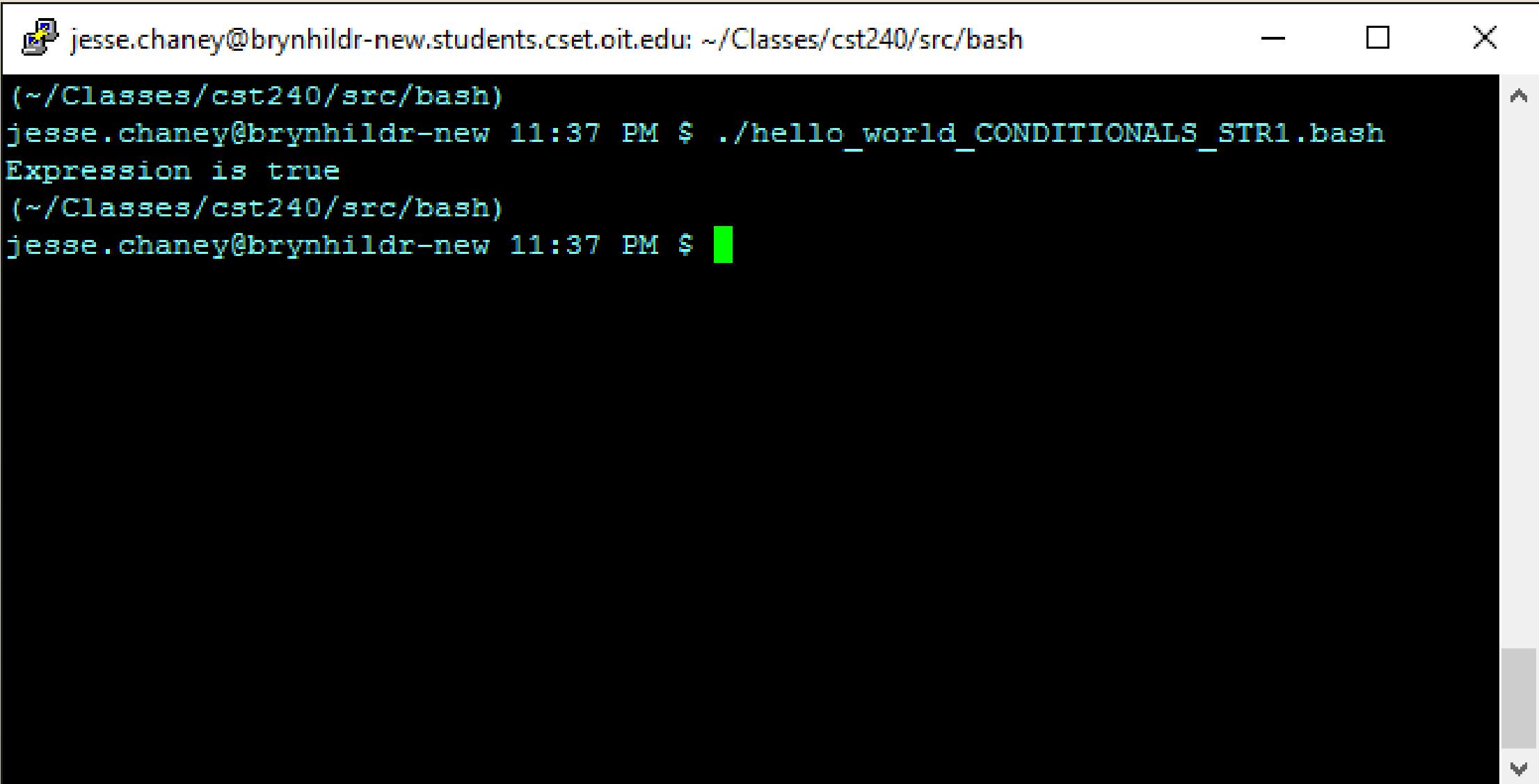
Notice that string comparison is done with the = sign, not ==.

The `if` statement is terminated with a `fi`.

hello_world_CONDITIONALS_STR1.bash

fee, fi, fo, fum!

Conditionals - if



The terminal window shows the following session:

```
jesse.chaney@brynhildr-new: ~/Classes/cst240/src/bash
(jesse.chaney@brynhildr-new 11:37 PM) $ ./hello_world_CONDITIONALS_STR1.bash
Expression is true
(jesse.chaney@brynhildr-new 11:37 PM) $
```

Conditionals - if

```
#!/bin/bash

if [ "foo" = "bar" ]
then
    echo "Expression is true"
else
    echo "Expression is false"
fi
```

You can use if/then/else logic as well.

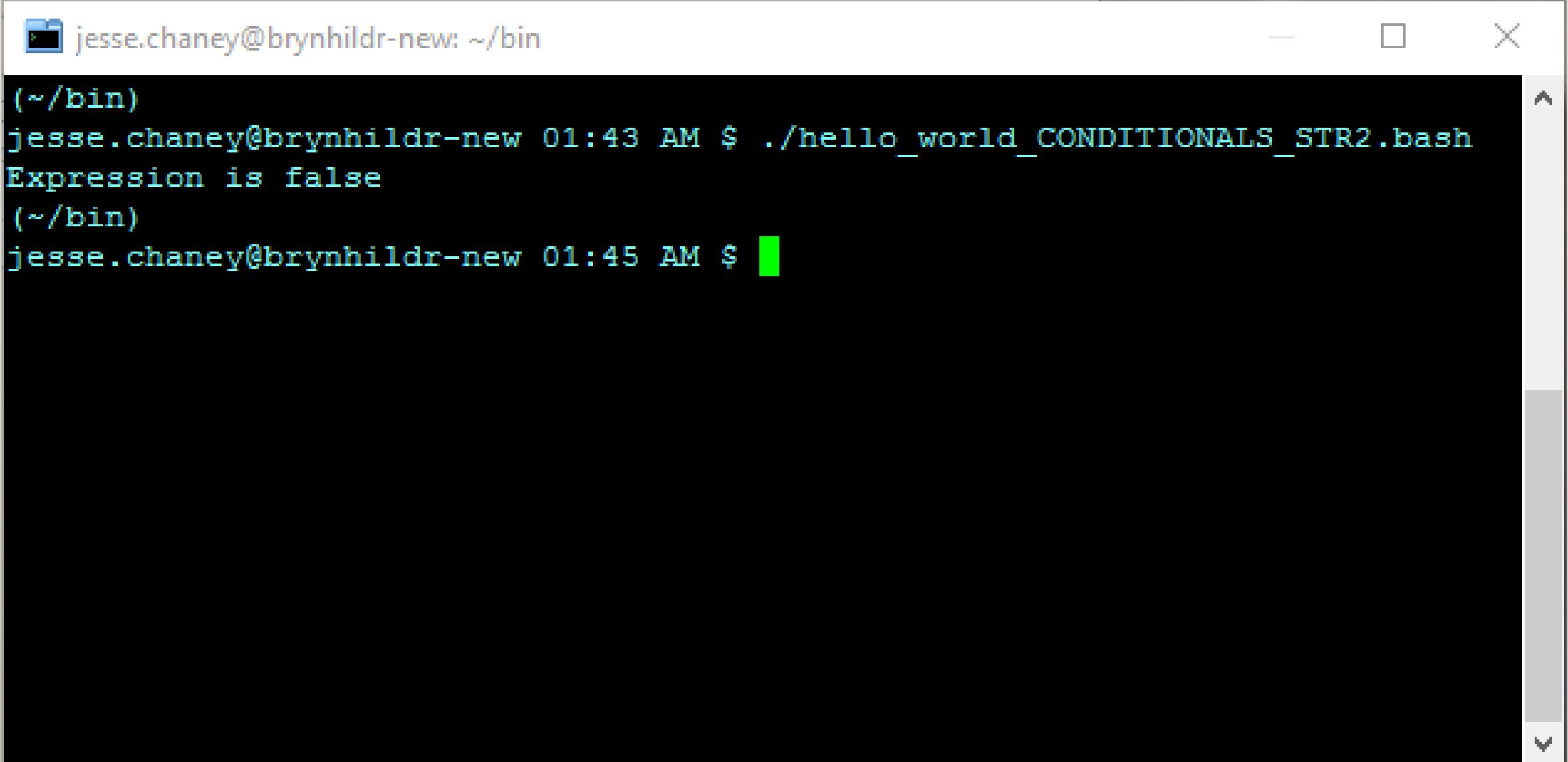
You can also use the bash elif for larger nested if/then/else statements.

hello_world_CONDITIONALS_STR2.bash



CS333 Intro Op Sys

Conditionals - if



The terminal window shows the following session:

```
jesse.chaney@brynhildr-new: ~/bin
(~/.bin)
jesse.chaney@brynhildr-new 01:43 AM $ ./hello_world_CONDITIONALS_STR2.bash
Expression is false
(~/.bin)
jesse.chaney@brynhildr-new 01:45 AM $ █
```

The terminal window has a dark background and light-colored text. It includes standard window controls (minimize, maximize, close) and scroll bars on the right side.

Conditionals – test

This is not a comprehensive list.

Operator	Description
-n STRING	The length of STRING is greater than zero.
-z STRING	The length of STRING is zero (ie it is empty).
STRING1 = STRING2	STRING1 is equal to STRING2
STRING1 != STRING2	STRING1 is not equal to STRING2
INTEGER1 -eq INTEGER2	INTEGER1 is numerically equal to INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 is numerically greater than INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 is numerically less than INTEGER2
INTEGER1 -ge INTEGER2	INTEGER1 is numerically greater or equal to INTEGER2
INTEGER1 -le INTEGER2	INTEGER1 is numerically less or equal to INTEGER2
-d FILE	FILE exists and is a directory.
-e FILE	FILE exists.
-s FILE	FILE exists and its size is greater than zero (ie. it is not empty).
-r FILE	FILE exists and the read permission is granted.
-x FILE	FILE exists and execute (or search) permission is granted.

String comparisons

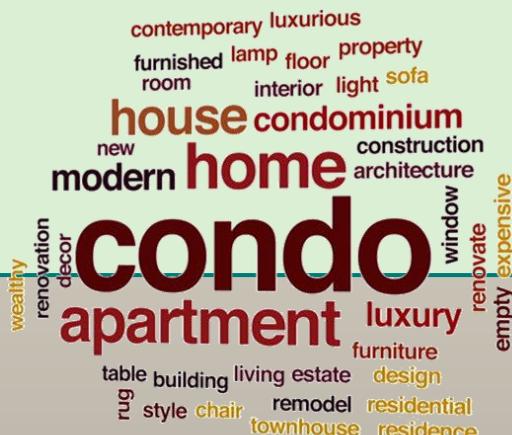
Notice the difference between how you do string comparisons and integer comparisons.

Conditionals - if

```
#!/bin/sh

FILE="hello world.bash"
if [ -e $FILE ]
then
    echo "The file $FILE exists"
    if [ -s $FILE ]
    then
        echo "The file $FILE has a size greater than zero"
    fi
else
    echo "The file $FILE does not exist"
fi
```

hello_world_CONDITIONALS_FILE.bash



Conditionals - if

```
jesse.chaney@brynhildr-new: ~/bin
(~/.bin)
jesse.chaney@brynhildr-new 02:05 AM $ ./hello_world_CONDITIONALS_FILE.bash
The file hello_world.bash exists
  The file hello_world.bash has a size greater than zero
(~/.bin)
jesse.chaney@brynhildr-new 02:05 AM $
```

Conditionals - case

Case statements are handy when you have several conditions to test and don't want to build up a large number of if/then/else statements.

Matches are made on a pattern.

The commands that execute must end with a double semicolon.

The entire case statement ends with an esac line.



Conditionals - case

Begin the case statement



hello_world_CASE1.bash

```
#!/bin/bash
```

```
VALUE=5
```

```
case $VALUE in
```

```
0)
```

```
    echo "A zero"
```

```
5)
```

```
    echo "A 5"
```

```
    ;;
```

```
10)
```

```
    echo "A 10"
```

```
    ;;
```

```
*)
```

```
    echo "A strange value"
```

```
    ;;
```

```
esac
```

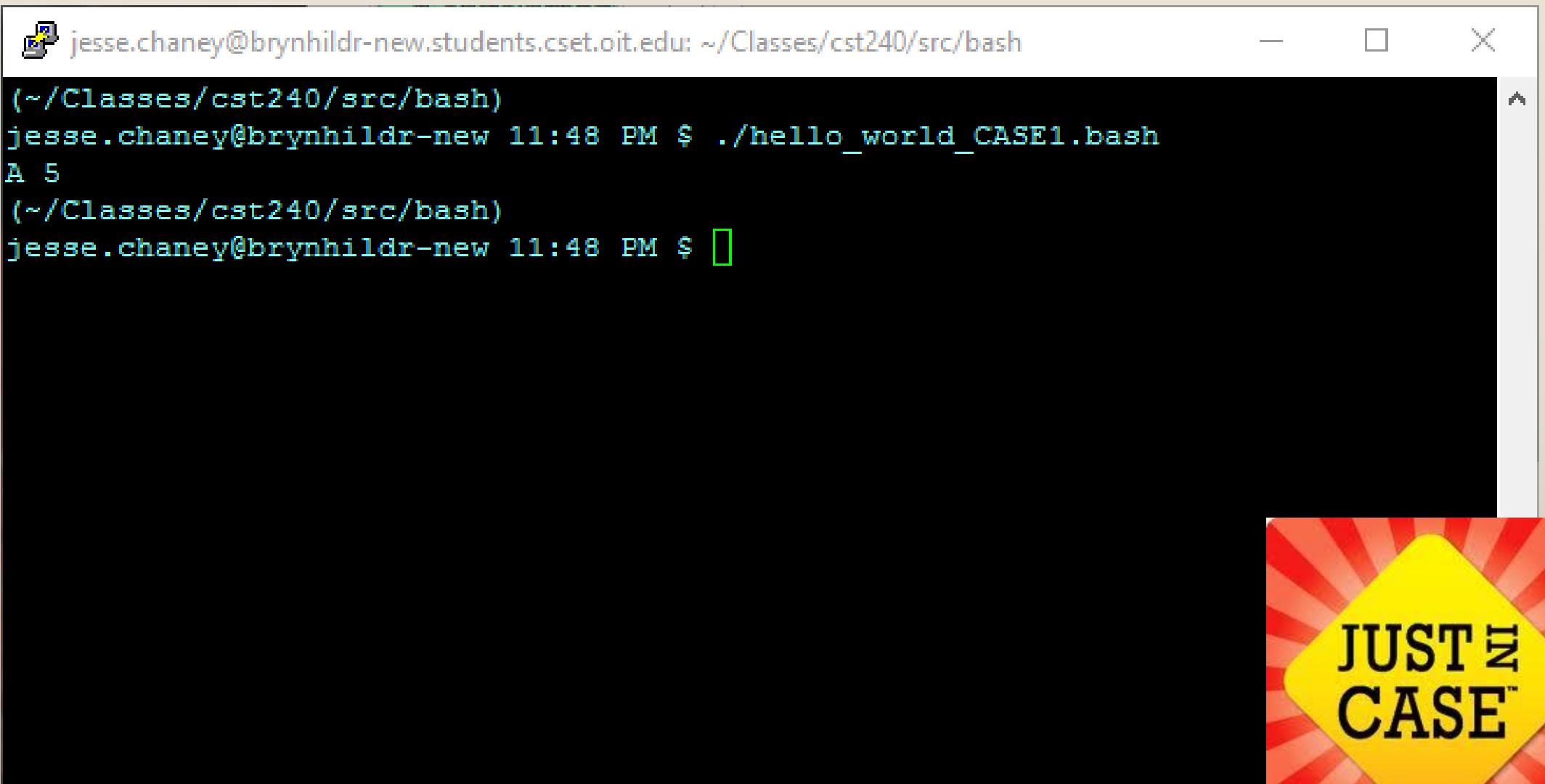
The case statement ends with esac.

A pattern match.

End of a pattern match block.

A catch-all pattern.

Conditionals - case



A screenshot of a terminal window titled "jesse.chaney@brynhildr-new.students.cset.oit.edu: ~/Classes/cst240/src/bash". The terminal shows the following command and output:

```
(~/Classes/cst240/src/bash)
jesse.chaney@brynhildr-new 11:48 PM $ ./hello_world_CASE1.bash
A 5
(~/Classes/cst240/src/bash)
jesse.chaney@brynhildr-new 11:48 PM $ 
```



Conditionals - case

Begin the case statement

```
#!/bin/bash
VALUE="A"
case $VALUE in
    "a" | "A" )
        echo "An A or a was seen"
        ;;
    "b" | "B" )
        echo "A B or b was seen"
        ;;
    *)
        echo "A strange value $VALUE"
        ;;
esac
```

Match on either an
“a” or an “A”

Match on either an
“b” or an “B”

End the case statement

hello_world_CASE2.bash

The catch-all pattern



CASE WESTERN RESERVE
UNIVERSITY EST. 1826

Conditionals - case



```
jesse.chaney@brynhildr-new: ~/Classes/cst240/src/bash
(~/.Classes/cst240/src/bash)
jesse.chaney@brynhildr-new 11:51 PM $ ./hello_world_CASE2.bash
An A or a was seen
(~/.Classes/cst240/src/bash)
jesse.chaney@brynhildr-new 11:51 PM $ 
```

Bash trap Command

A shell script can run into problems during its execution, resulting in an error signal that interrupts the script unexpectedly.

Errors occur due to a faulty script design, user actions, system failures, or **while testing programs that do not adhere to the specification**. A script that fails may leave behind temporary files that cause trouble when the script restarts.

The bash **trap** command allows you to catch signals and execute code when they occur.

<https://www.tutorialspoint.com/bash-trap-command-explained>

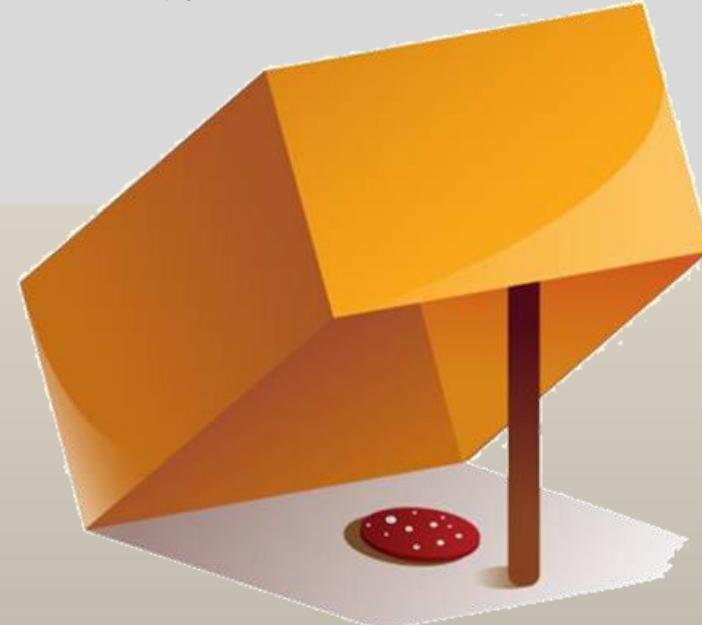
Bash trap Command

```
trap 'echo "This is bad"' SIGTERM SIGKILL SIGSEGV
```

```
trap 'echo "SIGINT caught;"' SIGINT
```

```
trap 'signalCaught;' SIGTERM SIGQUIT SIGKILL SIGSEGV
```

```
trap 'signalCtrlC;' SIGINT
```



Bash trap Command

```
#!/bin/bash

trap 'echo "Signal SIGINT caught"' SIGINT

while true
do
    echo "Press Ctrl-C to generate SIGINT signal"
    sleep 1
done
```

Bash trap Command

```
#!/bin/bash

trap 'echo "Command exited with non-zero status"; exit 1;' ERR

echo "This command will exit with a non-zero status"
false

echo "This command will not execute"
```

Bash trap Command

```
#!/bin/bash

cleanup() {
    echo "This is an exit handler for the script that can do cleanup."
}

trap cleanup EXIT

sleep 5
```

Bash timeout Command

A **timeout** is a command-line utility that runs a specified command and terminates it if it is still running after a given period of time. The **timeout** command allows you to run a command with a time limit. Like when testing and a process goes into an infinite

```
timeout [OPTIONS] DURATION COMMAND [ARG] ...
```

The **DURATION** can be a positive **integer or a floating-point number**, followed by an optional unit suffix:

s - seconds (default)

m - minutes

h - hours

d - days

<https://linuxize.com/post/timeout-command-in-linux/>

Bash timeout Command

A **timeout** is a command-line utility that runs a specified command and terminates it if it is still running after a given period of time. The **timeout** command allows you to run a command with a time limit.

```
timeout [OPTIONS] DURATION COMMAND [ARG] ...
```

The **DURATION** can be a positive **integer or a floating-point number**, followed by an optional unit suffix:

s - seconds (default)

m - minutes

h - hours

d - days

Bash timeout Command

```
time timeout 10s sleep 30
```

```
real      0m10.005s
user      0m0.003s
sys       0m0.005s
```

Bash timeout Command

```
timeout 5s ping google.com
```

```
PING google.com (142.251.211.238) 56(84) bytes of data.  
64 bytes from sea30s13-in-f14.1e100.net (142.251.211.238): icmp_seq=1  
ttl=118 time=4.23 ms  
64 bytes from sea30s13-in-f14.1e100.net (142.251.211.238): icmp_seq=2  
ttl=118 time=4.14 ms  
64 bytes from sea30s13-in-f14.1e100.net (142.251.211.238): icmp_seq=3  
ttl=118 time=4.34 ms  
64 bytes from sea30s13-in-f14.1e100.net (142.251.211.238): icmp_seq=4  
ttl=118 time=4.22 ms  
64 bytes from sea30s13-in-f14.1e100.net (142.251.211.238): icmp_seq=5  
ttl=118 time=4.35 ms
```

Bash timeout Command

If no signal is given, `timeout` sends the **SIGTERM** signal to the managed command when the time limit is reached.

You can specify which signal to send using the **-s (--signal)** option.

```
timeout -s SIGKILL 5s ping google.com
```

```
timeout -k 7s 5s ping google.com
```

Bash timeout Command

A call to **timeout** returns 124 when the time limit is reached. Otherwise, it returns the exit status of the managed command.

To return the exit status of the command even when the time limit is reached, use the **--preserve-status** option:

```
timeout --preserve-status 5 ping 8.8.8.8
```



Looping - for

The most common looping structures in bash scripts are `for` and `while` loops. However, the `until` loop does exist.

```
#!/bin/bash

for FILE in h*.bash
do
    echo "    $FILE"
done
```

A `do` line follows the beginning of the `for` loop.

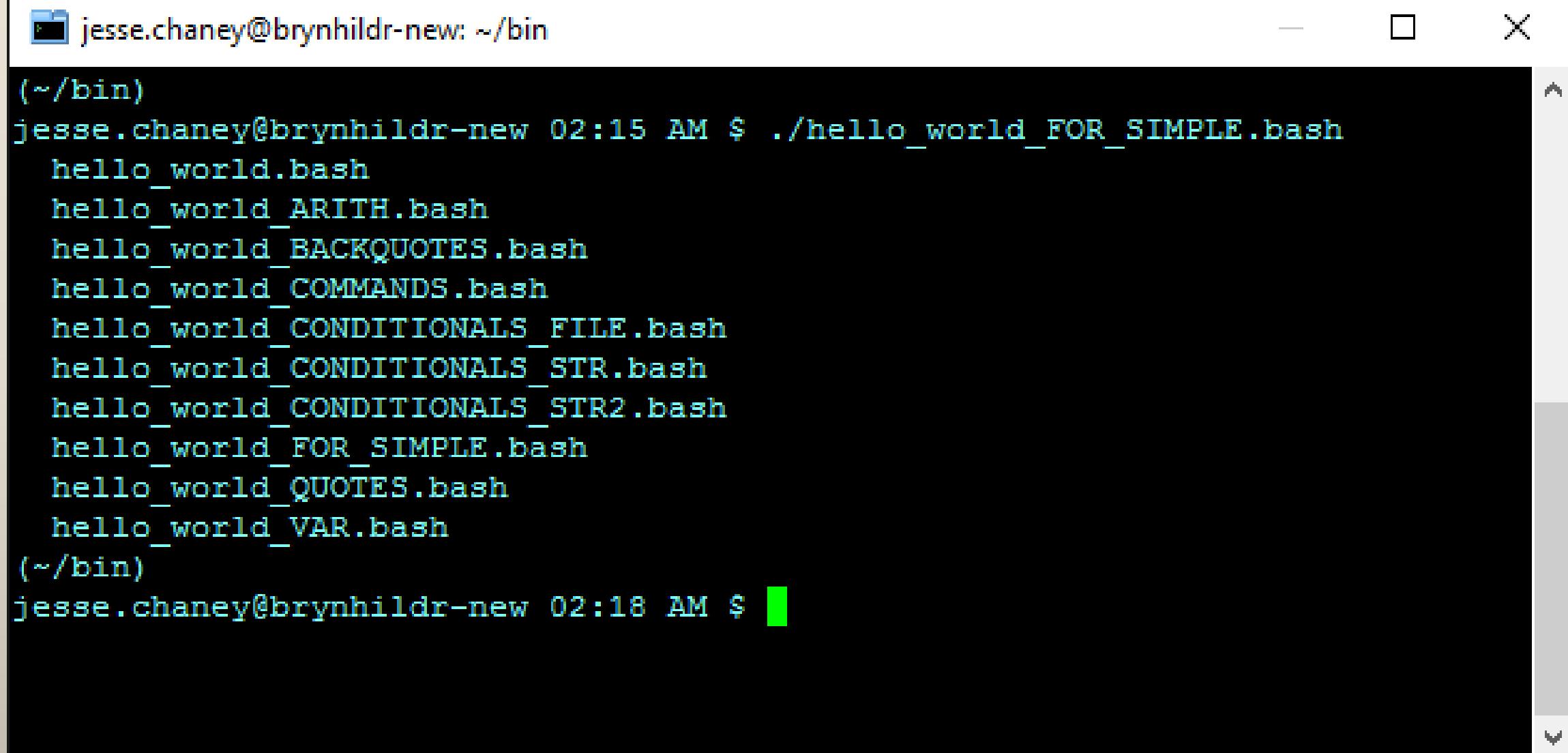
Loop through all files that have an `h` as the first letter and end with `.bash` and echo the file name.

A `for` loop is terminated with a `done`.

You can think of the `for` loop in bash as being a **foreach** loop.

hello_world_FOR_SIMPLE.bash

Looping - for



The screenshot shows a terminal window with the following text:

```
jesse.chaney@brynhildr-new: ~/bin
(~/.bin)
jesse.chaney@brynhildr-new 02:15 AM $ ./hello_world_FOR_SIMPLE.bash
    hello_world.bash
    hello_world_ARITH.bash
    hello_world_BACKQUOTES.bash
    hello_world_COMMANDS.bash
    hello_world_CONDITIONALS_FILE.bash
    hello_world_CONDITIONALS_STR.bash
    hello_world_CONDITIONALS_STR2.bash
    hello_world_FOR_SIMPLE.bash
    hello_world_QUOTES.bash
    hello_world_VAR.bash
(~/.bin)
jesse.chaney@brynhildr-new 02:18 AM $ █
```

The terminal window has a dark background and light-colored text. It includes standard window controls (minimize, maximize, close) in the top right corner.

Looping - for



```
#!/bin/bash

NAMES="Raymond Jesse Chaney"
for NAME in $NAMES
do
    echo $NAME
done
```

Loop through the space delimited list.

```
#!/bin/bash

for VALUE in {1..5}
do
    echo $VALUE
done
```

Loop through the range 1 to 5.
Notice the braces used to define the range for the loop.

Looping - for

```
jesse.chaney@brynhildr-new: ~/bin
```

```
(~/bin)
jesse.chaney@brynhildr-new 02:31 AM $ ./hello_world_FOR_ITER.bash
1
2
3
4
5
(~/bin)
jesse.chaney@brynhildr-new 02:32 AM $ hello_world_FOR_NAME.bash
Raymond
Jesse
Chaney
(~/bin)
jesse.chaney@brynhildr-new 02:32 AM $ █
```

Looping Through a Range of Numbers

```
#!/bin/bash

for VAL in {1..10}
do
    echo $VAL
done
```

When you want to loop through a range of values, you can do something simple like this.

```
#!/bin/bash
LL=2
UL=10
for VAL in {$LL..$UL}
do
    echo $VAL
done
```

However, when you try it with variables, it does not work!



Looping Through a Range of Numbers

```
#!/bin/bash
LL=2
UL=10
for VAL in `seq ${LL} ${UL}`
do
    echo $VAL
done
```

There are a couple of solutions.
Though perfectly good, this one seems
to be deprecated. :-)

```
#!/bin/bash
LL=2
UL=10
for (( VAL=$LL; VAL<=$UL; VAL++ ))
do
    echo $VAL
done
```

This looks more like the C/C++
syntax you are accustomed to using.



Looping Through a Range of Numbers

```
#!/bin/bash
LL=2
UL=10
VAL=$LL
while [ $VAL -le $UL ]
do
    echo $VAL
    $((VAL = VAL + 1))
done
```

For the **while** loving folks out there.
We'll look more at `while` in a
couple slides. Yes, you need all the
brackets.

Notice brackets here, not
parentheses or braces.

```
#!/bin/bash
LL=2
UL=10
for VAL in $(eval echo "{$LL..$UL}")
do
    echo $VAL
done
```

This evaluates the variables to
produce the original format.

Use of this one will either **elevate you in the
nerd herd** or be met with shock and disgust.



Looping - while

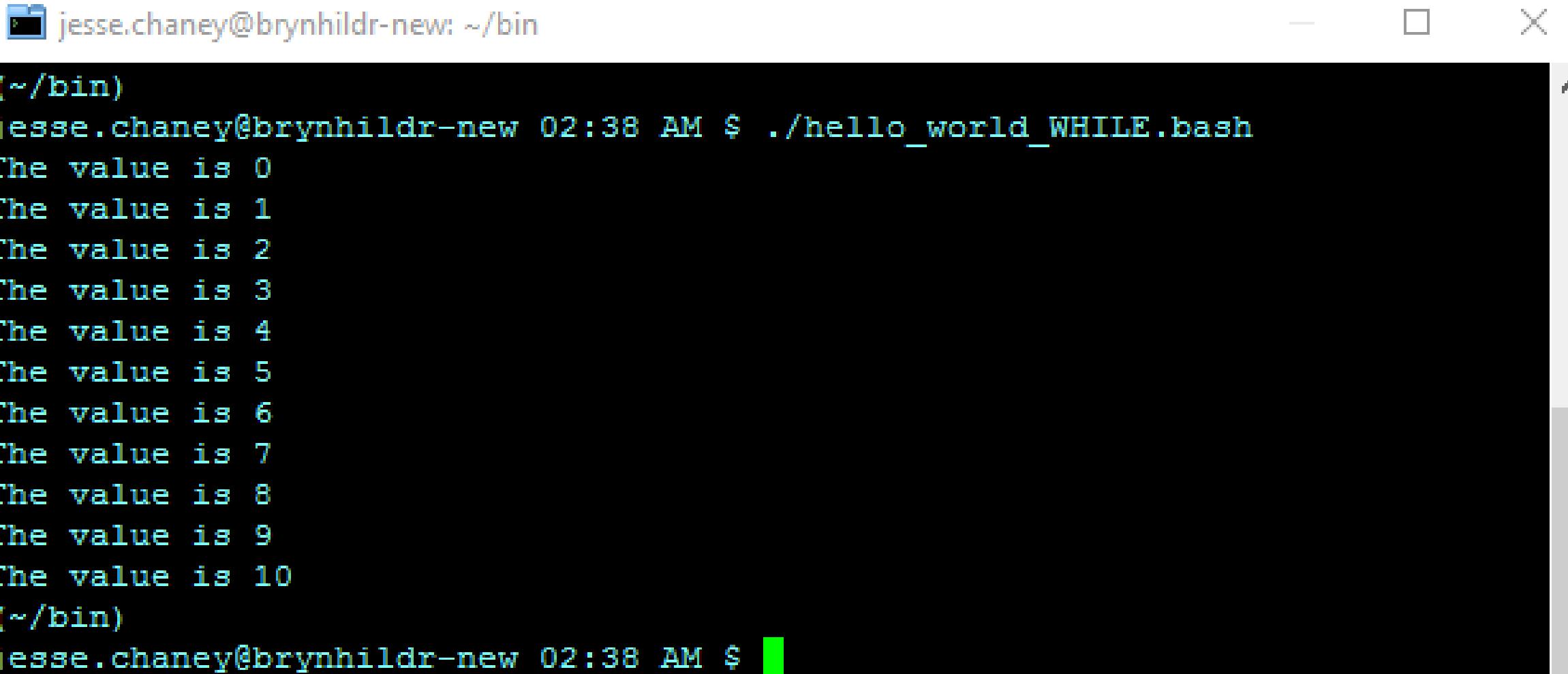
Like the for loop, a while loop has both the do and the terminating done, just as the for loop has.

```
#!/bin/bash
VALUE=0
while [ $VALUE -le 10 ]
do
    echo "The value is $VALUE"
    VALUE=$(( VALUE + 1 ))
done
```

While the value of VALUE is less than or equal to 10, loop.



Looping - while



The terminal window shows the output of a bash script named `hello_world_WHILE.bash`. The script uses a `while` loop to print the value of a variable `i` from 0 to 10. The terminal prompt is `jesse.chaney@brynhildr-new: ~/bin`.

```
(~/bin)
jesse.chaney@brynhildr-new 02:38 AM $ ./hello_world_WHILE.bash
The value is 0
The value is 1
The value is 2
The value is 3
The value is 4
The value is 5
The value is 6
The value is 7
The value is 8
The value is 9
The value is 10
(~/bin)
jesse.chaney@brynhildr-new 02:38 AM $ █
```

Prettier output with printf

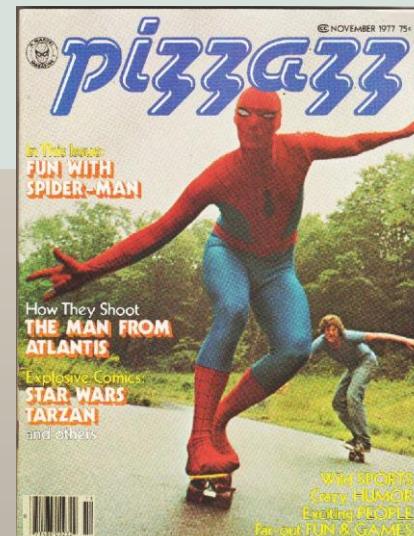
Sometimes you need something with a bit more control or *pizzazz* than you can get with the `echo` command/built-in.

In those times, reach for `printf`.

There is both a `printf` command and a `printf` bash built-in.

Both work about the same and use pretty much the same format specifies as does the C `printf` function.

```
printf "%s\n" 'Hello world!'
```



Some printf Format Specifiers

There are *oodles* of format specifiers you can use with printf.

For the most part, they mirror the ones you use for the C function printf.

Format Specifier	
%d or %i	Print the associated argument as signed decimal number (an integer)
%s	Interprets the associated argument literally as string
%b	Print the associated argument while interpreting backslash escapes in the string
%f	Interpret and print the associated argument as floating point number
%c	Interprets the associated argument as char : only the first character of a given argument is printed
%x	Print the associated argument as unsigned hexadecimal number with lower-case hex-digits (a-f)

Reading Input with `read`

Sometimes you need to get input from a file or from the user.

This is a good time to try the `read` bash built-in.

The `read` built-in will read a line of data from `stdin` and split it into fields.

Reads user input
from the keyboard
into the variables
`GName` and `FName`.

```
#!/bin/bash

echo "Enter your name, <given name> <family name>"
read GName FName

echo "Your fortune for the day ${GName} ${FName}."
fortune
```

`yourFortune.bash`

Reading
Is Fundamental

Reading Input with `read`

Did I hear you say that you'd like to read each line from a file and then echo it back out?

We've got a script for that.

```
#!/bin/bash
while read LINE
do
    echo ${LINE}
done < $1
```

Reads one line from the
stdin into the variable `LINE`.

Uses the file given on the command
line as `stdin` for the loop.

lineByline.bash



Reading Input with `read`

Wait, wait, don't tell me.

You want to read a line at a time, but output each value from the line individually?

We can do it.

```
#!/bin/bash

while read LINE
do
    for VAL in ${LINE} ←
do
    echo ${VAL}
done
done < $1
```

Splits each line into space delimited values.



valueBYvalue.bash

bash and arrays

bash can handle arrays, but the syntax is a bit ... awkward.



One dimensional arrays can simply be used without any prior declaration.

```
#!/bin/bash

ARRAY=( one two three )

for ELEMENT in "${ARRAY[@]}"
do
    echo $ELEMENT
done
```

Assigns the values into the array.
By default, the values are space delimited. Be sure to use parenthesis around the values.

Allows you to iterate through the elements in the array one at a time.

arrayLoop1.bash

CS333 Intro Op Sys

bash and arrays

You can also directly assign values into a 1D array.

```
#!/bin/bash

STRINGRAY[0]="zero"
STRINGRAY[1]="one"
STRINGRAY[2]="two"
STRINGRAY[3]="three"

for ELEMENT in "${STRINGRAY[@]}"
do
    echo $ELEMENT
done
```

Assigning individual values into the array.



arrayLoop2.bash

CS333 Intro Op Sys

bash and arrays

You can use a **non-numeric** value as an index.

This becomes an associative array for bash.



An associative array in bash must be declared before it is used.

```
#!/bin/bash

declare -A XRAY
XRAY[zero]="zero"
XRAY[one]="one"
XRAY[two]="two"

for ELEMENT in "${XRAY[@]}"
do
    echo ${ELEMENT}
done
```

Declare the array **XRAY** to be an associative array.

Assigning individual values into the associative array

arrayLoop3.bash

bash and associative arrays

bash can handle multi-dimensional arrays, as associative arrays, but you must be careful.

```
#!/bin/bash
```

```
declare -A ARRAY
```

```
ARRAY[0,0]="00"
```

```
ARRAY[0,1]="01"
```

```
ARRAY[1,0]="10"
```

```
ARRAY[1,1]="11"
```

```
for ELEMENT in "${ARRAY[@]}"
```

```
do
```

```
    echo $ELEMENT
```

```
done
```

Declare the array ARRAY to be an associative array.

Assigning individual values into the associative array

arrayLoop4.bash

bash and 2D arrays

Reading a file into a bash associative array.

The following reads from a file into a 2D array

```
#!/bin/bash
ROW=0; COL=0;
declare -A MATRIX
while read LINE
do
    COL=0
    for VAL in ${LINE}
    do
        MATRIX[$ROW,$COL]="$VAL"
        ((COL++))
    done
    ((ROW++))
done < $1
```

Declare the array ARRAY to be an associative array.

Read a line from the file.

Break the line into individual values (space delimited).

Assign an individual value into the associative array

The file identified on the command line to the script.

assocArray1.bash

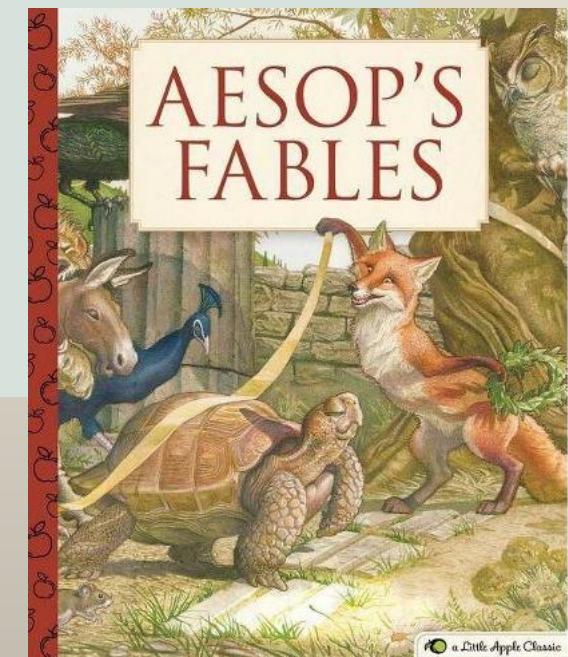
bash and 2D arrays

The script on the previous slide does not provide any output, due to a lack of room to show a longer script in a slide.

For a more robust example, look through the script `assocArray2.bash`

Try running:

```
./assocArray2.bash 2x2.mat  
./assocArray2.bash 5x10.mat  
./assocArray2.bash 10x10.mat
```



Looping – with Regular Expressions

```
#!/bin/bash

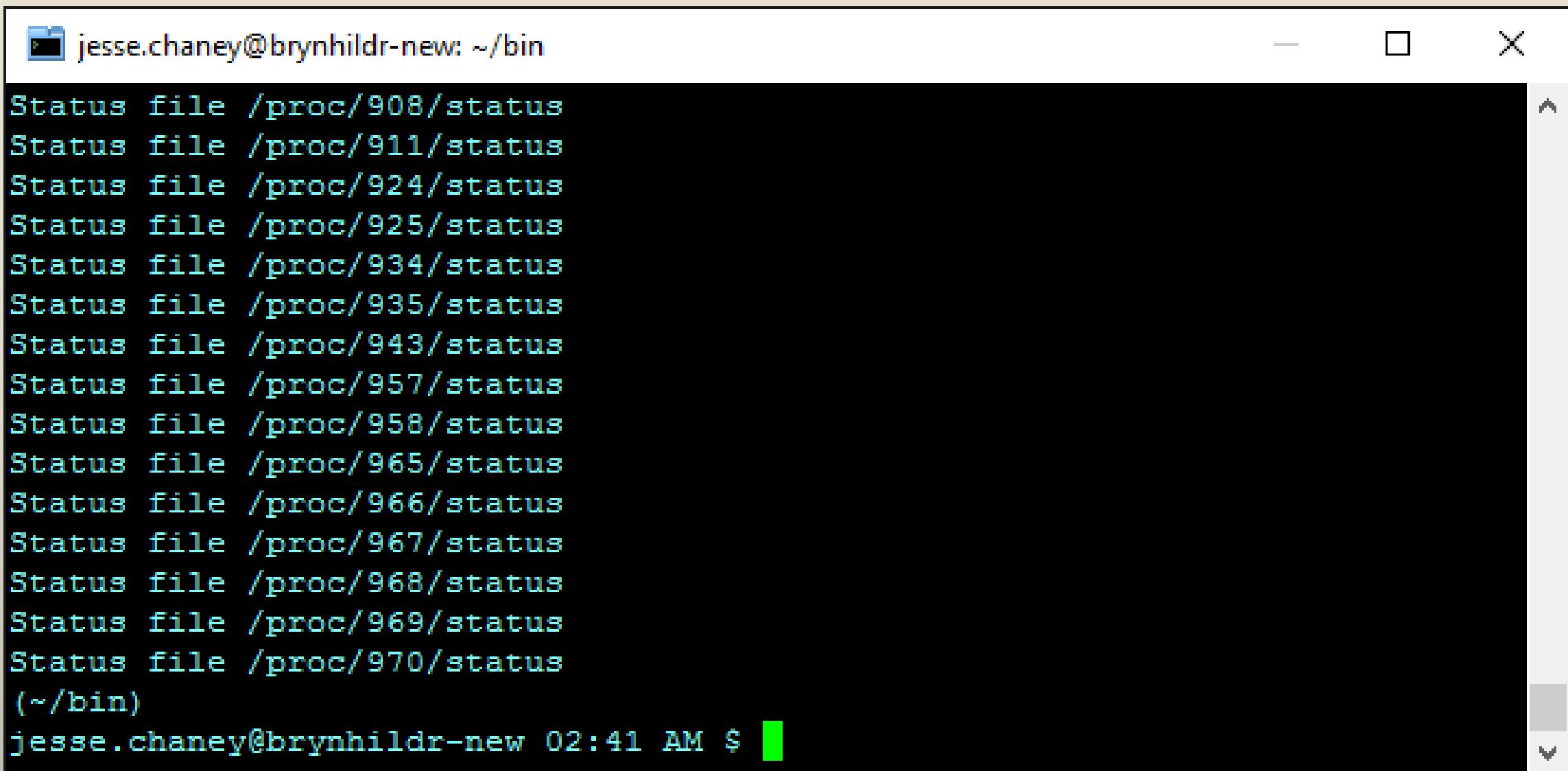
for FILE in /proc/[1-9]*status
do
    echo "Status file $FILE"
done
```

A simple regular expression for all the directories in the `/proc` directory that represent processes running on the system.

hello_world_FILE_REG.bash



Looping – with Regular Expressions



A screenshot of a terminal window titled "jesse.chaney@brynhildr-new: ~/bin". The window contains the following text:

```
Status file /proc/908/status
Status file /proc/911/status
Status file /proc/924/status
Status file /proc/925/status
Status file /proc/934/status
Status file /proc/935/status
Status file /proc/943/status
Status file /proc/957/status
Status file /proc/958/status
Status file /proc/965/status
Status file /proc/966/status
Status file /proc/967/status
Status file /proc/968/status
Status file /proc/969/status
Status file /proc/970/status
(~/bin)
jesse.chaney@brynhildr-new 02:41 AM $
```

Functions

```
#!/bin/bash

print_something()
{
    echo "  This is from the inside of a function"
}

echo "not in a function"
print_something

echo "not in a function"
print_something

echo "not in a function"
```

A function definition. You can also precede the function name with the `function` keyword.

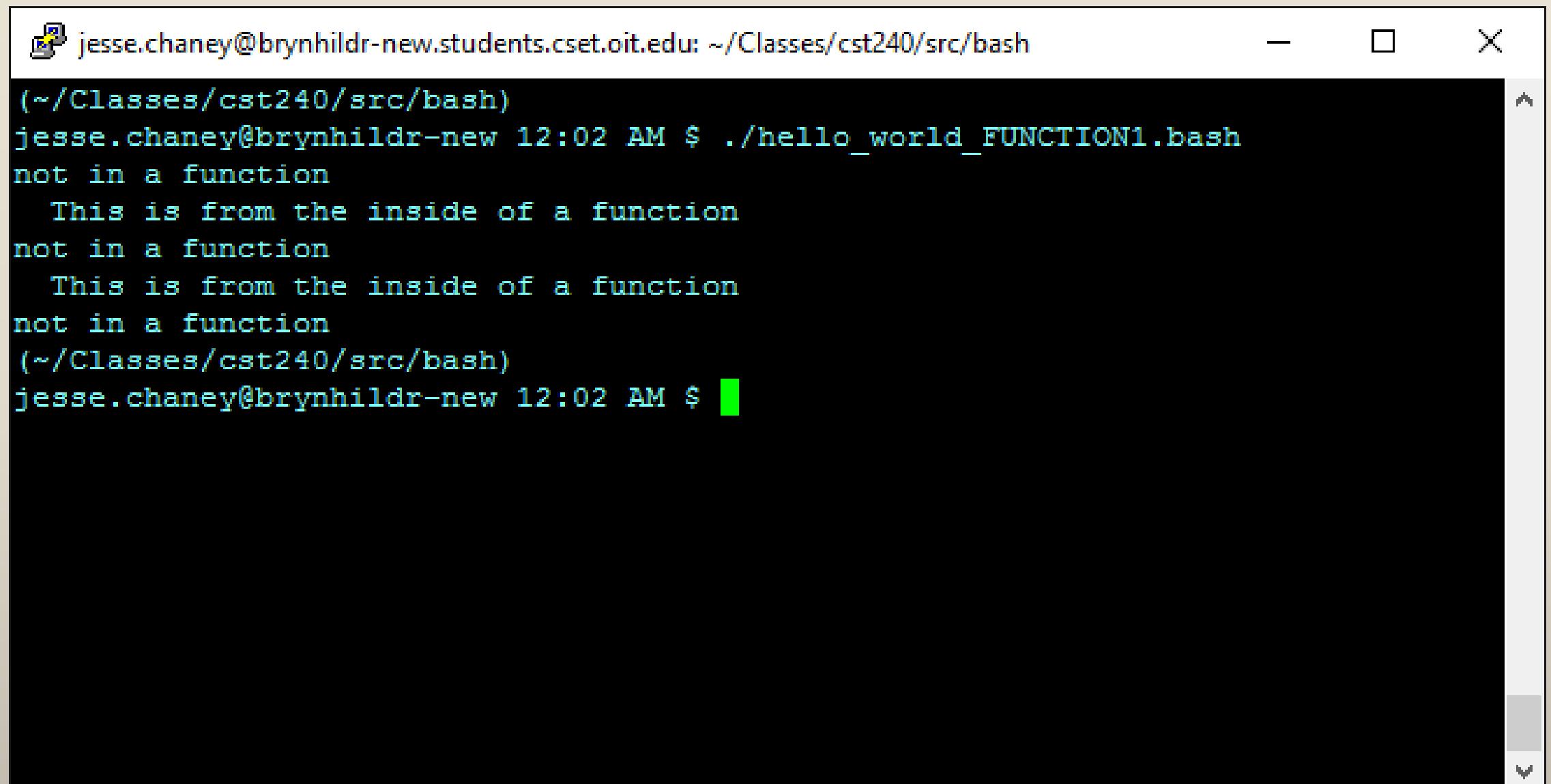
It is possible for a function to call `exit` to terminate the script.

Function calls.

You can pass parameters to functions.

Function

Functions



A screenshot of a terminal window titled "jesse.chaney@brynhildr-new.students.cset.oit.edu: ~/Classes/cst240/src/bash". The terminal displays the following output:

```
(~/Classes/cst240/src/bash)
jesse.chaney@brynhildr-new 12:02 AM $ ./hello_world_FUNCTION1.bash
not in a function
This is from the inside of a function
not in a function
This is from the inside of a function
not in a function
(~/Classes/cst240/src/bash)
jesse.chaney@brynhildr-new 12:02 AM $ █
```

The terminal window has standard window controls (minimize, maximize, close) at the top right.

The getopt Built-in

- One of the things we do a lot in *nix is parse and process the command line.
- We do for both shell scripts and for C programs.
- We do it a lot in this class.
- For example:

```
./shell_script.bash Raymond Jesse Chaney
```

- The command line options Raymond Jesse and Chaney are called positional parameters (because their order matters).
- The positional parameters can be accessed in a shell script as \$1, \$2, and \$3.
- Where \$1 = Raymond, \$2 = Jesse and \$3 = Chaney.

The getopt Built-in

- Many times, simple positional parameters are sufficient.
- However, we often need something more capable.
- The answer to the more capable is getopt or getopts.
- The getopt command is the old (more difficult) way to parse the command line.
- I highly recommend you make use of the bash built-in getopts.
- With getopts, you can do something like these:
 - ./shell_script.bash -f Raymond -m Jesse -l Chaney
 - ./shell_script.bash -l Chaney -f Raymond -m Jesse
 - ./shell_script.bash -m Jesse -l Chaney -f Raymond
- The parameters are no longer positional.
- They can be specified in any order

A getopt Example

```
while getopt "hv:s:" OPT
do
  case "${OPT}" in
    h)
      show_help
      ;;
    v)
      VAL=${OPTARG}
      ;;
    s)
      STR=${OPTARG}
      ;;
  esac
done
```

Command line options are given in double quotes.

getopt "hv:s:" OPT

"\${OPT}"

h)

show_help

;;

v)

VAL=\${OPTARG}

;;

s)

STR=\${OPTARG}

;;

esac

done

A **colon** following an option means that an argument for the option is **required**.

The bash built-in getopt is used by shell scripts to parse command line parameters.

builtin

The \${OPT} variable is set to the value of the option as the command line is parsed. If the option has an argument, its value is set to the \${OPTARG} variable.

The getopt Built-in

```
while getopt "hv:s:" OPT
do
    case "${OPT}" in
        h)
            show_help
            ;;
        v)
            VAL=${OPTARG}
            ;;
        s)
            STR=${OPTARG}
            ;;
    esac
done
```

- In the options string, "hv:s:" in this case, each command line option is a single letter.
- If the letter is followed by a colon (":"), then the option has a **required** argument. If the option is not followed in a colon, it cannot have an argument.
- The value of the option (h, v, or s in this case) is assigned into the shell variable OPT.

- If a command line argument has an option (it is followed by a colon), the value of the argument is placed in the shell variable OPTARG.
- If an option argument is missing, it is an error.

- The while loop allows you to iterate through each of the command line options.
- The case statement allows you to individually process each option and possible argument.

Redirection of **stdin/stdout/stderr**

The shell (bash or other) and many UNIX commands take their input from standard input (`stdin`), write output to standard output (`stdout`), and write error output to standard error (`stderr`).

By default, standard input is connected to the terminal keyboard and standard output and error to the terminal screen.

The way of indicating an end-of-file on the standard input, a terminal, is usually `<Ctrl-d>`.

Basic Redirection Operators

Character	Action
>	Redirect standard output into a file
2>	Redirect standard error into a file
2>&1	Redirect standard error to standard output
<	Redirect standard input from a file
	Pipe standard output to another command
>>	Append standard output into a file

Some simple examples:

```
$ who > names
```

- Direct standard output from the `who` command to a file named `names`. All `printf()` calls will automatically go into the `names` file.

```
$ cat < file.txt
```

- Direct the `file` `file.txt` as the `stdin` to the `cat` command. All calls to `scanf()` / `gets()` come from the `file.txt` file.

```
$ who | wc
```

- The `stdout` from the `who` command is sent to the pipe (**the `|` character**) and is directed as `stdin` for the `wc` command.

When used on UNIX/Linux command line, the vertical bar character is called a pipe.



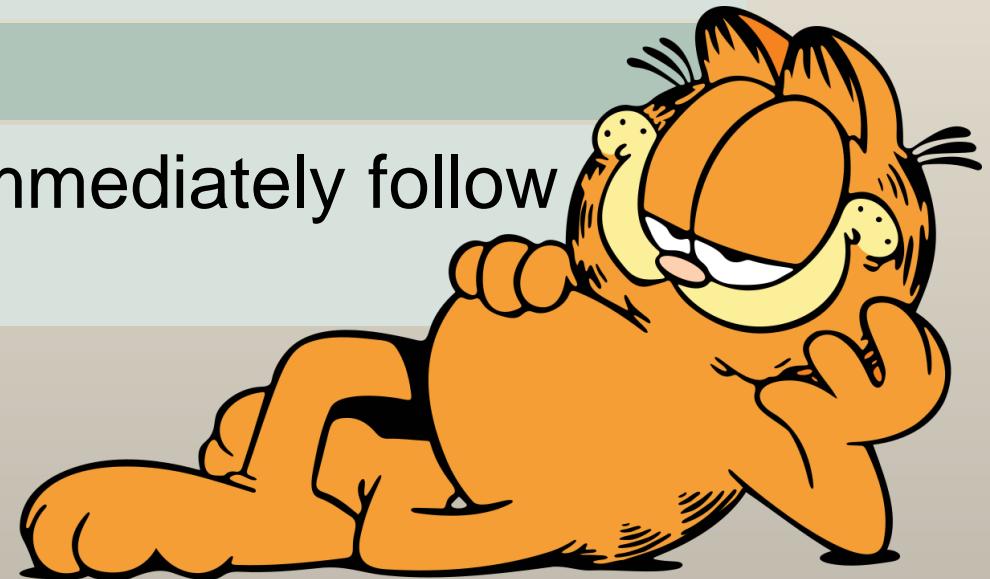
The cat Command

The `cat` command will read a file or files listed on the command line and display the contents to `stdout`.

If more than one file is listed on the command line, the files will be **con**cat**enated** in the output.

```
cat /etc/passwd /etc/group
```

The contents of the second file will immediately follow the contents of the first file.



The `diff` Command

The `diff` command allows you to **compare** 2 files line by line.

```
diff file1 file2
```

Lines from the files that differ are printed on stdout.

The **exit status** from `diff` is **0 if input files are the same, 1 if differences are identified**, 2 if trouble occurred during the comparison.

The `diff` command has a lot of other command line options you can read in the `man` page.



Head or Tails

The `head` command will display the first lines of a file. The default is to show the first 10 lines, but a command line option (as shown in the man page) can alter that.

```
head /etc/passwd
```

```
head -n 5 /etc/passwd
```

The `tail` command will display the last lines of a file. The default is to show the last 10 lines, but a command line option (as shown in the man page) can alter that.

```
tail /etc/passwd
```

```
tail -n 5 /etc/passwd
```



The find Command

The `find` command is very useful. It allows you to search for **file names** that match certain conditions. You can use the output of the `find` command within a `for` loop.

```
find /proc -ignore_readdir_race \
      -maxdepth 2 -user rchaney -name status
```

This is only a line continuation character.

Find all files under the `/proc` directory at a directory maximum depth of 2 that are owned by me and are named “status”



The tr command

The `tr` command allows you to translate (or delete) characters from the input stream (`stdin`).

```
tr '\n' ' ' # Translate each newline into a space
```

Replace (translate) all newline characters (the '`\n`') in the `stdin` stream into a space.

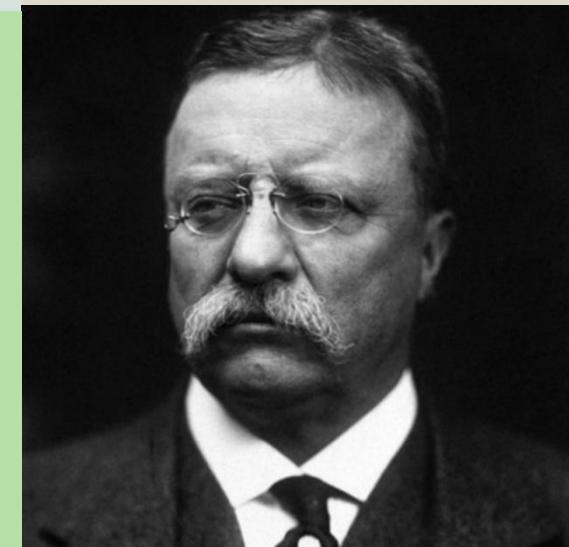
```
$cat geekfile | tr "[a-z]" "[A-Z]"
```

```
$cat geekfile | tr "[:lower:]" "[:upper:]"
```

```
$echo "Welcome To GeeksforGeeks" | tr [:space:] '\t'
```

```
$echo "Welcome To GeeksforGeeks" | tr -d 'w'
```

```
$echo "my ID is 73535" | tr -d [:digit:]
```



The wc Command

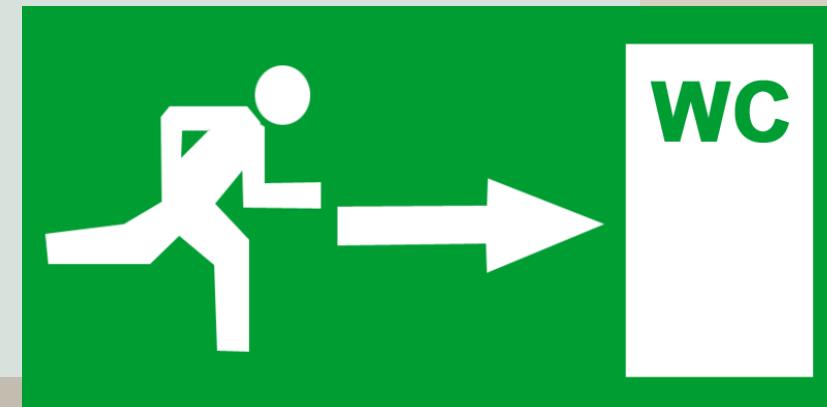
Print newline, word, and byte **counts** for each FILE, and a total line if more than one FILE is specified.

```
wc [option] file1 [file2 ... fileN]
```

A word is a non-zero-length sequence of characters delimited by white space.

Options:

- c print the byte counts
- l print the newline counts
- w print the word counts



The grep Command

The grep command is how you **search the contents** of a file (or files) for lines that **match** a string or pattern.

You can use **regular expressions** in your search pattern.

```
grep -s '^Name: \| ^State: \| ^Threads:' /proc/[1-9]*/status
```

Find all lines that **begin** with “Name:” **or** “State:” **or** “Threads:” in the status files in any numeric directory in the /proc pseudo-file system.

Ken Thompson wrote the original grep. Its first release was in November 1973.



A Little on Regular Expressions

The `grep` command supports a lot of flexibility for regular expressions.

Depending on the source, `grep` stands for **g**lobal **r**egular **e**xpression **p**rint.

The man page for `grep` has a lot of detail about the regular expressions that `grep` can handle, in the **REGULAR EXPRESSIONS** section.

```
grep -s '^Name: \| ^State: \| ^Threads: ' /proc/[1-9]*/status
```

Anchor the expression to the beginning of a line.

Stephen Kleene developed regular expressions in the mid-1950s.

Three different expressions are or-ed together. Notice how the or (the vertical bar or **pipe**) is escaped with a backslash (\).

```
/proc/[1-9]*/status
```

This regular expression is expanded by the shell, not `grep`. It means all directories that begin with a numeral 1-9 and are followed by 0 or more other characters.

**CREATIVE
EXPRESSION**

Meta Characters

Character	Meaning
<code>^</code> (circumflex or caret)	anchor expression to the start of a line, as in <code>^A</code> .
<code>\$</code> (dollar)	anchor expression end the end of a line, as in <code>A\$</code> .
<code>\</code> (back slash)	turn off the special meaning of the next character, as in <code>\^</code> . We often refer to this as “ <i>escaping the character.</i> ” Take away its special meaning and treat it as a regular character.
<code>[]</code> (square brackets)	match any one of the enclosed characters, as in <code>[aeiou]</code> . Use a hyphen <code>"-"</code> to represent a range, as in <code>[0-9]</code> . This is an inclusion set.
<code>[^]</code>	match any one character except those enclosed in <code>[]</code> , as in <code>[^0-9]</code> . This is an exclusion set.
<code>.</code> (dot or period)	match a single character of any value, except end of line.
<code>*</code> (star, asterisk, or splat)	match zero or more of the preceding character or expression. Commonly called the wildcard.



Some other interesting/helpful/fun commands:

- paste
- cut
- cal (Try `cal 9 1752` and what looks odd)
- awk
- date
- tac (Can you spell cat backwards?)
- rev
- true
- false
- yes
- uptime
- strings
- file
- last
- tty
- look
- sleep
- script
- whereis
- which
- sort
- uniq
- compress
- resize
- reset





The dwarf planet Makemake

Examples

An **excellent** place to find a couple example Makefiles (ones that follow the content in these slides) is:

`~rchaney/Classes/cs333/src/make`

Look at both **`Makefile`** and **`Makefile_vars`**

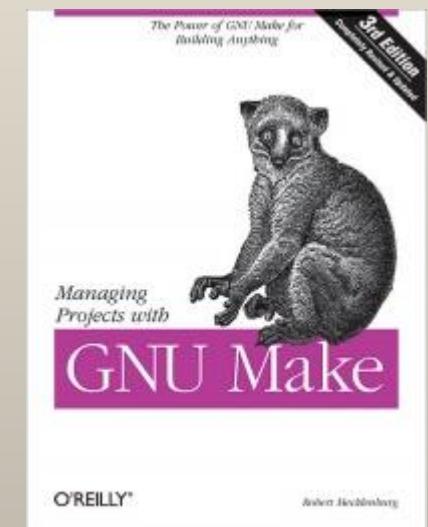
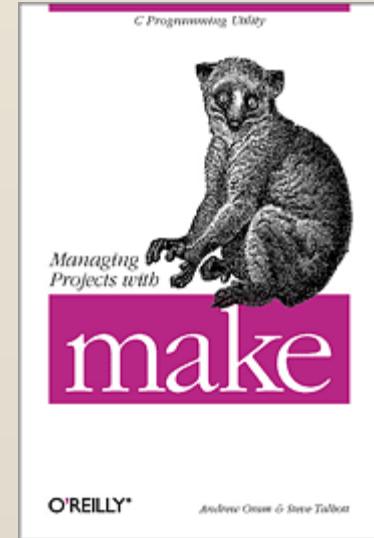
You'll need to create a **Makefile** to support your **C code** labs and assignments.

I have placed links to some web resources in Canvas (Resources for make).

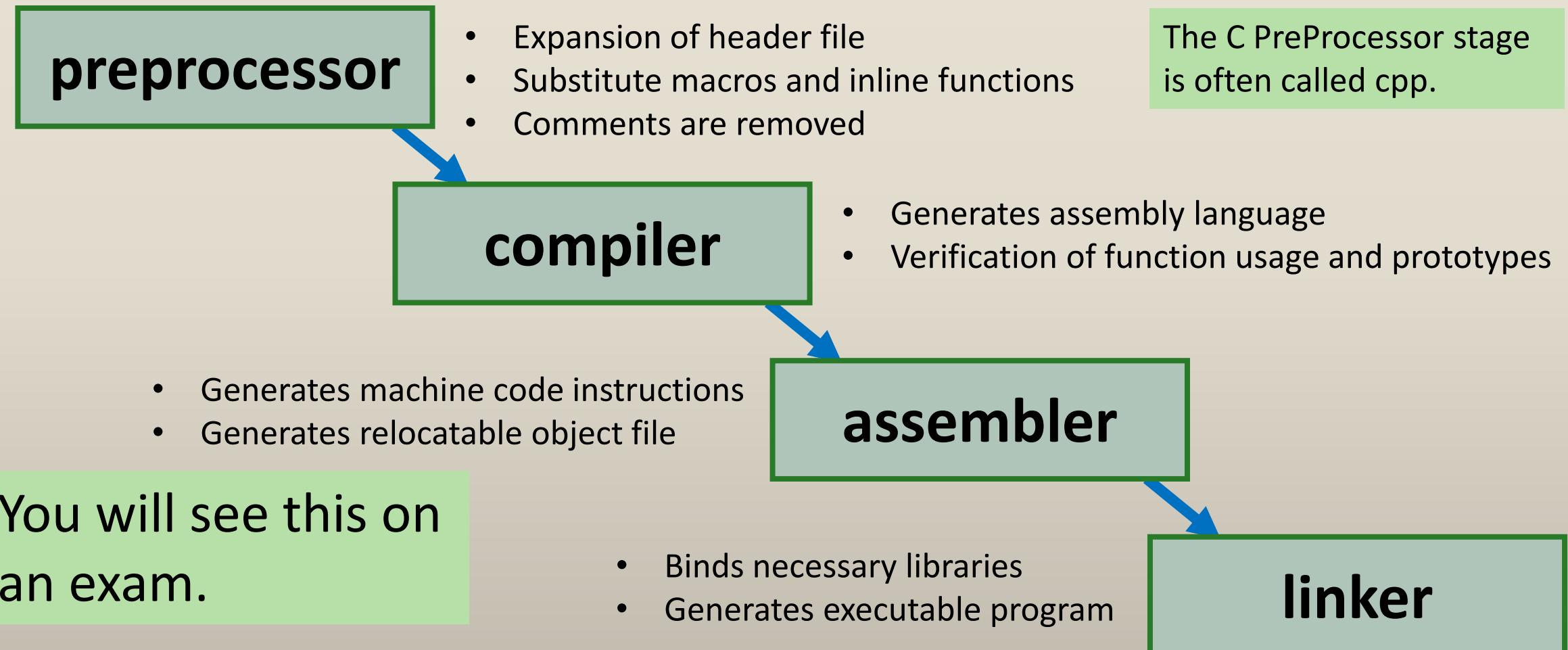
There are some books on make through the main Library and the GNU make book is **freely available** from O'Reilly as a pdf.

The GNU make manual is also online.

<https://www.gnu.org/software/automake/manual/make.pdf>



The 4 Stages of Compilation of a C Program

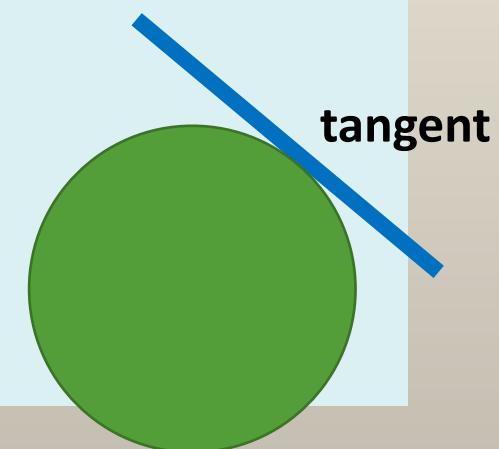


Common Command Line Options for gcc

There are a couple command line options for the gcc compiler that you are going to be using.

The man page for gcc is a tad bit lengthy (18,000+ lines), so I'm going to call out 3 options:

- O
- g
- c



Command Line Options for gcc

`-o file`

Place **output** from the `gcc` compiler in file `file`. This applies to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

If `-o` is not specified, the default is to put an executable file in file named `a.out`, the object file for `source.c` is `source.o`, its assembler file is `source.s`, a precompiled header file is `source.c.gch`, and all preprocessed C source on standard output.

With the `-o` option, you specify the name of the output that you want to create. We **want** to be specific.

Command Line Options for gcc

-g

Produce **debugging** information in the operating system's native format (stabs, COFF, XCOFF, or DWARF 2). GDB can work with this debugging information.

Debugging with GDB is not quite like using Visual Studio, but it can be a lot better than just using print statements in your code.

You will be using this command line option.



Command Line Options for gcc

-c **Compile or assemble the source files, but do not link.**

The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix .c, .i, .s, etc., with .o.

Unrecognized input files, not requiring compilation or assembly, are ignored.



This is the `gcc` option to use to create the object file (the .o) from the .c file. The name given to the .o file will be the same as the name of the .c file, with the .o substituted for .c.

The `make` utility in Unix is one of the original tools designed by S. I. Fieldman of AT&T Bell labs in 1977. There are several/many versions.

We will be using the GNU version of `make`, the default for Linux.

GNU `make` has a LOT of options and capability. We will actually only use a small portion of that.



make is a program building tool which runs on Unix, Linux, and other various like operating systems.

You can even use make for Visual Studio development!!!

make allows you to describe build **dependencies** between source code and object code or executables. It also allows you to **describe** how to bring dependencies **up to date**.



- When you are building a small program, with only a couple .c and .h files, a Makefile may not seem necessary.
- When you have a large program with many .c and .h files, a Makefile is a requirement. Especially when you have multiple developers working on the project.
- When you build a very large project that is made from many (maybe hundreds) files, you only want to rebuild the portions that have changed since the last build.



In `~rchaney/Classes/cs333/src/make` there are some sample files that will give you some practice with reading and writing Makefiles.

I like to make the first letter of my Makefile a capital M.

By default, when you run the `make` command it will look for a file called `GNUmakefile`, `makefile`, or `Makefile`, in that order.

You run `make` by typing '`make`' on the command line.



```
// The hellomake.c file

#include <stdio.h>
#include "hellomake.h"

int main( void )
{
    // call a function in another file
    myPrintHelloMake();

    return(0);
}
```

Both C files include
the hellomake.h
file.

```
// The hellofunc.c file
```

```
#include <stdio.h>
#include "hellomake.h"

void myPrintHelloMake( void )
{
    printf("Hello makefile!\n");

    return;
}
```

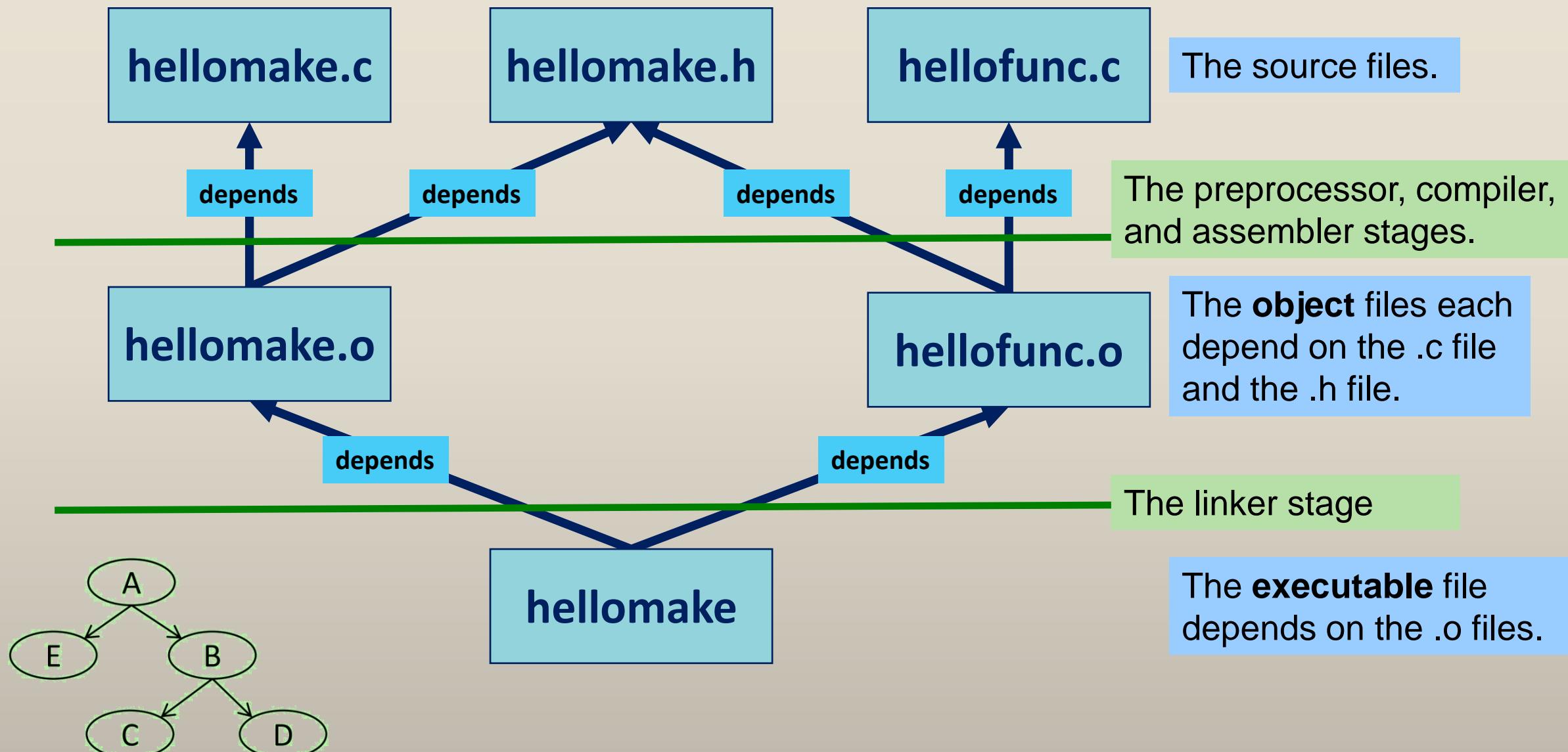
I want to build the program
hellomake using the 3 files
shown here.

```
// The hellomake.h file.

void myPrintHelloMake( void );
```



The File Dependency Tree



“Depends on” means that the target (dependent file or target) **must be rebuilt** any time one of the **prerequisite files** is **newer**.

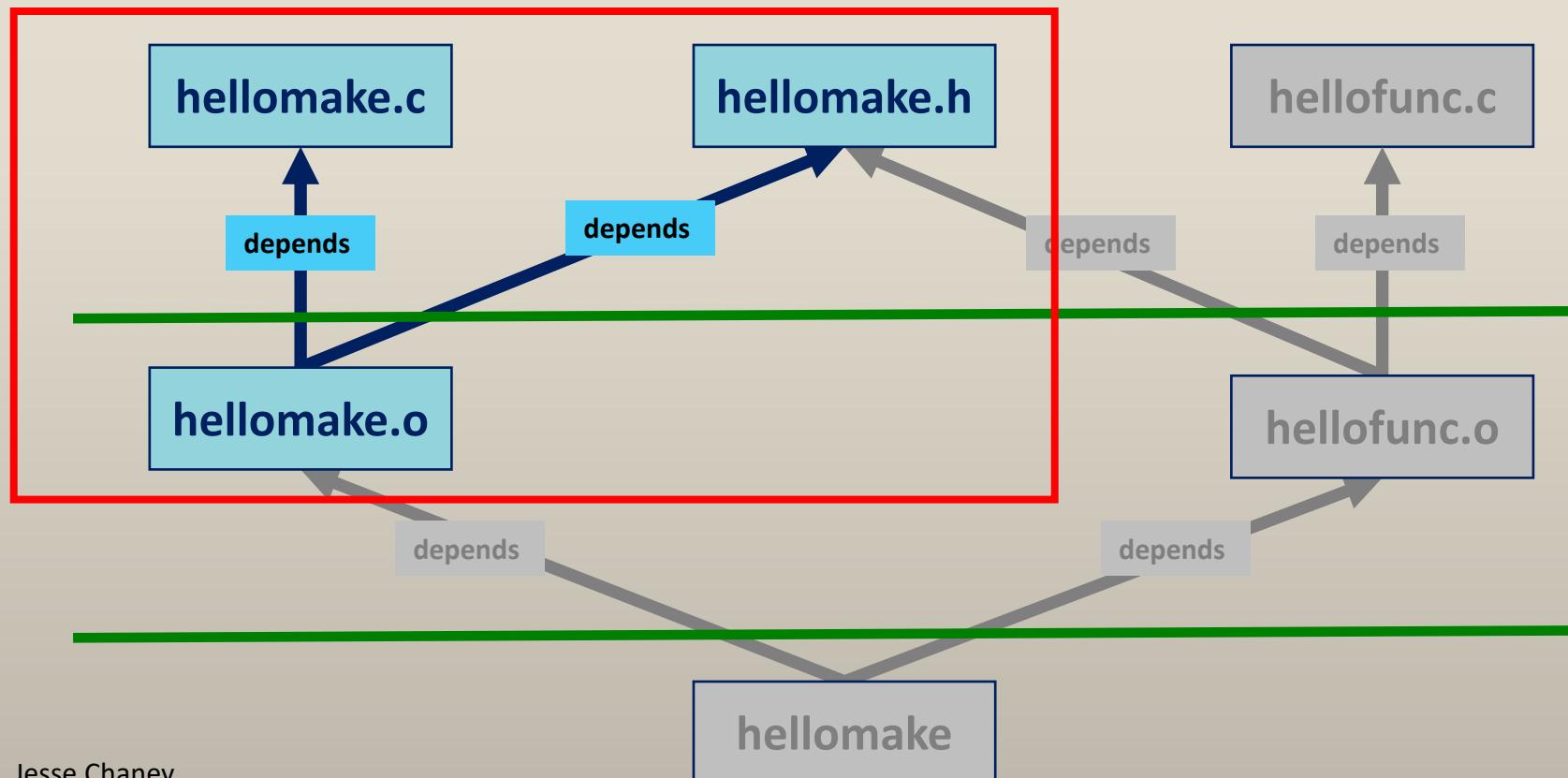
“Newer” is based on timestamp of the file.

A more recent timestamp is newer than an less-recent timestamp.

Today is more recent than yesterday.

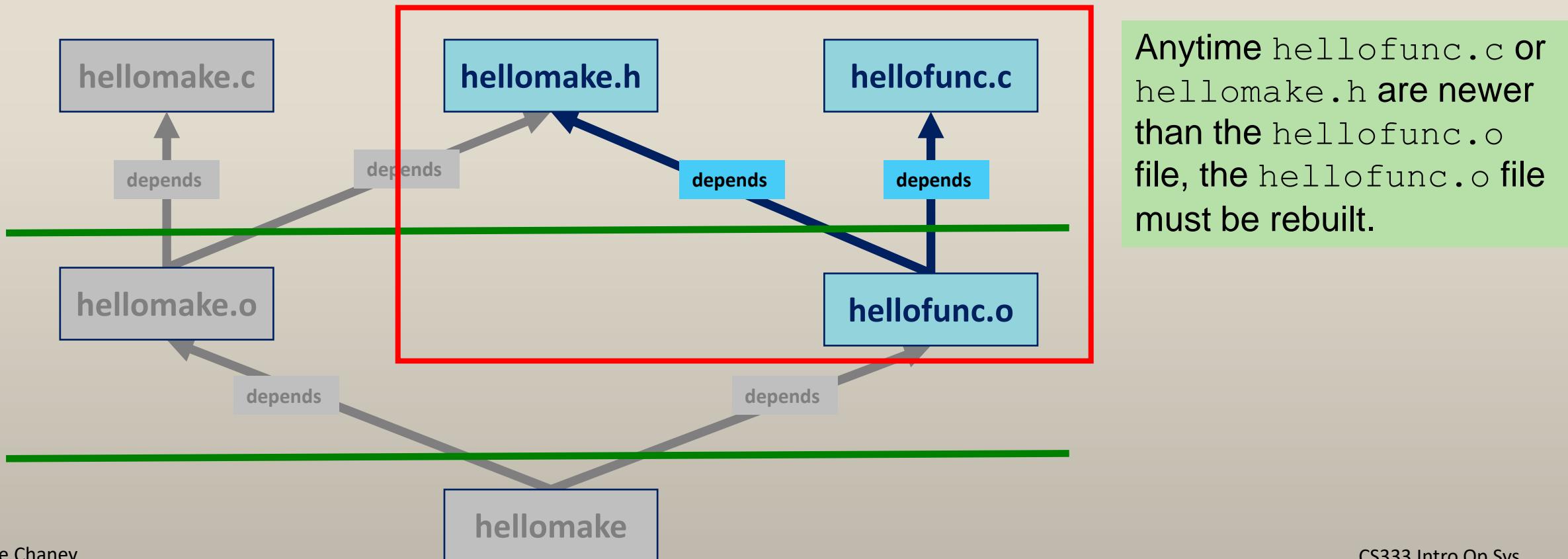
The File Dependency Tree

- The hellomake.o file depends on the hellomake.c file and the hellomake.h file.
- The hellomake.c file and the hellomake.h are **prerequisites** to the hellomake.o file.

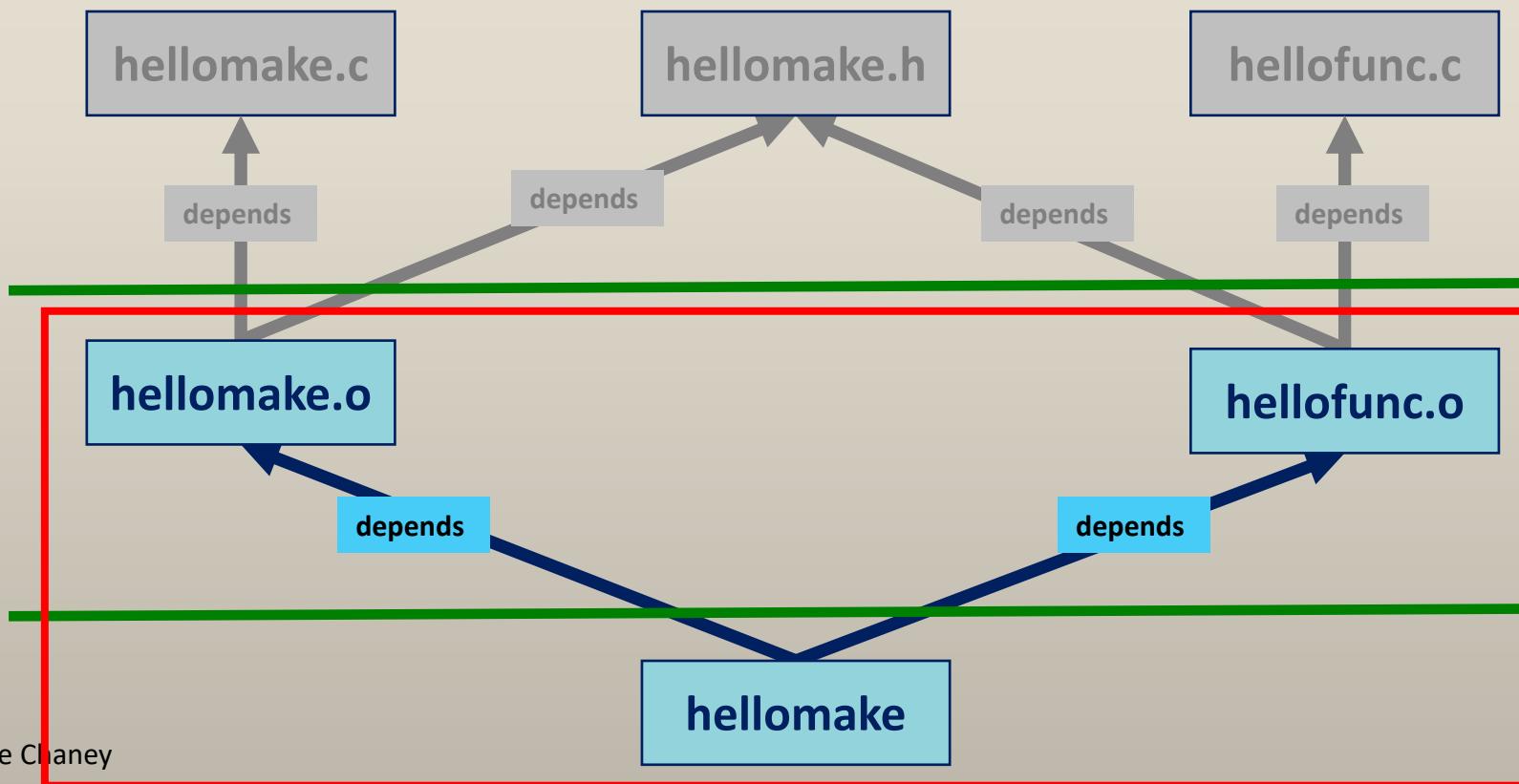


Anytime hellomake.c or hellomake.h are newer than the hellomake.o file, the hellomake.o file must be rebuilt.

- The `hellofunc.o` file **depends on** the `hellofunc.c` file and the `hellomake.h` file.
- The `hellofunc.c` file and the `hellomake.h` are **prerequisites to the** `hellofunc.o` file.

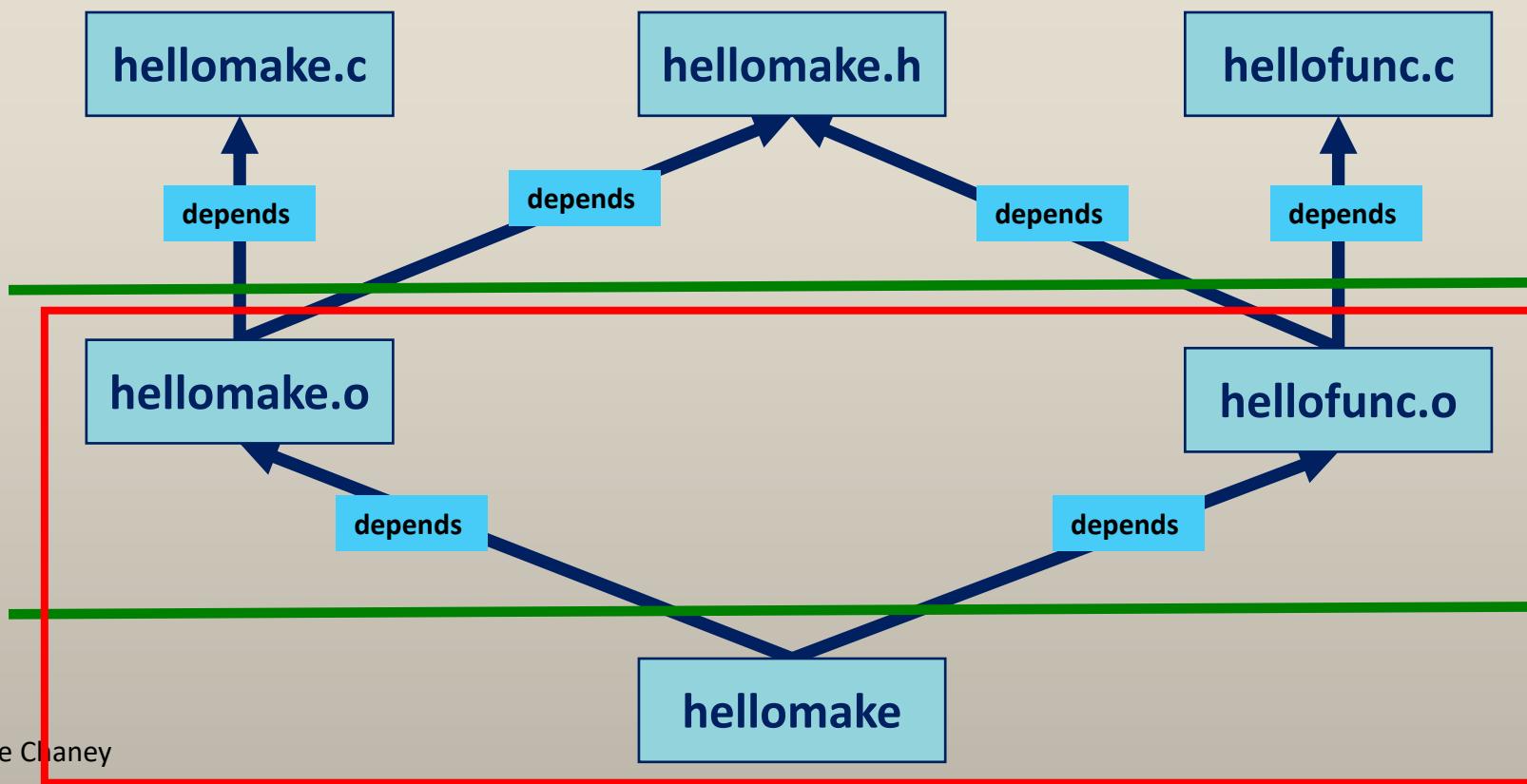


- The hellomake executable **depends on** the hellomake.o and hellofunc.o files.
- The hellomake.o and hellofunc.o files are **prerequisites to the** hellomake file.



Anytime hellomake.o or hellofunc.o are newer than the hellomake file, the hellomake file must be rebuilt.

- We can actually go a step further and say that the hellomake program actually depends on: hellomake.o, hellofunc.o, hellomake.c, hellofunc.c, and hellomake.h



Anytime `hellomake.o`, `hellofunc.o`, `hellomake.c`, `hellofunc.c`, or `hellomake.h` are newer than the `hellomake` file, the `hellomake` file must be rebuilt.

Identifying Prereqs in a Makefile

Target (what we want to build)

A single colon separates the target from the prerequisite files.

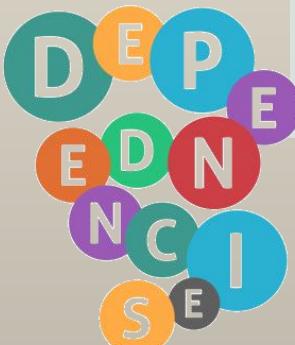
Prerequisites/Depends on

hellomake : hellomake.o hellofunc.o

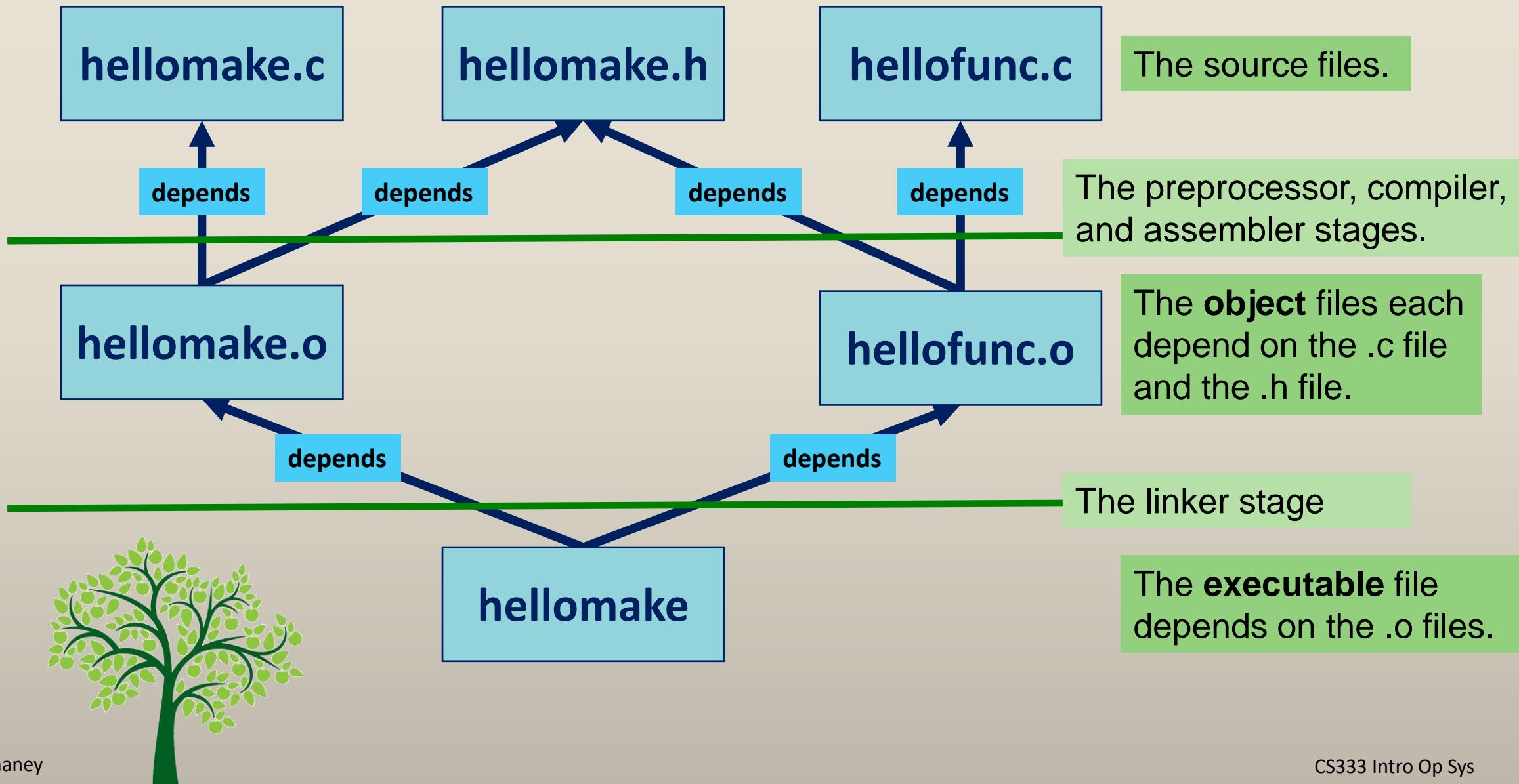
hellomake.o : hellomake.c hellomake.h

hellofunc.o : hellofunc.c hellomake.h

If any of the prerequisite files is **newer** than the target, the target must be rebuilt.

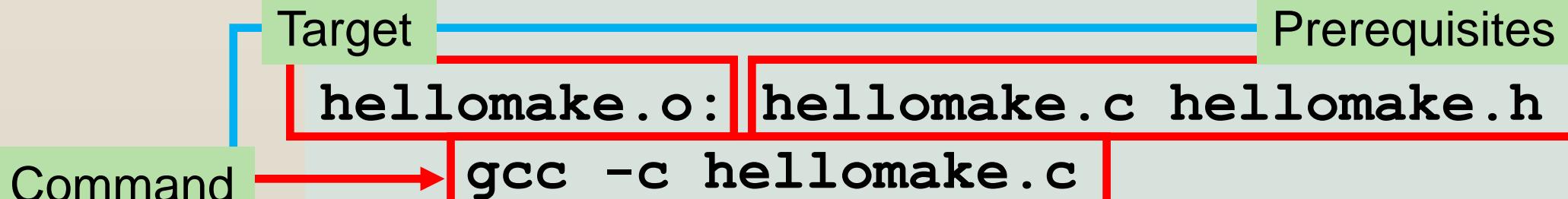


The File Dependency Tree



Rules

```
hellomake: hellomake.o hellofunc.o  
          gcc -o hellomake hellomake.o hellofunc.o
```



```
hellofunc.o: hellofunc.c hellomake.h  
          gcc -c hellofunc.c
```

A **rule** is composed of the target, the prerequisites, and the command(s) used to build the target.

make is **VERY** finicky about
tab characters.



```
hellomake: hellomake.o hellofunc.o
          gcc -o hellomake hellomake.o hellofunc.o
```

These **MUST**
be hard tab
characters!!!

```
hellomake.o: hellomake.c hellomake.h
          gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
          gcc -c hellofunc.c
```

Do **NOT** try and do indentation
before commands with spaces.

You **must** have a target in your Makefile called **clean**.
The **clean** target will delete a three different kinds of files.

1. The executable file/files
2. The object `.o` files
3. Any editor droppings files that are left around (`#*` files from `vi` and `*~` files from `emacs`).

A target without any prerequisites. This target has 2 names, `clean` and `cls`. Either can be used.

`clean cls:`

```
rm -f hellomake * .o *~ \#*
```

Note the `-f` option for `rm`. Why?



Each Makefile has a **default target**. The default target is the **first target** listed in the Makefile. Typically, that is a target called **all**. When you don't specifically identify a target for make, it will run the default target.

The all target should build all the executables in the Makefile.

A rule without any commands. It does have Prerequisites though.

all: hellomake

A Makefile can build more than 1 executable.

I really like to ask this on exams.



We'll go through
this simple
Makefile in detail.



```
all: hellomake

hellomake: hellomake.o hellofunc.o
    gcc -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.c hellomake.h
    gcc -c hellomake.c

hellofunc.o: hellofunc.c hellomake.h
    gcc -c hellofunc.c

clean cls:
    rm -f *.o hellomake *~ \#*
```

Notice the command line
options used with gcc.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
    gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
    gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
    gcc -c hellofunc.c
```

```
clean cls:
```

```
    rm -f *.o hellomake *~ \#*
```

These **MUST** be
tab characters!!!



Make sure there is at least 1 blank line between rules. The line should not have any spaces or tabs in it.

```
all: hellomake
hellomake: hellomake.o hellofunc.o
        gcc -o hellomake hellomake.o hellofunc.o
hellomake.o: hellomake.c hellomake.h
        gcc -c hellomake.c
hellofunc.o: hellofunc.c hellomake.h
        gcc -c hellofunc.c
clean cls:
        rm -f *.o hellomake *~ \#*
```

```
all: hellomake
```

The default target (all) is what runs when you just type make.

```
hellomake: hellomake.o hellofunc.o  
        gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h  
        gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h  
        gcc -c hellofunc.c
```

```
clean cls:  
        rm -f *.o hellomake *~ \#*
```

The rule to build
the executable file
`hellomake`.

`all: hellomake`

`hellomake: hellomake.o hellofunc.o`
`gcc -o hellomake hellomake.o hellofunc.o`

`hellomake.o: hellomake.c hellomake.h`
`gcc -c hellomake.c`

`hellofunc.o: hellofunc.c hellomake.h`
`gcc -c hellofunc.c`

`clean cls:`

`rm -f *.o hellomake *~ \#*`

all: hellomake

The targets and rules to build the object files from the .c files.

The hellomake.o file depends on the hellomake.c file and the hellomake.h file.

hellomake: hellomake.o hellofunc.o
 gcc -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.c hellomake.h
 gcc -c hellomake.c

hellofunc.o: hellofunc.c hellomake.h
 gcc -c hellofunc.c

clean cls:
 rm -f *.o hellomake *~ \#*

The hellofunc.o file depends on the hellofunc.c file and the hellomake.h file.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
          gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
           gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
           gcc -c hellofunc.c
```

```
clean cls:
        rm -f *.o hellomake *~ \#*
```

Clean up editor
droppings with the
clean target.

This is the order in which you'd typically place the entries in the Makefile.

all: hellomake

hellomake: hellomake.o hellofunc.o
gcc -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.c hellomake.h
gcc -c hellomake.c

hellofunc.o: hellofunc.c hellomake.h
gcc -c hellofunc.c

clean cls:

rm -f *.o hellomake *~ \#*

We'll step through this Makefile how it would be executed if we simply type `make` on the command line and the object files and executable file do not exist.

all: hellomake

hellomake: hellomake.o hellofunc.o
gcc -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.c hellomake.h
gcc -c hellomake.c

hellofunc.o: hellofunc.c hellomake.h
gcc -c hellofunc.c

clean cls:

rm -f *.o hellomake *~ \#*

We type `make` without any target, it will execute the default target, `all`.

The `all` target will execute the `hellomake` target.

all: hellomake

hellomake: hellomake.o hellofunc.o
 gcc -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.c hellomake.h
 gcc -c hellomake.c

hellofunc.o: hellofunc.c hellomake.h
 gcc -c hellofunc.c

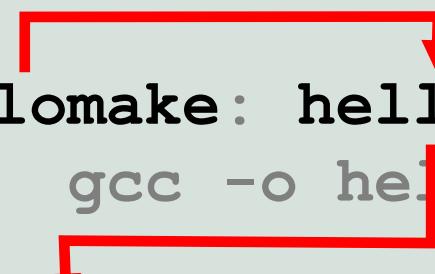
clean cls:

 rm -f *.o hellomake *~ \#*

The hellomake rule will check the hellomake.o file, find it is out of date, and will run the hellomake.o target.

```
all: hellomake
      hellomake: hellomake.o hellofunc.o
                  gcc -o hellomake hellomake.o hellofunc.o
      hellomake.o: hellomake.c hellomake.h
                  gcc -c hellomake.c
      hellofunc.o: hellofunc.c hellomake.h
                  gcc -c hellofunc.c

clean cls:
      rm -f *.o hellomake *~ \#*
```



The `hellomake.o` rule will check the `hellomake.c` and `hellomake.h` files, finding they are newer than the target.

There is not a `hellomake.c` target, so the command for the `hellomake.o` rule will be executed, building the `hellomake.o` file.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
```

```
    gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
→ gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
    gcc -c hellofunc.c
```

```
clean cls:
```

```
    rm -f *.o hellomake *~ \#*
```

The hellomake rule will check the hellofunc.o file, find it is out of date, and will run the hellofunc.o target.

all: hellomake

hellomake: hellomake.o hellofunc.o
gcc -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.c hellomake.h
gcc -c hellomake.c

hellofunc.o: hellofunc.c hellomake.h
gcc -c hellofunc.c

clean cls:

rm -f *.o hellomake *~ \#*

The `hellofunc.o` rule will check the `hellofunc.c` and `hellomake.h` files, finding it is out of date.

There is not a `hellofunc.c` target, so the command for the `hellofunc.o` rule will be executed, building the `hellofunc.o` file.

```
all: hellomake
```

```
hellomake: hellomake.o hellofunc.o
          gcc -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
          gcc -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
→ gcc -c hellofunc.c
```

```
clean cls:
```

```
      rm -f *.o hellomake *~ \#*
```

Now that the `hellomake.o` and `hellofunc.o` files are up to date, the command associated with `hellomake` will be run, producing the `hellomake` executable.

```
all: hellomake
      ↘
hellomake: hellomake.o hellofunc.o
→ gcc -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.c hellomake.h
gcc -c hellomake.c

hellofunc.o: hellofunc.c hellomake.h
gcc -c hellofunc.c

clean cls:
      ↘
rm -f *.o hellomake *~ \#*
```

The hellomake dependency and the all rule are now up to date. The hellomake executable has been built.

If you run make again right now, nothing will be rebuilt, as all targets are up to date.

all: hellomake

hellomake: hellomake.o hellofunc.o
gcc -o hellomake hellomake.o hellofunc.o

hellomake.o: hellomake.c hellomake.h
gcc -c hellomake.c

hellofunc.o: hellofunc.c hellomake.h
gcc -c hellofunc.c

clean cls:

rm -f *.o hellomake *~ \#*

In order to run the target `clean`, you issue the command:
`make clean`

Since there are no dependencies, the command will simply be run.

`all: hellomake`

`hellomake: hellomake.o hellofunc.o`
`gcc -o hellomake hellomake.o hellofunc.o`

`hellomake.o: hellomake.c hellomake.h`
`gcc -c hellomake.c`

`hellofunc.o: hellofunc.c hellomake.h`
`gcc -c hellofunc.c`

`clean cls:`

 `rm -f *.o hellomake *~ \#*`

The executable, object files, and editor droppings will be deleted.

- You put comments in your Makefile by using the # (octothorpe) character. Comments continue to the end of line.
- You can suppress showing the output from a command by putting a @ as the first character in the command.
 - **Don't do this. I will deduct points if you do.**

```
# This is a comment
clean cls:
    @rm -f *.o hellomake *~ \#*
```

Don't do this!



You can define variables to use in your Makefile.

Creates 2 variables you can use in the Makefile.

```
{ CC = gcc -Wall -g  
PROG = hellomake
```

```
all: $(PROG)
```

Yes, you can put spaces around the = operator in the assignment!

You must use braces or parentheses around variable names.

```
$(PROG) : hellomake.o hellofunc.o
```

```
$(CC) -o hellomake hellomake.o hellofunc.o
```

```
hellomake.o: hellomake.c hellomake.h
```

```
$(CC) -c hellomake.c
```

```
hellofunc.o: hellofunc.c hellomake.h
```

```
$(CC) -c hellofunc.c
```

```
clean cls:
```

```
rm -f *.o hellomake *~ \#*
```



Special make variables:

- \$@ - The name of the target of the rule (`hellomake.o`).
- \$< - The name of the first prerequisite (`hellomake.c`).
- \$^ - The names of all the prerequisites (`hellomake.c`
`hellomake.h`).
- \$? - The names of all prerequisites that are newer than the target.

This is NOT the same \$? you use in your bash shell scripts.



In addition to just being part of the ***cool crowd***, use of variables in your Makefile makes it easier to copy and reuse for other projects.

```
CC = gcc -Wall -g
```

```
PROG = hellomake
```

```
all: $ (PROG)
```

```
$ (PROG) : hellomake.o hellofunc.o
```

```
$ (CC) -o $@ $^
```



Special variables

The name of the target. The names of all the prerequisites.

```
hellomake.o: hellomake.c hellomake.h
```

```
$ (CC) -c $<
```

Special variable

The name of the first prerequisite.

```
hellofunc.o: hellofunc.c hellomake.h
```

```
$ (CC) -c $<
```

The name of the first prerequisite.

```
clean cls:
```

```
rm -f *.o hellomake *~ \#*
```

You can put other targets in your Makefile. The lazy among us might have variables and targets that look like these:

```
TAR_FILE = Lab3_${LOGNAME}.tar.gz
```

An environment variable.

```
tar: clean
      rm -f $(TAR_FILE)
      tar cvfa $(TAR_FILE) *.[ch] ?akefile
```

Notice that I use braces around environment variables and parenthesis around make variables. This is just my standard, not required.



Ever have your code “disappear” just before the due date? Revision control can help.

```
ci:  
    if [ ! -d RCS ] ; then mkdir RCS; fi  
    -ci -t-none -m"lazy-chicken" -l *.[ch] ?akefile
```

OR

```
git checkin chicken poultry:  
    if [ ! -d .git ] ; then git init; fi  
    git add *.[ch] ?akefile  
    git commit -m "my lazy git commit comment"
```



Dynamic Memory Allocation and Management with `malloc()` and Related Functions

`calloc()`

`realloc()`

`strdup()`

`free()`

and cousin `alloca()`

You've may have been using `new` and `delete` in your C++ programs to dynamically allocate and release memory while your program is running.

If you've only been using Java, you may have forgotten how to reclaim memory.

If you used Python, ... you have no idea.

In this class you'll be using `malloc()`, `calloc()`, `realloc()`, and `free()`.

You'll also probably like `strdup()`, a lot.

Kinds of Memory Allocation in C

Kind of memory	When is it allocated?	When is it deallocated?
automatic	created from the stack on a call to a function	deallocated on return from function
static	before main starts	when program terminates
dynamic	calls to malloc, realloc, or calloc	when the program calls free or terminates.

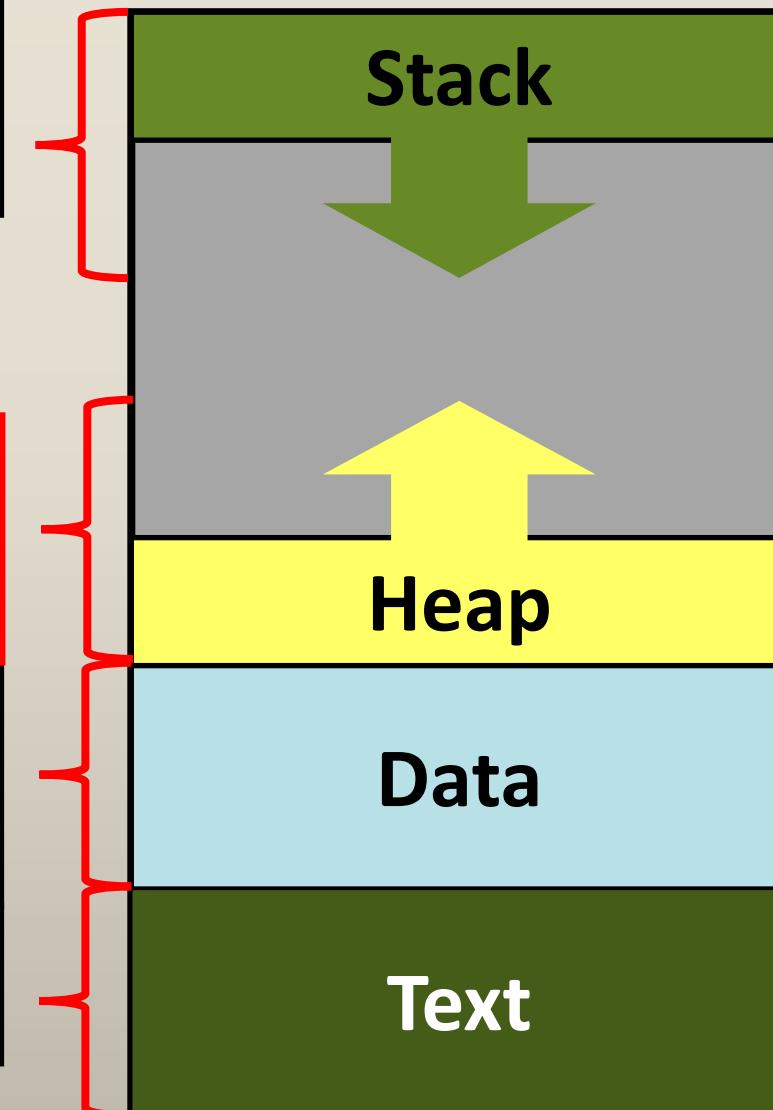
Memory Layout of a Process

Function call stack: activation records and automatic variables

Data returned from calls to **malloc**, **realloc**, or **calloc**

Initialized and uninitialized statically allocated data

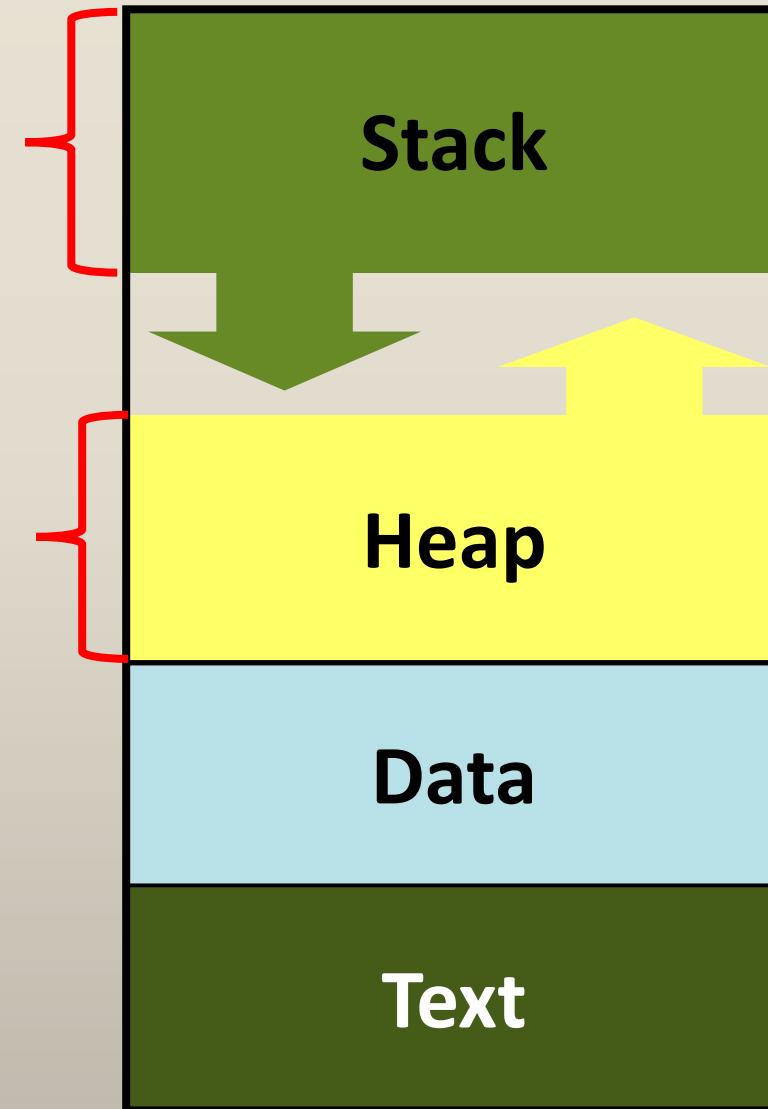
Your program instructions



Memory Layout of a Process

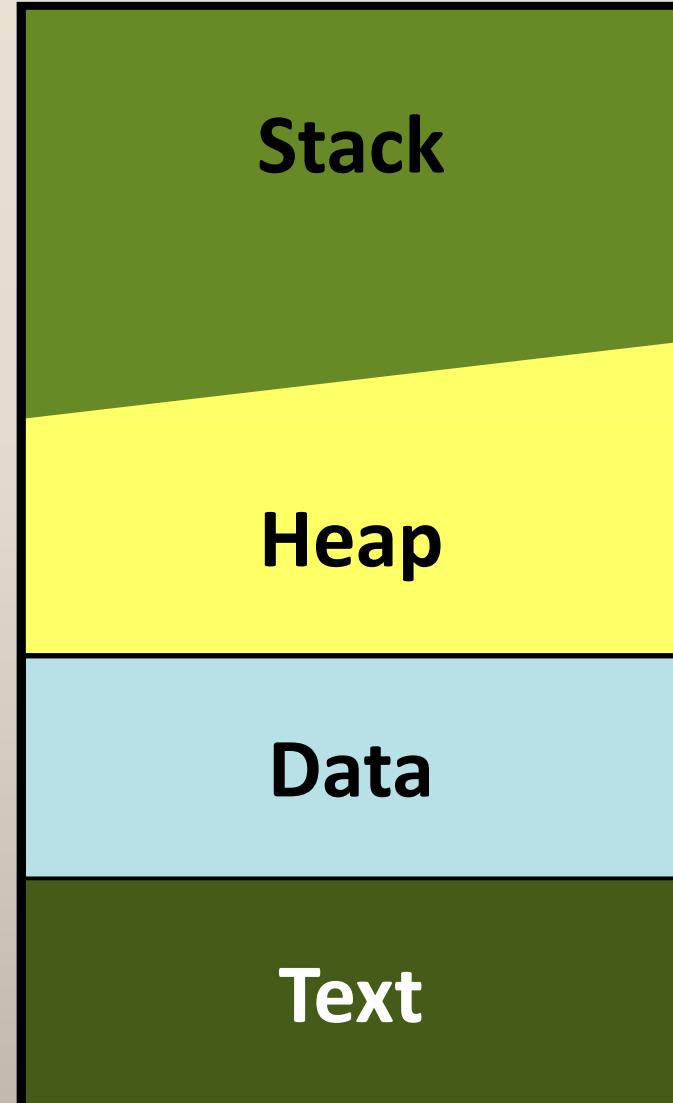
As your program calls functions and returns, the stack grows and shrinks.

As your program allocates dynamic memory, the heap grows. Your program consumes the heap.



Memory Layout of a Process

If the stack and heap meet or worse overlap, baaaaad things happen to your process.



In what Segment will the Address be Located

```

static int x;
int y = 0;

int main (int argc, char *argv[]) {
    int a = 0;
    foo (a);
    printf("a: %d\n", a);
}

void foo (int d) {
    int b = 2;
    int *z = malloc(sizeof(int));
    static int c = 0;
    *z = 5;
    printf("d: %d z: %d\n", d, *z);
    c++;
}
  
```

Address	Process Segment
x	
y	
main	
argc	
argv	
a	
foo	
b	
d	
z	
*z	
c	

The `sizeof()` Operator

Returns the **size** of a variable **or** datatype, **measured in the number of bytes** required for the type or structure.

I really like to ask about the `sizeof()` operator on exams.

I have an example in
`~rchaney/Classes/cs333/src/sizeof/sizeof.c`

The sizeof() Operator

```
char c;  
char *cp;  
int i;  
int *ip;
```

```
sizeof(char);           // Guaranteed to be 1.
```

```
sizeof(c);
```

```
sizeof(int)
```

```
sizeof(i);
```

```
// System dependent, but 4 for us.
```

```
sizeof(char *);
```

```
sizeof(cp);
```

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

```
sizeof(ip);
```

```
// All pointers are the same size.
```

```
// On our system, pointers are
```

```
// 8 bytes (64 bits).
```

The `sizeof()` Operator

```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;
```

```
account_t act;  
account_t *act_p;
```

Remember, all pointers
are the same size.

The sum of the sizes of the members of a structure may differ from the `sizeof()` for the structure.

```
sizeof(account_t);  
sizeof(struct account_s);  
sizeof(act);
```

```
sizeof(account_t *);  
sizeof(struct account_s *);  
sizeof(act_p);
```

The malloc () Call

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

What is a `void *`?

```
void *malloc(size_t size);
```

The allocated
memory is not
initialized

The number of **bytes** you
wish to allocate.

The `malloc()` function allocates **size** bytes and **returns a pointer to the allocated memory**.

If the call to `malloc()` fails, it returns a **NUL pointer**.

The allocated memory is not initialized.

The allocated
memory is not
initialized

Straight from the heap.

The allocated
memory is not
initialized

The void * Pointer

The void * pointer is a **generic** pointer in C.

A void pointer is a pointer that has no associated data type with it. A void pointer can hold an **address** of any type and can be **type cast** to any type.

The type of the data on the other side of a void * pointer is **opaque**, until the void * pointer is **type cast** to another type.

Wanna bet this is an exam question?

Type Casting in C

Type casting is a way to convert a variable from one data type to another data type.

```
float a = static_cast<float>(5)
      / static_cast<float>(2);
```

This is the C++ way to do a type cast.

```
float a = ((float) 5) / ((float) 2);
```

This is the C way to do a type cast.

The type **to** which you want to cast the value.

Type Casting in C

Type coercion is the **automatic** conversion of a datum from one data type to another within an expression.

```
double x = 1;
```

Type casting is an **explicit** type conversion **defined within a program**. It is defined by the user in the program.

```
double da = 3.3;
double db = 3.3;
double dc = 3.4;
int result = (int)da + (int)db + (int)dc; //result == 9
```

Type Casting in C

```
int i;  
int *ip = &i;  
void *vp;
```

```
vp = (void *) ip;
```

Type cast the int pointer
to a void pointer.

```
ip = (int *) vp;
```

Type cast the void pointer
to an int pointer.

Back to malloc()

I want to allocate a block of memory large enough for 1,000 characters and I want a block of memory for 1,000 integers.

```
char *cp = NULL;  
int *ip = NULL;
```

```
cp = malloc( 1000 * sizeof( char ) );  
ip = malloc( 1000 * sizeof( int ) );
```

Sadly, many compilers now let you get away without an explicit cast.

The malloc () Call

I want to allocate a block of memory large enough for 100 **pointers** to character arrays.

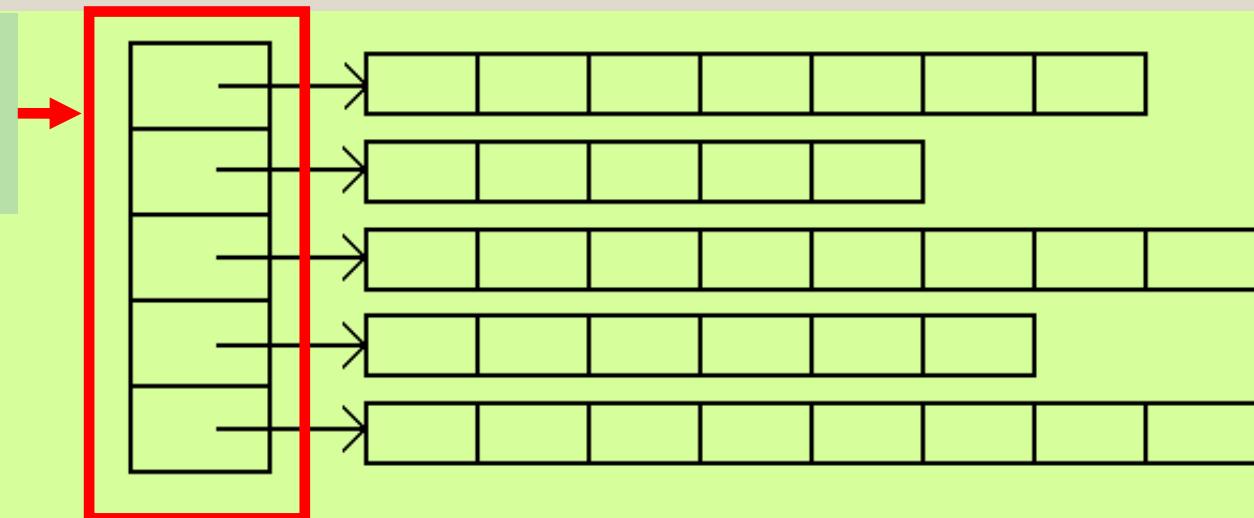
```
char **cp;
```

Notice this is a `char *`

```
cp = malloc( 100 * sizeof(char *) );
```

The call above only allocates this portion.

Does this look like a
ragged array?
It should!



The malloc () Call

I want to allocate a block of memory large enough for 1,000 `account_t` structures.

```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;  
account_t *act_p;  
  
act_p = malloc( 1000 * sizeof(account_t) );
```

The structure type.

The malloc () Call

```
typedef struct account_s {  
    int account_number;  
    char first_name[50];  
    char last_name[50];  
    float balance;  
} account_t;  
int i;
```

How you access the elements in the allocated array of account_t?

```
account_t *act_p = NULL;  
act_p = malloc( 1000 * sizeof(account_t) ) ;  
for (i = 0; i < 1000; i++) {  
    act_p[i].account_number = i;  
}
```

Notice the use of the dot, not
use of the ->

The memset () Call

The memory returned from a call to malloc () **is NOT initialized.**

You must assume that it is garbage.

If you want to set it to a constant value, use

Guess what often immediately follows a call to malloc () ?

NAME

memset - fill memory with a constant byte

This is a valid call:

```
memset(arr, ' ', sizeof(arr));
```

SYNOPSIS

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

```
void *memset(void *s, int c, size_t n);
```

DESCRIPTION

The memset () function fills the first n bytes of the memory area pointed to by s with the constant byte c .

RETURN VALUE

The memset () function returns a pointer to the memory area s .

Other mem* () Calls

In addition to the ever useful `memset()` call, there are other functions that work directly on memory.

NAME

memcpy - copy memory area

SYNOPSIS

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);
```



Not all memory is nice a NULL terminated string.

NAME

memcmp - compare memory areas

SYNOPSIS

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

The calloc () Call

```
#include <stdlib.h>  
void *calloc(size_t nmemb,  
            size_t size);
```

The allocated
memory **IS** initialized

Number of elements
in array.

The size of **each**
element of array.

size_t size

The `calloc()` allocates memory for an array of `nmemb` elements of `size` bytes each and returns a pointer to the allocated memory.

The allocated memory **IS** initialized to all zeroes.

The `free()` Call

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

```
void free(void *ptr);
```

De-allocates memory. Returns it so that it can be reused.

The `free()` call de-allocates the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc()`, `calloc()` or `realloc()`.

If `free(ptr)` has already been called before on that pointer, **undefined behavior occurs.**

Don't free memory that has already been deallocated.

The `free()` Call

The argument to `free()` **must** be the address of the **beginning** of a currently allocated block in the **heap**.

Blocks of memory deallocated by `free()` become available for future reallocation by `malloc()`, `realloc()`, or `calloc()`, this is a kind of recycling.

Calling `free()` with a `NULL` pointer is okay.



Common Mistakes Related to `free()`

1. Attempting to `free()` stack or static memory.

- Don't do it, `free` works only with heap memory. If you are lucky, `free()` will only silently fail.

2. Attempting to `free()` only part of a block.

- Don't try to free a pointer that points into the middle of a dynamically allocated block. It will be U-G-L-Y.

3. Attempting to `free()` a block that is already free.

- This usually causes trouble.

Common Mistakes Related to `free()`

Another big mistake related to `free()` is

Wait for it....

Forgetting to call `free()`!

This kind of mistake is called a memory leak.

The easy way to check for memory leaks is to run your code through valgrind. We'll cover valgrind in a lab.

The realloc() Call

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



```
void *realloc(void *ptr, size_t size);
```

Changes the size of the block of memory pointed to by `ptr` and it **copies** the contents of the old block into the new block.

The `realloc()` function changes the size of the memory block pointed to by `ptr` to `size` bytes.

The contents of the old pointer are **copied** into the newly allocated area.

The old block of memory will be automatically deallocated.

The realloc() Call

realloc **copies** a number of bytes from the beginning of the old block to the beginning of the new block. The number of bytes copied will be the old block size or the new block size, **whichever is smaller**.

When the new size is larger than the old size, the additional bytes are **not initialized** and should be assumed to contain garbage.

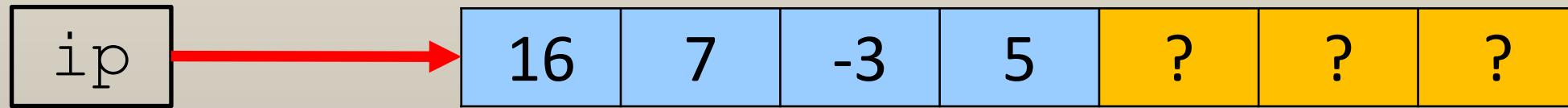
The realloc() Call

```
int *ip = NULL;  
ip = realloc( ip, 4 * sizeof(int) );  
ip[0] = 16; ip[1] = 7; ip[2] = -3; ip[3] = 5;
```



After the call to `realloc()` and some assignments.

```
ip = realloc( ip, 7 * sizeof(int) );
```



After the call to `realloc()`.

This works for pointers to pointers;
such as **ragged** arrays.

The `strdup()` Call

```
#include <string.h>

char *strdup(const char *str);
```

The `strdup()` function returns a pointer to a new string which is a **duplicate** of the string `str`.

Memory for the new string is obtained with `malloc()`, and should be deallocated with `free()`.

The amount of memory allocated will be only as large as is necessary to hold the data.

The `strdup()` Call

```
char *cp1, *cp2;
```

```
cp1 = strdup("This is a string");
```

```
cp2 = strdup(cp1);
```

```
...
```

```
free(cp1);
```

```
free(cp2);
```

Being a string, the
NULL is copied as well.

Since the memory was allocated
with `malloc`, it should be
deallocated with `free`.

The alloca () Call

```
#include <alloca.h>
```

Use of alloca () should be done only with great care.

```
void *alloca(size_t size);
```

The alloca () function allocates size bytes of space in the **stack frame** of the caller. This temporary space is **automatically freed** when the function that called alloca () returns to its caller.

Do not attempt to free () space allocated by alloca () !

The alloca () Call

Unlike the memory returned by malloc, calloc, and realloc, the memory returned by alloca is not from the heap, but **from the stack**. Do not try and use alloca for large chunks of memory.

Unlike memory returned by malloc, calloc, and realloc, you do not need to free the memory returned by alloca. It will be automatically returned to the stack when the function creating it returns.

The alloca () Call

- The alloca () function is super cool. However, you must be very careful about when you use it.
- Probably the best thing about alloca () is that its use does not fragment the heap.
- Since the memory allocated by alloca () comes from the stack, not heap, the heap will not become fragmented by its use.
- Do not use alloca () in a recursive function call.

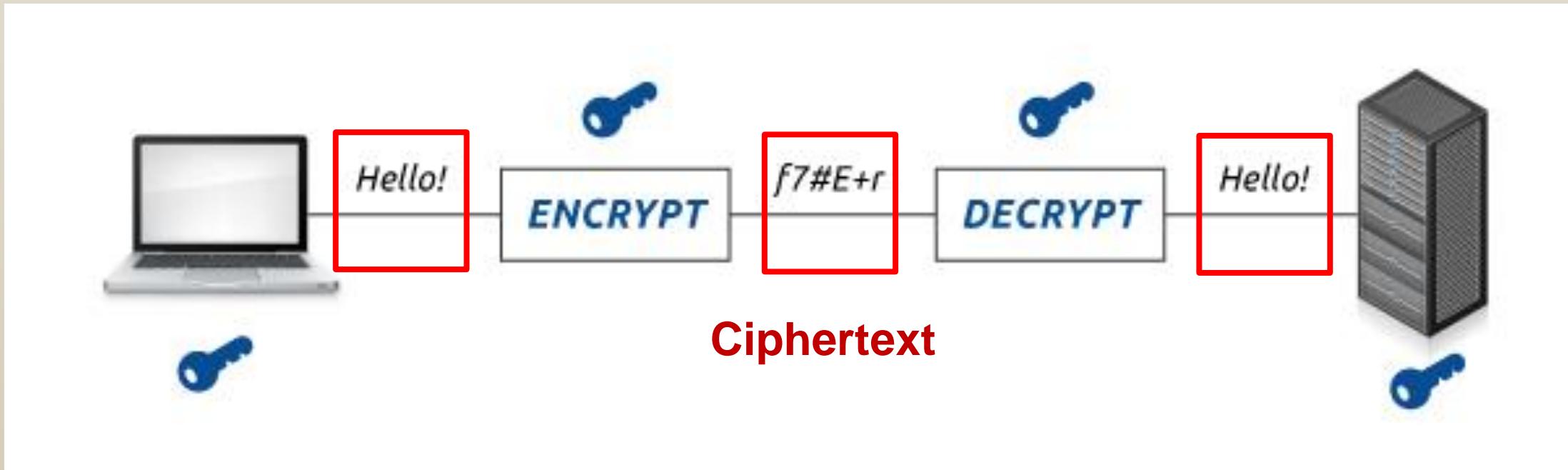


Encryption: What is it?

- The purpose of encryption is to **transform data in order to keep it secret from others**, e.g. sending someone a secret letter that only they should be able to read, or securely sending a password over the Internet. **Rather than focusing on usability, the goal is to ensure the data cannot be consumed by anyone other than the intended recipient.**
- Encryption **does not of itself prevent interception**, but denies the message content to the interceptor.
- With encryption, the intended communication information or message, referred to as **plaintext**, is encrypted using an encryption algorithm, generating **ciphertext** that can only be read if decrypted.

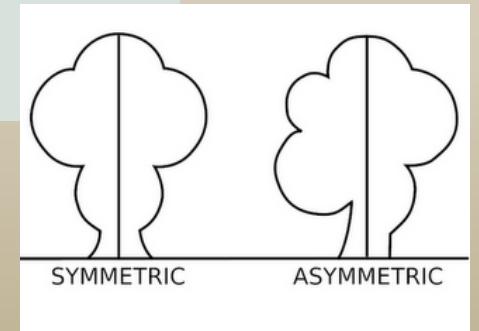
- **The purpose of encryption** is to ensure that **only somebody who is authorized** to access data (e.g. a text message or a file), **will be able to read it, using the decryption key.**
- Anyone not authorized is excluded, because they do not have the required key, without which it is impossible to read the encrypted information.





Symmetric-key algorithms are algorithms for cryptography that use the **same cryptographic keys for both encryption of plaintext and decryption of ciphertext**. The keys may be identical or there may be a simple transformation to go between the two keys.

Public key cryptography, or **asymmetric cryptography**, is any cryptographic system that uses **pairs of keys**: public keys which may be disseminated widely, and private keys which are known only to the owner.



Popular Encryption Algorithms

- **Advanced Encryption Standard (AES)** is the standard when it comes to **symmetric key encryption**, and is recommended for most use cases, with a **key size of 256 bits**. Also known by its original name **Rijndael**.
- **Data Encryption Standard (DES)**, is a **symmetric-key algorithm** for the encryption of electronic data.
 - **Now considered insecure**, it was highly influential in the advancement of modern cryptography. Published in 1977, has been replaced by AES. It uses a **56 bit key**.
 - DES was followed by **Triple DES**, which used **three 56 bit keys**.

- **Blowfish** is a algorithm designed to replace DES.
 - Blowfish symmetric cipher splits messages into blocks of 64 bits and encrypts them individually and a variable key length from 32 bits up to 448 bits.
 - Blowfish is known for both its speed and overall effectiveness, as many claim that it has never been defeated.
 - It is free availability in the public domain.

- **Twofish** has a block size of 128 bits with key sizes up to 256 bits.
 - It is a symmetric encryption technique, so only one key is needed.
 - Twofish is regarded as one of the fastest and well suited for use in both hardware and software environments.
 - Twofish is related to the earlier block cipher Blowfish.
 - Like Blowfish, the Twofish algorithm is free for anyone to use without any restrictions

RSA (Rivest–Shamir–Adleman) is a **public-key encryption** algorithm and a standard for encrypting data sent over the internet.

- It was one of the first practical public-key cryptosystems and is widely used for secure data transmission.
- Unlike Triple DES, RSA is considered an **asymmetric algorithm** due to its use of a pair of keys.
- It also happens to be one of the methods used in PGP programs.
- RSA is a **relatively slow algorithm**. Due to this, it is not commonly used to directly encrypt user data. More often, **RSA is used to transmit shared keys for symmetric-key cryptography**, which are then used for bulk encryption–decryption.

- **Pretty Good Privacy (PGP)** is the most popular public key encryption algorithm. PGP combines symmetric-key encryption and public-key encryption.

The message is encrypted using a symmetric encryption algorithm, which requires a symmetric key.

- Each symmetric key is used only once and is also called a session key.
- The message and its session key are sent to the receiver.
- The session key must be sent to the receiver so they know how to decrypt the message, but to protect it during transmission, the key is encrypted with the receiver's public key.
- Only the private key belonging to the receiver can decrypt the session key.

Hashing: What is it?



Hashing serves the purpose of **ensuring integrity**, i.e. making it so that if something is changed **you can know that if it has been changed**.

Hashing is a **one-way process**.

You hash plaintext into hashed text without the intention of ever returning the hashed text back into plaintext.

In fact, the objective is that you cannot turn the hashed text back into plaintext.

Hashing: What is it?

Technically, hashing takes arbitrary input and produces a fixed-length string that has the following attributes:

1. **The same input will always produce the same output.**
2. **Multiple disparate inputs should not produce the same output.**
3. **It should not be possible to go from the output back to the input.**
4. **Any modification of a given input should result in drastic change to the hash.**



Hashing is used in conjunction with authentication to produce strong evidence that **a given message has not been modified**.

- It can be accomplished by taking a given input, hashing it, and then sending the hash with the sender's private key.
- When the recipient opens the message, they can validate the signature of the hash with the sender's public key and then hash the message themselves and compare it to the hash that was signed by the sender.
- If they match it is an unmodified message, sent by the correct person.

- **Hash algorithms are one way functions.** They turn any amount of data into a fixed-length "fingerprint" that **cannot be reversed**.
- They also have the property that if the input changes by even a tiny bit, the resulting hash is completely different.



```
hash("hello") =  
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824  
  
hash("hblllo") =  
58756879c05c68dfac9866712fad6a93f8146f337a69afe7dd238f3364946366  
  
hash("waltz") =  
c0e81794384491161f1777c232bc6bd9ec38f616560b120fda8e90f383853542
```

Use Cases

- **Use a hash function when you want to compare a value** (ensure integrity) but can't store the plain text representation (for any number of reasons). **Passwords** fit this use-case very well since you don't want to store them plain-text for security reasons (and shouldn't).
- **Use encryption whenever** you need to get the input data back out.





Hashing Passwords

A key feature of **cryptographic hash functions** is that they should be very fast to create, and very difficult/slow to reverse (so much so that it's practically impossible).

For passwords, you don't want to get the plaintext back out, you just want to match with what you have.

However, you are not guarding against **rainbow tables or brute force attacks**. The hash function was designed for speed. It's easy for an attacker to just run a dictionary through the hash function and test each result.

A **dictionary attack** is a technique for defeating a cipher or authentication mechanism by trying to determine its decryption key or passphrase by trying hundreds or sometimes millions of likely possibilities, such as words in a dictionary.

A **rainbow table** is a precomputed table for reversing cryptographic hash functions, usually for cracking password hashes. Tables are usually used in recovering a plaintext password up to a certain length consisting of a limited set of characters.



The best way to protect passwords that you store in your database is to employ **salted password hashing**.

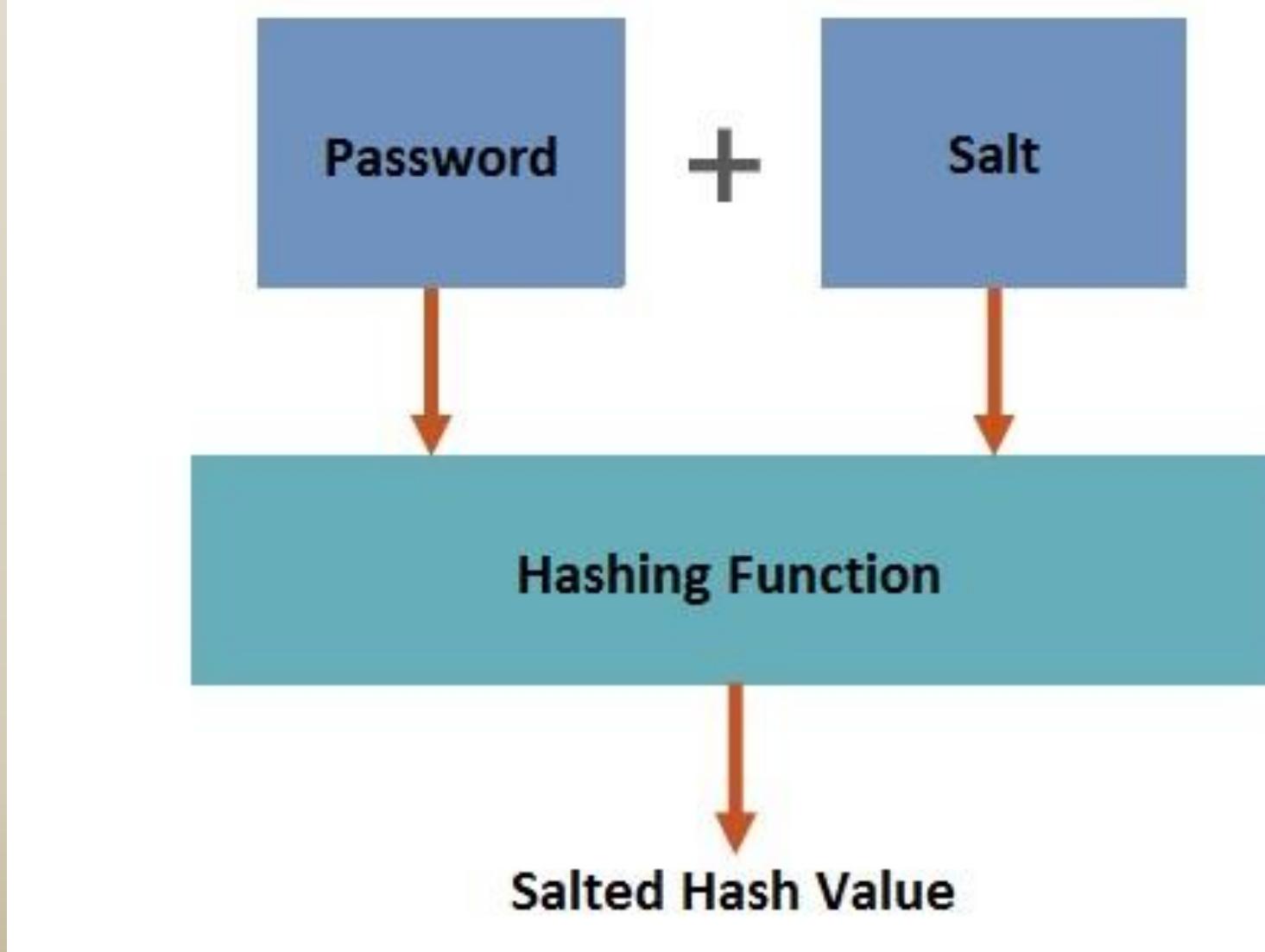


- In cryptography, a **salt is random data** that is used as an additional input to a one-way function that "hashes" a password or passphrase.
- The primary function of a salt is to defend against dictionary attacks or against its hashed equivalent, a pre-computed rainbow table attack.



- Lookup tables and rainbow tables only work because when each password is hashed in exactly the same way.
- If two users have the same password, they'll have the same password hashes.
- We can prevent these attacks by **randomizing each hash**, so that when the same password is hashed twice, the hashes are not the same.
- We randomize the hashes by appending or prepending **a random string, called a salt**, to the password before hashing.





- **A new salt is randomly generated for each password.** Typically, the salt and the password are concatenated and processed with a cryptographic hash function, and the resulting output (but not the original password) is stored with the salt in a database.
- Since salts do not have to be memorized by humans they can make **the size of the rainbow table required for a successful attack prohibitively large without placing a burden on the users.**
- Salts are different in each case, they also protect commonly used passwords, or those who use the same password on several sites, by making all salted hash instances for the same password different from each other.

How does the application authenticate a user with a password hash?

- When the application receives a username and password from a user, it performs the hashing operation on the **password with the salt** and compares the resulting hashed value with the password hash stored in the database for the particular user.
- If the two hashes are an exact match, the user provided a valid username and password.
- **The benefit of hashing is that the application never needs to store the clear text password.** It stores only the hashed value and the salt.

- It is common for an application to store in a file/database the hashed value of a user's password.
- Without a salt, a successful attack may yield easily crackable passwords.
- Because many users re-use passwords for multiple sites, the use of a salt is an important component of overall web application security.





The **WRONG** Way: Short Salt & Salt Reuse

- The most common salt implementation errors are reusing the same salt in multiple hashes, or using a salt that is too short.
- A common mistake is to use the same salt in each hash. Either the salt is hard-coded into the program, or is generated randomly once. This is ineffective because if two users have the same password, they'll still have the same hash.
- If the salt is too short, an attacker can build a lookup table for every possible salt.
- A good rule of thumb is to use a salt that is the same size as the output of the hash function.
 - For example, the output of SHA256 is 256 bits (32 bytes), so the salt should be at least 32 **random** bytes.

The salt should be generated using a **Cryptographically Secure Pseudo-Random Number Generator (CSPRNG)**.

- **CSPRNGs** are very different than ordinary pseudo-random number generators, like the C language's `rand()` function.
- **CSPRNGs** are designed to be cryptographically secure. They provide a high level of randomness and are completely unpredictable.
- You don't want your salt to be predictable, so you must use a **CSPRNG**.



The following table lists some CSPRNGs that exist for some popular programming platforms.

Platform	CSPRNG
PHP	<code>mcrypt_create_iv</code> , <code>openssl_random_pseudo_bytes</code>
Java	<code>java.security.SecureRandom</code>
Dot NET (C#, VB)	<code>System.Security.Cryptography.RNGCryptoServiceProvider</code>
Ruby	<code>SecureRandom</code>
Python	<code>os.urandom</code>
Perl	<code>Math::Random::Secure</code>
C/C++ (Windows API)	<code>CryptGenRandom</code>
Any language on GNU/Linux or Unix	Read from <code>/dev/random</code> or <code>/dev/urandom</code>

- The salt must be **unique per-user and per-password**.
- Every time a user creates an account **or changes their password**, the password should be hashed using a new random salt.
- **Never reuse a salt.**



```
# ./thread_crypt -h
help text
./thread_crypt ...
Options: i:o:hva:l:R:t:r:
-i file           input file name (required)
-o file           output file name (default stdout)
-a [01356byg]    algorithm to use for hashing
                  see 'man 5 crypt' for more information
-l #              length of salt
                  valid length depends on algorithm
-r #              rounds to use for SHA-256, SHA-512, or bcrypt
                  valid rounds depends on algorithm
-p str            parameters to use for yescript or gost-yescrypt
                  (default is "j9T")
-R #              seed for rand_r() (default 3)
-t #              number of threads to create (default 1)
-v               enable verbose mode
-h               helpful text
```

-a [01356byg] algorithm to use for hashing
see 'man 5 crypt' for more information

- 0: DES – the default
- 1: md5
- 3: NT
- 5: **SHA-256**
- 6: **SHA-512**
- b: bcrypt**
- y: yescrypt**
- g: gost-yescrypt**

```
// DES
// AAA:ejb9F5m6VukDU
//
// md5
// AAA:$1$cjPb7q2Y$wguL9h9L8HOdi梓VyMegC5/
//
// sha256
// AAA:$5$rounds=5000$kUU/R2mSMwWNGRXk$/R1BqMHS6yqMQavJtYV9BDs7c2g2RBCz6LOJeHSilD
//
// sha512
//
AAA:$6$rounds=5000$kUU/R2mSMwWNGRXk$.USaGis3Nj7XKqufpEdnu3SEjNozE1Zi1qHppwm5e1I8y2
/XPmYbYstYEkTqjKPtnxP.G8jgKOEs1Nm1N0XYt.
```

```
// md5 (algorithm 1)
// ./thread_crypt -a 1 -i words10.txt -R 3
// AAA:$1$cjPb7q2Y$wguL9h9L8HOdizVyMegC5/
//
// ./thread_crypt -a 1 -i words10.txt -R 3
// AAA:$1$cjPb7q2Y$wguL9h9L8HOdizVyMegC5/
//
// ./thread_crypt -a 1 -i words10.txt -R 4
// AAA:$1$qhdT904r$.A9oIPMG/UxbZHVz3u1ch1
//
// ./thread_crypt -a 1 -i words10.txt -R 5
// AAA:$1$0Y8iMhNp$TDZcBe5Q8ze3E3XfyMUOg/
//
// ./thread_crypt -a 1 -i words10.txt -R 6
// AAA:$1$.Krsz2g2$eIrfBwQ4hML88fs.hV7371
```

Using the same seed to rand() results in the same salt, resulting in the same hashed password value.

Using a different seed to rand() results in a different salt, resulting in a different hashed password value.

Delimiters for the fields within the result from `crypt()`.

```
// sha256
//
// AAA:$5$rounds=5000$kUU/R2mSMwWNGRXk$/R1BqMHS6yqMQavJtYV9BDs7c2g2RBCz6LOJeuHSi1D
```

The algorithm used.
In this case sha256

The salt. In this case, it is 16 characters long, the maximum for both sha256 and sha512.

The hashed password. The plaintext password was combined with the salt and hashed 5000 times to produce this value.

The number of times the password was hashed by the algorithm. A higher number takes longer, making the passwords harder to crack.

```
#include <crypt.h>

char *
crypt(const char *phrase, const char *setting);

char *
crypt_r(const char *phrase, const char *setting, struct crypt_data *data);

char *
crypt_rn(const char *phrase, const char *setting, struct crypt_data *data
      , int size);

char *
crypt_ra(const char *phrase, const char *setting, void **data, int *size);
```

You **won't** be using `crypt()`. It is not thread-safe.

I used **crypt_rn()**

From `man crypt`

The data argument to `crypt_r` is a structure of type `struct crypt_data`. It has at least these fields:

```
struct crypt_data {  
    char output[CRYPT_OUTPUT_SIZE];  
    char setting[CRYPT_OUTPUT_SIZE];  
    char phrase[CRYPT_MAX_PASSPHRASE_SIZE];  
    char initialized;  
};
```

They **lie** like a rug!

The data member listed in the man page as **phrase** is actually **input** in the `crypt.h` include file.



There is a `crypt_gensalt()` function, but **you must NOT use it**, for 2 reasons:

1. I want your code to be repeatable. This usually makes it easier to debug.
2. I want you to see how a salt is actually generated. It is not very hard.

To this end, you will be using the `rand()` function to “randomly” select characters from the valid salt characters.

Yes, I know this not a CSPRNG, we are setting that aside for some learning right now.

```
# define SALT_CHARS \
    "./ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyz"
static const char salt_chars[] = {SALT_CHARS};

switch (salt_algo) {
case 1: // md5
case 5: // sha256
case 6: // sha512
...
    ...
    ...
    for(int i = 0, rand_value = 0; i < salt_len; ++i) {
        rand_value = rand();
        rand_value %= strlen(salt_chars);
        s[i] = salt_chars[rand_value];
    }
    sprintf(salt, "$%d$%s%s$", salt_algo
        , (rounds_str ? rounds_str : ""), s);
    break;
default:
    rand_value = rand();
    rand_value %= strlen(salt_chars);
    salt[0] = salt_chars[rand_value];
    ...
    ...
}
}
```



The Never Ending Story

Even if you have the very best salted, hashed, and hash algorithm in place, it will get old and frail.

Your customers' data security is only as safe as your vigilance will maintain.

