Basic C

Tuesday, 22 February 2022 9:00 AM

Main

In C, there is always a main function

C files should end in .c
Filenames are case sensitive

Vim used for editing code on all linux/unix machines

Can use other coding environments but some systems may only have vim as it's been around for a long time and very lightweight.

Printing

printf is printing with formatting

Escape Characters

Start with /
e.g.

- \n
- \t
- \r
- \\

Place Holders

Start with %
These replace an expression

e.g
printf("3+5=%d\n",8

Output

Numbers

Type	Symbol		
int	%d or %i		
unsigned int	%u		
double	%e	$12.34 \rightarrow 1.234000e+01$	Scientific notation
	%f	$12.34 \rightarrow 12.340000$	
	%g	$12.34 \rightarrow 12.34$	Combination
char	%c	'c'→c	
		99→c	ASCII

- More C types later

Casting

(type) expression

e.g.

(int) log10(54)+2

Manual Pages

man
Is the manual for commands in C

Man sections

e.g.

$ man log10

- #include - which header file is needed
- Link with - do we need to include additional libraries

GCC Options

$ gcc digit.c -lm -o digits

- -lm - link in the m (maths) library (libm.so)
- -o digits - call the output file digits instead of default

Expressions

An expression is a fragment of code which can be evaluated to get a value

e.g.
Literals
1.2, "strings here", `A`
Variables cost

Expressions can be combined
a+b
y+m*x
Function calls
Get_cost(3)
Costs[3]

Types

In C, expressions and variables have explicit types

- Int - integer
- Unsigned int - unsigned (non-negative) integer
- Char - character
- Float - single precision floating point number
- Double - double precision floating point number
- …

Variables must be declared before use (different from Python)

- Int x
- Int y = 5;

Global variables do not require initialisation
Local variables should be initalised before use in C as it is not guaranteed what their value will otherwise be

Arrays

Int numbers[100];
Char str[50];

Accessing members: use [] notation

Int first = numbers[0];

Note

- The size of the array must be specified when creating it
- The size is part of the type so int[3] and int[4] are different types
- Cannot compare whole arrays. Need to compare element by element

Array Initialisation

Arrays can be initialised when declared

```
int a[3] = { 1, 3, 5 }; // or just int a[] = {1, 3, 5};
```

Size need not be given when an initaliser is given - size can be inferred

```
char str[] = "hello";
char str[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

Strings are an array of characters - with a null character to terminate them

```
// The following declarations are all the same
char str[6] = "hello";
char str[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
char str[] = "hello";
char str[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

For loops

C uses for not a for each loop as in python and java

```
for (int i = 0 ; i < 4; ++i) {
    totalA += values[i];
}
```

1. `int i = 0` — Done once at the beginning of the loop
2. `i < 4` — If this is false, stop the loop
3. `totalA +=...` — Loop body
4. `++i` — Done after the loop body. Now jump to 2

However, all three parts of the loop header are optional

```
for (;remain > 0;)
```

```
for (;;) // loop forever
```

```
for (A; B; C) {
    BODY
}
```

is equivalent to:

```
A;
while (B) {
    BODY;
    C;
}
```

While loops

```
do {
    BODY
} while (TEST);
```

This loop will execute BODY at least once. Vs:

```
while (TEST) {
    BODY;
}
```

Which might not execute BODY at all (If TEST fails the first time)

Booleans

Thursday, 3 March 2022 10:53 AM

The bool type in C can be used for variables where there are only two possible values (i.e. true and false)

In order to use **literals** true, false and the bool type in C, you need to

#include <stdbool.h>

This is because C didn't always have bools, and it was added later on. So in order not to break existing programs, it is included as a library.

In C, any expression which produces a numeric value can be interpreted as "true" or "false"

== 0 --> false
!=0 --> true

Pointers

Thursday, 3 March 2022 10:56 AM

Parameters of main()

Main always takes 2 parameters - together they describe an array of strings

- Int argc: the number of strings in the array
- Char** argv or char* argv[]: the array itself
- Argv[0] is the program being run

See arg1.c

- %s - placeholder for a string
- C arrays are not range checked
- Generally you can't reliably ask how big an array is (hence argc)

What are Pointers

A pointer

- Is a value
- That is the memory address where another "thing" can be found (e.g. your address is both a value and a reference to the location where you live
- Has a type
    - What sort of thing does the pointer "point to"
    - Int* and char* are different types but both are pointers

Declaring Pointers

Int* var;

Declares var to be a variable which stores a "pointer-to-int"

As a short hand, programmers will often refer to variables by the type they store

But understand the difference

Getting valid pointers

- Malloc() - find me some memory I can use and give me a pointer to it
- P gives the value(address) of the pointer
- *p gives the thing pointed to
- Malloc() doesn't ask what type of value you want to store (just how much space you need)
- Use sizeof to find out how much space something takes
- The return value from malloc should (probably) be cast
- Memory acquired via malloc() is called **Dynamic Memory**

Malloc() and friends

```
$ man malloc

NAME
       malloc, free, calloc, realloc - allocate and free
                dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *malloc(size_t size);
       void free(void *ptr);
       void *calloc(size_t nmemb, size_t size);
       void *realloc(void *ptr, size_t size);
       void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

Clean up

Memory leaks
Malloc'd memory is not cleaned up until

- It is explicitly released using free()
- The program ends
- Memory can be free'd in another function, provided the pointer is known

Dangling Pointers

Don't do the following

```
p3 = p2;
free(p2);
// use p3;
```

Once any pointer to a chunk of memory is passed to free(), all pointers to that address are invalid. You can't tell this by looking at them!

Dynamic Arrays

For a 10 element array of int

```
int* arr = (int*)malloc(sizeof(int) * 10);
...
// arr[0]...arr[9] will be valid.
...
free(arr);
```

The name of the array can be treated as a pointer to the first element

Some Pointer Arithmetic

```
// arr points to an array of 5 ints
int* arr=(int*)calloc(5, sizeof(int));

// [ ] operator works on pointers too!
arr[0] = 5;

// "Last" element in the array
arr[4] = 7;

//hmm
arr[400] = 5;
```

Negative Indices

Negative indices do not count from the end like they do in Python

Negative indices in C literally go backwards from the 0th element

Pointers and Array Equivalence

To find an indexed location in an array, calculate

&(arr[index]) → start of array + index * size of element

When you add an integer to a pointer, C moves forward in multiples of the size of the type pointed at

e.g.

```
int* p=1024;
char* c=1024;

c + 1 == 1025    // sizeof(char)==1
p + 1 == 1028    // on 32bit machines
```

Pointers don't check how much memory they point at, so you can construct a pointer to any location based on where it is relative to a known pointer

```
// arr points to an array of 5 ints
int* arr=(int*)calloc(5, sizeof(int));

// These are the same
arr[0] = 5;
*arr = 5;

// so are these
arr[4] = 7;
*(arr + 4) = 7;

// and so are these
int *p=&(arr[1]);
p[-1] = 42;
arr[0] = 42;
```

Null Pointer

There is a special pointer called the null pointer

It **never** points to anything valid
It can be written in a number of way, but the standard says that **0** is where you expect a pointer will be treated as the null pointer

New pointers should be initialised to null unless you have a proper value for them.

```
int* v = 0;

if (v == 0) ...
```

<stddef.h> defines the macro NULL - use it!!

```
int* v = NULL;
if (v == NULL) { ...

// This is also very common idiom
if (!v) {...
```

The last part with the if statement is just saying if the pointer v is not equal to nothing (opposite from the first if) but more concise and easier to read).

Structs

Thursday, 24 March 2022 11:00 AM

Pointers and Structs

Structs are variables that can hold several data items of different kinds. Kind of like an array but arrays can only hold data of the same kind.

You might use a struct to represent a record like a student instance

- Student ID
- Room #
- Degree studying
- Origin
- Name
- DOB

Etc.

Here is an example:

```
struct Data {
    int length;
    char* str;
};

    struct Data d1;
    struct Data* d2=malloc(sizeof(struct Data));
        // modify the length field
    d1.length = 4;
    *d2.length = 4;      // ?? - no
    (*d2).length = 4;    // legal
    d2->length = 4;      // easier to read
```

-> the pipe is just accessing a member of a struct

It's the same as using `a.length <-> a->length`
But easier to read

Here's an example of it being used

```c
struct Node {
    int value;
    struct Node* next;
};
```

```c
// Want to change the value 3 hops along from n
    Node* n = ...
    ...
    (*(*(*(*n).next).next).next).value = 4;
// vs
    n->next->next->next->value = 4
```

Pointer related functions

```c
void *memset(void *s, int c, size_t n);
```

- Set a chunk of bytes to a chosen value

```c
void *memcpy(void *dest, const void *src, size_t n);
```

- Copy bytes at one pointer to another I Buffers must not overlap! I use **memmove()** if they do, or might

Where are the variables stored?

Main places

- Global variables and literals (etc)
- Function local variables (includes function parameter variables)
  - Allocated on the stack - a downwards growing temporary storage space
  - Space allocated when the function is called, released when exiting/returning from the function
- Dynamically allocated storage (malloc(), free ())
  - Heap
  - Only cleaned up when explicitly told to or the program finished
  - Can store much bigger things than the stack can

Pointers to Existing Variables

```c
int v1 = 7;
printf("v1=%d is at address %p", v1, (void*)&v1);
int* v2 = &v1;
*v2=14;
printf("v1=%d is at address %p", v1, (void*)&v1);
```

Note: pointers can pointer anywhere in memory, including to the stack.

Void*

Void is used to indicate the lack of something:

- Void fn… - this function doesn't return a value
- Int fn(void) - this function doesn't take any parameters

Void* is a pointer without a type

- Do not dereference a void*
- How C deals with functions which take varying types

The `%p` placeholder wants a void pointer (hence the case in the previous example)

Parameter Passing

All C function parameters are passed by value

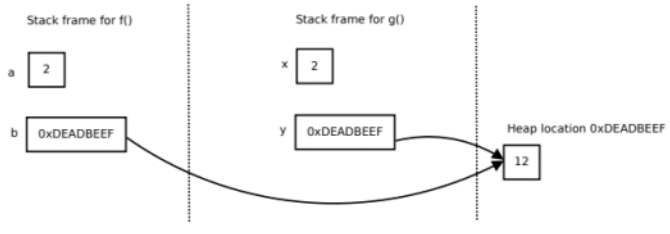
You just need to know what is being passed

```
void g(int x, int* y);

// inside some other function f()
int a;
int* b = malloc(sizeof(int));
g(a, b);
```

The values of a, b will be copied into x, y respectively.

This applies to the contents of structs.

```
void g(int x, int* y) {
    x = 5;
    *y = 10

void f(void)
    int a=2;
    int* b =
        malloc(sizeof(int));
    *b=12
    g(a, b);
}
```



In this example, when g returns to f:

- A will be 2
- B will be 10

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

In this example, when swap returns, the variables passed in won't be swapped because they are only being swapped in a local scope.

To do this, you would need to pass in the memory address of each of x and y. Then in the swap function, change the values at those memory locations

Strings

Thursday, 24 March 2022 11:00 AM

Strings

- Stored in arrays of char
  - Shortened (incorrectly) to "are arrays of char"
- Well-formed strings end in a terminator byte "\0"

Not all arrays of char hold proper strings

e.g.

```
char buffer[7]; strcpy(buffer, "Hello");
```

```
buffer[0]=='H'
buffer[4]=='o'
buffer[5]=='\0'
buffer[6]====
```

- Strings don't store their length
- String length must be calculated every time
- Strlen(s) - finds length by counting chars until a terminator is found. e.g.

```
int len(char s[]) {
    int i=0;
    for (;s[i] != '\0'; ++i) {
    }
    return i;
}
```

```
char buffer[6];
strcpy(buffer, "Hello");
char* greet=buffer;
printf("%s %s\n", greet, buffer);
greet=&(buffer[2]);
printf("%s %s\n", greet, buffer);
```

See greet.c

The type of strings is most commonly given as char*

Pitfall declaring multiple pointers

Int x, y, z;

Declares x, y, and z to be ints. Ok. So what about

Int* x, y, z?

The * only affects the variable directly to its right. So we'd end up with x being int* and y and z being ints

Approaches. If you wanted them all to be pointers

- Int* x, *y, *z
- Typedefs

Typedef

Typedef actualtype newname
e.g.

```
typedef char* cptr;

// later

cptr x, y, z;
```

All three vars are char* this way.

Structs

Can also use typedef on structs

```
typedef struct {
    int id;
    double gpa;
    char* name;
} Student;

// later

Student s1;
```

```c
typedef struct Node {
    int value;
    struct Node* next;
} Node;

// Later

Node n;
```

Note that you can't use typedef'd name inside the struct.

Revision Control - Subversion (SVN)

Wednesday, 9 March 2022 1:17 PM

Version Control Systems

- Store development history of a project
  - As a series of commits/revisions
- Allow retrieval of previously committed states.
- Report differences between revisions
- Multiple parallel lines of development (branches) / merging
  - Beyond the scope of this course

Subversion is one VCS among many, but it is the only supposed one in this course

SVN Pieces

Repository

- Where history is is stored
- Only manipulate via `svn` commands
- Located via a URL
- In this course you each have a repo at
  - https://source.eait.uq.edu.au/svn/csse2310-sem1-s4698512
  - Not for browser to access. Use SVN client
  - You can browse it however, via https://source.eait.uq.edu.au/viewvc/csse2310-sem1-s4698512
- Working copy / copies
  - Where you do your editing / compiling / testing
  - Record "good" states back to the repo with commits
  - Nothing you modify in your working copy affects the repo unless you commit.
  - You can delete your working copy and it will not affect the repo (will lose any uncommitted changes though)
  - There could be multiple working copies checked out.

SVN Operations

- `Svn checkout URL [working directory]`
  - Make a workinc copy of the most recent version
  - e.g.: `svn checkout https://source.eait.uq.edu.au/svn/csse2310-sem1-s4698512/play 2310play`
    - Creates directory `2310play` that is the working directory of the given part of the repo
- `Svn add filename`
  - Tells svn to track changes to this file
  - Doesn't take effect until you commit
- Svn mv oldname newname
  - Rename or move a file
  - Need to cmmit to make the change in repo.
- Svn rm fname
  - Remove file locally and remove it from future repo versions
  - Can not remove it from past versions

- Svn status
  - Shows which files have pending changes
  - M - modified
    A - untracked file to be added
    D - file to be removed
    ? - don't know anything about this file
- Svn diff
  - Show the lines which have changed
- Svn commit
  - Send pending chnages from working copy to the repo
  - Willa sk for a log message (edit and save)
  - Or svn commit -m "Meaningful message about changes here"
  - Can commit only specified files with: svn commit f1 f2 f3
- Svn log filename
  - Show the log messages for filename
- Svn revert filename
  - Undo any pending changes to filename
  - Can't be reversed
- Svn update
  - Bring working copy up to date with the latest version in the repo
  - Generally only needed if you've been using multiple working directories
  - Can lead to conflicts.

Putting it back

- Svn update -r14 filename
- Make a copy of that version of the file
  - Cp filename backup
- Svn update filename
- Copy the older version over
  - Cp backup filename
- Svn commit filename

Revision Control Best Practices

Revision control is most useful when you use it well

- Every commit should at least compile
- We recommend never being more than an hour (or less!) from working cod
  - Commit early, commit often
- Ideally, every commit should be runnable (i.e. don't break stuff)
  - This allows something called bisecting
- Commit messages **must be meaningful**
  - Describe **what** and **why** rather than **how** the code has changed
    - Change in the code itself can be easily seen (svn diff …)
    - No need to mention the files that have changed - also easily seen - but it *is* ok to mention function names, data types etc.
- Commits should be atomic
  - Try not to have multiple changes in one commit, e.g. don't fix a bug and add features in one commit.
    - Make roll-back and bisecting easier, e.g. if you introduce a new bug when adding features.

Aside - Bisecting

For large projects, this is incredible useful but only if

- Each commit can compile and run
- You have a good way of testing

Bisecting is so useful that git supports it natively.

You can use bisecting to figure out which commit broke the program if the above conditions are met.

Writing Good Commit Messages

- Think in terms of the following questions
    - Why will the change be made?
        - Fix bug? Add feature? Change style? Refactor? Updated documentation?
    - What effect will the change have?
        - What is the change in functionality? Or What bug is fixed?
- Write in active voice, present tense
    - Phrase the message to complete the sentence "If applied, this commit will…"

MakeFiles

Wednesday, 9 March 2022 2:19 PM

What are MakeFiles?

- Automates building of your program - all you need to do it type **make** or perhaps **make target**.
- Make will look for a file called **Makefile** or **makefile** in the current directory.
- The makefile specified the steps needed to build your program
- Can specify prerequisites (dependencies so that only build if dependency changes
- More detail later in the semester, but you will need a simple makefile for your first assignment, so we'll cover the basics here.

A Basic MakeFile

- Makefiles are made up of a set of rules
- A rule has
  - A **target** - the file to be generated or the name of an action to be carried out
  - **Prerequisites** or **dependencies** - files that must exist , or actions that must be carried out before building the target.
    - No prerequisite means recipes will always run if building target.
  - Recipes - how to build the target once prerequisites are satisfied
    - Zero or more commands to be executed
    - These are indented with a **tab** character

```
target: prerequisite ...
        recipe
        recipe
        ...
```

Example makefile

```
all: tictactoe
tictactoe: tictactoe.c tictactoe.h
        gcc -Wall -pedantic -std=gnu99 -o $@ $<
        # $@ = name of target, $< = name of first prereq
        # line above same as
        # gcc -Wall -pedantic -std=gnu99 -o tictactoe tictactoe.c

clean:
        rm tictactoe
```

- First target is the default target (target to be built if you type make)
- Comments begin with #

Multi-dim Arrays

Saturday, 19 March 2022 9:50 AM

An $m \times n$ array of int
Three options

1. Int arr[m][n] - 2d array, size fixed at compile time
    a. Not convertible to int*
2. Int* arr = malloc(sizeof(int)*m*n) - fake it with a 1D array
3. Int** arr = malloc(sizeof(int*)*m) - array of arrays.

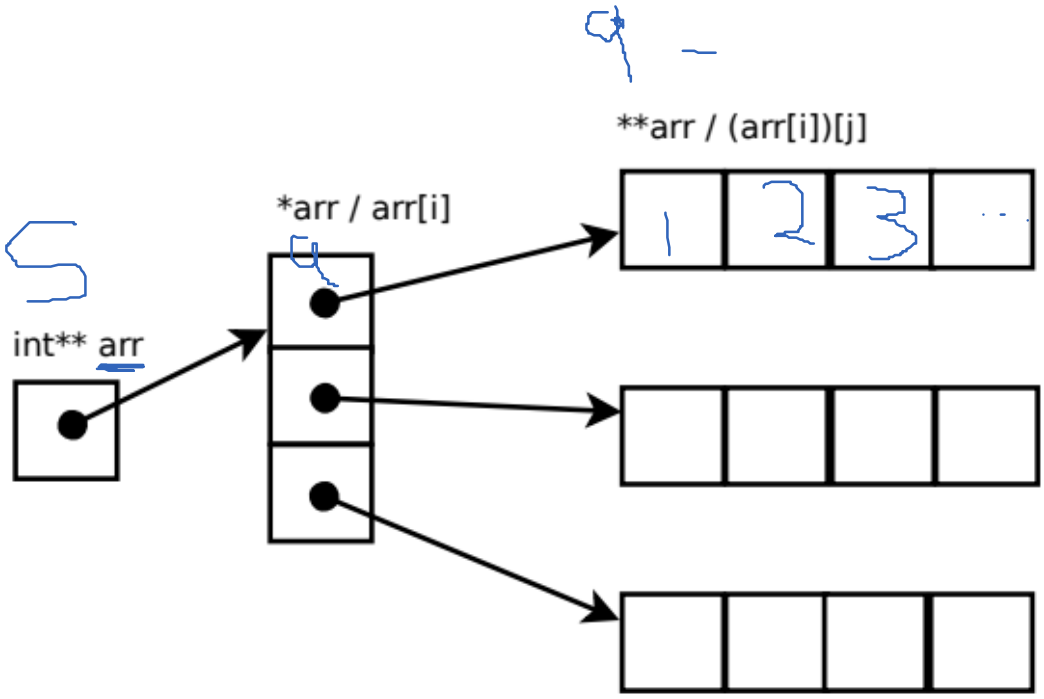
Fake 1D Version

int* arr = malloc(sizeof(int)*M*N); // lookup arr[i][j] arr[i*M+j] free(arr);

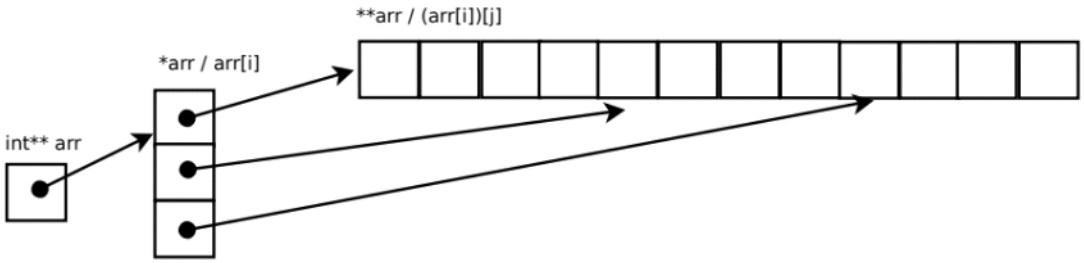
Array of Arrays

```
int** arr = malloc(sizeof(int*)*M);
for (int i=0;i<M) {
    arr[i]=malloc(sizeof(int)*N);
}

// lookup arr[i][j]
arr[i][j]

for (int i=0;i<M;++i) {
    free(arr[i]);
}
free(arr);
```

Think of it like this



Can also do it like this but not as good



This is because to resize the array would be a pain

FILEs

Saturday, 19 March 2022 10:04 AM

FILE*

- The type for C standard I/O is FILE*
  It should be treated as a n opaque type (i.e don't try to dereference it or look inside)
- To interact with a file, use fopen() to get a FILE* for it
- Fclose() the FILE* when you are finished with it
- Stdio.h defines three special FILE * variables that are always available
  - Stdin
  - Stdout
  - Stderr

Fgetc()

Fgetc takes a file and reads one char from it

- It returns either a char or the special value EOF
- This is why it needs to be an int
- For fun, try printf("%d\n", EOF);

When is it EOF?

Foef(f) is true when the program has tried to read from f and failed because it was at the end

Stdio tests for the edge of a cliff by asking "are we calling?".

Comma operator for fun and profit

See copyf2.c

Evaluating expr1, expr2

- Evaluate expr1
- Throw the result away
- Evaluate expr2

Is (5+3), 7 pointless?

- Yes
- Useful if the first expression has a side effect (or affects the second expression)
- Eg: ++things, things > 2

Error Checking

Fpoen() returns the null pointer if it can't open the file.

It wills et the errno variable to indicate what the problem was

```
#include <errno.h>

FILE* fin = fopen(…);
If (fin == 0) {
```

```
        Perror("Opening file:"); // what happened
        Return;
}
```

It will give a name you can look up in man pages

See copyf4.c

Output functions

Fprintf(FILE*, const char* format, …)
Fputc()
Fputs()
Fwrite()

Consult man pages for parameter order

Input Functions

Fgetc()
Fgets()
Fread()

Fscanf()

See scandemo.c

Notes:

- Need to pass pointers to the variables you want to input into.
- Need to use a different placeholder for doubles

Yes, there is a scanf()

Not as wonderful as it lookks
In particular, it makes error handling difficult

Consider using sscanf() instead

See scandemo2.c

Buffered Output

See buffo.c

Aside: 'watch' - useful shell command
$ watch -n seconds command

Just because eyou have printed something doesn't mean it has actually left the buffer yet.

See buffo2.c

"Hey I found a way to disable buffering…"
Generally not a good idea. Buffering exits for a reason.

What if I don't close a file?

- There is a system limit on how many files you can have open at any given time. A long-running program with lots of files open might prevent you from opening any more.
- If your program exists "normally" i.e.
    - Return from main();
    - Call exit();
      All open FILE*s will be closed
- If your program terminates abormally, then the file will be closed but no flushing will occur

Pre-processor

Wednesday, 23 March 2022 10:20 PM

Pre-processor Macros

The pre-processor runs before the main compile and deals with the # directives

#define PI 3.141

Every occurrence of PI will be replaced with 3.131. The compiler will never actually "see" PI. It will only see the replacement value.

Be careful though.
This can create problems using a debugger because the code you see is not exactly the code that was compiled.

Macros with Parameters

#define CUBE(X) ((X) * (X) * (X))

- These can look like a function call, but are expanded by the pre-processor
- Be careful with ()
  - Could we have #define CUBE X*X*X?
  - Yes but
  - CUBE(2+3) --> 2+3*2+3*2+3 == 17 and not 125.
- Beware of side effects
  - Int x= 1; int y = CUBE(++x);
  - If it were a function, we would expect the answer to be 8
  - (++x) * (++x) * (++x) = 2*3*4 = 24 (and you've broken x)

Why??

Beyond the scope of this course

- Can emulate generic programming and C++ - like templates (sort of) e.g.
  - #define MAX(X,Y) ((X) > (Y) ? (X) : (Y))
- Used carefully it can reduce repitious and complex code structures, but can be very hard to debug.
- Often used to force inline function code (faster) but modern compilers recognise the inline keyword

Pre-processor abuse is a staple technique in the International Obfuscated C Competition - https://www.ioccc.org/

See ditdah.c

Conditional Compile

```
#define BOB
```

Tells the pre-processor that "BOB" is a symbol it should recognise but doesn't actually give it a value. These can also be defined on the command line with -DBOB

Conditional Compile

```
#ifdef OMP_SUPPORT
Void stuff_that_only_works_under_omp();
#endif
```

Most frequent usage - Include/header guards

```
#ifndef BOB_H
#define BOB_H
        // Bob things
#endif
```

See h1.h, h2.h, h.c, vs h1.h, g2.h, g.c

Enums

Wednesday, 23 March 2022 10:21 PM

Enums

Bool allows us to have a variable which stores oone of a set of named values {true, false}

What about

- Days of the week? {SUNDAY, MONDAY, TUESDAY, WEDNESDY, THURSDAY, FRIDAY, SATURDAY, SUNDAY}
- States in a statemachine? {SETUP, CONNECTED, WORKING, DISCONNECTED, ERROR}
- …

```
Enum Day {
    SUNDAY,
    MONDAY,
    …
};

Enum Day d = TUESDAY;
```

Behind the scenes, the compiler will chose an int value for each member of the enum.

Fixed Values?

```
Enum Errors {
    E_OK = 0,
    E_TOO_MUCH = 1,
    E_NOT_A_NUMBER = 2
};
```

Switch Statements

Wednesday, 23 March 2022 10:21 PM

Switch

```
Enum State s;

Switch (s) {
        Case SETUP:
                Printf("Doing setup\n");
                Break;
        Case CONNECTED:
                Printf("Doing connected\n";
                Printf(" more connected\n");
                Break;
        Default:
                Printf("Doing other states\n");
};
```

See switch.c

Things to watch for

- Switch can be used with any integer-like type - deosn't work with strings or floats
- Case statements must be constants
- Missing break statements!

If you are feeling brave, check out Deff's Device but please don't use it in your assignments.

Break and Continue

Wednesday, 23 March 2022 10:21 PM

Break and Continue

Break

Break will jump out of the inner-most loop or switch statement

e.g.

```
for (int num=2; num<100; ++num) {
    bool prime=true;
    for (int factor=2; factor<num; ++num) {
        if (num%factor == 0) {
            prime=false;
            break;
        }
    }
    if (prime) {
        printf("%d ", num);
    }
}
```

Continue

Continue jumps to the next iteration of the inner-most loop

e.g.

```
while (fgets(buffer, 80, input)) {
    if (strlen(buffer)<5) {
        continue;    // read the next line
    }
    // more processing
}
```

Types

Wednesday, 23 March 2022 10:21 PM

Types

- Signed integer types
  - Char[4] ≤ short int ≤ int ≤ long int ≤ long long int
- Unsigned integer types
  - Unsigned char ≤ unsigned short int ≤ unsigned int ≤ unsigned long int ≤ unsigned long long int
- Floating point types
  - Float ≤ double ≤ long double …
- Boolean
  - Bool, … numeric types

The header file <limits.h> defines the min and max values for most (all?) types, regardless of architecture

<limits.h> excerpt

```
/* Number of bits in a 'char'. */
# define CHAR_BIT 8

/* Minimum and maximum values a 'signed char' can hold. */
# define SCHAR_MIN (-128)
#define SCHAR_MAX 127

/* Maximum value an 'unsigned char' can hold. Minimum is 0.)  */
#define UCHAR_MAX  255

/* Minimum and maximum valeus a 'signed short int' can hold. */
#define SHRT_MIN (-32768)
#define SHRT_MAX 32767
```

Function Pointers

Wednesday, 23 March 2022 10:22 PM

Function Pointers

Why?

Sometimes we want to put functions into variables

- Callbacks: when particular thing happens call this function
  - GUIs and other event driven tasks
- Flexible functionality
  - Change how one part of an overall task is done
  - E.g.: sorting how ordering is defined? (see man 3 qsort)
- …

In C the name of a function (without () is treated as a function-pointer for it)..

- Similarly to the name of an array being a pointer to the first element
- See fp1.c for a possible error message.

Syntax

A type of g is
Int (*)(void)

A function which takes two ints and returns an int has type
Int (*)(int, int)

(*)(…) is your cue that a function pointer is invoked.

The name of the variable[6] goes with the (*)

Int (*vname)(int, char)
Vname is a variable storing a pointer to a function which returns an int and takes int and a char as parameters.

See fp2.c and fp3.c

See why people might want to use typedefs: fp4.c

What about
Void qsort(void* base, size_t nmemb, size_t size, int (*compar)(const void*, const void*));

Operating Systems

Wednesday, 23 March 2022 10:22 PM

What does an Operating System Do?

Abstraction

- OS provides an abstraction of hardware (include CPU) to application ("user mode") code
- Allows programs to ignore low-level details by providing *generic interfaces*
    - What sort of disk is it? SSD? HDD? USB? SATA?. It doesn't care: it just read/writes files
    - What kind of network interface is it? It doesn't care, just send/receive it
    - Please give me some memory. I don't care where it is

Multitasking

The OS permits the orderly sharing of resources among multiple users/processes

- CPU
- Memory
- Hardware
- Graphics
- Storage
- Etc.

Arbitrate this shared access according to some defined policies

Standardised Interfaces

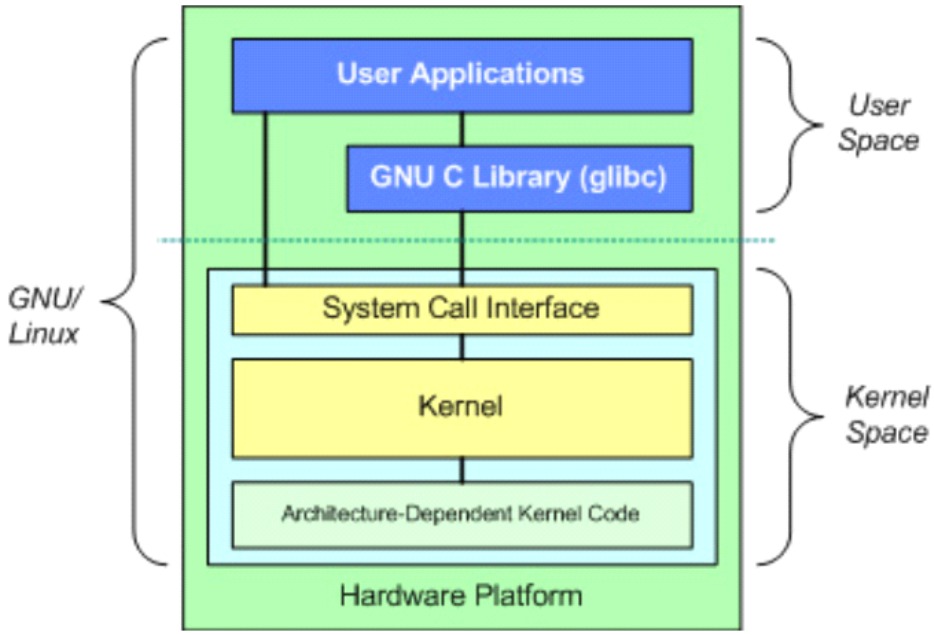
- Portability is the degree to which programs can be built and run on a variety of systems
- Code written properly for e.g. Linux, is generally portable across a wide range of CPU architectures
- The POSIX (portable Operating System Interface) standard defines a common subset of interfaces for Unix-like operating systems.

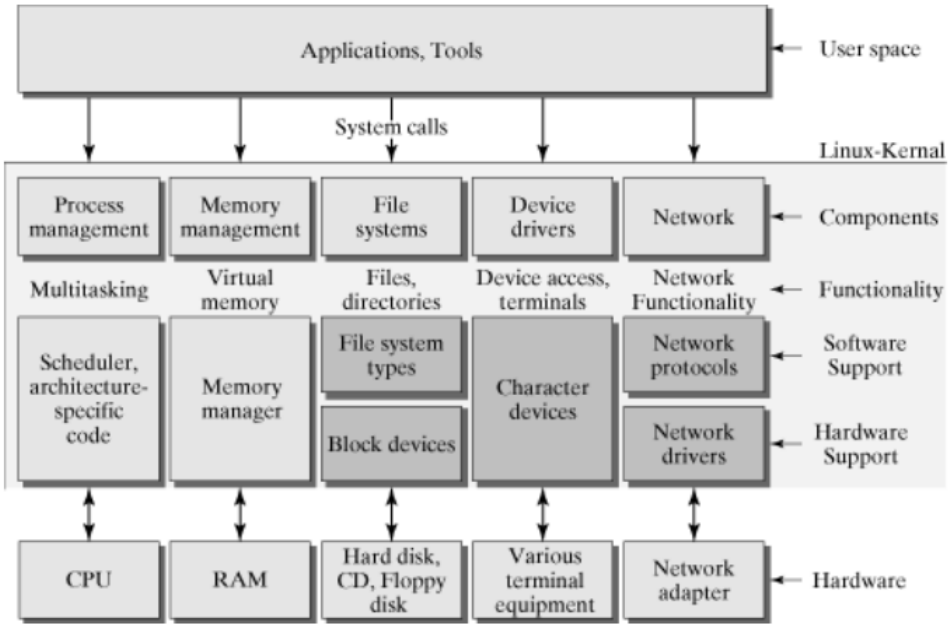
In Summary

An OS lets your program to pretend it is alone on the CPU…

… unless otherwise specified and permitted

The Linux Architecture - Graphically





User vs Kernel Space

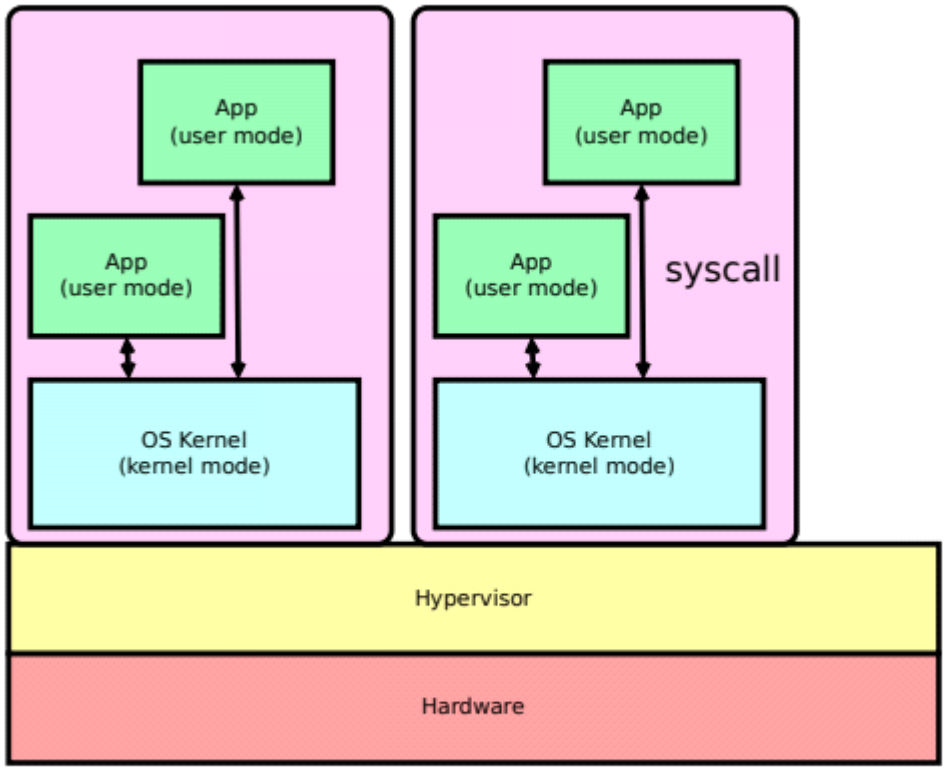
Most CPU architectures define at least two privilege levels

- **User mode** - unprivileged
  - Can do normal CPU instructions like load, store, arithmetic etc
  - Not allowed to access hardware directly, disable interrupts,…
  - All memory accesses are controlled and protected by the memory management unit (MMU)
- **Kernel mode** - full privileges - user mode plus..
  - Modify CPU state - interrupts, modify MMU configuration
  - Access anywhere in memory or IO address space

- In short, just about anything

Modern CPUs with support for hardware virtualisation take this deeper.

User / Kernel / Hypervisor



Getting into Kernel Mode

- System calls
  - Deliberately triggered by user code via special op codes
  - Visible because the user program asked for them (often presented as a function call).
- Exceptions
  - Result from program actions (eg illegal floating point operation)
  - Visible because the user code wakes up in a handler
  - Not related to language based software exceptions (although one might trigger the other)
- Interrupts
  - Hardware -> CPU (e.g. network interface -->)
  - CPU timer interrupts triggers process scheduling
  - Not directly visible to user programs

Kernel Mode, Library calls and System calls

C Library - nanosleep.c

```c
#include <time.h>
#include <sysdep-cancel.h>
#include <non-cancel.h>

int
__nanosleep (const struct timespec *req_time,
        struct timespec *remaining) {

  return SYSCALL_CANCEL (nanosleep, req_time,
     remaining);
}
```

C library - sysdep.h

```c
// A bunch of pre-processor macros ...
#define SYSCALL_CANCEL(...) \
...
syscall(__NR_nanosleep, ...)
```

C library - x86_64/syscall.S

```
ENTRY (syscall)
        movq %rdi, %rax         /* Syscall numr -> rax.*/
        movq %rsi, %rdi         /* shift arg1 - arg5.*/
        movq %rdx, %rsi
        movq %rcx, %rdx
        movq %r8, %r10
        movq %r9, %r8
        movq 8(%rsp),%r9        /* arg6 on the stack. */
        syscall                 /* Do the system call.*/
        cmpq $-4095, %rax       /* Check for error. */
        jae SYSCALL_ERROR_LABEL
        ret                     /* Return to caller.  */
PSEUDO_END (syscall)
```

Shells

Shells are:

- An unprivileged program
- Are an interface between users and the kernal
- Provide scripting capabilities
- Are oftne (but not required to be) text based

Examples

- Thompson Shell (sh) - the original UNIX Shell
- Bourne Shell (Sh) - form UNIX v7 (1977)
- Bourne-Again shell (bash) - superset of sh
    - What you're probably using on moss
- Korn Shell (ksh)
- Z shell (zsh)
- C shell (csh)
- TENEX C shell (tcsh)
- Scheme shell (scsh)

In this course we are assuming bash

Startup

- Read startup files (e.g. .bashrc, /etc/bashrc)
- Read commands
    - From standard in (for interactive use - e.g. a login with putty or ssh)
    - From a script file (when running as a script interpreter)

To be able to run text files as scripts, they need to have both rx permissions

- Need to read so the shell can *read* the script.

Internal/External Commands

- Some commands are built into the shell
    - Cd
    - Alias

    - Type
    - Which
    - …
- Other commands are executable programs on the filesystem
    - Ls
    - Gcc
    - Vim
- Use type to see whether something is a builtin
- To find where an external command is located, use which
- Echo $PATH to see what directories are searched for commands in what order.

Variables

Variables:

- Are always strings (some programs and shell commands interpret those strings as numbers)
- Are either local ("shell variables")
    - Variable a scope is the current shell process only
    - Not passed on to programs started by the shell (child process)
- Or environment variables
    - Passed on to child process

Shell variables can be promoted to the environment with export
e.g. export COURSES

See env.c

Variable Syntax

- Variables do not need to be declared (they are created by assigning something to them

BOB=7

There is now a variable called BOB

- Do not put cosmetic whitespace around operators
- To read the value out of a variable, use $
    - e.g. NPATH=$PATH: ~/bin
    - NPATH now stores the contents of PATH followed by : ~/bin
- Variable names are case sensitive
- Use braces for extra clarity and explicitness - ${PATH}

Important Variables

- PATH - directories to search for commands
- LD_LIBRARY_PATH - direcetories to search for dynamic libraries
- UID - current user's (numeric) userid
- USER - current user's login name
- HOME - path to current user's home directory
- $? - exit status of the most recent command
- $# - argc - 1
- $0 - argv[0]
- $1 - argv[1] … etc

Env will give dump of current environment variables

Special Characters

- Wildcards for filenames
  - * - stands for zero or more characters
  - ? - stands for exactly one character
  - Be careful - the shell attempts to expand these before calling your program (see demo)
- # - comment to the end of the line
- & - run a command in the background
  - Sleep 10 &
- ; - run command sin sequence
  - Sleep 10 ; ls

Quote / Escape Characters!

comment to the end of line

\ escape the next character

```
$ z="A B C" # treat everything as a single item
      # (escapes spaces, does not escape $)
```

```
$ z='A B C' # like " but escapes more things
```

```
$ z=`cmd` # evaluates to the output of
      # the command in quotes
```

```
$ z=$(cmd)     # Cleaner syntax
```

eg:

```
$ cat $(which a1test.sh)
```

Show the contents of the script `a1test.sh`, whereever it may be

Other Useful Stuff

Writing shell scripts

#!/bin/bash

FILEs=$(ls $*)

For f in ${FILES}; do
    Md5sum $f
Done;

Redirecting and Piping

- Cmd < in - stdin uses file "in"
- Cmd > std.out - stdout to file "std.out"
- Cmd 2> err.out - stderr to file "err.out"
- Cmd1 | cmd2 - stdout of cmd1 feeds into stdin of cmd2

e.g. run naval without user input

$ cat player.turns

A1
C3
…

./naval test.rules plyr.map cpu.map cpu.turns < player.turns

File all C files in this directory under subdirs, and count number of lines

$ find . -name '*.c' | xargs wc -1

Note

The shell commands covered in the Linux tute are examinable.

Debugging

Friday, 25 March 2022 5:34 PM

Debuggers

Debuggers allow you to examine a running program

- Look at variables
- Look at the contents of memory
- Pause the program at a particular point (breakpoint)
    - Line of code
    - Function call
    - On some condition (e.g. variable has a certain value)
- Step through a program line by line or function by function

Can be used to debug a program (obviously)
Can also be used to "crack" a program - get around software protection
Can be command line based or graphical (integrated with an IDE)

gdb = GNU Debugger

Available on **Moss**

- Make sure your program includes debugging information (symbols) - e.g. variable names associated with particular memory locations
    - Ensure that you compile using gcc with the -g flag
    - Without this it will be much harder to get useful information
    - It includes extra symbol information which for example lets you find out which particular line numbers things occur at, etc.
- You can strip out symbols with the strip command or use the -s flag to gcc
- Run gdb using
    - gdb program-executable-name

Useful gdb Commands

- Run [cmd-line-args] - starts your program running
- Break - sets a breakpoint (program stops at that point)
    - Break function-name - break at start of function
    - Break filename.:23 at line 23 in file
        - Can omit filename if it is the "current" file
    - Break file.c:56 if i==10 - break conditionally
- Backtrace or bt - show the function call stack
- Up/down - move up and down the call stack
- Next/step - execute next line after stopping
  step steps info functions, next steps over them
- Continue - continue running (until next breakpoint)
- List - show the code around the current stopping point
- Print expression - print value of expression
- Info, help, quit, enable/disable, display, ignore, …
- Ctrl-C - doesn't exit your program - it interrupts it

Valgrind

Valgrind is "an instrumentation framework for building dynamic analysis tools"

- Default tool is **memcheck** - looks for
  - Memory errors - e.g. accessing addresses outside those allocated
  - Memory leaks - failure to free memory
- Other analysis tools include thread error detectors, profilers, etc…

Valgrind Examples

On **Moss**

- Leaky.c
- Debugcopy.c

Code Coverage

- Useful to know if all parts of your code have been exercised by your tests
- On Linux you can use **gcov**
  - Must compile with gcc arguments
    -fprofile-arcs -ftest-coverage
  - Run tests on your executable
  - Then run gcov filename.c
  - This prints a summary and puts results in file filename.c.gov
- Complete code coverage does **not** necessarily mean your code is right!

Profiling

- Shows where your program is spending time
- Not a debugger - can use it to optimise your program
- On Linux, you can use **gprof**
  - Must compile and link with gcc arguments **-pg**
  - Run your executable
    - Data is recorded in the background (in a file called gmon.out)
  - Run **gprof executable-name** to generate a report
- Reports based on statistical sampling - will not be accurate for short-lived programs.

Time Utility

Simple command to report CPU usage

- Time spent in user mode
- Time spent in kernal mode

Assignment 2

Sunday, 27 March 2022 11:45 AM

Assignment 2 Information

- Assignment 2 (Binary Bomb) now available
  - Specification is on Blackboard
  - Files are on Moss

Abstraction

Monday, 28 March 2022 2:04 PM

Processes

A process is

- An instance of a program in execution
  - There is one /opt/local/bin/bash on moss
  - There could be many executing instances of bash
- An abstraction of a computer
  - Interactions (e.g. with files) are via system calls to the kernel
  - Other processes resources are not visible
  - Processes' memory is not accessible by other processes
  - Other processes' CPU activity shouldn't influence yours
  - Indirectly aware of other non-related processes
    - e.g. files on system

The kernel keeps track of all the things the process is doing

- Files open

How

Separating:

- Resources (e.g. open files)
  - Each process can have their own table of resources
  - At lower level, files are identified by integers
- Memory
  - Virtual memory (later on)
- CPU Activity (From Week 4)
  - Whenever a CPU switches to kernel mode, registers are saved
  - (possibly much later) when the process is put on the CPU, registered are restored
  - The process is none the wiser

Process States

From the kernel point of view, a process could be in one of the following states

- Running - The process is currently executing on the CPU
- Ready - The process could run but isn't (something else has the CPU)
- Blocked - Process is not ready because it is waiting for something
  - e.g.: waiting to read from a file
  - Sleep?
- Ended(?) - Process has exited or terminated and needs to be cleaned up
  - May have become a zombie.

Unix Processes

Every process has

- A unique ID (PID)
  ```
  pid_t getpid(void);
  ```
- A parent process
  ```
  Pid_t getppid();
  ```

A process may have zero or more children
e.g.

```
uqjwill1@uqjwill1-8048:~$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
uqjwill1     230     229  0 08:50 pts/2    00:00:00 -bash
uqjwill1     305     230  8 08:53 pts/2    00:00:30 gummi lec08_09_Processes.tex
uqjwill1     344     230  0 08:59 pts/2    00:00:00 ps -f
```

Ps can tell you about the processes running
-f gives full information
-e gives all processes running on the system

The Family Tree

All processes in a Unix system have a parent:

- `init` (ID 1) is special - it is the first userspace process created after the kernel finishes booting
- `$ ps -e` will show us all processes in the system
- `$ pstree` shows the tree structure visually
- PID 0 is the kernel itself - `init` and `kthreadd` have the kernel as a a parent

Creating new Processes

`fork()` - Asks the kernel to create a new (child) process

- Called from one process (the parent)
- Returns in two processes (two processes "remember" calling it)
  - Parent's `fork()` call return value is the PID of the child or -1 if no child was created (and sets errno)
  - Child's `fork()` call returns 0.

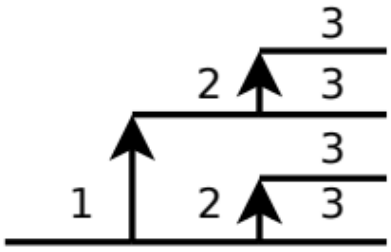
```
pid_t pid = fork();
if (pid) {
    // parent
} else {
    // child
}
```

Down the Rabbit Hole

- Child has an **exact, but distinct copy** of the parent's memory space
- Child has exactly the same files open (including `stdin`, `stdout`, `stderr`)
- See `f1.c`

```
void X(void) {
    int p=5;
    if (fork()) {
        ++p;
    } else {
        --p;
    }
    printf("%d\n", p);
}
```

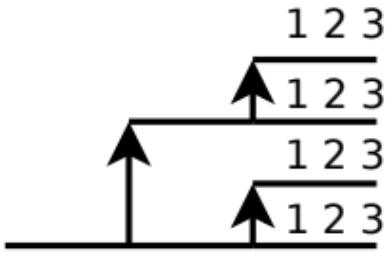
Child processes are processes and can fork as well
See f2.c



Watch out for buffering

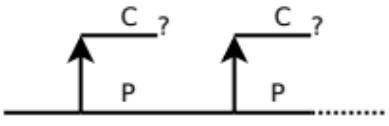
The stdio buffers are part of the process's state
--> they get copied in the fork()

See f3.c vs f4.c



The parent process could fork to make workers

See f5.c



Fork Bombs

Wednesday, 6 April 2022 1:33 PM

Fork Bombs

```
for (int i=0; i<20; ++i) {
    if (!fork()) {
        printf("Child\n");
    }
}
```

The above example will stop eventually because it has an upper bound of 20

- Limited number of simultaneous processes per user
  - Ulimit -u on bas
- Use kill or pkill?
  - In most caoses, that needs to fork again.
  - Which you can't do
  - How do you know which process IDs to kill?
- Can you do anything quickly enough
  - Your useful programs will be competing with all these clones

Fork Bomb Consequences

Depends on the system configuration. On Moss

- May slow *your* processes down
- Probably won't affect other users.
- But moss is being slow
  - Moss almost always I/O bound
  - Unless sysadmins say otherwise, Someone else fork bombing themselves is unlikely to be causing problems for you

Ending a Process

Exit()

Is a system call which ends the *current* process (you can't use it to end a different process)

Useful things:

- The parameter to exit is the "exit status" of the process.
- Any open output streams are flushed
- Exit hooks are executed
  - Int atexit(void (*function)(void));

Don't want those things? (e.g. if you think you have inherited unflushed buffers)

- Man _exit()

Program crashes (or receives a signal)

- Exit doesn't happen and there is no exit status

Zombies

Wednesday, 6 April 2022 1:32 PM

Zombies

Zee zom1.c

The system needs to keep a record of the what happened to a process in case the parent is interested.

- Did it exit normally?
  - With what status?
- It terminated rather than exiting normally.
  - Which signal caused that?

The memory and resources the process was using have already been released but part of the process still hangs around. A process in that state is called a zombie.

Reaping a Zombie

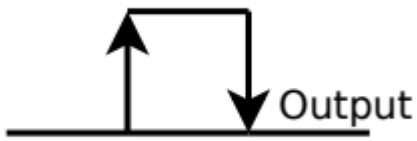
Reaping - when a process' parent has asked about the cause of termination.

- Zombie will be removed

To reap, the parent process calls wait(). Wait blocks until either:

- A current child process becomes (or already was) a zombie.
- The parent has no child processes (returns error).

See zom2.c



Zombies can't be killed - there is no code left to handle a signal

So how do you stop a zombie

You drop a wait on it.

Reaping and Adoption

If a process

- Is "alive"
- Has zombie children
- Doesn't reap them

Then those zombies will stay on the system (the parent might ask about them eventually). For long running processes like servers, this could be a problem.

What if a process dies and has running child processes?

See adopt.c

- Only the direct parent of a process can reap it.
- The children will continue to run
- All that process' children will be adopted by process 1.

Decoding "status"

The information returned by **wait** needs to be decoded.

- WIFEXITED(status) - true if the process exited normally
  - Then WEXITSTATUS(status) - the exit status of the process
- WIFSIGNALED(status) - true if the process was terminated by a signal
  - WTERMSIG(status) - the signal which caused the process to terminate

See f7.c

Wait can be used to check for things other than a process ending, but we don't need them now.

Waitpid()

See f8.c

The third argument will be 0 for now

Later we may use W_NOHANG.

Changing a Script

Wednesday, 6 April 2022 1:32 PM

Changing Script

A process can change which program it is running

Int exec1(char* path, char* arg0, char* arg1, char* arg2, …, NULL)

- The last thing in the list must be a NULL pointer
- Replaces the old process image with a new one.
  - The old stack and heap are gone.
  - The old program instructions are gone
  - Resources on the kernel side (e.g. files) are kept
- Returns -1 if the operation fails.
- Never returns anything but -1

See exec1.c

Pathing

- From exec1.c
  - ./exec1 ls doesn't work
- To the contents of the PATH variable into account, use execlp.
- In this course, you should always use the p forms of exec.

Execvp

See exec2.c

Int execvp(char* arg0, char** argv)

- More useful for varying numbers of commandline arguments
- The last element in argv must still be a null pointer

There is also execvpe() but we don't need them in this course.

Summary

So in order to run another program

1. We clone ourselves
2. Replace the clone's memory with a new program

Is this overly complicated for a reason?
Yes

Signals

Wednesday, 6 April 2022 1:33 PM

Signals

- Signals are very simple messages from the kernel to a process.
  - They don't carry information other than "One or more of some event has happened"
  - If a previous signal of that type hasn't been handled yet, additional signals will be ignored
- The kernel will send signals
  - On its own initiative
    - The process has accessed invalid memory - i.e. a segfault (SIGSEGV)
    - The process tried to write to a destination that won't accept input (SIGPIPE)
    - …
  - Because a process has asked it to
    - Shell kill command
    - C kill system call

Kill

- Processes have signal handling functions registered with the kernel.
  - When a signal is delivered, the kernel fakes a function call in the process (see exceptions from last week)
- If your program doesn't set up new ones, the default handler will be used.
  - The default handler usually terminates the process
- Kill commands are actually "send a given signal to a process"
  - … often that results in the process not being alive.

Signal Numbers

To see signals available on your system in bash

Kill -l (n.b. dash el)

1. SIGHUP
2. SIGINT
3. SIGQUIT
4. SIGILL
5. SIGTRAP
6. SIGABRT
7. SIGBUS
8. SIGFPE
9. SIGKILL
10. SIGUSR1

Sigaction

From man **sigaction**

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

There are older functions like:

- Signal
- Sigset

Do not use them

Sig.c

See sig.c

How do I stop valgrind from complaining about my sigaction variable?

Add the following after declaration

Memset(&sa, 0, sizeof(struct sigaction));

SA_RESTART?

- If a signal arrives while a system call is running, restart the sys call

How do I stop it?

What is a process is trapping SIGINT?

- ^\ sends SIGQUIT --- very useful
- Kill -3 PIL / kill -QUIT PID - sends SIGQUIT
- Kill -9 PID - sends SIGKILL
    - SIGKILL can never be blocked or caught

Kill -9 all the time?

- Not ideal
- Better to give processes a chance to clean up

SIGCHLD

Wait() will bock until a child ends
Not useful if you want to do other work while the child is running

- You could use
  result = waitpid(pid, &status, W_NOHANG)
    - What if there are lots of children? Need to check each one?
- SIGCHLD is sent to the parent process when something happens to one of its children
    - You can set a handler to act when SIGCHLD arrives.

See child.c

Notes

Wednesday, 6 April 2022 1:33 PM

Notes

- Ideally your signal handlers should not
  - Take a long time to execute
  - Acquire locks
- Sigwait() may be useful if your main program needs to wait for a signal
- Can't we just do … and zombies will never be created?
  - The documentation says that techniques vary across systems
- "Core dumps" - in ye olden days segfaults (and some other signals) would cause a copy of memory (core) to be put into a file:
  - These were useful for
    - Debugging
    - Using all your quota
  - Most modern linux systems have core dumps disabled.

Other Signals

- SIGSEGV - surprise segfault
- SIGWNCH - your terminal window changed size.

Summary so far

Wednesday, 6 April 2022 1:28 PM

- Fork() creates new processes
  - Child gets ()
  - Parent gets the pid of the new child
  - Child is a perfect clone of the new child
- Exec() replaces the current process with a new program
- Wait() and waitpid() lets parents know what happened to their children
- Kill() sends signals
- Sigaction() lets us catch (some of) them
- Getpid() returns our PID
- getppid() returns parent PID

File Descriptors

Wednesday, 6 April 2022 1:33 PM

Everything is a "File"?

- Unix systems make many aspects of the system available by file interface.
  - e.g. cat /proc/cpuinfo
- IO devices
  - e.g Disks /dev/sda1

Just because you can see it on the filesystem, doesn't mean it is actually stored there (or anywhere for that matter).

e.g. /dev/random

Unix Files

To be treated as a file, the kernel needs to know what to do with a few basic requests

- Open, close
- Read/write - bytes only
- Seek - move around in the file

Note that some calls might return errors

e.g. moving backwards in stdin from keyboard won't work.

File Descriptors

- Unix systems (at a lower level then the standard IO functions) use integers to refer to open files.
- When you ask the kernel to open a file, it gives back a "file-descriptor"
- When ever the program wants to interact with that file, it includes that number in requests to the kernel

Open()

See fd.c

- Open() takes numeric constants instead of the string form used by fopen()
  - O_RDONLY, O_WRONLY, O_RDWR <-- be careful
  - Can | in other flags (bitwise or in other flags)
    - O_APPEND
    - O_CREAT
- Read() calls deals in fixed numbers of bytes
- Ssize_t read(int fildes, void *buf, size_t, nbyte)

See fd2.c

Writing

See write.c

- Ssize_t write(int fildes, const void *buf, size_t nbyte)
- We need to specify what "mode" (i.e. permissions) the newly created file has.
    - S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR - user has rwx
    - S_IRWXG = S_IRGRP | S_IWGRP | S_IXGRP - group has rwx
    - S_IRWXO = S_IROTH | S_IWOTH | S_IX|TH - other has rwx
- If we want the file created, we need to ask for that.
- If we want the file truncated on creation, we need to ask for that
    - O_CREAT vs (O_CREAT | O_TRUNC)

Moving Around

- Off_t lseek(int fd, off_t offset, int whence)
    - Go somewhere (maybe no where) and tell us where we land
- Examples
    - Lseek(fd, n, SEEK_SET); // Move to byte `n` in the file
    - Pos = lseek(fd, 0, SEEK_CUR); // Where are we?
    - Filelen = lseek(df, 0, SEEK_END); // Where is EOF?
- Remember O_RDWR?
    - There's only one file offset - you have to keep track of where you are for reading or writing.
    - So there's **not** a file offset for reading independent of a file offset for writing.

Fds and fork()

See fdc.c

- Any open files in the parent at time of fork() will also be open in the child
- The parent and child share the same offset
    - They actually share the file descriptor at the kernel level
- If opening the same file multiple times (i.e. different file descriptors) moving one does not move the other.

IO Redirection

How does the **shell** do this?

$ **cmd < stdin.txt > stdout.txt 2> stderr.txt**

See redir.c and redir2.c

- Stdin -> descriptor 0 - or STDIN_FILENO
- Stdout -> descriptor 1 - or STDOUT_FILENO
- Stderr -> descriptor 2 - or STDERR_FILENO

Note that dup2() is "copy" not "move", so it's a good idea to close the original.

There is also a dup() which copies a descriptor but doesn't put it in a specified place.

Piping

Wednesday, 6 April 2022 1:54 PM

Piping

$ ls | sort -r

The standard output of ls **ls** is connected (somehow) to the standard in of the second process

- Write (1, … --> read(0…

What about

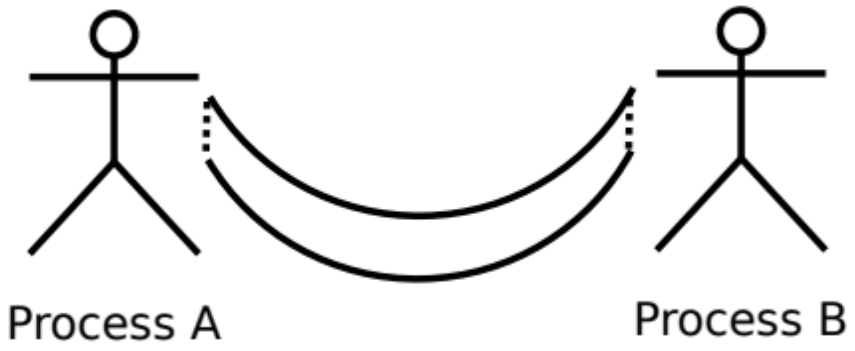
$ (ls > tempfile &) ; (sort -r < tempfile)

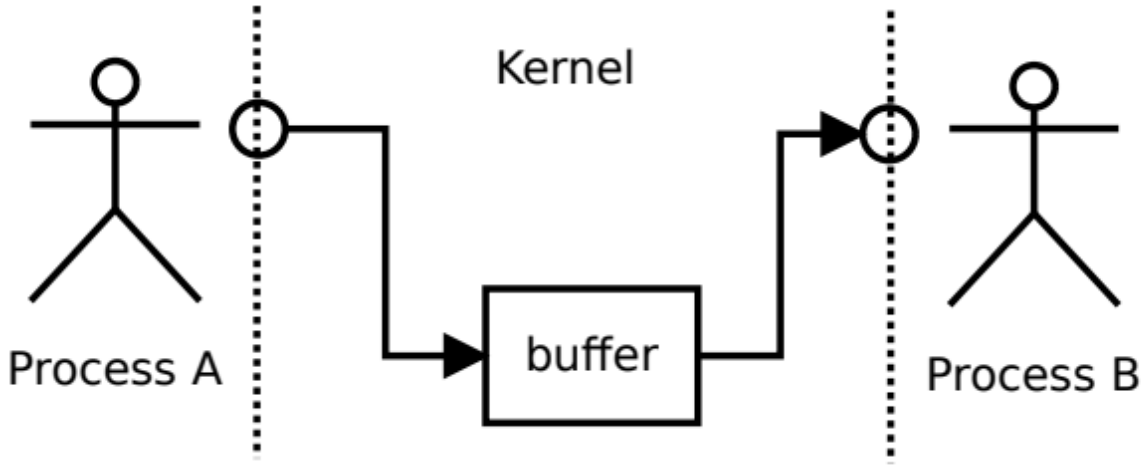
Problems:

- Assumes the process has write permission to the current directory
- The second process could hit EOF before the first process has finished writing

We want something which doesn't use "disk" files

Pipes in C - Behaviour





Reading from a Pipe

- If no bytes in the pipe --> block
- EOF only when it is impossible to write to the pipe
    - In the simple case when the writing process closed its fd. But beware later.

Writing to a Pipe

- If the pipe is full --> block
- If it is impossible to read from pipe (and you are attempting to write) --> you win a SIGPIPE from the kernel.

Physical Intuition

Note:

- Bytes that are written go into a kernel buffer (the pipe buffer), they do not travel directly to the other process.
- The other process must actively read (there is no push delivery)
- There is no concept of pressure (you can't write bytes harder to push them into the other process)
- You can't connect two pipes together (end-to-end) and read from the far end to suck bytes out of the earlier pipe

More about Pipes

File descriptors are just numbers, they don't have any meaning outside a given process's context

- You can't pass a file descriptor to another process - it won't mean anything to it

You can't force an fd onto another process:

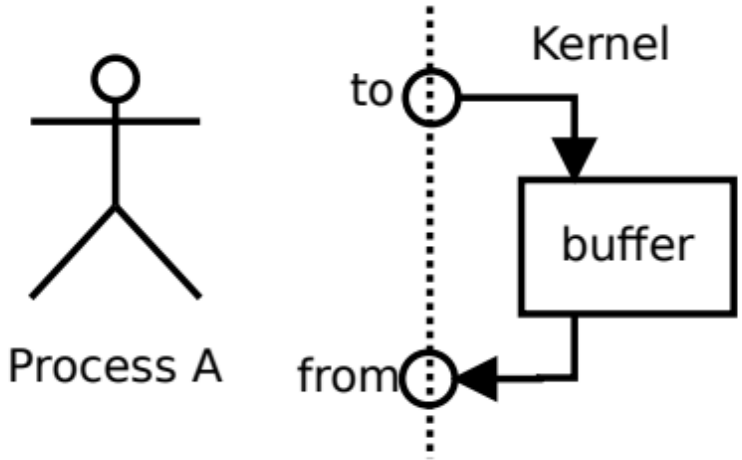
- You can't force a pipe through the wall of someone's house
- If the program is not expecting to interact with it, it won't ever refer to it.

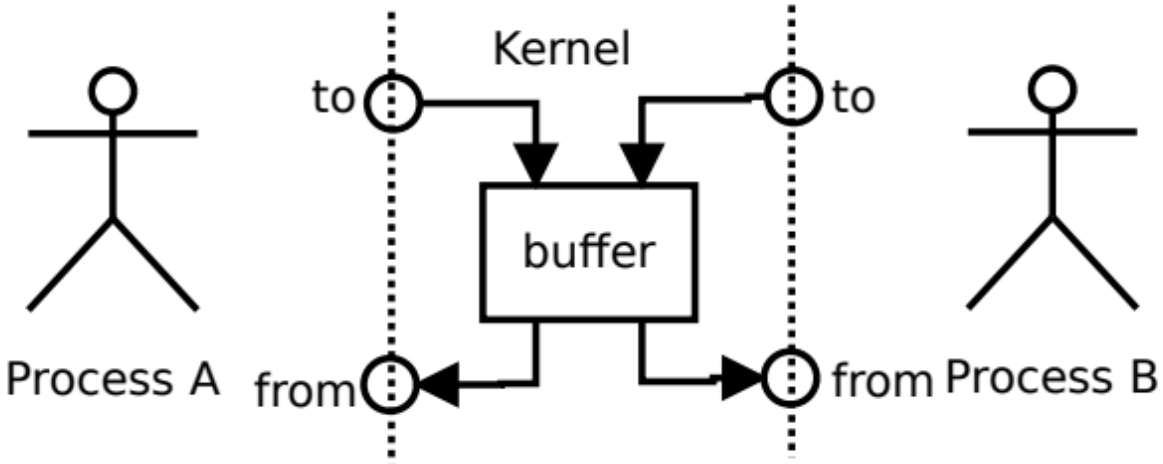
Pipes How?

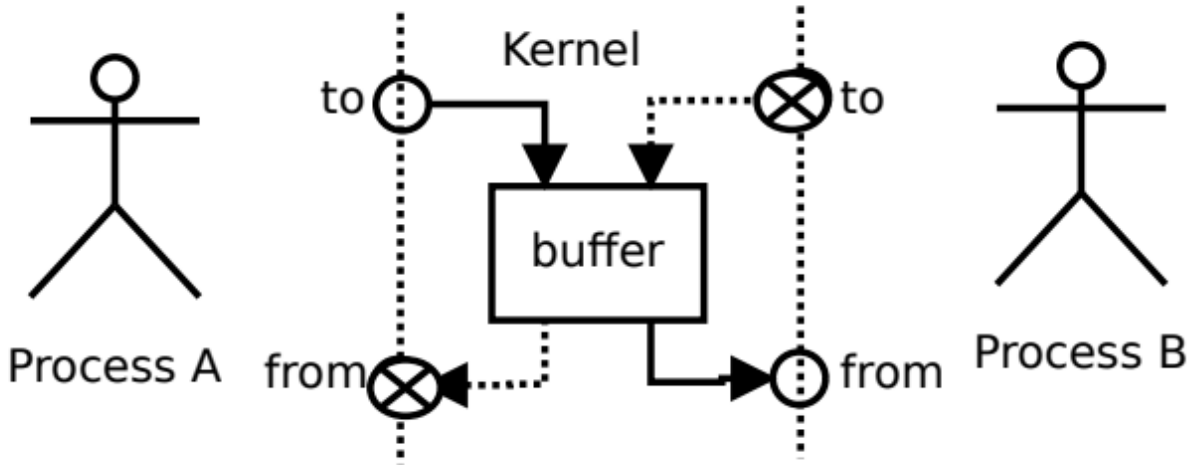
Int pipe (int fd[2]);

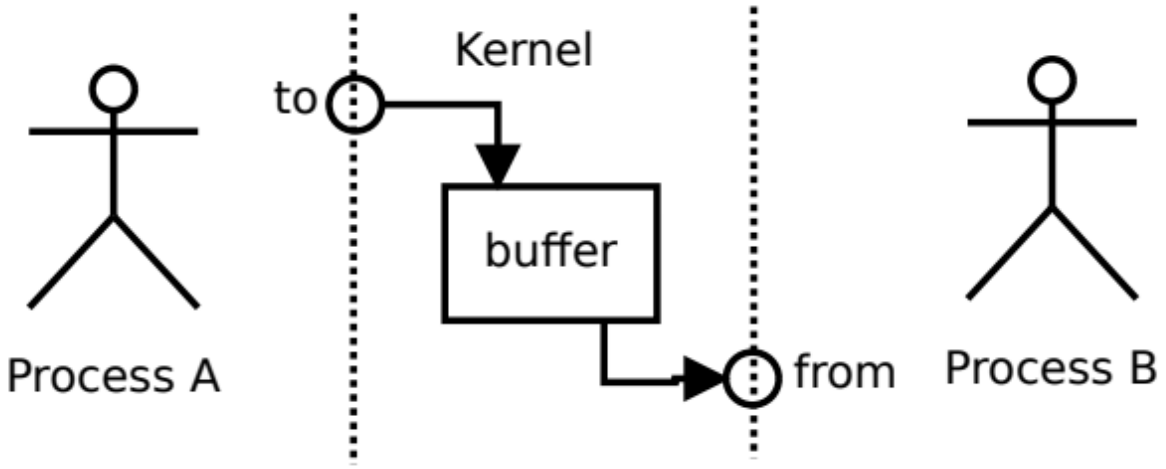
- Pass in an array to hold two fds (integers).
- A successful pipe() call fills in this array.
- Fd[0] is the read end and fd[1] is the write end
- Congratulations, you are now holding both ends.

Demo


Kernel
to
buffer
Process A
from


Kernel
to
to
buffer
Process A
from
from
Process B


Kernel
to
to
buffer
Process A
from
from
Process B


Kernel
to
buffer
from
Process A
Process B

Notes

Wednesday, 6 April 2022 1:54 PM

Notes

- You need to create the pipe in the parent **before** you fork (otherwise the child won't inherit the fds.
- You need to close off the ends of the pipe each process won't be using.
  - Otherwise the reader won't know when the writer has finished.
- If you dup2() and fds make sure to close the original
- SIGPIPE could kill your program?
  - The file process at the other end of the pipe has terminated - there's nothing there to read
  - Look at the return value of write/printf?
  - Block/handle SIGPIPE?

FILE*

How do fds and FILE* interact?

- If you are setting up fds and then exec0ing on top, there is no issue (since the new process will create its own FILE* stdin, stdout, stderr).
- FILE* fdopen (int fd, const char *mode) will wrap a FILE* around an fd.
- Int fileno(FILE *stream) will get the file descriptor.
- If you call fclose(), then the underlying descriptor wil be close()d.

Don't do this

```
int get_int(int fd) {
    FILE* f = fdopen(fd, "r");
    int result;
    fscanf(f, "%d", &result);
    return result
}
```

Once you wrap an fd, you should forget about it and use the FILE* instead.

In the above code, every time you call get_int, a new buffer will be created. This probably results in input going "missing".

Threads

Monday, 11 April 2022 1:37 PM

Challenges with Multiprocessing using fork()/pipe() etc

- Debugging multiple processes simultaneously is hard
- Pipes are useful but also limiting
  - Unidirectional, stream of bytes
  - Not appropriate for some tasks (you can fake it with more abstraction)
- Higher overhead
  - Process creation/destruction is time consuming
  - Context switching between processes is time consuming
  - Data passes through the kernel (pipes) - multiple copies and kernel entries

Threads

Assignment 3

- Threads and synchronisation is for **assignment 4 only**
- Do not use this material in Assignment 3

Working in a Process

To run code, we need

- A process container
  - Address space
  - Open files
  - …
- Code / Instructions
  - "text segment"
- Global variables / constants
- Heap
  - Dynamic storage
- Stack
  - Function local variables
- CPU and its registers

Multiple workers?

Suppose we want to have multiple workers doing things? Fork()

- Open files are shared (see last week)
- Everything else is separate
- Needed if you want to have the new worker running a different program

Different processes can cooperate on tasks (especially since they can start off with the same knowledge).

- But communicating results is more complicated
  - Pipes?
  - "Network connection"? Similar complexity to pipes
  - Shared memory?
  - …

- But processes are safe from each other

Threads?

A thread is a worker in a proces
So every process has at least one thread (even if it doesn't explicitly use a thread library).

- Likely you will only interact with these via libraies.
- Different systems may have distinctions between "Threads / Light weight processes / ...
    - We will avoid talking about these
- Threads can be classified as "native" or "green"
    - We will get to this later

Multiple Threads

If we want multiple threads operating in the same **process**, we would want them to be able to share information easily.

Item	Shared?
Process	Yes
Code	Yes
Global variables	Yes
Heap	Yes
Stack	No
CPU & registers	No

Threads and Memory

Any thread running in a process can interact with any variable used by any other thread if they know where it is (i.e. they have a pointer to it).

The kernel won't stop this because if a page is valid for one thread, it is in the page table and so valid for all of the other threads.

Thread Implementations

Two main approaches here:

1. "native" or "kernel" threads
    - Threads are known by the kernel
    - Threads can be scheduled and run independently of each other
        - Scheduling is out of the scope of CSSE2310
2. "green" threads
    - Kernel just sees a single threaded process
    - User space library switches saves registeres and modifies program counter to switch between activities
    - Not actually using multiple cpu cores
    - A kernel call in one means you can't switch until you get back to user mode
        - Blocking I/O?
    - e.g. python threading

Why use threads?

1. A way around blocking I/O
    - I need to listen to 5 network connections and I don't know what order they will return in
2. A way to use more than one core worth of CPU power.

Blocking I/O

Situation
Need to interact with multiple I/O FILE*/fds

```
while (!done) {
    read from A and process
    read from B and process
    read from C and process
}
```

Problem: if input is available from C but not A, we won't ever get there.

Threading Approach

```
... interact(FILE* src) {
    while (...  ) {
        read from src
        process input
    }
}

...
   run interact(A) in thread    // not actual syntax
   run interact(B) in thread
   run interact(C) in thread
```

Each thread blocks on its own input source and doesn't interact with others

Non-threaded Approach

Non-blocking I/O calls exist in C/Unix, so you could do this

```
while (!done) {
    if (input_avilable_on(A)) {
        read from A and process
    }
    if (input_available_on(B)) {
        read from B and process
    }
    if (input_available_on(C)) {
        read from C and process
    }
}
```

Problem: *busy waiting*. This loop may run hundreds of times before input actually arrives.

Event Driven Programming

```
while (!done) {
    sleep_until_input_available_on(A, B, C)
    process input
}
```

These sorts of calls exist and deal with the busy wait problem
One core may be plenty - since most of the time is spent waiting
So why aren't we using them?

- You need to work with FDs and read(). No FILE* niceties
- It is trickier to keep track of where you are up to with multiple tasks (instead of each task having its own thread context)

More CPU Power

In applications which do not do much I/O (e.g. scientific computing), non-blocking operations are not going to help.

There you either need multiple processes or multiple threads to access more cores.

Pthreads

The C standard defines no particular threading library - it is OS/context dependent

- We are using the POSIX-thread API (pthreads)
- Implementations exist for most OS (including Windows)
- Pthreads wrap OS thread calls

From now on, any mentions of "threads" actually means "pthread threads"

More about pthreads

- Threads have no parent child relationships
- Any thread can "join" (i.e. wait on) any other thread in the same process
- Threads can't interact with threads in **other processes**.
- Any thread calling exit() will end the whole process including all other threads
- Can you fork()
    - Discuss later
- Signal handlers
    - These are possible

Thread Functions

- Each thread is created to run a function
- When that function ends / returns, so does the thread
- The function needs to have the correct signature

```
void* foo(void*) {
    // do thread things
    return (void*)0;
}
```

So the function can

- Accept any parameter (as long as you can express it as a void*)
- Send back anything to whoever join()s on it (as long as it's a void*)

First Thread Program

See thread1.c

Note: to compile a program using pthreads you need to add -pthread to the gcc command line.

- Do this for both compiling and linking
- Do not link the pthreads library directly - gcc needs to do special things if using threads

Creating Threads

Man pthread_create

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

- The id of the new thread is stored at the first argument
  - If the call is successful - see return value
- We'll ignore the second argument for now (attributes)
- The function we want to call
- The argument we want to pass it

First Thread Program - Revisited

- What is the call to sleep() doing there?
- See thread2.c

- We can end an individual thread using pthread_exit();
- See thread3.c

Passing Values into Threads

- So far we've passed in strings (char*).
  - Char* -> void* -> char*
- What about int
  - See thread4.c

- Why bother malloc()ing?
  - See thread5.c
    - This is an example of a "race condition"

Abusing Pointers

Ints fit in void* right?

What about thread6.c

- We could pass long instead to get around the size issue.
- The real problem here is that it works
  - In this case
  - It encourages people to ignore warnings

Multiple Parmeters

Suppose we have a function which we would like to run in a thread

```
void do_things(char** items, char* s, int limits);
```

It doesn't match the required signature.

Declare a struct type to hold all the values:

```
struct Params {
    char** items;
    char* s;
    int limits;
};
```

Add a wrapper function

```
void* do_things_wrapper(void* v) {
    struct Params* p =(struct Params*)v;
    do_things(p->items, p->s, p->limits);
    free(v);    // can't free earlier without copying the struct
}
```

Elsewhere

```
    struct Params* p = malloc(sizeof(struct Params));
    p->items = items1;
    p->s = "target";
    p->limits = 10;
    pthread_create(&tid, 0, do_things_wrapper, p);
```

What is a Thread ID?

- <pthread.h> declares a pthread_t type
- It is an **opaque** type
  - Makes printing it in debugging awkward
  - (On moss it turns out to be unsigned long but you can't rely on this

Where does thread return go?

See thread_calc.c

Note:

- Since the thread function returns void*, to allow pthread_join to modify a variable, you need

to pass in void**.
- Most pthread functions return error code, which you should check
- Pthread_exit(Val) ends the thread and sets its result to Val:

Zombie Threads?

- Yes, the doco indicates these exist.
- Pthread_join() deals with them in a similar way to wait() reaping zombie processes.
- Threads don't terminate independently due to signals
  - Unhandled signals will take out the whole process
- Pthread_detach(tid) tells the system to clean up zombie threads automatically
  - You can use the second argument of pthread_create to start a thread detached

Killing Threads

Can you "kill" threads?
- Not using signals
- Pthread_cancel(tid) exists for a reason

In general, it is difficult to do safely

A "safer" approach is to use a shared variable that the thread checks

```
void* thread_fn(void* v) {
    bool* stop=...      // get this some how
        // other things
    while (!*stop) {
        // do some work
    }
}
```

Pthread_self()

```
pthread_t pthread_self(void);
```

Pthread_self() returns the thread id of the current thread

Racing

See race1.c

Problem: value++ is not atomic

This means that if one thread reads the value, another thread could read that same value before the first thread increments it and then writes the value back. Here, two threads only achieve one increment.

Locks

What about if we use a lock?
See race2.c

This achieves the same outcome as locks are not atomic either

How can we do it then

See race3.c

Using **semaphores**.

Thread Coordination

Thursday, 14 April 2022 3:36 PM

Thread Coordination

Problems

The race2 implementation fails to do two desirable things

1. Ensure mutual exclusion ("mutex")
    a. Much simpler with hardware assistance
    b. Normally access this via library calls
2. Avoid busy waiting

Note: For a mutex algorithm that doesn't rely (too much) on hardware assistance, see Peterson's algorithm.

Semaphores

A *semphore* is an opaque type which represents an integer value.

Two atomic operations

- *Sem_wait()*
    - If the value is > 0, decrement it by 1
    - If the value == 0, stop the process until value > 0, then attempt to decrement
- *Sem_post()*
    - Increment the value

If the *p* threads are waiting on the semaphore and a post() occurs, only one of the threads unblocks, the others stay blocked.

Does everyone get a turn eventually?

- **Starvation** is beyond the scope of this course

Semaphores in Action

See race3.c

- Sem_init()
    - Sets the initial value of the semaphore
- Sem_wait()
    - Decrement the semaphore
- Sem_post()
    - Increment the semaphore
- Sem_destroy()
    - Clean up

Note: always pass pointers to semaphores, do not copy the value itself

Semaphores for Mutual Exclusion

- Sem_init(): set value to 1
- Sem_wait(): acquire the lock / mutex. Only one thread can succeed until post
- Sem_post(): release the lock

Provided that all paths into the critical section(s) require waiting and all paths out post, this will ensure mutual exclusion.

Semaphores for non-busy waiting

- Sem_init(): set value to 1
- Sem_wait(): block until semaphore is available
- Sem_post(): let the other thread know it happened

Other things to do with semaphores

Limit maximum threads active (not as common)

- Sem_init(): set value to N (max threads you want)
- Then at most N threads can pass (wait) until one or more threads leave (post)

Producer and consumer tasks

- Sem_init(): set value to 0
- Each time the "producer" adds a job to the queue, post
- Consumer threads all wait on the semaphore

Notes:

- In this case, some threads only wait and other threads only post
- You still need a separate mutex to control accessing the queue

Volatility and Thread Safety

Thursday, 14 April 2022 3:06 PM

Volatility Example

Consider the following code

```
total=0;
if (*a > 0) {
    total++;
}
if (*a > 0) {
    total++;
}
if (*a > 0) {
    total++;
}
// total is either 0 or 3?
```

Or

```
total = 0;
ref = *a;
total += *a;
total += *a;
total += *a;
total += *a;
// total is 4 * ref?
```

The compiler won't see any reason that it can't optimise the calculation (and most of the time it would be correct).

But

- What if another thread also knows pointer *a* and modifies it?

The **volatile** keyword warns the compiler that the value of the variable may change in ways (and at times) when the compiler won't predict.

e.g.

```
volatile int* p;
            // p is a pointer to a volatile int
volatile const int x=6;
            // yes this is legal
```

Use the **volatile** keyword when any variable may be modified by one thread and read in others

Not Volatile

```
void f(...) {
    int a=...;   // not volatile

}
```

Even if multiple thread falls f() at the same time, they will each see a different (local) variable

Thread Safety

An operation is "thread safe" if multiple threads can have active calls to the function at the same time

Things to look for

- A value could be modified by one thread while another is using it.
    - This included feeing and mallocing, removing entries from lists, … etc
    - Can by tricky to spot (e.g. i++) if you are used to thinking of them as atomic

Look for calls to non-threadsafe functions

- How to tell. Look at the man page (documentation)
- e.g. rand_r
    - _r normally means "reenterant
    - Man 3 rand_r
    - On moss, says rand() and rand_r() are both thread safe. This is because rand_r is actually now obselete on modern systems. But in older versions of C, rand_r was required if using it a multi-threaded application

Some functions make used of hidden static state which might not be obvious

Non-obvious non-atomics

Suppose many (n) threads execute the following at the same time:

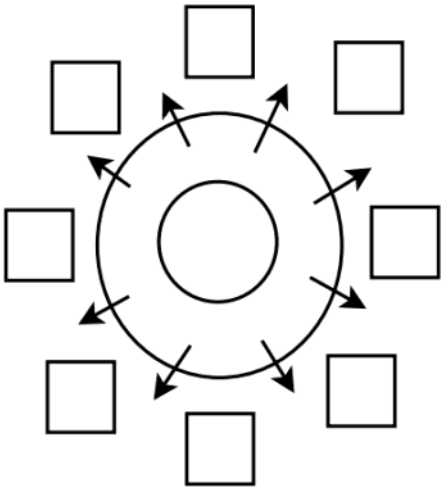
```
    *p = xn;          // i.e. x1...xn
```

Thread 1 stores x_1, … Thread n stores x_n.

What happens?

- There is a race condition where it is not known which thread will write last.
- Will *p even store one of $\{x_1, \ldots, x_n\}$?
    - It depends on the size of x_i vs the size of a processor word.
    - e.g. long double on moss
    - Long on 32-bit systems

Dining Philosophers



Potential problems

- Deadlock
- Livelock
- Starvation

Deadlock

Thread 1:

```
wait(sem1);
wait(sem2);
    // do something
post(sem1);
post(sem2);
```

Thread 2:

```
wait(sem2);
wait(sem1);
    // do something
post(sem2);
post(sem1);
```

- For simple cases. Everyone should request resources in the same order
- More complex cases: beyond the scope of CSSE2310.

Misc Pthreads

Thursday, 14 April 2022 3:34 PM

Fork() and threads

According to documentation, calling fork() in a multithreaded program only duplicates the thread which called fork().

See forknthread.c as an example.

You must also be careful of locks etc.

More on Semaphores

- Sem_trywait()
  - Either lock immediately or error
- Sem_timedwait()
  - Error if lock can't be acquired before timeout
  - Note: this function takes an **absolute time** and not a delta
- Semaphores between processes (not in 2310)

Mutexes and Condition Variables

Pthreads also specifies

- Pthread_mutex_t
  - Only mutual exclusion
  - Can only unlock from the thread when locked
- Pthread_cond_t
  - For waiting for something
  - Each condition var is linked to a mutex.

Makefiles (Revisited)

Saturday, 7 May 2022 6:11 PM

Makefiles can get quite complicated with large projects

- Lots of files
- Dependencies
- Compilation steps

Targets and Dependencies

- A non-file target will always be built (i.e. all recipes will be run) if you 'make' that target
- A file target will be built (i.e. recipes will be run) if you 'make' that target and if
  - The target does not exist, or
  - Any dependencies are newer than the target.
- Dependencies may need to be built/run first

This saves time during compiling when developing as it will only compile files/objects that have changed.

Default Targets, Phony Targets

```
.PHONY: all
.DEFAULT_GOAL := all

all: hello
# build hello from hw.c.      <- remember # begins a comment
hello: hw.c
    gcc -Wall hw.c -o hello
```

- Without arguments, make will run the first rule by default. You can override by defining the .DEFAULT_GOAL variable
- Phone targets are targets which have no associated output file
  - e.g. all doesn't create any output files, it just runs/builds other targets (hello in this case)
- Phony targets are specified as dependencies of .PHONY

Variables

- You can create a variable using <VARNAME>=<VALUE>
  - Without the <>
  - Spaces are ignored
- Variables are accessed using $ (VARNAME)
- We can use targets to modify variables
  - e.g. Change CFLAGS if we make careful

```
CC=gcc
CFLAGS= -Wall -pedantic -std=gnu99
.PHONY: all careful
.DEFAULT_GOAL := all
all: hello

careful: CFLAGS += -Werror
careful: all

hello: hw.c
	$(CC) $(CFLAGS) hw.c -o hello
```

Variable Assignment

Single equals

Variable is recursively expanded at the time of use

```
CFLAGS=-Wall $(DEBUG)
...
DEBUG = -g
```

Just think, the difference is when the variable gets computed

Colon equals

Value is set at the time of definition

```
.DEFAULT_GOAL := all
```

Plus equals

Value is appended to the previous definition (as per usual coding)

```
CFLAGS += -I/local/courses/csse2310/include
```

Some Best Practices/Rules of Thumb

- Use variables to reduce duplication, especially compiler flags
- Have a clean target to remove executables and .o files
- Set a DEFAULT_GOAL and all target to control makefile execution
- Use the CC and CFLAGS built-in variables to set the default C compiler and flags

```
CC=gcc
CFLAGS= -Wall -pedantic -std=gnu99

.PHONY: all clean careful
.DEFAULT_GOAL := all

all: hello

careful: CFLAGS += -Werror
careful: all

# build hello from hw.c
hello: hw.c
    $(CC) $(CFLAGS) hw.c -o hello

clean:
    rm -f hello
```

Compile and Link Separately

- When you have many source files, you don't want to compile all of them every time
- Separate out compiling each source file by creating rules for each source and object file pair
- We then have a single rule per program to link the object files together

```
CC=gcc
CFLAGS= -Wall -pedantic -std=gnu99
.PHONY: all clean
.DEFAULT_GOAL := all

all: hello

# link hello from object files
hello: hw.o
   $(CC) $(CFLAGS) hw.o -o hello

# compile hw.c to an object file
hw.o: hw.c
    $(CC) $(CFLAGS) -c hw.c

clean:
    rm -f hello *.o
```

Multiple Source Files

To add more source files

- Add the object file as dependencies in the linking step (in hello for this example)
- Add a new rule for each source file (here we added hw2.o)
- Update the clean rule (note the additional *.o in the rm)

```
CC=gcc
CFLAGS= -Wall -pedantic -std=gnu99
.PHONY: all clean
.DEFAULT_GOAL := all

all: hello

# link hello from object files
hello: hw.o hw2.o
    $(CC) $(CFLAGS) hw.o hw2.o -o hello

# compile source files to objects
hw.o: hw.c
    $(CC) $(CFLAGS) -c hw.c
hw2.o: hw2.c
    $(CC) $(CFLAGS) -c hw2.c
clean:
    rm hello *.o
```

Another View of Makefiles

A makefile defines a dependency tree and make traverses it

- If all dependencies are met for a node, that node can be built (if out of date)

```
CC=gcc
CFLAGS= -Wall -pedantic -std=gnu99
.PHONY: all clean
.DEFAULT_GOAL := all

all: hello

# link hello from object files
hello: hw.o hw2.o
    $(CC) $(CFLAGS) hw.o hw2.o -o hello
# compile source files to objects
hw.o: hw.c
    $(CC) $(CFLAGS) -c hw.c
hw2.o: hw2.c
    $(CC) $(CFLAGS) -c hw2.c
clean:
    rm hello *.o
```

Implicit Rules

Make has some built-in rules to build certain types of files

- For example, it knows how to build
  - .o files from .c files
  - Executables from .o files
  - Executable from .c files
- We can simplify our makefile by only listing the .o file dependencies
- Note: The built-in rules require the CFLAGS and CC variables to be set appropriately
  - Also LDFLAGS if you want make to do the linking for you

Implicit Compilation

Example

```
CC=gcc
CFLAGS= -Wall -pedantic -std=gnu99
OBJS=hw.o hw2.o
PROG=hello
.PHONY: all clean
.DEFAULT_GOAL := all

all: $(PROG)

$(PROG): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o $(PROG)

clean:
    rm $(PROG) $(OBJS)
```

Special Variables

Many including

- &@
  - The file name of the target
- &<
  - The name of the first prerequisite
- &?
  - The names of all prerequisites newer than the target
- &^
  - The names of all prerequisites

Modularity

Using multiple Source Files

One file main.c

```c
typedef struct {                        // Data structure
  ...
} MyList;

void print_list(MyList *list);          // Function declaration
...
    MyList *list = (MyList *)malloc(sizeof(MyList)); // Use in code
    ...
    print_list(list);
...

void print_list(MyList *list) {     // Function definition
    ...
}
```

list.h

```c
#ifndef LIST_H
#define LIST_H
// Data structure
typedef struct {
  ...
} MyList;
// Function declaration
void print_list(MyList *list);
#endif
```

list.c

```c
#include "list.h"
// Function definition
void print_list(MyList *list) {
    ...
```

main.c

```c
#include "list.h"
...
    // Use of function in code
    MyList *list = (MyList *)malloc(sizeof(MyList))
    ...
    print_list(list);
    ...
```


Now the makefile

Makefile

```
CC=gcc
CFLAGS= -Wall -pedantic -std=gnu99
.PHONY: all clean
.DEFAULT_GOAL := all
all: main

# link main from object files
main: main.o list.o
    $(CC) $(CFLAGS) $^ -o $@

# compile source files to objects (note header file dependency)
main.o: main.c list.h
list.o: list.c list.h
clean:
    rm main *.o
```

Header Guards

Protect a header file from being included more than once

- Header guards must be unique (often FILENAME_H or similar)
- Forgetting them can cause a lot of interesting and hard-to-find errors

Example using header guards

```
#ifndef UNIQUE_NAME
#define UNIQUE_NAME

    // Declarations go here

#endif // UNIQUE_NAME
```

Linker Modularity vs Function Pointers

Linker Modularity

- Multiple source files
- Same function is implemented differently in each file (but same signature)
- The linker (part of gcc) will choose the functionality based on input files at **compile time**.

english.c

```
void greeting(char* msg) {
    printf("Good morning %s!\n", msg);
}
```

dutch.c

```
void greeting(char* msg) {
    printf("Goedemorgen %s!\n", msg);
}
```

afrikaans.c

```
void greeting(char* msg) {
    printf("Goeiemore %s!\n", msg);
}
```

language.h

```
void greeting(char* msg);
```

main.c

```
#include "language.h"

int main(int argc, char** argv) {
    if (argc < 2) {
        return EXIT_FAILURE;
    }
    greeting(argv[1]);
    return EXIT_SUCCESS;
}
```

```
$ gcc -o main-e main.c english.c
$ gcc -o main-d main.c dutch.c
$ gcc -o main-a main.c afrikaans.c
```

Example

Building multiple programs which each have subtly different functionality

```c
void english(char* msg) {
    printf("Good morning %s!\n", msg);
}

void dutch(char* msg) {
    printf("Goedemorgen %s!\n", msg);
}

void afrikaans(char* msg) {
    printf("Goeiemore %s!\n", msg);
}
typedef void (*GreetFn)(char*);
```

```c
int main(int argc, char** argv) {
    GreetFn greetings[] = {
            english, dutch, afrikaans
            }
    if (argc < 2) {
        return EXIT_FAILURE;
    }

    for (int i = 0; i < 3; i++) {
        greetings[i](argv[1]);
    }
    return EXIT_SUCCESS;
}
```

Function Pointers

- Multiple different functions - all have the same type signature
    - Same parameter types and ordering
    - Same return type
- Function selected at **runtime**.

Example

Reducing code duplication and changing functionality based user input

More on Make

- Official manual for GNU make

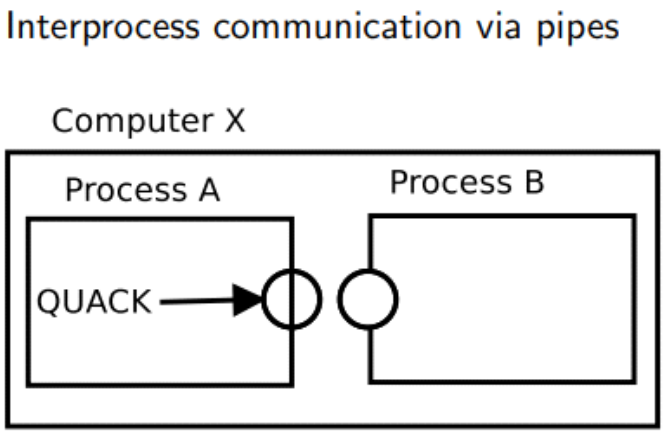
https://www.gnu.org/software/make/manual/make.html

- Can build dependency lists using the compiler (-M and -MM arguments to gcc)

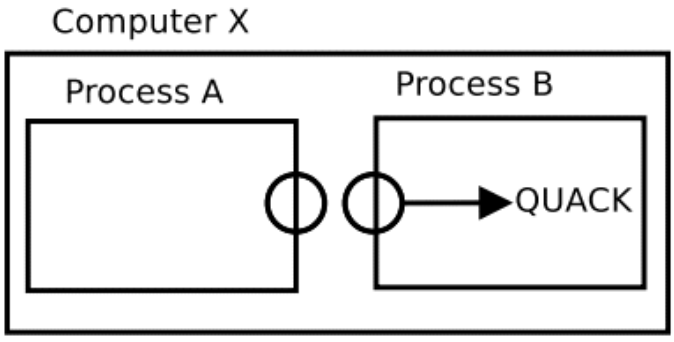
Introduction to Networks

Saturday, 7 May 2022 6:12 PM

Overview: Communication via pipes

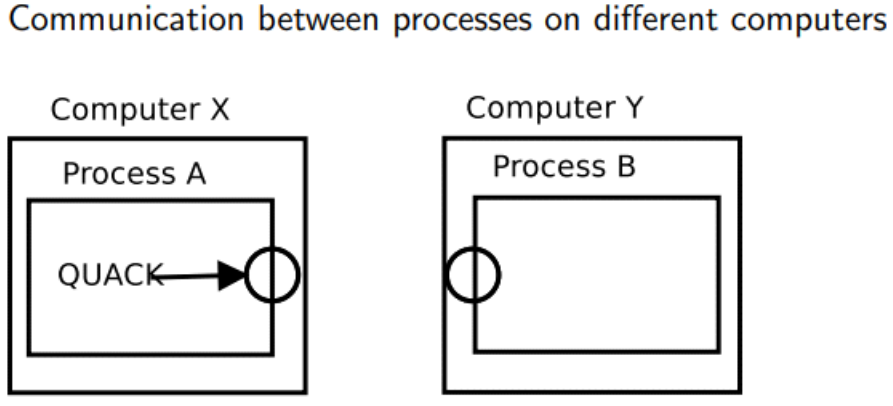


Then

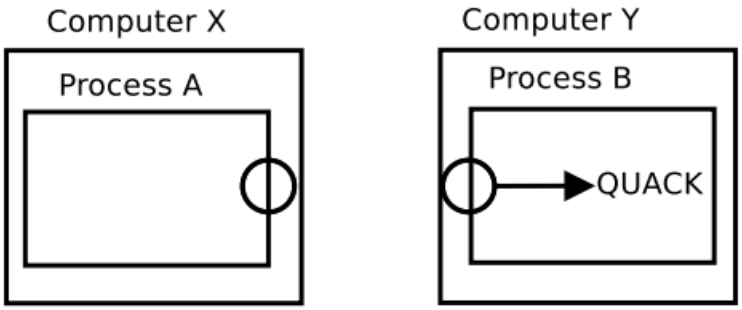


Internals are hidden from the programmer

Now we are looking at communication between computers



Then



Software Layers

Rather than developing software in one go, it may be easier (though not always), to write in layers

- Make each part of the program more manageable
- Add capabilities of modes of operation the lower level doesn't allow
- Hide procedural aspects
- Make one system act like another
- Allow lower levels to be replaced without disrupting the whole system

A connected set of layers is a "stack".

Layers Example: C Standard Input/Output

Standard C I/O is via "streams" (FILE*)

- Act as an unstructured sequence of bytes
- Can read from stream fgetc(), fscanf(), …
- Can write to stream using fputc(), fprintf(),…

Underneath might be

- Operating System
  - Then device, e.g. terminal, disk, network
- Just a library that interacts directly with hardware.

C Standard I/O Library

- Hides lower level details (e.g. OS)
  - From the application programmer
- Provides useful services, e.g.
  - Buffering

Operating System (if there is one)

- Hides lower level details (e.g. device details)
  - From the C standard I/O library programmer
- Provides useful services, e.g.
  - Access control

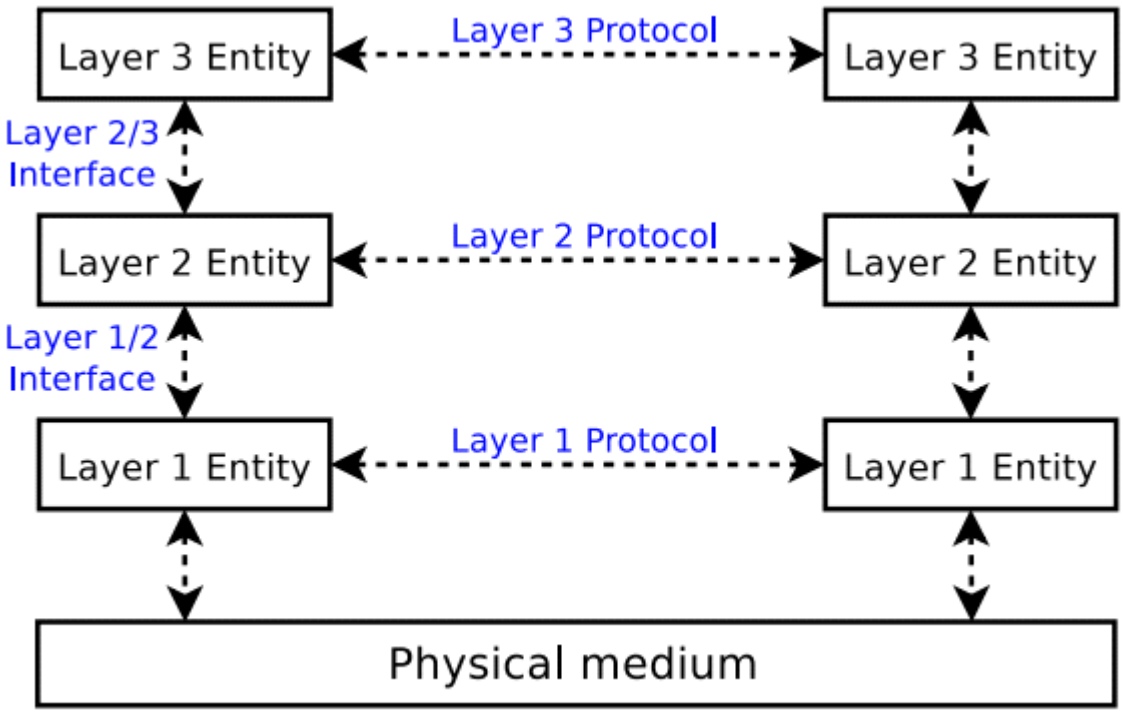
Protocols

Definition

(non computing) system of rules about the correct way to act in formal situations

(communications) a protocol is a set of rules governing exchange of data between two entities.

Protocol Stacks



- Higher layers are shielded from lower level implementation details
- Inter-layers interfaces can vary from machine to machine - they're not part of the protocol

Entities

Definition

Entity: (computing) *Active* elements in each layer

- Can be software (e.g. library code or kernel code or separate process)
- Can be hardware (integrated circuit)

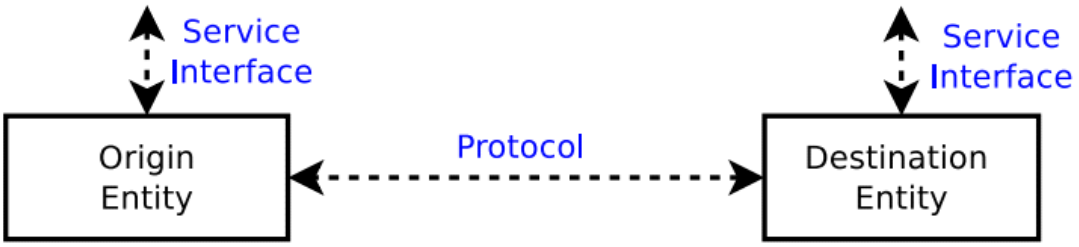
Peer Entities

- Same layer - different machine
- Sometimes called **peer processes**

Services vs Protocols

Distinction is important

- Service (interface)
  - Operation(s) available to a higher level entity
- Protocol
  - Rules for communication between peer entities



- Protocols are between entities
- Services are between layers

Service Categories

Connection

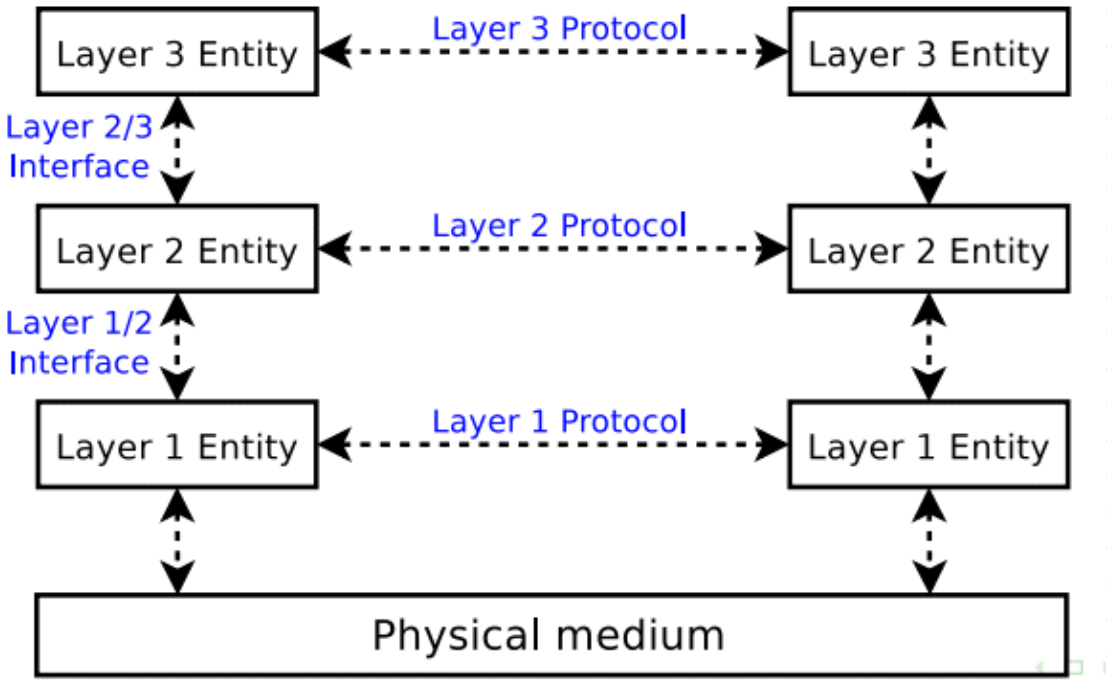
- Connection-oriented
  - Modelled after telephone system
  - Service user - establishes connection, uses connection, releases connection
- Connectionless
  - Modelled after postal service
  - Service user - specifies full destination address for each message sent

Reliability

- Reliable
  - Service provider makes "best effort" attempt to communicate
  - This doesn't mean 100% guarantee. "Best effort" is what we mean here. So if the computer is off, this isn't going to work despite our best effort.
- Unreliable
  - Talk about this later
  - UDP

Virtual Protocols

- No data is directly passed between peer entities
- Each layer passes data and control information to layer below until the lowest layer is reached
- Actual communication occurs via physical medium (wire, optic fibre, air)



Headers/Envelopes

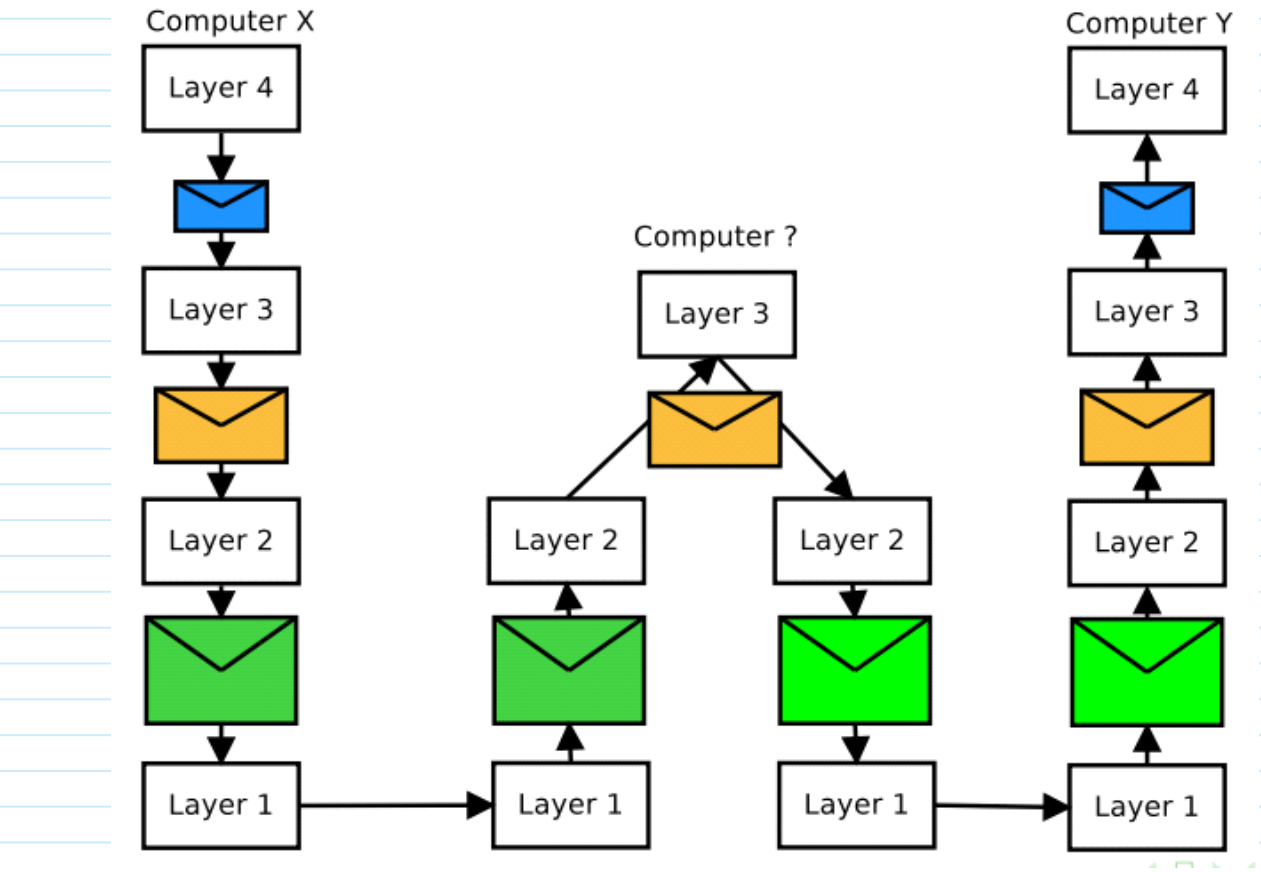
In order to work with lower layers, the "message" may need

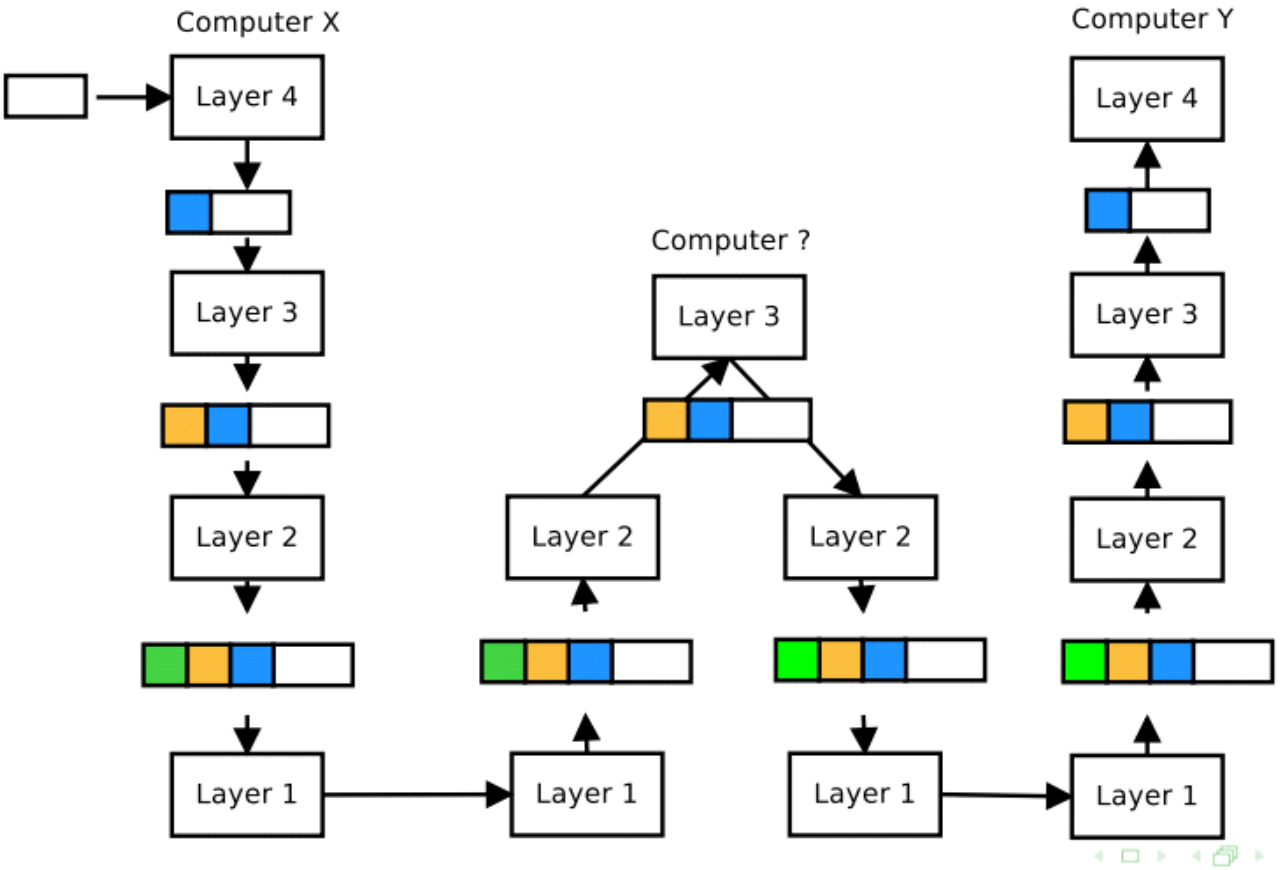
- To be encoded (e.g. bytes sent via an optical network -> light pulses)
- Have extra information
    - Headers (who is the message to? From?
    - Footers (ethernet uses both)

That extra information

- Will (usually) be removed at the other end.
- Might be the only part of the message that level understands
    - So we use an envelope as the analogy (you can't see into the envelope)

View 1 - Envelopes





Networks of Networks

An *internetwork* (**internet**) is an internet connected set of networks.

The global **Internet** (upper-case 'I') is the most famous example of an internet (lower-case 'i')

Internet

Based on the TCP/IP protocol family

- **IP** (Internet Protocol)
  - Provides basic addressing scheme and unreliable delivery capability of packets (datagrams) from host to host
- **UDP** (User Datagram Protocol)
  - Uses IP to provide unreliable datagram delivery from process to process
- **TCP** (Transmission Control Program)
  - Uses IP to provide reliable byte streams from process to process
  - Has acknowledgements and retransmissions built-in

Network Stack

Thursday, 19 May 2022 1:29 PM

Network Stack

We will talk about the **5** layer "TCP/IP" stack

- Application Layer
- Transport Layer
- Network Layer
- Link Layer (or Data-link layer)
- Physical Layer

1. Physical

Layer 1 is the "Physical" layer

This is the medium through which signals travel through, e.g.

- Current in a wire
- Infra-red
- Microwave
- Audio
- Carrier pigeons?

1. Data-Link

Layer 2 is the (Data-)Link layer

- Transfers data between two nodes on a network segment (link) across the physical layer

Peers can communicate directly via messages:
- Unicast or Broadcast?
- Need addressing information
- Timing or other information?

Example
- Ethernet frames
- WiFi

Addresses:
- MAC Addresses = "Medium Access Control" (Controlling access to the physical medium)
- Ethernet MAC Addresses are 48 bits

1. Network

Layer 3 is the Network layer

Exchange messages with any other host on the "internet"

Uses the internet protocol (IP):
- Messages are "IP datagrams"

    - Often called "IP packets"
- IPv4 addresses are 32 bits (written as dotted quads" e.g. 130.102.72.9)

Sends messages via the link layer.

- Messages may travel through multiple devices before they reach their destination
- The network layer needs to know "which direction" to send a message to get it to its destination
- How the network layer works this out is beyond the scope of this course.

1. Transport

Exchange messages with a **process** on a host on the internet

Two protocols to choose from

- UDP = User Datagram Protocol (Diagrams)
- TCP = Transmission Control Protocol (Segments)

Sends messages using the network layer

Transport Addresses

Addresses? Ports = a 16 bit integer

- Web -> 80
- SSH -> 22
- SSL -> 443

- Ports below 1024 are restricted on Unix type systems
- Port 0 doesn't do what you think it does
- High numbered ports are "ephemeral" (short-lived)
    - Internet Assigned Numbers Authority (IANA) suggests 49152 to 65535
    - Many Linux kernels use 32768 to 60999
    - e.g. used by clients in client-server communication

1. Application

"Everything else"

- Web servers and clients (browsers) - communicating via HTTP
- Ssh servers (e.g. sshd) and clients (e.g. putty)
- Games?
- SMB (files and printers)

Sends messages using via the transport layer
Addresses?

- URL/URI?
- …

Why so many addresses?

- Application specific addresses: differentiate between resources (e.g. URLs)
- Port: To differentiate between processes on the same computer
- IP: which computer is the process on?
- MAC: Which device is this direct message to?

MAC vs IP?

So why do we need IP and MAC?

- The internet was designed to connect lots of networks together
- … at a time when people had all sorts of different network tech.
- Different hardware was not going to understand addresses from other systems
- Need a separate "global" addresses that new software/hardware could process even if the local network didn't understand it
- Ethernet MAC is "supposed to be" globally unique but it isn't hierarchical
- IP Addresses are hierarchical - blocks are associated with organisations - sub-blocks are associated with sub-units

Note:

- IP and MAC addresses technically don't identify *devices* - they identify interfaces.

IP, TCP and UDP

Thursday, 19 May 2022 1:30 PM

TCP

The network layer deals with packets (datagrams). We'd like:

- Streams of bytres
  - Or at least not needing to worry about how big a message can be
- "Reliable delivery"
  - This is actually impossible but the internet will try
  - Basically, send some packets, if you don't get an acknowledgement, send them again.

The transport layer's TCP protocol does these things

TCP Connections

TCP is connection oriented

- In order to communicate with something, you establish a connection first
- (Not actually a physical circuit)
  - The internet is still packet switched
- Making a connection requires messages to travel there and back
- Closing a connection is polite
  - There will be a timeout as a fallback
- Connections are bi-directional

UDP

UDP deals with discrete messages

- With a maximum size
- No guarantee of delivery or acknowledgement

So why would you want this?

Why UDP?

What about

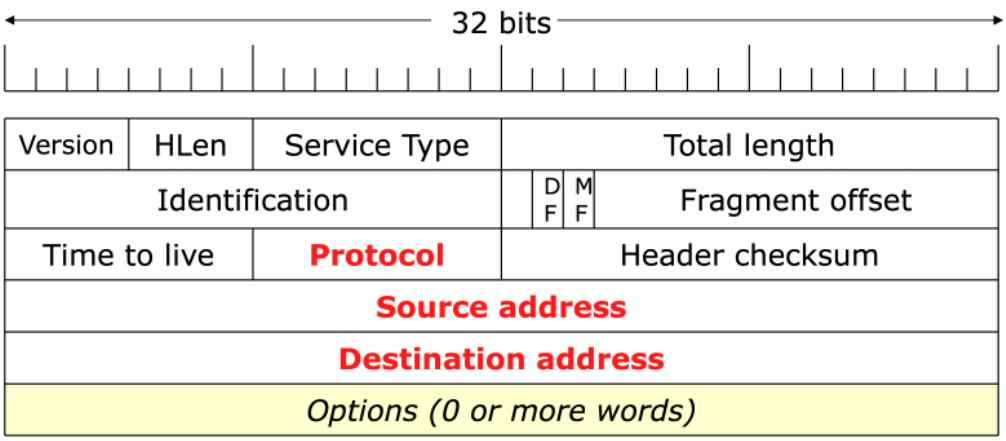
- Streaming video
  - One corrupt frame or audio segment is between than stalling the whole video stream until a perfect copy is found.
- Games or interactive environments
  - Sync up every so often
- Congested networks?
  - Some commands getting through in a timely fashion better than being late?
- Many small operations when you don't want to set up a connection for each one.

IP - Internet Protocol

- Connectionless model of delivery
- Unreliable - no guarantees of successful transmission
- IP Datagrams
  - Sometimes called packets

- Consist of
  - Header
  - Body part (data or "text")
- Header consists of
  - 20 byte fixed part
  - 0 to 40 byte optional part

Internet Protocol (IPv4) Header



We're particularly interested in the fields in **red** - protocol, source and destination address.

IP Header (v4): Protocol Field

- 8 bits
- Range: 0 to 255
- Identifies higher level protocol entity to which packet should be passed, e.g.
  - 6 = TCP
  - 17 = UDP
- Defined by IANA:
  - https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml
- See also in /etc/protocols on a Linux system
  - Moss

IP Header (v4): Address Fields

Source and Destination Address

- 32 bits each
- Destination address
  - Used to route the datagram to intended destination
- Source address
  - Destination can choose whether to receive datagram
  - Destination knows whom to reply to

IP Addresses

- 32 bit IP addresses (v4) often write in dotted decimal notation (for human consumption)
  - Each 4 bytes written in decimal, e.g. 130.102.72.9
- Some addresses ave special meanings
  - e.g. broadcast to all hosts on a particular network (more about this in week 11)
- Where do addresses come from?

    - Internal configuration
        - Device administrator chooses one - but it may not work on that network
    - External configuration
        - On startup/connection, the device can ask what addresses should be used
        - DHCP - Dynamic Host Configuration Protocol.

UDP

- Header is 8 bytes in size
    - 2 bytes - source port
    - 2 bytes - destination port
    - 2 bytes - datagram length (including header)
    - 2 bytes - checksum
- UDP datagrams must fit in IP datagrams so max UDP payload size is limited by max IP datagram size (65535 bytes)
- Max UDP payload = 65535 - 8 - 20 = 65507 65507 bytes
    - Note: large IP datagrams get fragmented (MTU on most links is 1500 bytes) and every fragment must arrive successfully for the UDP datagram to be received
    - So best to send smaller datagrams and avoid fragmentation
    - MacOS limits UDP datagrams to 9,216 bytes for example

TCP

- Connection-oriented
- Reliable (over unreliable IP internetwork)
- Byte-stream
    - Not message stream
- Full-duplex
    - Bidirectional
    - Can send messages in both direction
- Point-to-point (end-to-end)
    - Not multicast or broadcast
- Each machine has a TCP transport entity
    - User process or part of kernel

TCP Responsibilities

- Sender
    - Accepts byte streams from local processes
        - Message boundaries are not preserved
    - Breaks data into pieces < 64k bytes (max IP datagram size)
        - In reality - pieces limited by MTU for lowest level (1500 bytes typical for Ethernet)
    - Sends each piece as IP datagram
    - Time-out and retransmit if no acknowledgement received.
- Receiver
    - IP datagrams containing TCP data are passed to TCP entity for reconstruction
    - Acknowledgement receipt
    - Reassemble datagrams in proper sequence

TCP Protocol

TCP entities exchange **segments**

- Data + 20 byte header, including
    - Source & destination port numbers (16 bits each)

    - Sequence number (for reordering)
    - Acknowledgements
- Segments with zero data are valid
- Segment size limited by
    - IP datagram size
    - Network MTU

TCP Connections

Identified by combination of

- Source IP address
- Source port number
- Destination IP address
- Destination port number

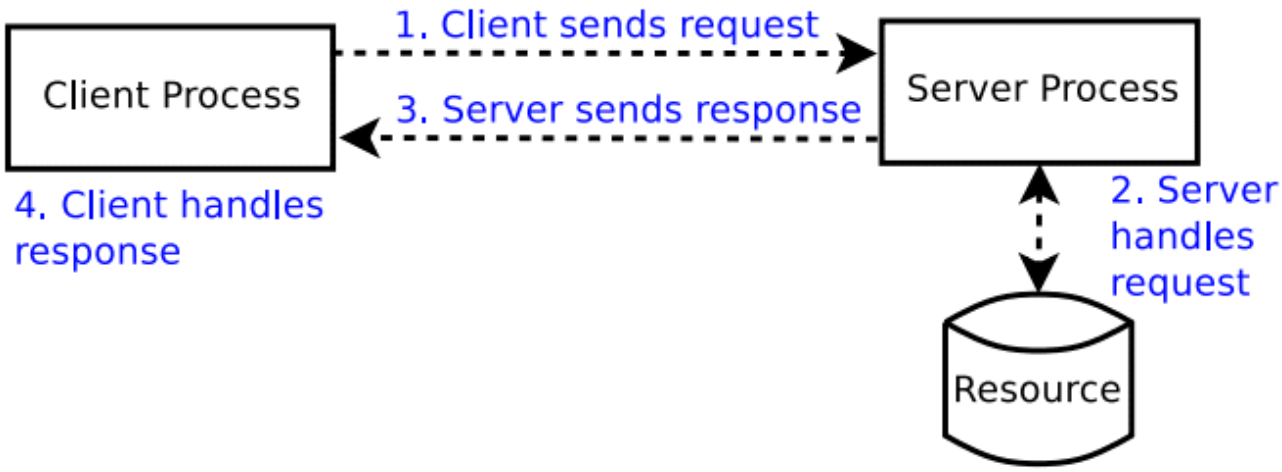
Client/Server

Thursday, 19 May 2022 1:14 PM

Client ↔ Server

Most network applications based on the **client-server model**

- A **server** and one or more **clients**.
- Server manages some **resource**
- Server provides some **service** by manipulating resource for clients



Server — a **process** that waits for requests from clients

- e.g.: A web server waits on port 80 for browsers to connect and request pages
- Sshd waits for connections on port 22

Client — a **process** that submits requests to the server

- A web client connects to a sever on port 80 and requests pages
- Ssh or putty connects to the server on port 22

Note

A single process **can** be both a server and a client

Beware of Terminology

- "Server" refers to a process running on a machine
  - Note: Server on port x, not port x on the server
- "Server" is also often used to refer to the hardware that the server process runs on

TCP Connections

- The client/server distinction applies to single connections
- For any TCP connection, there is always a client and a server
  - Client initiates connection
  - Server listens for incoming connections
- An application acting in a peer-to-peer mode, may have some connections for which it was the server and others for which it was the client
- Once a connection has been established, there is no difference between what a client can do and what a server can do

Coming Up

- Next time - Network Programming
  - How to deal with addresses
  - Sockets
  - Typical client
  - Typical servers, (e.g. multi-threaded, multi-process)

- Other Protocols (DNS, IMCP, HTTP, …)
- Internet and Ethernet

Contact - Network Programming 1

Saturday, 7 May 2022 6:12 PM

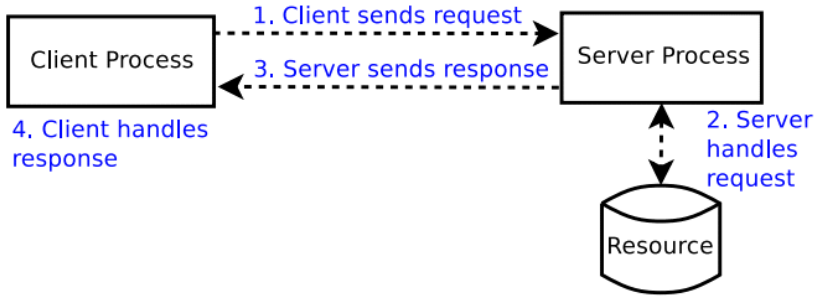
Outline

- Network programming APIs
- Sockets API
- Clients and Servers
- IP Addresses and Socket addresses in C
- Example Client

Client ↔ Server

Most network applications based on the **client-server model**

- A **server** and one or more **clients**.
- Server manages some **resource**
- Server provides some **service** by manipulating resource for clients

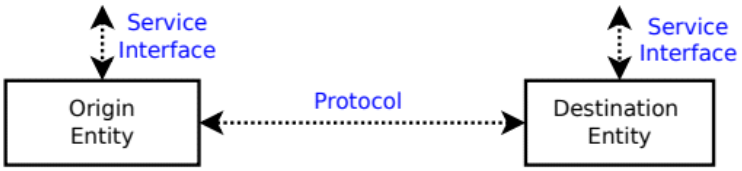


Network Programming APIs

- API = Application Programming Interface
  - Set of procedures/functions/methods that provides access to some service
- Service of interest to us - transport
- Many network transport APIs exist
  - NetBIOS
  - TLI
  - XTI
  - Sockets
  - Winsock

Network Transport APIs

- Hide details of network layer (routing etc.)
- Provide simple (?), clean interface for application developers
- Ideally, hide protocol details
  - Remember the distinction between services and protocols



Sockets

- Introduced in 4.2BSD (1983)
- Originally C based - many languages now
- A **socket** is a communication endpoint
  - Associated with a file descriptor in UNIX - can do file I/O on most sockets
  - Main distinction between regular file I/O and socket I/O is how the application "opens" the socket
- Many types of sockets
  - We're only interested in **Internet sockets**.

In Linux
- Sockets are bidirectional
- While pipes are unidirectional

Internet Sockets

Two types of interest to us

- **Stream** sockets
  - Full-duplex, byte streams
  - Reliable, connected
- **Datagram** sockets
  - Unreliable, connectionless
  - Limited-size messages

Another type - **Raw** sockets
- Allows direct access to network layer
- See raw(7) man page

Socket Primitives

Socket(...)
Creates a new communication end-point

Bind(...)
Attached a local address to a socket

Listen(...)
Indicate willingness to accept connections

Accept(...)
Wait for a connection attempt to arrive

Connect(...)
Attempt to establish a connection

Send(...) or **write(...)**
Send data over the connection

Send(…) or **write(…)**
Send data over the connection

Recv(…) or **read(…)**
Receive data over the connection

TCP

Sendto(…)
Send datagram

Recvfrom(…)
Receive datagram

UDP

Close(…)
Destroy the socket

Shutdown(…)
Close down one side of connection (or both sides).
Probably won't use this one

Typical Server (Stream-based)

- Create socket
  - Socket(…)
- Bind to address/port
  - Bind(…)
- Specify willingness to accept connections
  - Accept(…)
- Loop
  - Accept(…)
  - Block waiting for a connection
    - Accept(…) returns a net socket
    - Original socket continues to listen
  - Deal with request (possibly within a new thread/process)
    - Communicate with client
      - Send(…) or write(…)
      - Recv(…) or read(…)
  - Close connection
    - Close(…)

Typical Client (Stream-based)

- Create socket
  - Socket(…)
- Connect to server at particular address/port
  - Connect(…)
- Send and receive data as necessary
  - Send(…) or write(…)
  - Recv(…) or read(…)
- Close connection (when finished)
  - Close(…)

Clients don't normally call bind(…)

- Don't care what the outgoing port is

Typical Datagram Applications

Typical Datagram Receiver

- Create datagram socket
  - Socket(…)
- Bind to address/port
  - Bind(…)
- Receive messages
  - Recvfrom(…)
- Send reply
  - Sendto(…)
- Close socket
  - Close(…)

Typical Datagram Sender

- Create datagram socket
  - Socket(…)
- Send message
  - Sendto(…)
- Close socket
  - Close(…)

IP Addresses in C

- 32-bit IP addresses are stored in an IP address struct:
- Struct in_addr

- On linux, defined as follows

```
typedef uint32_t in_addr_t;
struct in_addr {
    in_addr_t s_addr;
};
```

- IP Addresses (and port numbers) are stored in **network byte order** - i.e. big-endian

(Some systems have a union inside the struct)
(Linux on x86 uses little-endian to represent numbers

IP Address Functions 1

Useful byte-order conversion functions

- **htonl()** - convert long int from host to network byte order
  - Long int is 32-bit. Used for IP Addresses
- **htons()** - convert short int from host to network byte order
  - Short int is 16-bit. Used for port numbers
- **ntohl()** - convert long int form network to host byte order
- **ntohs()** - convert short int from network to host byte order

IP Address Functions 2

Functions that deal with strong (ASCII) representations of IP addresses, e.g. 1.2.3.4 (dotted decimal)

```
in_addr_t inet_addr(const char *cp);
```

- Converts string (cp) to network-byte ordered IP address
  - Best to avoid this function - error return value (-1) corresponds to return value from valid IP address
  - 255.255.255.255 is -1 so you wouldn't be able to differentiate between an error and a valid IP address

```
int inet_aton(const char *cp, struct in_addr *inp);
```

- Converts string (cp) to network-byte ordered IP address (inp)

```
char *inet_ntoa(struct in_addr in);
```

- Converts network byte ordered address to string (returns pointer to statically allocated space - not thread safe).

IP Address Functions 3

Other functions

- Inet_ntop()
  like inet_ntoa() but
  - Thread safe (destination string is passed as an argument)
  - Handles IPv6 as well
- Inet_pton()
  like inet_aton() but
  - Handles IPv6 as well

"p" = presentation (i.e. how it can be presented to users)

See addr1.c, addr2.c

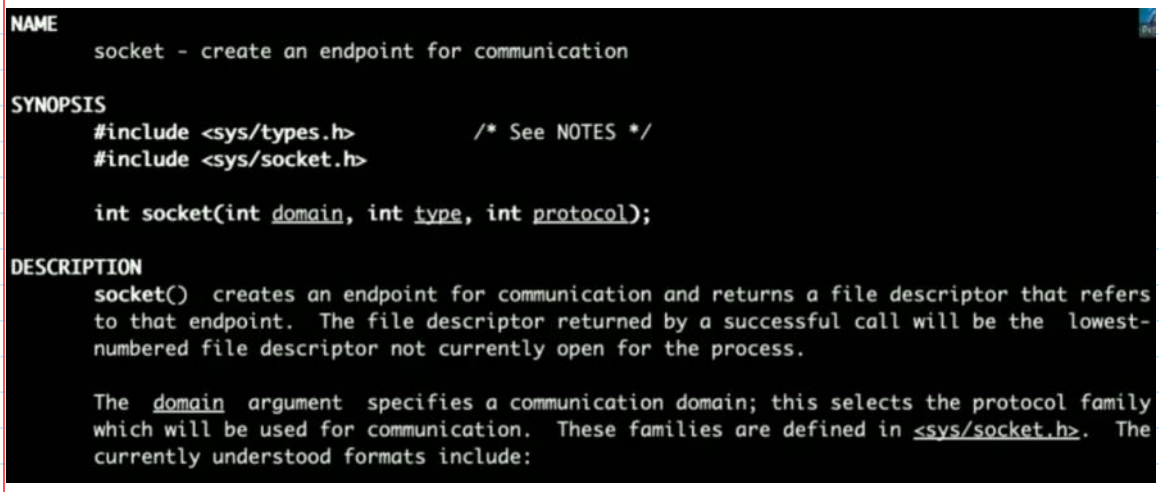
Creating a Socket

Socket(...)
Creates communication end point and returns a socket file descriptor

```
int fd;

if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("Socket creation failed");
    exit(1);
}
```

- AF_INET - protocol family - Internet protocol family (IPv4)
- SOCK_STREAM - socket type - reliable byte stream connection
- Third argument is specific protocol - 0 = default (TCP in this case)

- Socket() returns -1 on failure



```
NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int socket(int domain, int type, int protocol);

DESCRIPTION
       socket()  creates an endpoint for communication and returns a file descriptor that refers
       to that endpoint.  The file descriptor returned by a successful call will be the  lowest-
       numbered file descriptor not currently open for the process.

       The  domain  argument  specifies a communication domain; this selects the protocol family
       which will be used for communication.  These families are defined in <sys/socket.h>.  The
       currently understood formats include:
```

Socket Address Structures

Generic socket address[3]

- For address arguments to connect(), bind, and accept()

```
struct sockaddr {
  unsigned short  sa_family;    // protocol family
  char            sa_data[14];  // address data
};
```

[3]Structured this way because C did not have generic (void*) pointers when sockets interface was designed

Internet-specific socket address
Must cast struct sockaddr_in * to struct sockaddr * for connect(), bind() and accept()

```
struct sockaddr_in {
  unsigned short  sin_family; // address family (AF_INET for us)
  unsigned short  sin_port;   // port num in network byte order
  struct in_addr  sin_addr;   // IP addr in network byte order
  unsigned char   sin_zero[8];// padding to sizeof(struct sockaddr)
};
```

ncat / nc

You want to be able to debug your network code without needing both your client and your server working. Netcat to the rescue.

To start a server (it's -ell):

```
$ nc -4 -l 43210
```

Using IPv4 listen on port 43210.

To connect to your server

```
$ nc -4 localhost 43210
```

Using IPv4, connect to port 43221 on a computer called **localhost** (your computer).

Notes

- Connections are bi-directional (you can type into either and it gets sent to the other end).
- Ports are machine-wide so you will need to pick a number that no one else is using.
- On Moss, you can also use the machine name **localhost4** which only has a IPv4 address.
- **Localhost** is defined on most systems to give **an** IP address of the machine you are on.
    - Remember, machines can have multiple names and multiple Ips
    - **Localhost** is **usually** 127.0.0.1, however, 127.0.0.1 is less portable, e.g. it wouldn't work on an IPv6 machine.

- There are multiple versions of netcat out there.
- We will be using the version installed on Moss. (In Debian this is the netcat-openbsd package).
- You may need to check the doco for the version installed on your system.

Client Steps

- Find the address of the machine you wish to connect to
- Make a socket
- Connect() to the server
- Wrap socket descriptor for nicer I/O
    - You should dup() the descriptor before calling fdopen() and use one FILE stream for reading and another for writing

Client Example

See net1.c and net2.c

Net1.c is just connecting to a server
Net2.c is actually sending and receiving data both ways - Look at this for assignment 4

So struct addrinfo

- Contains a struct sockaddr*
- Which is actually a struct sockaddr_in*
    - Which contains a struct in_addr
        - Which contains a in_addr_t which is an unsigned integer of some sort.

Coming up

Network programming continued - network servers, including

- Multi-process server
- Multi-threaded server

Network Programming 2

Thursday, 19 May 2022 12:00 PM

Contents

- Network servers
- Multi-process network server
- Multi-threaded network server
    - This will be a starting point for assignment 4

Client

- Client code from last time revisited

Server

See net3.c

Server Steps

- Make a socket
- (Optionally) set socket options
- Bind() the socket to a port (and maybe an interface).
    - Otherwise it becomes
    - I have a server and can you guess where.
- Set the socket to listen() for connections
- Call accept() to get allow a connection
    - Use the new fd to interact with the client

See net3b.c for options

Ephemeral Ports

Last week we mentioned port 0 was special…

- In principle, port 0 can be valid, but its use is very rare an implementation dependent.
- In practice, and in Unix socket programming, requesting port 0 is actually saying
    - Give me a port, any port
- The system will create a socket on a valid, unused, non-reserved port
- You need to do some checks to actually figure out what this port number is

See net4.c

This would not be normal for a server

- Normally you want clients to connect to a **known** port number
- We will do it to avoid port collisions during assignment 4

Multi-programming

Accept() is a blocking call. If you want to be able to work on a connection **and** wait for new connections, you need extra workers.

Possibilities

- Fork() and let the new child handle the new connection
- Create a pthread to handle the new connection
- Use a non-blocking call to check for I/O and connections (i.e. using select())

Concurrent Servers

Want to be able to handle communication with more than one client at a time

Option

- Multi-process
- Multi-threaded
- I/O multiplexing with select().

Muli-process Server

- Server accepts connection - gets a socket (file descriptor) for client communication
- Creates child process to handle communication

Key issues

- Prent must close the client socket after fork.
    - Or resources won't be reclaimed when the client finishes (refcount on fd still 1)
- Child must close the listening port after fork
- Parent must reap dead children
    - Don't want zombies for a long-life server!
    - Accept() system call willl be interrupted on SIGCHLD (unless use SA_RESTART)
    - Or ignore SIGCHLD (see note on sigaction() man page)

Multi-process Server Example

See server.c - to be modified

Multi-process Server Pros and Cons

+ Handles multiple connections concurrently
+ Relatively simple to code
- Additional overhead for process control
- Difficult to share data between processes (need pipes, or shared memory, etc).

Multi-threaded Server Example

See server.c - again to be modified

Multi-threaded Server Pros and Cons

+ Handles multiple connections concurrently
+ Relatively simple to code
+ Easier to share data between threads
- Though may need mutexes or semaphores

- Some overhead for thread control
- Can be difficult to debug

I/O Multiplexing using select()

Select() system call.

- Can wait on many file descriptors at once (provide "set" of fds of interest to select())
- Return when data available or event happens on one or more fds (and updates set of fds)

Pros and Cons of using Select()

+ Handles multiple connections concurrently
+ one logical flow (single process, single thread)
+ can therefore be easier to debug
- more complex to code (managing sets of file descriptors)

Realities

Large scale servers

- May have combination of approaches: process/threads/select()
- May have pool of worker processes/threads and not terminate them after each client
    - May terminate and restart processes occasionally to avoid memory leak issues
- SO_REUSEPORT socket option allows multiple listeners on a socket - on BSD systems & Linux >= v3.9 (Supported on Moss)
- May use system specific functions for performance (e.g. epoll() on Linux) or common frameworks for portability (e.g. libevent - which uses various system specific functions)

Coming Up

- Secure and defensive programming
- IP Addresses
- Some other protocols (e.g. HTTP)

Contact - Secure and defensive programming

Monday, 23 May 2022 12:55 PM

Why does this matter?

- Cyber security is everybody's business
  - Economic impact -> US $10T by 2025
  - Social impact
- Often times, people are the weakest link
  - Password reuse
  - Phishing and other scams
  - Ransomware
  - Plugging in random USBs found
- However, as developers we make software artifacts that do Important Things and handle Sensitive Data
  - We have an ethical and possibly legal responsibility to do that as safely and securely and robustly as possible

Prevention is better than cure

Most software is developed under time and cost constraints

Historically, the focus has been on features and availability

"Developing secure software is expensive!"

The subtext here is

"Customers pay for features, not security updates"

As a community, we are finally beginning to realise that it's **more** expensive to

- Release vulnerable software
- Have it exploited
- Pay for the consequences
- Fix the vulnerabilities

Guidelines for Secure Programming

Software Engineering Institute CERT C Coding Standard, e.g.

- 4.1 EXP30-C. Do not depend on the order of evaluation for side effects
- 4.2 EXP32-C. Do not access a volatile object through a non-volatile reference
- 4.3 EXP33-C. Do not read uninitialised memory
- 4.4 EXP34-C. Do not dereference null pointers
- …
- 5.1 INT30-C. Ensure that unsigned integer operations do not wrap
- 5.2 INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data
- 5.3 INT32-C. Ensure that operations on signed integers do not result in overflow.

The Classic Buffer Overflow

See **bufferoverflow.c**

How can we protect against this?

- Don't use **gets()**
- Consider using **strncmp()** instead of **strcmp()**
- Compile with **-fstack-protector**

Printf exploitations

```
printf("%x %x %s\n", intvar1, intvar2, stringvar)
```

- Copies of the values intvar1 etc are stored on the stack
- Printf() gets called
- Printf() uses C majic to walk the stack
- Interpreting what it finds, according to the format string specifiers

What about

```
printf("%x %x %s\n")
```

%x %x %s will access the first things it finds in the stack

So doing this

```
fgets(buffer, size, f);
...
printf(buffer);
```

Can be dangerous. Like what if buffer contains %s format specifiers **giving user direct access to the stack!**

Demo

See **stacksniff.c**

HTTP

Thursday, 19 May 2022 12:01 PM

Outline

- Admin
- HTTP
- Assignment 4
- IP Addresses
    - Subnets
    - Special addresses
    - Routing

HTTP

HTTP = HyperText Transfer Protocol

- The protocol for shipping web stuff around (but not just web stuff)
- Currently v1.1 (Standardised 1997)
    - HTTP/2 published in 2015
    - HTTP/3 in the works
        - Already supported in common browsers
- Not the same as HTML
- Runs on top of TCP (so it's layer 5)
- Request-Response protocol

Netcat for Fun

Echo -e "GET / HTTP/1.1\n\n" | nc student.eait.uq.edu.au 80

And the server responds with

```
HTTP/1.1 400 Bad Request
Server: nginx/1.18.0
Date: Fri, 13 May 2022 02:26:18 GMT
Content-Type: text/html
Content-Length: 157
Connection: close
X-Frame-Options: SAMEORIGIN
X-Request-Id: 0922102v87dp73tjob9g

<html>
<head><title>400 Bad Request</title></head>
<body>
<center><h1>400 Bad Request</h1></center>
<hr><center>nginx/1.18.0</center>
```

```
<center><h1>400 Bad Request</h1></center>
<hr><center>nginx/1.18.0</center>
</body>
</html>
```

We sent

- GET / HTTP/1.1
  - GET == I want a page (there are other methods such as POST too)
  - / == This is the name of the page (top page level)
  - HTTP/1.1 == the protocol we will be using
- A blank line
- Body containing HTML content telling us it was a bad request (in case we are displaying the result).

We got:

- HTTP/1.1 400 Bad Request
  - HTTP/1.1 == Response is in this protocol
  - 400 = status code
  - Bad Request == Human readable version of the status
- Headers (key: value pairs)
  - Content-Type: text/html == what sort of data is being sent back in the **body** of the response
  - Content-length: 157 == how big is the data
- Blank line (separates header information from the data
- Body containing HTML content telling us it was a bad request (in case we are displaying the result).

Fix Request

Echo -e "GET / HTTP/1.1\nHost: student.eait.uq.edu.au\n\n" | nc student.eait.uq.edu.au 80

- There is one compulsory request header in HTTP1.1 when communicating with a web server (Host).
- A browser would send a lot more information

- May need --no-shutdown argument to nc so that it waits for the server response after EOF on stdin

Response

HTTP/1.1 301 Moved Permanently

Location: https://student.eait.uq.edu.au/

- This is not an error
- It is telling us that we should use https, not http
- Your browser would make a new request automatically

How do we encrypt the message?

Netcat (some versions) support the --ssl argument

Let's try

Echo -e "GET / HTTP/1/1\nHost: student.eait.uq.edu.au\n\n" | nc --ssl student.eait.uq.edu.au 443

Note: port 443 instead of port 80
Does this work? Not all the time

Line Termination

Note: HTTP standard required \r\n (carriage-return and newline (or linefeed character)) at the end of each line but we got away with just \n here

Standard (RFC7230) says:
Although the line terminator for the start-line and header fields is the sequence CRLF, a recipient MAY recognise a single LF as a line terminator and ignore any preceding CR.

Best to use \r\n to be safe.

HTTP Requests and Responses

Request
Request Line (Method Address Protocol-Version)
Request Headers
(Blank-line)
Body of request (optional - needed for PUT and POST requests).

e.g.

GET /path/to/index.HTML HTTP/1/1\r\n
Host: www.domain.com\r\n
\r\n

HTTP Request Method

- **GET** - asks server to return document specified by address
- **HEAD** - asks server to return just the HTTP response headers, not data
- **POST** - for "putting" data to the server (e.g. form submission)
- **PUT**, **DELETE**, **OPTIONS**, **TRACE**, **CONNECT**

Sample HTTP Request Headers

- **Content-Length**: how much data in body part of request - mandatory if body data is being sent.
- **Host**: - virtual host to retrieve data from
- **User-Agent**: - name and version of client (e.g. browser)
- **Cookie**: - cookie(s) for the URL
- **Accept-Language**: - language that client will accept
- **Connection**: - whether or not connection stays open after current transfer
- *Many others*

HTTP Requests and Responses

Response:
Status Line (Protocol version Status-code Explanation of status)
Response Headers

(Blank-line)
Body of response (optional - needed for PUT and POST request)

e.g.

HTTP/1.1 301 Moved Permanently\r\n
Location: https://www.itee.uq.edu.au/newpage.html\r\n
\r\n

Content length header required if there is a body present so server can parse the body text.

Sample HTTP Status Codes

- **100's** - Informational
- **200's** - Client request successful
  - **200** - Ok - URL found, data returned
  - **204** - No Response, Request was OK but no data to return
- **300's** - Redirectional, further action needed
  - **301** - Moved - URL permanently moved
- **400's** - Client request incomplete
  - **401** - Unauthorised - user must produce authorisation
  - **403** - Forbidden - authorisation failed
  - **404** - Not found - address does not exist
  - **418** - I'm a teapot - added as port of the Hyper Text Coffe Pot Control Protocol (1 April 1998)
- **500's** - Server errors

Many more codes out there. These are the most common.

Sample HTTP Response Headers

- **Content-Type**: - MIME type of data (e.g. text/html)
- **Server**: - name and version of server software
- **Last-Modified**: - date document was last changed
- **Set-Cookie**: - cookies to be set (if client permits)
- **Content-Length**: - Number of bytes of data in body of response
- **Location**: - new location for redirection response
- *Many others*

IPv4 Addresses

Friday, 20 May 2022 2:32 PM

IPv4 Addresses

- IPv4 Addresses = 32 bits
- Divided into network and host parts
  - Network part starts with the most significant bit
  - e.g.: moss is 130.102.72.10
  - 130.102 | 72.10
  - 10000010.01100100 | 01001000.00001010
- UQ public addresses look like 130.102.x.x

Subnet Size

- Increasing the number of bits in the network part means a "smaller" network
- e.g. 16 bits for the network part means 16 bits for host addresses = 65536 possible host addresses
- 18 bit network part would allow more networks but each network has "only" 16384 possible host addresses.

Subnets

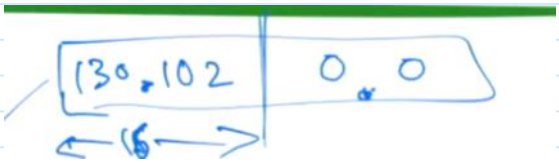
- An organisation's network can be divided into subnets
- A host can directly communicate with everything on the same subnet
- Broadcasts will reach all hosts in the subnet

(For the rest of this discussion, we'll use network and subnet interchangeably)

To communicate, a host needs to know both its IP addresses and which subnetwork it belongs to
Can describe the subnet in two ways

- CIDR notation
- Subnet mask

Method 1 - CIDR



Classless inter-domain routing

e.g. 130.102.0.0 / 16

- Set all host bits to 0
- The value after / is how many bits are in the network part

130.102.12.0 / 24

- Subnet of all addresses starting with 130.102.12.xx

CIDR

/x networks, x does not need to fall on a byte boundary (though it does often make it easier to interpret)
/8 /16 /24

These describe different networks

- 130.102.12.0 / 24 = "roughly" 254 host addresses
- 130.102.12.0 / 23 = "roughly" 510 host addresses

/24 => 130.102.00001100.???? ????
/23 => 130.102.0000110?.???? ????

So 130.102.00001101.0000 0110 == 130.102.13.6
Would belong to 130.102.12.0 / 23
But not 130.102.12.0 / 24

"roughly"?

Each subnet will have two addresses reserved for special uses

- All host bits = zero (minimum host address)
  - Network address (e.g. 192.168.0.0 for a 192.168.x.x network)
- All host bits = one (maximum host address)
  - Broadcast address (e.g. 192.168.255.255 for a 192.168.x.x network

So subnet $A.B.C.D\ /\ x$ has $32\ -x$ host bits and $2^{32-x}-2$ usable host addresses

Here, $/31$ is a special case

Method 2 - Netmask

A netmask = a bit pattern which will map under bitwise AND any IP addresses to the corresponding network address

1. Set all network bits to 1
2. Set all host bits to 0

For example /24
Mas would be 255.255.255.0

130.102.24.17 ->130.102.24.0
130.102.24.250 -> 130.102.24.0
130.102.21.16 -> 130.102.21.0

In general:

- Network address = IP address & netmask

Example

130.102.160.0 / 20 (160 = 128 + 32 = 160 0b10100000)

130.102.10100000.00000000

Network bits are
130.102.1010 0000.00000000

Netmask
255.255.1111 0000.00000000

So the netmask is 255.255.240.0 (in decimal)

Typical Exam Question

Are these IP addresses part of 130.102.160.0 / 20?

- 130.102.163.19
- 130.102.171.99
- 130.102.176.14

Valid netmasks

A valid netmask must have all ones followed by all zeros

i.e.
1111 1111 1100 0000

You can't have something like

1111 1110 1100 1000

Example

Which of the following are valid netmasks?

- 255.255.255.192?
- 255.208.0.0?
- 224.0.0.0?

255.255.255.192 = 11111111.11111111.11111111.11000000
Valid

255.208.0.0 = 11111111.11010000.00000000.00000000
Not valid

224.0.0.0 = 11100000.00000000.00000000.00000000
Valid

Exercise

What is the broadcast address for use by

117.98.141.19, netmask 255.254.0.0

117.98.141.19
255.254.0.0
--->
01110101.01100010.10001101.00010011
11111111.11111110.00000000.00000000
---------- (AND)
01110101.01100010.00000000.00000000
=
117.98.0.0

Broadcast address is 1's in host part

Bitwise OR of network number with 1s in the host part of

Solution

- 130.102.163.19

  Netmask of 130.102.160.0 / 20 is
  255.255.1111 0000.0000 0000

  IP address is
  130 = 130 (left as decimal)
  102 = 102 (left as decimal)
  163 = 128+32+2+1
  19 = 19 (left as decimal)
  We have

  130.102.10100011.19
  255.255.11110000.0
  ---------- (AND)
  130.102.10100000.0
  =
  130.102.160.0 = network address
  Therefore, this IP address is part of the network

- 130.102.171.99

  Netmask
  255.255.1111 0000.0000 0000

  IP address is
  130 = 130
  102 = 102
  171 = 128 + 32 + 8 + 2 + 1
  99 = 99
  We have

  130.102.10101011.99
  255.255.11110000.0
  ---------- (AND)
  130.102.10100000.0
  =
  130.102.160.0 = network address
  Therefore, this IP address is part of the network

- 130.102.176.14

  Netmask
  255.255.1111 0000.0000 0000

  IP address is
  130 = 130
  102 = 102
  176 = 128 + 32 + 16
  14 = 14
  We have

Broadcast address is 1's in host part

Bitwise OR of network number with 1s in the host part of the address

117.01100010. 0. 0
0. 00000001.255.255
---------- (OR)
117.01100011.255.255
117.99.255.255 is the broadcast address

14 = 14
We have

130.102.10110000.14
255.255.11110000.0
---------- (AND)
130.102.10110000.0
=
130.102.176.0 != network address
Therefore, this IP address is not part of the network

Exercise 2

Give the CIDR form and netmask for the largest network which

- Includes
  - 100.89.19.80
  - 100.89.19.82
- But does not include
  - 100.89.19.97

80 = 64 + 16
82 = 64 + 16 + 2
97 = 64 + 32 + 1

100.89.19.01010000
100.89.19.01010010

100.89.19.01100001

Different network address for the last one

100.89.19.01000000 / 27 (8 + 8 + 8 + 3 = 27 bits)

100.89.19.64 / 27 is the CIDR notation for the network that fits the given conditions

Netmask

255.255.255.11100000
255.255.255.224 is the netmask

Broadcast address (question didn't ask but as an example)
100.89.19.01011111 -> 64+16+8+4+2+1 = 95
100.89.19.95 is the broadcast address

Special Networks

From RFC 6890

Non-routable / "link local" addresses
Address from the following networks should not be used on the public internet

- 10.0.0.0/8
- 172.16.0.0/12
- 192.168.0.0/16

- 169.254.0.0/16
    - For auto config when you can't get a real address (i.e. if your ethernet cable is unplugged)
    - First and last 256 addresses (169.254.0.0/24 and 169.254.255.0/24) are reserved for future use

Loopback

All addresses in 127.0.0.0/8 are "loopback" addresses

- Including but not limited to 127.0.0.1
    - Yes, that is $2^{24} - 2$ addresses.

Contact - IP Address Exercises & DNS

Sunday, 5 June 2022 12:56 PM

See workbook for formulas and examples

Routing

Sunday, 5 June 2022 2:31 PM

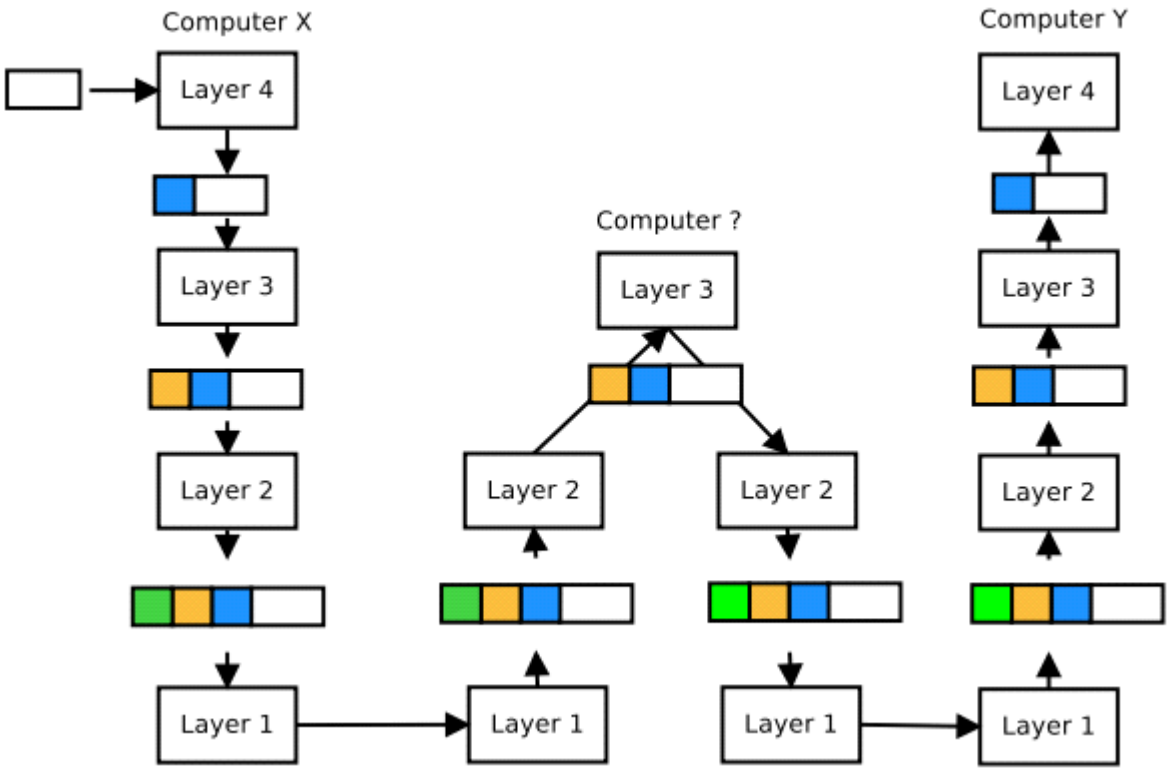
When sending a message, the network layer needs to make a decision:

- Send direct to the destination?
  - Find the MAC of the destination
- Send via another machine?
  - Find the MAC of the intermediary

Your computer does this by performing a bitwise AND with the network mask

Option #1 will only work if the destination is directly reachable at Layer 2

ARP (Address Resolution Protocol) operates at Layer 2 to support finding MAC addresses.



ARP - Address Resolution Protocol

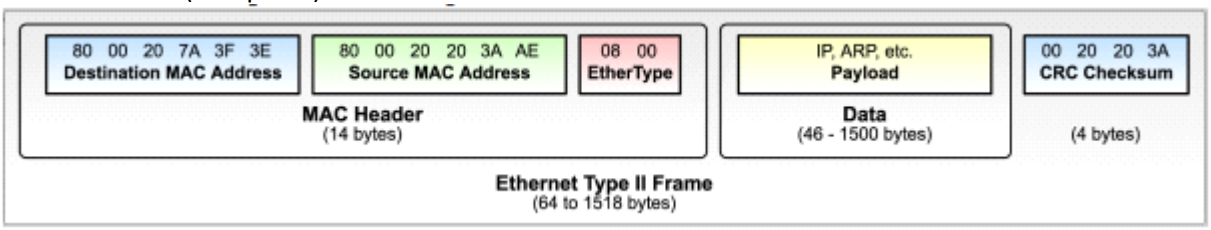
- IP datagrams must be sent via Layer 2 (say ethernet)
- ARP - discovers link layer addresses (MAC addresses) given an IP address
- ARP operation - what is MAC address for IP address x.x.x.x?
  - If MAC address is already known (cached) - return it.
  - Otherwise, send Ethernet broadcast message (to MAC address FF:FF:FF:FF:FF:FF)
    - Who owns IP address x.x.x.x? My IP and MAC address are …
  - Owner responds - my MAC address is yy:yy:yy:yy:yy:yy
  - Answer is cached (on both ends)

- Try running arp on moss (or arp -a on windows)

Ethernet

Ethernet now refers to a family of standards (IEEE 802.3)

Ethernet Frame (Wikipedia)



Network Devices

- **Hubs** - operate at physical layer
  - All entering frames sent to all other ports
- **Bridges** - operates at data-link layer
  - Connects two networks, learns which addresses are on which side and forward or filters all messages
- **Switches** - operates at data-link layer
  - Multi-port bridge
  - Frames transmitted only to device(s) for which frame intended
- **Routers** - operate at network layer
  - Connects networks together
  - Route datagrams based on IP address

Hubs and Bridges are not really used any more. Most of the time, a network can be expanded with a single router and multiple switches.

NAT - Network Address Translation

Sunday, 5 June 2022 5:20 PM

NAT - Overview

Problem: There aren't enough public IP addresses!
Solution: Create private networks behind routers using address spaces like 10.0.0.0/8 or 192.168.0.0/16

Problem:

- Host X = 10.0.20.15 wants to connect to address Y (on the public internet).
  - Address information will be: {src-ip=X, src-port=sp, dest-ip=Y, dest-port=80}
- Packet arrives at Y (maybe)
- Y tries to reply, with:
  - {src-ip=Y, src-port=80, dest-ip=X, dest-port=sp}
- Reply doesn't go anywhere because X is not routable on the public internet.

This is where NAT - Network Address Translation comes in

X --> ... --> R --> ... --> Y
{src-ip=X, src-port=sp, dest-ip=Y, dest-port=80}
Packet arrives at R
R **modifies address information**
{src-ip=R, src-port=np, dest-ip=Y, dest-port=80}
...
T receives packet and replies {src-ip=Y, src-port=80, dest-ip=R, dest-port=np}
...
R **receives packet and modifies info:**
{src-ip=Y, src-port=80, dest-ip=X, dest-port=sp}
X receives the message

X doesn't really know the substitution of address information has really happened. It doesn't really care so long as the information it needs is received.

- The above process only works because R remembers that port np corresponds to port sp on X
- R does not needs to be directly connected to X or Y
  - It needs to be somewhere before the packets with local addresses leak onto the public internet
- NAT is processor and memory intensive
  - All inbound/outbound port address pairs must be stored
  - Every single packet through the router requires a lookup
  - Packet header checksums must be recalculated for every packet
- How does the router know when it can discard a port/address mapping?
- What about servers listening behind NAT? How do external clients connect without a prior outbound packet? Port Forwarding

ICMP - Internet Control Message Protocol

Sunday, 5 June 2022 5:52 PM

Internet Control Message Protocol

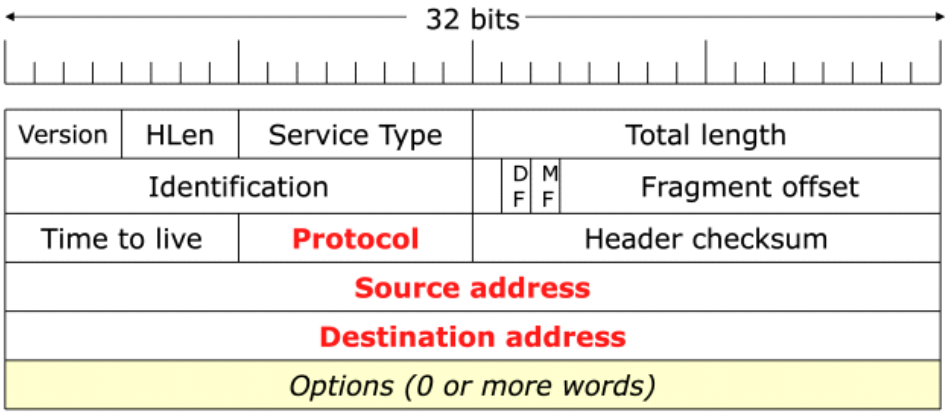
Networking layer protocol

- Networking layer protocol
  - But based on IP (i.e. delivered using IP datagrams - protocol number 1)
- Used for diagnostic and control information about IP datagram delivery
- Errors are returned to source IP address
  - E.g. destination unreachable

Ping

Send a message to device
ICMP echo request
Hopefully it sends a message back
Calculate travel time

IP Datagram Header (from week 9)



- Time to live (TTL) field lets us do interesting things
  - Reduced by 1 each time the packet reaches an interface
  - Packet is dropped if TTL reaches 0
    - Returns ICMP time exceeding message

Traceroute

Traceroute takes advantage of TTL functionality
It sends out a packet with TTL=1 from N0
Packet reaches N1. Packet dropped and returned along with routing time.

This is repeated for N=1 to N=destination node
Can then get a table of routing times to each hop/node on the route

Contact - DNS

Sunday, 5 June 2022 5:57 PM

Internet Phonebook

- IP packets need to use IP addresses
- People (and some systems) would prefer to use names like moss.labs.eait.uq.edu.au
- We need a way to map names -> IP Addresses
    - Source.eait.uq.edu.au --> 130.102.79.227

First version

1. Make a text file with every computer on the internet and their addresses in it
2. Put a copy of the text file on every computer using the internet
3. Send out updates when the internet changes

- It's called a "host's file" and it is likely /etc/hosts on linux
    - Or C:\Windows\System32\drivers\net\etc\hosts on Windows

- It is still used (but not expected to have the whole internet in it)!

DNS - Distributed Phonebook

DNS = Domain Name Service

- Each "domain" will have at least two **nameservers** which know the name --> address mapping for that domain
- Have a collection of root nameservers (computers just need to know some of these)
- The root servers know the information for the nameservers for the TLDs (top level domains)
    - .com
    - .net
    - .au
    - .uk
    - .edu
    - …
- Those servers each know the nameservers for the subdomains (e.g. theforce.net, .com.au, .net.au, …)

Suppose you want to contact source.eait.uq.edu.au

1. A root server knows the NS for .au
2. .au knows .edu.au
3. .edu.au knows uq.edu.au
4. …
5. And so on until
6. Source.eait.uq.edu.au is known

DNS queries

Queries are UDP messages

Servers could operate in:

- Iterative - I don't know but go ask that machine
    - .au NS <-- source.eait.uq.edu.au
    - .au NS --> Go ask .edu.au

- Recursive
    - I'll go and find out for you
    - Local nameservers are probably doing this
- Nslookup and dig are two useful commandline utilities for querying DNS

Notes

- DNS responses have TTL (time-to-live - how long is the data valid for)
    - Name servers can cache answers to reduce load
    - More stable mappings will have longer TTL
- Load balancing
    - Different requests for the same name could get different answers
    - Give answers which are close to the query source (e.g. CDNs - Content distribution networks)
- Change machine without changing contact details
- DNS domains are independent of networks
    - Nothing requires www.uq.edu.au to have an IP address in the 130.102.0.0/16 network

To get your own domain you need to convince the super domain to add you. You will need to pay for this and price can vary depending upon the domain.

Abstraction of Memory

Monday, 23 May 2022 2:03 PM

We have assumed that there is a way for processes to have their own view of memory

We'd like

- Protection
  - Other processes should not be able to interfere with another process
- Sharing
  - For inter-process communication (IPC)
  - Avoiding redundancy, e.g.
    - Unmodified parts of fork() ed children
    - Every program loads/links **libc.so** - how many copies?
      - Libc.so is the library that is required when a C program is run on Linux. ~ 10MB. Wasteful to load this for each process. Linux uses optimisations to only load this once.
- Optimisation
  - Demand paging allows us to load things only as they are required.
  - Also known as swap memory
  - This is how malloc(1000000000) can work even though there isn't this much RAM on the server

- Varying overall allocation
  - Don't want to fix memory allocation at start of runtime
  - Want to be flexible as to how much mem is needed at different times
- Exceeding physical memory
  - Use secondary storage to store "idle" memory

Virtual Memory

Monday, 23 May 2022 2:10 PM

Let's have a look at **/proc/pid/maps**

- Smallest address?
- Largest address?
- What's the range?
  - Much larger than the amount of physical memory installed on Moss
- Does this machine have this much memory?
  - No
- What's going on
  - OS is doing some trickery here

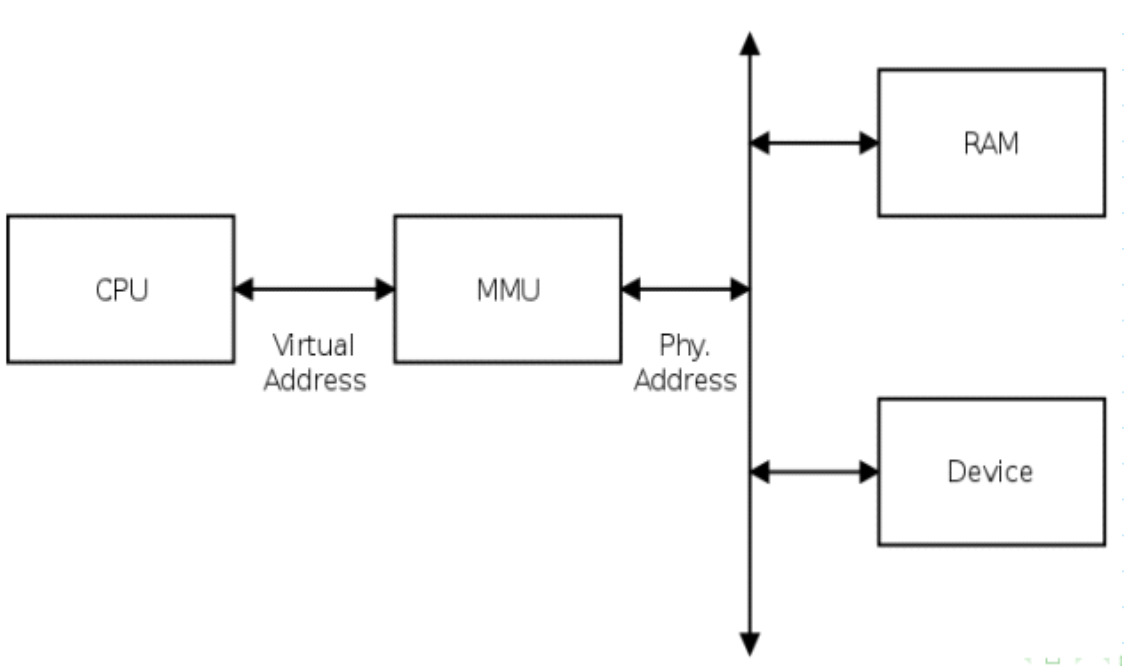
Two Types of Addresses

- **Virtual Addresses** - Used by the CPU when running user processes, e.g.
  - Pointers in your code
  - Instruction fetches
  - Also called **logical addresses**
- **Physical Addresses** - Locations in physical RAM

Hardware support to allow dynamic translation between them without the program needing to be aware of it.

Ideally, the kernel doesn't have to get involved very often either.

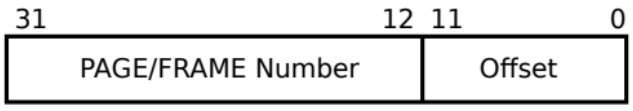
Memory Management Unit (MMU) within the CPU



Page Table

- The virtual address space is divided into equal sized pieces called "pages"
- The physical address space is divided into "frames"
- Frames and pages are the same size

- Page/frames sizes being a power of two means addresses can be easily split, e.g.
  4k pages in a 32-bit address space
  (4096 = 2^{12})



A **page table** is a map from (virtual) pages to (physical) frames

- Kernel maintains a page table for each process
- There does not need to be any relationship between virtual and physical layout

E.g.

Virtual	Physical
...	
20	120
21	122
22	95
23	94

Virtual	Physical
...	
20	121
21	123
22	250
23	251

Note: contiguous virtual addresses doesn't always require contiguous physical pages.
Addresses within pages/frames are always contiguous

Address Translation

Split a virtual address into a page number and an offset.

Hypothetical example
(page size = 4096 = 2^{12})

Virt Addr.	Page	Offset
0x00000000	0x00000	0x000
0x00000001	0x00000	0x001
0x00C00234	0x00C00	0x234
0xFFFFF64B	0xFFFFF	0x64B
0x81430FFF	0x81430	0xFFF

- Page # = VA / pageSize
- Offset = VA mod pageSize

Page# | offset → frame# | offset

i.e.

Page# | offset → PT(page#) | offset

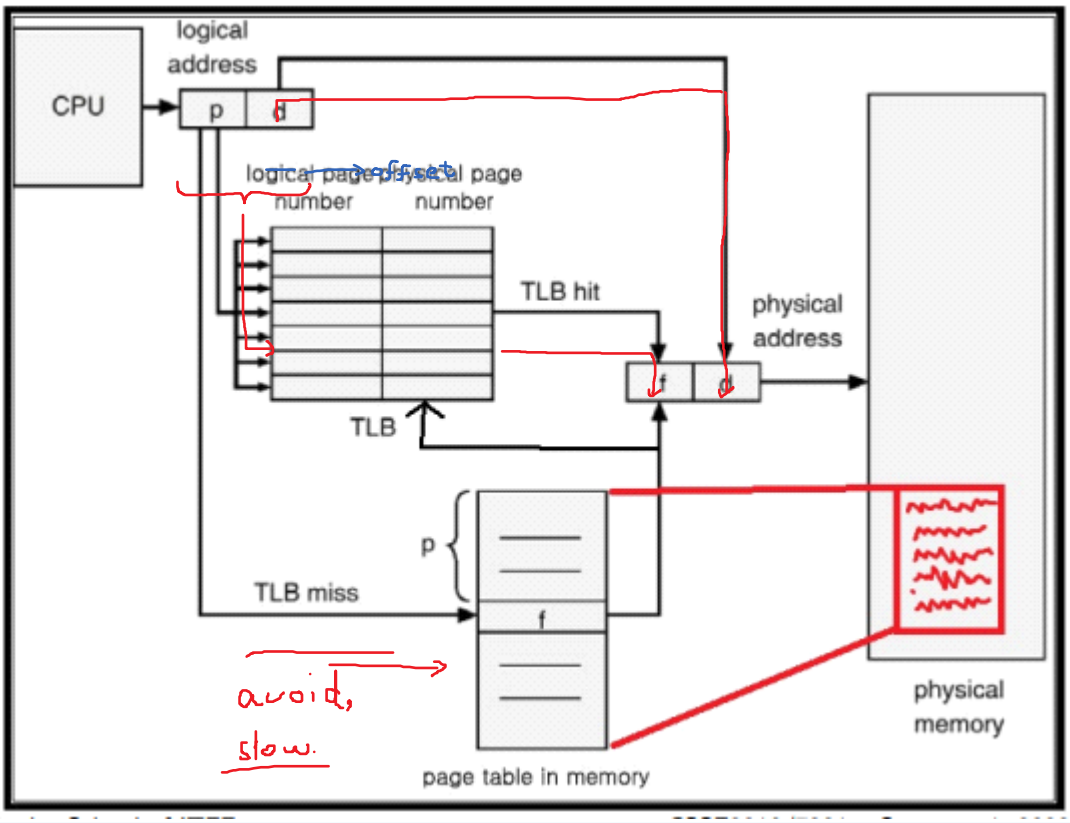
Implementation

Page tables are stored in memory

- Suppose 32-bit virtual addresses with 4kiB page size.
- A single process' page table would need: $\frac{2^{32}}{2^{12}} = 2^{20}$ page size table entries
    - To map the entire address space
- For a 4 Byte PTE (page table entry), this potentially means a 4MiB table, per process
- In reality
    - The kernel only maps the pages that are required, as they are required
    - Clever data structures are used to minimise overhead

TLB

- Needing to lookup a frame in the page table means that each memory access from a program would need two memory accesses
    - One into the page table
    - One for actual memory access
- Reduce the burden using a **Translation Lookaside Buffer (TLB)**
    - Hardware cache for (page → frame) mappings
    - Associative (content addressable) memory
    - Fast
    - Hardware only goes to actual memory-based table if there is a TLB "miss"
        - Hardware dependent
        - Some CPU architectures can "walk" memory-based page tables, others can't so the kernel does it for the CPU



Page Faults

When there is no frame corresponding to a given page, a "page fault" occurs

Causes

- The page is legal for that process
  - It hasn't been loaded yet
  - It has been *"paged"* out to disk (sometimes called *swapping*)
    The kernel needs to suspend the process until it can get that page back into RAM and find a frame to put it in
    If we are under memory pressure, this usually means somebody else's pages get swapped out → "thrashing"
- The page or page access is not legal for that process, e.g.
  - Followed the null or some other bogus pointer
  - Writing to a read-only page
  - Instruction fetch from a non-executable page

Kernel needs (probably) to inform the process (and possibly kill it)

Null Pointers?

- Core idea: legal vs illegal decisions are made at the level of **pages** not individual addresses.
- If page 0 (and possibly other very low numbered pages) are always marked as invalid, then hardware makes sure the null pointer will cause a segmentation fault

Page Replacement Algorithms?

Beyond the scope of this course - do COMP3301

Separate Address Spaces

- This comes as a consequence of each process having its own page table
- Shared pages happen when the kernel maps the same frame into multiple page tables
- Memory protection:
  - Pages are only valid if the kernel maps them to a frame
  - A process can not construct a pointer to another process' frames
    *((char *)(rand()) = 0x0; // ;)
  - Processes can ask the kernel to do it… shared memory, mmap() and friends

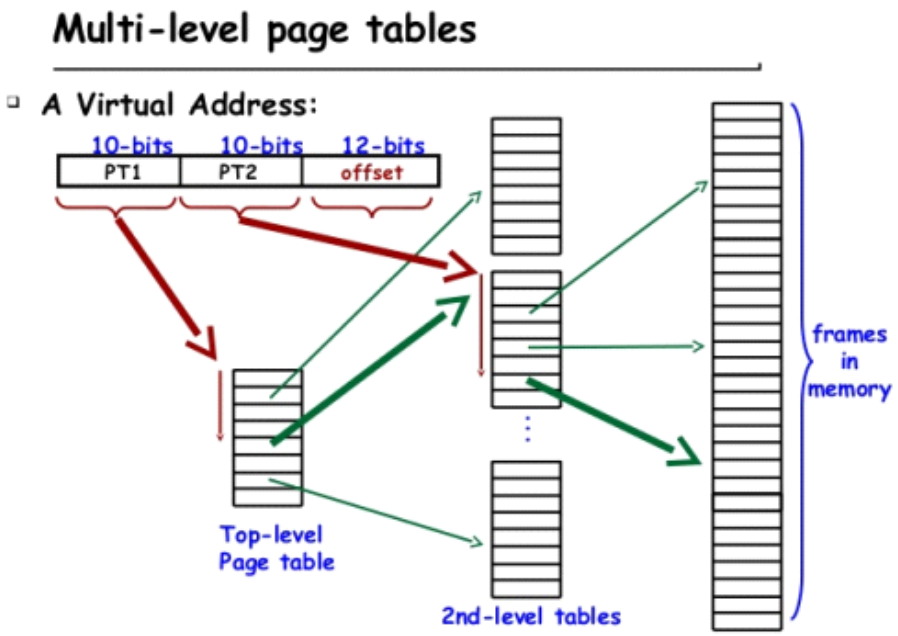
Multi-level Tables

Remember how we said a 4kiB pages in a 32-bit address space required 4 MiB per process? We can be a lot smarter than that.

- Split the page numbers into sub fields
- Entries in the top level of the table can be "empty". That is, there are no valid pages in the range for that top-level entry.

So the table doesn't need to explicitly store every entry. This saves a lot of memory.

Gaps



This concept can be extended to an arbitrary number of levels, e.g.

- 3 level page table
- 48 bit virtual addresses (Vas), 8 byte page table entries (PTE)
- 4KiB pages/frames (12 bits of offset)

Working backwards

- Each page (4KiB) can hold 512 = 2^9 PTEs
  - Each PTE field in the VA is 9 bits
- VA structure will be
  -

Lvl1	Lvl2	Lvl3	Lvl4	offset
9 bits	9 bits	9 bits	9 bits	12 bits

Note that there are relationships among field sizes and the other parameters of the system:

- PTEs must be large enough to uniquely identify a page or frame:
  - At least (PA size - offset size) bits wide
  - Additional bits in PTE are often used for storing flag such as permissions (RW, RO, EXEC, etc) among other things
  - In CSSE2310, **we will specify PTE sizes in bytes so you don't need to work this out**.
- Page size and PTE size define the PTE field size, e.g.
  - 4KiB pages and 4 bytes PTEs
    - 1024 PTEs per page
    - 10 bits per PTE field in the virtual address
  - Each PTE table is exactly one page in size (first level table can be an exception, see below)

Example

A trickier example

- 4 level page table
- 64-bit virtual addresses (VA), 8 byte page table entries (PTE)
- 8 KiB pages/frames (13 bits)

Working backwards

- Each page (8 KiB) can hold 1024 = 2^{10} PTEs
  - Each PTE field in the VA is 10 bits
- VA structure will be
  -

??	Lvl1	Lvl2	Lvl3	Lvl4	offset
11 bits	10 bits	10 bits	10 bits	10 bits	13 bits

That's only 53 "useful" bits - what about the 11 most significant bits?

- This is architecture dependent. We could
  - Expand the first level table to capture all of the remaining bits (like a single level page table) e.g.
  -

Lvl1	Lvl2	Lvl3	Lvl4	offset
22 bits	10 bits	10 bits	10 bits	13 bits

  - Ignore these bits
  - Force them to zero or some other value
- For CSSE2310, you can assume that the first level table is sized to account for all of these bits, unless we specify otherwise

User Space Memory Management

Sunday, 5 June 2022 6:06 PM

Memory Layout

Top of memory
- Kernel memory
- Bottom of stack
- Top of stack
- ...
- ...
- Top of heap
- Bottom of heap
- Other data
- Text "segment"
- Forbidden

0
- Forbidden

Memory mapped content goes somewhere between the heap and the stack

Lots of room in 64-bit?

It's possibly quite crowded in 32bit address space.

In a 64-bit address space, there is more room. The precise location of heap and stack could vary if ASLR is enabled

ASLR = Address Space Layout Randomisation

Kernel-Heap Interaction

The kernel only cares about where the top of the heap is. If the heap neds more space, then a system call (sbrk()) will allocate more valid pages to the process.

Malloc is a userspace function (which will ask the kernel for more pages if needed).

Some fun with /proc/pid/mem

/proc/pid/mem is a virtual file representing a processes' entier virtual memory space.

/proc/pid/mem always maps to the current process.

```
$ ls -al /proc/self/mem
-rw------- 1 uqjwill1 uqjwill1 0 May 23 11:20 /proc/self/mem
```

This file is writeable!

See target.c and writeat.c!

File Systems

Sunday, 5 June 2022 12:56 PM

Terms in File Systems

- File - bytes recorded in secondary storage
    - For this lecture, we are not talking about other uses of the file descriptor interface
    - e.g., not, network socket, pipe, keyboard, …
- Disk - Spinning magnetic storage
    - Most discussion also applies to SSD/flash storage (we aren't going down to the storage level - see COMP3301)

File System

A file system is a data structure which manages:

- Contents of files - data
- Information about files - meta-data
- Free space
    - File systems have a size
    - Files can be added
    - Files can change in size

Data structures need to exist somewhere

- Usually on a disk
- As a file on another system (.iso file)
- Mac .dmg files

FS vs OS

How programs can manipulate files may depend more on the file system than the operating system.

But, if

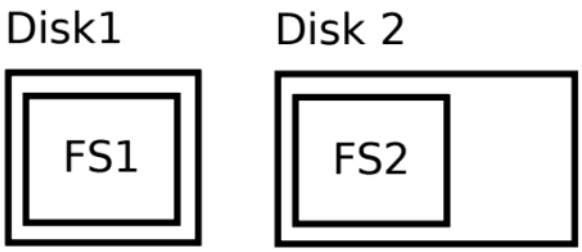
- The OS only provides one main FS
- That FS is hard to use on other OS

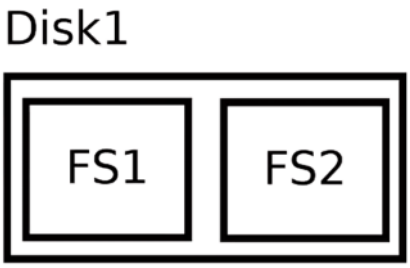
Then the difference might not be obvious
e.g.

- NTFS - NT file system
- HFS+ - OSX
- ISO - cdrom
- Ext2, ext3, ext4, btrfs, sfx, … - unix

Files vs Disks

Disks and File Systems are not 1-1





Why?

- Dual boot
- File-system limitations
- Separate data from OS
    - Some OS make this easier than others
- Capacity increase
- RAID and other capacity + safety schemes

Files

- Sequence of bytes (start at the beginning and read until the end)
- Storage for those bytes?

Variants

Lots of variety

- Sequence of bytes
    - Record based systems
- Single Sequence
    - Old mac - resource fork vs data fork
    - NTFS - multiple streams
- Store bytes
    - Sparse files? (see **bigfile.c**)
    - Union and overlay file systems

Metadata

Files are more than their data contents
Metadata is information **about** data which is not part of it.

- Name - does the content change if the name does?
    - Is name unique?
    - Case sensitive?
    - Case preserving?

- Location/path
  - A derived property from a sequence of dirs?
  - What meaning can be inferred from structure?
  - Is it unique?
- Size
- Type of content?
  - Infer from name?
    - Windows stage 1
    - Some linix GUIs but check contents
  - Encode type with file?
    - Old mac
  - Guess from contents
    - File using /etc/magic

Permissions
- By role
  - Unix systems - files have one "owner" and one "group"
  - Rrxr-x---
  - Owner can
  - Group can
  - Everyone else can
- Access Control Lists (ACL) - by user
  - Fine-grained access control - grant a specific user specific permissions, outside group/owner/other structure
  - Windows, unix also support it as an option

Permissions Reminder

- Change permissions on the command line using chmod
  - u+r: change the owner (user) permissions to add read
  - g+r: change the group permissions to add read
  - o+r: change the other permissions to add read
- X permissions on (normal) files
  - Needed to exec
  - For interpreted scripts (e.g. shell scripts) also need r
- Directory permissions
  - x: needed to interact with anything in the directory
  - r: needed to see what is in the directory
  - X with no r, can access things in the directory if you already know what they are called
- To follow a path, you need x on all directories in the path

Spinning Disks

Why are hdds slow
Consider polar coordinates
- Changing r - moving the head towards or away from the centre
- Changing θ - waiting for disk to rotate (rotational latency)

The more widely scattered operations are, the most cost incurred
Why spinning disks? Still need them for
- Lower cost
- Large capacity

Fragmentation

Files

- Reading through files leads to jumping around the device

Space

- External fragmentation
  - Free space is spread out over the drive
  - Could lead to fragmented files in the future
- Internal fragmentation
  - Unused space inside allocated blocks

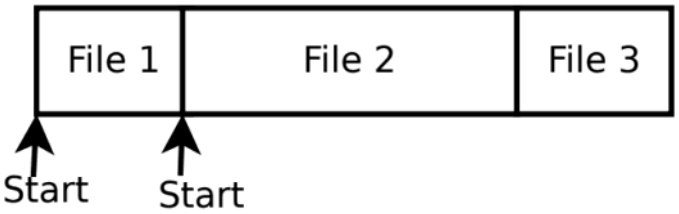
Solid State Drives (SSDs)

Features

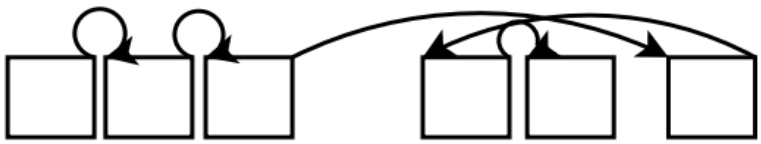
- Large arrays of non-volatile (NAND flash) memory
- Re-use legacy physical and logic interfaces
- Wear-levelling

Storage structures

We're dealing with all of these in abstract
Array-like



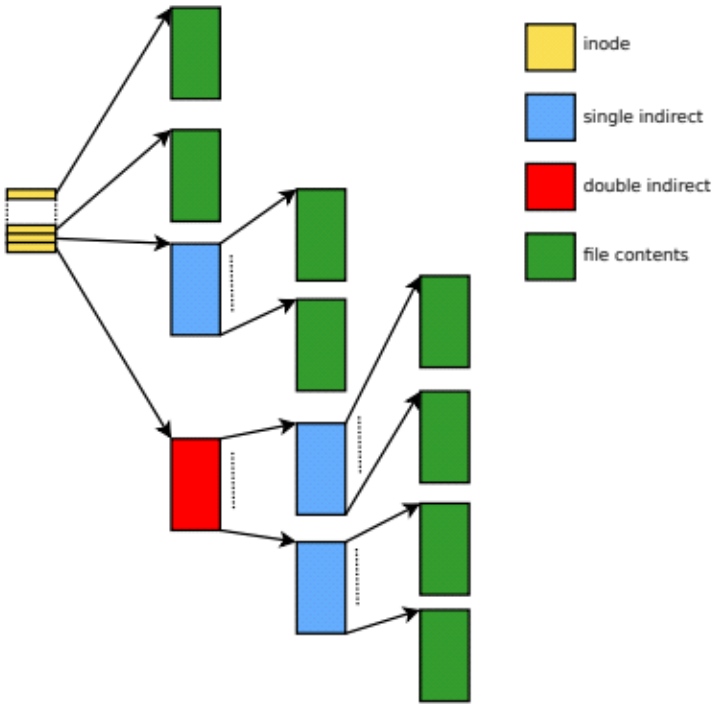
Linked storage



Kind of like a linked list in memory.

Indexed Storage

Most modern Unix-like file systems use this kind of structure. This is a good balance between performance and being able to store both small and large files effectively.



Directory Trees

"Typically" the directories on a file system form a tree.

- Avoid cycles means recursive traversals will eventually terminate
- Removing a subdirectory shouldn't remove the directory you are in because something is its own grandparent.

Directories are accessed through a different set of system calls (not open/read/write/close)

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

DIR* opendir(const char* pathname);
struct dirent* readdir(DIR* dp);
void rewinddir(DIR* dp);
int closedir(DIR* dp);
```

Hard Links

Consider a file **A.c**.

```
$ ln A.c B.c
```

Adds B.c to the directory

```
$ diff A.c B.c
```

Shows no difference,

But after modifying A.c, diff still shows no difference

- (same inode used)
- Link count increases by 1

Ln

- In the filesystems we use, directrory entries are (hard links) i.e. and name:-i-number mapping
- All the explicit properties of a file (apart from a name) are stored in an "i-node" (see indexed data structure above)
- The i-number lets the system find the i-node
    - i-nodes could be in a table
    - Or i-number could indicate where on the disk the node is
- The internals of i-nodes can vary with File System
    - For our purposes we'll assume a structure like that shown earlier (diagram)

- Ls -I will show the i-numbers for each directory entry
- Ls -l will show many many links there are to a file
- All hard links are equal, as long as there is at least one link to a file, it will remain on disk
    - The system call to get rid of a directory entry is unlink
    - Files will be kept with a link count of zero if a process has them open.
- Hard links can't cross into other filesystems

Directories

The link count for a directory is 2 plus the number of direct subdirectories it has. 2+ because one is the name of the directory and the other is the . Within that subdirectory. Add in any sub-subdirectories and you have your link number.

E.g. /tmp/bob will have one link in /tmp pointing "down"

For the other $ ls -a /tmp/bob shows directories

- . <- second link - points to itself
- .. <- points to parent

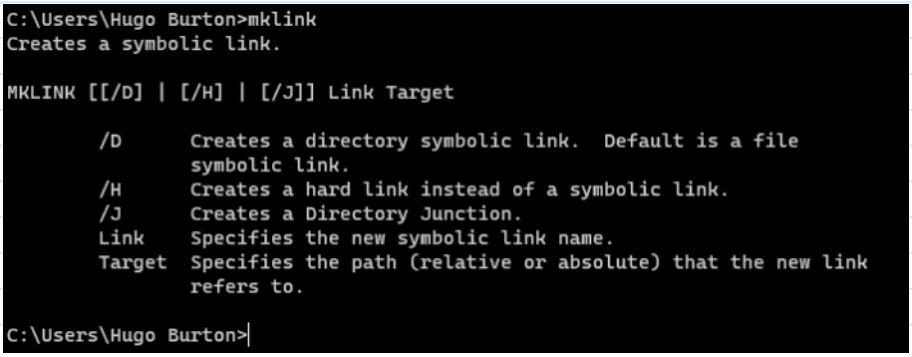
If we mkdir /tmp/bob/sub,

Then there will be another link added from /tmp/bob/..

Hard linking directories is not allowed.

Symbolic Links

- Symlinks point from one name to another (as opposed to name to file contents as in hard links).
- You've seen these used in the testing framework (highlighted cyan on moss)
    - Testfiles/ is a symbolic link to /local/courses/csse2310/resources/tests/…
- Create with ln -s target newname
    - Or on windows:

-


- Can cross file systems (uses paths and not low level ids)
- Have 'l' in the type field
- Permission of '**rwxrwxrwx**'
  - The actual permissions are just those of the target
  - Full permissions are set and ANDed with the target permissions
- A **symlink** will not prevent a target being deleted
- If the target moves or is deleted, the **symlink** won't work
- **Symlinks** can target directories

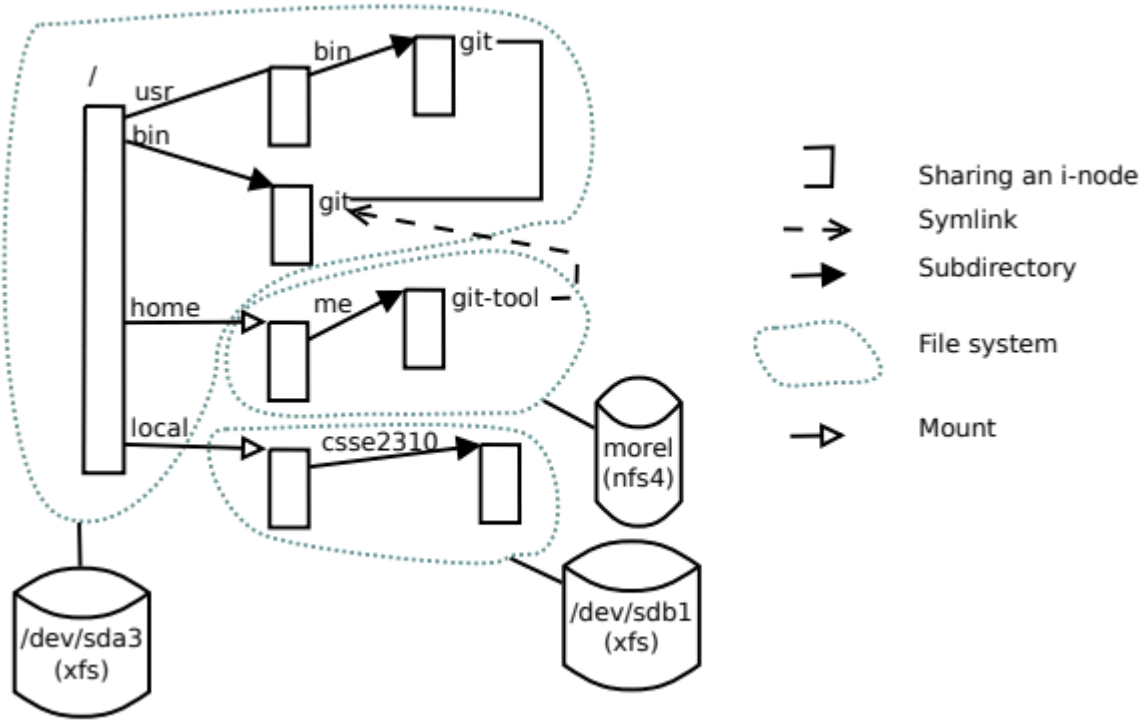
Mounting

- To allow the system to interact with the contents of the file system, the FS must be "mounted"
- Normally, file systems should be 'unmounted' before being removed
  - Unmount, eject, safely remove
  - Why
    - The File System may be auditing info or performing other background tasks
    - Buffering may mean changes haven't yet been written
    - Fflush means it is out of your process
    - Sync "should" mean written to disk (not always)
    - Windows will detect USBs and other removable drives and prevent caching. On Windows, there's really no need to eject. However, on MacOS, eject is important
- Unix **mount** command will list mounted file systems

Mount Points

- Windows
  - Forms a forest of "trees"
  - C:\, D:\, E:\, …
  - UNC paths
- Unix
  - All the directories of all mounted FSs form a single unified tree rooted at '/'
  - Can mount into any directory, chosen by root
  - Temporary mounts e.g. /media
- OSX
  - See unix
  - Tends to be under /Volumes

Moss Mounts



This has changed in the years since this was created though.

Summary

```
ls -ali
 1441875   drwxr-xr-x   30   joel   grp   4096    Jun 12 2018    .
 1441800   drwxr-xr-x   40   joel   grp   4096    Oct 15 18:31   ..
 1441196   drwxr-xr-x   2    joel   grp   4096    Jun 12 2018    bin
 1446457   drwxr-xr-x   5    joel   grp   4096    Jun 12 2018    build
 1446461   -rw-r--r--   1    joel   grp   13364   Jun 12 2018    config.log
 1446433   -rw-r--r--   1    joel   grp   203     Jun 12 2018    CREDITS
 1444671   drwxr-xr-x   8    joel   grp   4096    Jun 12 2018    cusplibrary
 ...
 A         BC           D    E      F     G       H              I
```

A	i-number	F	File group
B	Type	G	Size
C	Permissions	H	Modification time
D	Link count	I	Name
E	File owner		

Contact - File System Calculations

Sunday, 5 June 2022 12:57 PM

Unix Indexed File System

- Block pointers: 4 Bytes
- Block size: 2 KiB
- i-node has
  - 8 direct pointers
  - 1 single indirect pointer
  - 1 double indirect pointer

Questions on the exam

1. What is the maximum possible file size
2. Which block number of a file would require use of double indirect?
3. How much capacity would be added if one of the direct pointers was replaced with a single indirect?
4. How many blocks would need to be read to access block #502

Question 1

Need to work out how many pointers per block

$$\begin{aligned} \text{p\_per\_block} &= \frac{\text{blocksize}}{\text{pointersize}} \\ &= \frac{2048}{4} \\ &= 512 \end{aligned}$$

Then work out total blocks

$$\begin{aligned} \text{Totalblocks} &= 8 + \text{p\_per\_block} + \text{p\_per\_block}^2 \\ &= 8 + 512 + 512^2 \\ &= 262664 \end{aligned}$$

And finally, multiply be the block size to get the max size a file can be

$$\begin{aligned} \text{maxsize} &= \text{blocksize} * 262664 \\ &= 525328 KiB \end{aligned}$$

Question 2

- We need to use all of the previous pointers
- Single indirect block stores 512 pointers
- There are 8 direct pointers.
- So once we use 520 blocks, the next one will need to use the double indirect
- Since blocks are numbered from zero, block #520 will require the double indirect

Question 3

Replace a direct with a single indirect?

- We lose one direct pointer (1 block)
- We gain 512 block pointers via the indirect
- Net increase of 511 blocks
- Unless explicitly told otherwise, do not give answers in blocks
- Increase in capacity = 511 * 2 KiB = 1022 KiB

Question 4

We need to work out which block numbers sit in the boundaries

- #8 is the first block using single indirect
- #520 is the first block to use double indirect
- #502 is between them so it is stored somewhere under single indirect
- To read #502, we would need to read the indirect block and then #502.
- So a total of 2 blocks

More rigorous examples and formulas in notes.