This is a compilation of all of the lecture notes given to us. The exam review is FIRST. I tried to remove any recaps or project discussion, as well as highlight any important topics/functions.

Review of topics

- data types

```
- determining size of data: sizeof()
    sizeof(int) sizeof(struct addrinfo)
    sizeof(int *)  sizeof(struct addrinfo *)

    sizeof(variable)  <- don't be confused!

    int array[20];
    sizeof(array) == 20 * sizeof(int)

    int *array = malloc(20 * sizeof(int));
    sizeof(array) == sizeof(int *)

- strings vs char arrays
    strings always end with a terminator ('\0')
    character arrays may have no terminator or many terminators
    the difference is how they are used

        when we call read() we get raw data; not necessarily a string

    strcpy(), strcmp(), strdup(), strlen(), strcat()
    strncpy(), strncmp(), strndup(), strnlen(), strncat()
        - "n" variants take a maximum length
        - useful if we can't guarantee the presence of a terminator
        - useful if we aren't sure destination of strcpy() is big enough

- creating types: struct, union, enum, typedef
    struct bundles together multiple values
    union gives us a choice between types - hard to use correctly
    enum - fancy integer constants
        enum cardinal_direction { north, south, east, west };
    typedef - abbreviate type names
        typedef void *(*thread_fun)(void *);

        now we can write
            int pthread_create(..., thread_fun, ...);
```

- pointers

```
- size of object being pointed to
- assigning to pointer variable vs copying data
    int *p;
    p = q;  // makes p point to the same int as q
    *p = *q;  // copies the value q points to into the place p points to

    char *msg = "Hello";
```

```
msg = "Goodbye";  // changes which string literal msg points to
strcpy(msg, "Goodbye"); // segmentation violation!
   // msg points to a string in the data segment (read-only memory!)

char buf[] = "Empty";  // buf names an array of 6 chars
buf = "Full";   // not allowed!
strcpy(buf, "Full");  // copies characters to buf

char *nothing;
strcpy(nothing, "Something");  // wrong! nothing does not point to anything

char *something = malloc(10);
strcpy(something, "Something");  // ok, because we allocated enough space

char *somethingelse = malloc(10);
somethingelse = "Something";   // memory leak!
   // we no longer have a reference to the 10-byte object from malloc()
   // therefore, we have no way to free() it
```

- dereferencing (*)

left side: *p = x; // writes to the object p points to
right side x = *p; // writes the value that p points to

- address-of (&)

&variable - address of a variable
&array[index] - address of specific item in array
array[0] - value of first element
array - address of array / address of first element
&array - address of array / address of first element
&array[0] - address of first element

p[n] == *(p + n)
- prefer array index notation to pointer arithmetic

&struct_ptr->field - address of field in structure pointed to by struct_ptr

- arithmetic: incrementing/decrementing pointers

some_type *p = malloc(sizeof(some_type) * n);

p - address of first item / object that p points to
p + 1 - address of value after the one that p points to

*p - value of object that p points to / first item in array
*(p + 1) - value of object after the one that p points to
- same as p[1]
- also the same as 1[p], if you like confusing people

- contrast with array variables

array variables are always static or on the stack
pointer can point to anything

array variables cannot be reassigned

pointers may be freely reassigned

compiler knows how big the array for an array variable is
only you know how big an array a pointer points to

- **void ***
  point to anything; effectively the same as char *
  C automatically casts pointers to and from void *
  we can't meaningfully dereference them
    - we cast to the right kind of pointer and then use that
    - compiler assumes that what we are doing is correct
    - no guaranteed behavior if we accidentally cast to the wrong type

- **variables**
  - scope/visibility
    - associated with a particular block or the top-level
    - visible after being declared
    - variables in an inner scope "shadow" variables with the same name in outer scopes

```
int foo; // global

void fun(void)
{
   int foo;  // local; shadows global foo (no way to refer to global foo)

   foo = 5;  // refers to local foo, not global foo
}

void fun2(void)
{
   foo = 5;  // refers to global foo
}
```

  - **binding with object**
    top-level (global) variables bind to static (process-lifetime) objects
    local variables are usually stack (function-lifetime) objects
      static local variables have process lifetime but local scope

  - **initialization vs assignment**

```
int a[4] = {1,2,3,4};  // array initializer
a = {1,2,3,4};  // syntax error (cannot use initializer syntax for assignment)

int *p = "Foo";  // pointer initializer (p points to string literal)
p = "Foo";   // pointer assignment (p points to string literal)
*p = "Foo";  // type error (*p expects char, but "Foo" is char *)
```

- **objects: anything stored in memory**
  - lifetime: static, function, arbitrary

static - stored in data segment; exists while process runs
function - stored in stack; created when function starts, destroyed when function exits
arbitrary - stored in heap; created by malloc(), destroyed by free()
- location: data segment, stack, heap
- direct reference (variable name) vs indirect reference (pointer)
- malloc(), calloc(), realloc(), free()
  - avoiding leaks
  - detecting errors

```
p = realloc(p, new_size); // risky! if realloc() fails, we lose our pointer
```

- memcpy(), memmove(), memset()

- file IO
- buffered operations (FILE *, fopen(), fprintf(), etc.)
- non-buffered operations (int, open(), write(), etc.)
- modes: read, write, append, read/write
- file pointer/cursor: keeps track of where we last read/wrote
  - control with lseek(), fseek()
- use of file IO for non-files: pipe(), socket(), accept()
  - some operations restricted to specific kinds of file
- opendir(), readdir(), closedir(), struct dirent
- stat(), fstat()
- dup(), dup2() - additional file descriptor for existing open file

- preprocessing, compilation, and linking
- what goes in a header file
- #include vs linking
- #define macros

- processes
- process ID
- fork() - create a child process
- wait() - wait for a child process to terminate
- zombie process, orphan process, zombie-orphan process
- execl(), execv() - change the program a process is executing
- many things shared between parent and child process
  - what happens if we fork() when a file is open?
- many process attributes are preserved when we exec()

- signals
- setting a signal handler
  - signal(), sigaction()
- blocking signals

- threads
- pthread_create(), pthread_join(), pthread_detach()
  - if nothing joins or detaches a thread, it becomes a zombie when it stops
- coordination
  - mutex, condition variable, barrier, semaphore
- deadlock: 4 necessary conditions

- mutual exclusion
    - hold and wait
    - no preemption
    - circular wait

- **file system**
  - inodes
    - use of indirection to allow:
      1. constant size of inode
      2. minimal wasted space for small files
      3. possibility to represent very large files
  - directories (association of names to inodes)
  - paths
    - paths-inode relation is many-to-one!
  - file permissions/modes

- **The command line**
  - running programs: bare name vs path
    $ program_name
      use search path to find program
    $ path/to/program_name
    $ ./program_in_current_dir
      specifies program; no need to search

  - **file globs**
  - controlling stdin/stdout/stderr: pipes, redirecting
    cat file | ./ww 20
      output from first program sent as input to next program
    ./program < input
    ./program > output

  - **utility programs**
    - man
    - cat, more, less, head, tail
    - file, wc
    - cmp, diff
    - grep
    - ps, top, jobs
    - tee
    - echo

  - **file management utility programs/commands**
    - ls, cp, mv, rm
    - cd
    - mkdir, rmdir
    - chmod, chown

- **networking**
  - 7-layer OSI model
  - 4-layer internet model
  - addresses at various layers

link layer: identify specific machine on local network (e.g., MAC)
network layer: identify machine globally (IP address)
transport layer: identify process on machine (IP address + port)
application layer: identify things relevant to application (e-mail address, URL)
- not all devices interact with the entire stack
    - switches and hubs live at link level; invisible to network and above
    - routers live at network layer; mostly invisible to transport and above
    - user programs live in application layer
      (sockets talk to transport layer but do not expose details)

- sockets
- listening vs connection sockets
    listening - can use accept()
    connection - can use read(), write()
- socket(), bind(), listen(), accept()
- socket(), connect()
- domain name vs IP address; getaddrinfo(), getnameinfo()
- what can go wrong?

Types

e.g., int, float, char

C provides several "primitive" types

integer-like types: int, char, long int, short int, unsigned int,
    unsigned long int, etc.

    char is just an integer that takes a single byte
    differences between integer types:
        sizes: char < short < int < long
            on the iLab, short is 2 bytes, int is 4, long is 8
        "signedness": signed integers have negative values, unsigned do not

floating-point types: float, double

pointers <- more on these

Data in C

all data in C is either an integer, a floating-point rational, a pointer,
or a bundle of these

    what about Bool? We don't have them in C; we just use integers
        0 is false, everything else is true (use 1 by default)

    what about char? these are just (one-byte) integers
        'A' <- character literal; behaves the same as 65

what about strings? there is no string type in C; we use arrays of chars

we have many sizes of integer

char (1 byte)
short int (at least 2 bytes, 2 on the iLab)
int (at least as big as short but not bigger than long; 4 on iLab)
long int (at least 4 bytes, 8 on the iLab)

"short" or "long" by themselves mean short int and long int

we have two kinds of integer: signed and unsigned
signed is default
unsigned means no negative values

"signed" or "unsigned" by themselves mean signed int and unsigned int

unsigned x = -1; <- will not do what you expect

"unsigned long" means "unsigned long int", etc.

unsigned ints can have larger positive values than signed ints

unsigned x = 1025U; <- the "U" means unsigned
you almost never have to worry about this

we have two sizes of floating-point value

float and double

C does not specify what these are
all modern compilers use IEEE floating point

float is 4 bytes
double is 8 bytes

the only reason to use float is to save space; don't bother in this class
just use double when you need floating-point

1.0 floating point literal
1.0e-8 floating point literal with scientific notation
1 * 10^-8

Declaring variables
declare a variable by giving a type and a name

int i; <- declares a variable named "i" of type "int" we can also initialize a variable when we declare it

int i = 0; <- declares i and sets it to 0

Difference from Java: in C, uninitialized variable has an indeterminate value
if we don't say what is in it, then it contains "garbage"
<- whatever happened to be in that part of memory

it is possible to read from a variable that has never been initialized
<- but you never want to do this

Why does this exist?
Originally, C required you to declare all local variables at the top of the function, and there might be variables that you don't end up using
initializing variables you don't end up using is inefficient (but safer)

C always chooses speed over safety!

It is never wrong to initialize a variable, so you might as well do it

Literal values
how we write specific values in source code
integer literals:
decimal: 1, 0, 1000, -15, etc.
octal (starts with 0): 010, 0127 <- base 8, not base 10
hexadecimal (starts with 0x): 0x123, 0xABCD <- base 16
suffixes: L (for a long int), U (for unsigned)
you usually don't need the suffixes, because the compiler will promote values

situations where you might want to use a suffix:
the value is too big for a default (signed) int
0x12345678 <- 4 byte value (written in hex)
0x123456789 <- too big for an int (requires at least 5 bytes)
0x123456789L <- at least 5 bytes (long int is okay for this on iLab)

you might want to force a promotion

```
int x;
long y = x * 100000000;  // possibly a problem if the product is too big
                         // to store in an int

long y = x * 100000000L; // forces compiler to promote x to a long int
                         // and use long int multiplication

long y = (long) x * 100000000;  // also "casts" x to long int before multiplying
```

these are both pretty rare

floating-point literals
0.123, 123.5
1.23e-18

character literals <- actually integers
'A', 'a', '\n', etc.
these behave the same as integers
'A' - 1

int x = 'A'; <- sets x to 65 (4 bytes)
char x = 65; <- sets x to 'A' (1 byte)

char x = '\0'; <- sets x to '\0'
char x = 0; <- sets x to '\0'

assume char literals are ASCII characters
support for modern text representations is more complicated

string literals <- pointers to constant arrays of chars terminated by '\0'

char *p = "hello"; <- p refers to a (constant) array of six chars
printf("%d\n", n);

%d - format code, tells printf to print an integer in decimal
\n - escape sequence, compiler will replace with newline character

printf("%s\n", p); // prints "hello" followed by a newline

Enums

enums -- defines a set of named integer values that can be used as constants

enum direction {left, right, up, down, forward, back};
<- declares (creates) the type "enum direction"
<- creates constant values left, right, up, etc.
these are just integers
by default, left = 0, right = 1, up = 2, etc.

enum direction input; <- "input" is a variable that stores a direction

input = left; // same as input = 0;

if (input == up) { ... }

```
switch (next_direction) {
    case left:
        ...
        break;

    case right:
        ...
        break;
```

...
}

enum values are not unique (different enums may reuse the same integer represenation)

enum color {red, green, blue};
// by default, red = 0, green = 1, blue = 2

input = red; // nonsense, but technically possible

left == red // true, but you may get a warning from the compiler

enum values cannot be reused in different types

enum bad_color { yellow, red, purple}
// not allowed, because red is already defined

you can set your own values, if you want
enum other_thing { good = 0, bad = 1, awful = 10 };

enum names are not kept at runtime, so you can't print them as their names
if you want to print an enum value by name, you have to write your own function
printf("%s\n", input) <- this is a type error
<- modern compilers will catch this and report an error
<- older compilers will let this through, and then your code will crash at runtime

you can also print them as integers

printf("%d\n", input); // prints the numeric value

why use enums?
makes your code clearer
fewer "magic numbers"
self-documenting code
<- says what the expected values are
some support from type system

can you do without them?
sure, just use constants

Structs

structs <- bundle data into a package

a struct type has multiple "fields", each field has a name and type

struct rgb_color {

```
        int redness;
        int greenness;
        int blueness;
        int transparency;
};
        // declares a type "struct rgb_color"
                // bundles together 4 integers
        // declares four field names
                // field names cannot be reused
                // we will see lots of field names with prefixes to avoid name collisions

struct rgb_color background;
        // declares a variable of type struct rgb_color
        // this can be a local variable; the struct value will be stored in the stack
        // i.e., this value is managed by the compiler

struct rgb_color x, y;

        x = y;   <- copies fields from y to x


struct rgb_color background = { 100, 0, 0, 15 };
        // special syntax for initializing struct variables
                // you can only do this when initializing a variable
        // standard only allows you to give fields in order
```

field access using .

```
        x.redness = 25;  <- you can set the values of fields
        y.redness = x.blueness;  <- you can access the values of fields

        if (x.redness == y.redness) { ... }
```

struct fields can be any type, including structs

```
        struct circle {
                struct point center;
                double radius;
                struct rgb_color fill;  // this stores the actual fields of the color in the circle
        };

        struct circle my_circ;

        my_circ.fill.redness = 235;  // accessing the field of a field
```

structs cannot be recursive!

```
        struct list {
                struct list next;  // not allowed!
                data_t data;
        };
```

// a structure that contains itself would be infinitely large!
// this is where we would want to use pointers (coming soon)

Unions

unions <- lets us combine multiple types
that is, use the same variable/field for different types of data

```
union int_or_float {
	int inty;
	float floaty;
}

union int_or_float f;
f.inty = 16;

if (f.floaty < 0.0) { ... }
```

similar to a struct, except that only one "field" exists at a time
all the fields are stored in the same/overlapping chunk of memory

unions do not store which field is currently active!
there is no way to check at runtime whether f contains a float or an int

some people use this as a way to sneakily get bit representations of types,
but not all compilers support this

this is not the same as casting!

Why would anyone do this?
to save space
if we have two variables/fields that we know we won't need at the same time,
and we only use them in situations where we know which one we have,
then we can use the same storage for both

somewhat justifiable for unions of large structs
lets us fake subtyping; where certain fields are only needed for certain data

```
struct circle_data { struct point center; double radius; };
struct rect_data { struct point topleft; struct point bottomright; }

struct shape {
	enum {CIRCLE, RECT} type;   // indicate which kind of shape
	union {
		struct circle_data circ;   // only store data appropriate to that shape
		struct rect_data rect;
	} data;
};
```

```
struct shape foo;
foo.type = CIRCLE;
foo.data.circ.center = p;
foo.data.circ.radius = 15;
```

generally a headache; you will rarely if ever use union

Arrays

arrays <- contiguous collection of (same-type) values

declare an array variable using []

```
int times[14];  // declares an array containing 14 integers
```

array size should be constant (e.g, a number or const)

array variables are managed by the compiler (you do not allocate/delete them)
each array variable points to a single unique array

access elements of an array using []

```
times[0]  <- first element of array
times[0] = 14;
times[1] = times[0] + 2;
```

"times" by itself can be treated like a pointer (coming soon)

you cannot assign to the array itself

```
array1 = array2;  <- not allowed; you would have to write a loop
```

multi-dimensional arrays are just arrays of arrays

```
int matrix[4][4];  // "matrix" will be 16 contiguous integers

matrix[0][0]  // access row 0, column 0
matrix[0]     // refers to the complete first row


float foo[20][20][20];  // 3-dimensional array
```

Objects and variables

In C, an "object" is anything that we store in memory integers, structs, arrays, even functions

What do we know about an object?
1. its address (where it is in memory)
2. its size (how many bytes it occupies)
3. its type (what sort of data it is)

<- but only the address is available when your program is running

Variables in C are names for specific objects

int x; // create an integer object in memory that I can refer to using the name x

the variable name/object mapping is fixed / we can't change it

We can use & to get the address of the variable's object

&x <- the location of x's object

Every variable has an associated object
Not every object is associated with a variable
<- malloc creates objects without variables

An "object" is anything that is stored in memory
-> this can include code, depending on how you think about it

What do we know about objects generally?
1. location in memory (the address)
2. type of data
3. size (bytes needed to store data)

Generally, this information is tracked by the compiler or the run-time system
-> C does not have "introspection", so we have no way to ask questions about objects in general

Objects can be grouped into three categories

global or static objects
- allocated when process starts, deallocated when process ends
- cannot create new ones during runtime, or change sizes
- global variables
- string literals
- static local variables

- compiler always knows where these are
  - no need to track these at run-time; program is written implicitly knowing where globals, etc. are located

- implicitly initialized to 0 (all zero bits), or NULL

stack objects

- associated with the "call stack"
    - stores the return address; says where to go when a function exits
- most languages store local variables and function arguments in the stack
- the stack as a whole is dynamically sized
- individual stack frames are usually static
    - the number/size of local variables doesn't change during run-time
    - there are ways to dynamically allocate stack space, but we won't use them

- compiler knows where these are, relative to the stack pointer

    compiler tracks local variables using offsets from the stack pointer

        e.g. for a local variable, int x, compiler will have table that says
            x is 4 bytes and located in a specific part of the stack frame

- program does not need to manage these
    - compiler and run-time manage allocation/deallocation of stack frames
    - these are also called "automatic" variables

        i.e., auto int x; // you can write auto explicitly, but it is implied

        by default, local variables are always auto

- stack objects are not initialized by default
    - unless you explicitly initialize, they will contain unspecified data
        before they are first written to
    - never read from an uninitialized variable
    - C gives you the option of not initializing, because you might want to
      save time by not initializing a variable that you won't need

heap objects

- not associated with variables
- explicitly allocated/deallocated by the program
    - malloc reserves space for an object
    - free marks that space as no longer in use
- dynamic memory management
    - run-time system keeps track of what parts of memory are in use/available
    - no reliable way for the program to obtain this information

- in Java, "new" allocates a heap object

    the garbage collector detects unused objects and frees them

when I call malloc(), I provide a size (the number of bytes I want)
and I get back a pointer to an object that is at least that big

    we can call this a "heap pointer"

when I no longer need the object, I pass that pointer to free()
-> the only pointers I should pass to free() are ones I got from malloc()
e.g., do not pass free() the address of variables, or pointers to the middle of a heap object

---

aside: static variables

- essentially global variables that are private to a single function

the first time you call strtok(), you pass it a pointer to a string
each subsequent time, you pass it NULL, and it remembers the string it was working with and where it was in the string

another use: random number generation
set a seed, then update the seed each time it is called

static variables are rarely useful

```
// this function returns an increasingly large number each time it is called
int count()
{
	static int c = 0;   // c is preserved across function calls
	return c++;
}
```

----

malloc/free

- why do we need free?
  if we never deallocate memory, we could run out (a memory leak)
  for short-lived programs, this is not much of an issue
  -> but it is good to practice cleaning up after yourself

- why do we need malloc?
  - we might not know how big an object needs to be until run-time
    e.g., an array of data read from a file
    global and local variables are statically sized (we can't make them bigger or smaller at run-time)
  - we might want to create data structures like trees/linked lists/graphs that require indirect references

----

aside: sizeof

sizeof() is an operator that returns the size (in bytes) of a type
sizeof(int) might be 2 or 4

```
        sizeof(double) usually 8
        sizeof(int*) might be 4 or 8
    sizeof() is not a function, and does not have any run-time cost:
        the compiler replaces it with an appropriate constant

    if I write something like 100 * sizeof(int), the compiler will replace
    sizeof(int) with the actual size and then do the multiplication
        -> on the iLab, it will be as efficient as just writing 400
        -> but using sizeof() guarantees that we get the correct size

    we can also use sizeof() with variables, but this always gives us the size
    of the variable's object

        int a[4];   // local array of 4 ints
        int *p = malloc(sizeof(int) * 4);  // pointer to object that holds 4 ints

        sizeof(a)  is equivalent to 4 * sizeof(int), because a in an array variable
        sizeof(p)  is equivalent to sizeof(int*), because p is a pointer variable

        sizeof(*p) is equivalent to sizeof(int), because the type of p is int*
```

----

array variables vs array objects

array variables always have a static (constant) size

```
    int a[400];   <- dimension must be known at compile-time
```

array variables alway refer to global or stack objects

technically, the type of "a" is int[400]

when I want a dynamically sized array, I have to allocate a heap object

```
    int arraylen = ...;

    int *p = malloc(sizeof(int) * arraylen);
        // allocates a heap object big enough to store arraylen ints
        // or returns NULL if not enough space is available
        // technically unsafe to use a pointer from malloc if you haven't checked
        // whether it returned NULL
```

the type of "p" is int*
    <- note that it is a pointer to an int, not a pointer to an array

in C, array indexing works using pointer arithmetic
    what happens behind the scenes is that we multiply the index by the size of
    the array element type and add to the pointer

p[10] <- adds 10 * sizeof(int) to the address of p to get the address of p[10]

p[10] means "10th integer after p"
this is why pointers are typed - so we know how big the elements are

nothing checks whether this address points to anything/the correct thing
<- no bounds checking!
<- you can go past the end of an array and never find out
<- if you are lucky, your program will crash with a segfault

<- the only way to avoid this is to make sure your algorithms are correct
<- tools like valgrind or AddressSanitizer can detect some errors

negative array indexes are allowed!
p[-1] reads an integer before the start of p
this is almost never useful, and never necessary
but it won't be checked for
recommendation: never do this on purpose

4[p] <- actually the same as p[4], because early C compilers were weird

in general, x[y] translates to *(x + y)
<- the + turns into pointer aritmetic, if x or y is a pointer
<- the * dereferences the new pointer
<- conclusion: 4[p] is the same as p[4] is the same as *(p + 4)

this is trivia: don't write code like this; it is confusing for no reason

what about array accessing?

a[1] works the same way as p[1]

"array decay" transforms array references to pointers in most circumstances

often, we ignore the difference between array variables and array objects

differences
pointers can point to different arrays
array variables always refer to the same array
multidimensional arrays

int a[4]; // a is a fixed stack object
int *a = malloc(4 * sizeof(int)); // *a is a new heap object

aside: char **argv
What is this?
argv refers to an array of pointers

argc gives us the length of the array

argv is a char** (double-pointer)
argv[0] is a char* (pointer to a char or string)
argv[0][0] is a char (the first byte in argument 0)

The C language standard guarantees that the pointers in argv always point
    to '\0'-terminated strings

we can call this a multi-level array, because each "row" can have a different
    length

why doesn't C do bounds checking?
- we don't keep the size information anywhere at run-time
    -> the language can't check, even if it wanted to
- most of the time, your algorithms already rely on the size of the array

if I write this loop,

```
for (i = 0; i < arraylen; ++i)  {
    total += a[i];
}
```

note that I already check i each iteration to make sure we didn't go past the end
the C philosophy says that having the array access do the check a second time
    is wasteful
the Java philosophy says that safety is more important (what if arraylen isn't
    actually the length of a?)

C puts you in charge of safety, so don't screw up

If you want, you can write your own functions

```
struct safearray {
    unsigned length;
    int *data;
};

int saferead(struct safearray *a, int i) {
    if (i >= 0 && i < a->length) {
        return a->data[i];
    }

    return 0;  // or explode with exit(EXIT_FAILURE) or something
}
```

Memory management

----

global objects - referred to using variables (direct) or pointers (indirect)
stack objects - referred to using variables (direct) or pointers (indirect)
heap objects - always referred to indirectly (using pointers)

Use & to get the address of a variable
    -> turn direct reference to indirect reference

Use (unary) * to follow a reference (dereference - turn indirect to direct)

Use malloc to create heap objects
    -> returns pointer to newly created object

Use free to deallocate heap objects
    -> give pointer to object we want to delete
    -> do not attempt to free global or stack objects!

Other useful functions

void *calloc(size_t num_elems, size_t elem_size);
- allocates space and clears all the bytes (sets all bits to 0)
- number of bytes allocated is num_elems * elem_size
- avoids any problems with forgetting to initialize values
    - allocate an array or struct with all elements/fields set to 0

- having two arguments may be to prevent integer overflow if you are allocating a lot of memory
    - but then why does malloc only take one argument? unclear

```
int *A = calloc(100, sizeof(int));
    // allocate space for 100 integers and set them all to 0
```

- why have calloc and malloc?
    malloc is faster because it doesn't need to write to the allocated space
        i.e., maybe you are going to immediately initialize to non-zero values
    calloc is safer because it ensures a known value of the data

- calloc is essentially just malloc + memset

void *memset(void *p, int c, size_t n);
- sets the value of bytes in memory
    - p - pointer to start of object
    - n - the number of bytes in the object
    - c - actual byte (char) to write
        only the lower 8 bits of c will be used

- may be faster than writing a loop yourself
    - possibly takes advantage of OS operations; hardware accelleration

- returns the same pointer it was given

void *realloc(void *p, size_t size);

- changes the size of a heap object
  - shrinks in place
  - grows in place if possible, or allocates new space & moves data

arguments:
p - pointer to object we are resizing
size - number of bytes we want the object to be

returns pointer to object - may have moved!
for this reason, you should not use the old pointer value after a call to realloc

- realloc is only safe for objects with a single reference

recall: ArrayList in Java
array that you can grow

We can implement ArrayList in C using a pointer and two integers

```
// global variables
int *list, size, used;

	// to have multiple array lists, create a struct that holds these

// set up array list: allocate space for data, set variables
// (essentially a constructor)
void init(int init_size)
{
	size = init_size;
	list = (int *)malloc(size * sizeof(int));
		// cast is unnecessary; just making sure I have the right pointer type
	used = 0;
}


// to append onto the array, we increase used
	// used: number of array elements "in use" (initially zero)

	// if used == size, we use realloc to make the array bigger

// add value onto the end of the array list
	// make additional space if necessary
void append(int i)
{
	if (used == size) {
```

```
        size = size * 2;
        list = (int *)realloc(list, size);

    }
    list[used] = i;
    used++;
}

int remove()
{
    if (used > 0) {
        --used;
        return list[used];
    }

    return 0; // or throw an error or something
}
```

void *memcpy(void *dest, void *src, size_t bytes);

- copy data from one object to another
arguments:
  dest - where data will go
  src - where data comes from
  bytes - how many bytes to copy

returns the destination pointer

- you must ensure that src and dest point to objects of the correct size
- src and dest must not overlap

- could be implemented as on O(n) loop, or use OS/HW tools for speed

- memcpy copies bits: does not care about types
  - does not convert values

- you must ensure that src and dest point to appropriately typed data

something like

```
char foo[] = "Hello";
```

is equivalent to doing;

```
char foo[6];
memcpy(foo, "Hello", 6);
```

void *memmove(void *dest, void *src, size_t bytes);

- same as memcpy, except that source and destination may overlap
- may be less efficient than memcpy
    - may be more efficient than writing a loop yourself

example: removing elements from the start/middle of an array

```
int a[100];
...
memmove(a, &a[1], 99 * sizeof(int));
        // copy 99 integers from a[1] .. a[99] into a[0] .. a[98]
                // note that a[99] will retain the same value
```

does a = &a[1] work?
    - no: compiler will not let us assign to an array variable

what about pointers?

```
int *p = malloc(100 * sizeof(int));

p = &p[1];   // allowed, but now we don't have a pointer to the original object
// unless we do free(p - 1);

int *q = p;

q = &q[1];
q = q + 1;
++q;
        // be careful when doing pointer shenanigans!
        // if you lose the pointer to a heap object, you won't be able to free it
```

char *strcpy(char *dest, char *src);
- similar to memcpy, but we don't specify a size
    - instead, we copy from src until we reach an all-0 byte ('\0')

- still must ensure that dest and src do not overlap
- still must ensure that dest is large enough to hold data

char *strncpy(char *dest, char *src, size_t n);
- copies up to n bytes from src to dest
    - stops after '\0' byte, or after n bytes

- note that strncpy may result in a non-terminated string, if src is longer than n bytes
    - you can explicitly set the last byte to '\0' if there is a problem

```
char foo[100];

strncpy(foo, some_string, 99);  // copy up to 99 chars
        // adds null terminator if some_string is less than 99 chars long
foo[99] = '\0';   // ensure that foo is null-terminated
```

```
        // redundant if some_string is < 99 chars long, but that is okay
```

Deciding when to use memcpy, memmove, strcpy, strncpy is fairly straightforward

    memcpy and memmove specify exactly how many bytes to move
    strncpy may copy fewer bytes
    strcpy is potentially more efficient than strncpy

    in general: pay attention to how long your strings may be
        are you copying data from disjoint objects or within an object

Example: duplicating a string

```
char *str_dup(char *str)
{
    char *dest = malloc(strlen(str) + 1);
        // allocate space for the whole string, plus null terminator

    strcpy(dest, str);

    return dest;
}

char *str_dup(char *str)
{
    int len = strlen(str) + 1;
    char *dest = malloc(len);
        // allocate space for the whole string, plus null terminator

    memcpy(dest, str, len);
        // faster than strcpy because it doesn't need to look for '\0'
        // dest is new object, so it can't overlap with str

    return dest;
}

// or use strdup from Posix library
```

Use man for more details
documentation is also available on-line

C Pre-Processor & separate compilation

The C Pre-Processor (CPP) runs before the compiler starts compiling
- fancy find/replace stage that modifies your source code

- can be used for code generation or for defining constants

Lines starting with # are preprocessor directives

```
#include <std_file.h>
#include "my_file.h"
```

#include copies the content of a file into your file
<filename> says to look for the file in the "include" directory
(e.g., /usr/include)
"filename" says to look relative to the current directory

Primarily used to include "header files"
typically end in .h
include function prototypes, struct declarations, typedefs, etc.

When I write
#include <stdio.h>
CPP will copy the definitions in stdio.h into my file
now I can call printf(), etc, without needing to declare them

Typically at the top of a file, but can occur anywhere
but included text is included at that point

Note that this is different from linking, but related
include brings in declarations from other files
linking connects our code to definitions from libraries

```
#include "something.txt"
    // works fine
```

```
#define MACRO optional_value
```

declares a "macro", which is used for text substitution
i.e., replacing a constant with its value

```
#define MEMSIZE 100

...

    int mem[MEMSIZE];  // after preprocessing, MEMSIZE will be replaced with 100
```

Using macros allows us to keep things consistent
If you see 100, you might not know what it means
MEMSIZE tells you

If we change MEMSIZE, we can do it in one place and all references will update

Any time you have numbers other than 1 and 0, consider defining a macro

MEMSIZE * 2

will be replaced with

```
100 * 2
```

and the compiler will turn that into

```
200
```

```
#include
#define

#define BUFSIZE 256

    ...
    char buf[BUFSIZE];

    for (i = 0; i < BUFSIZE; ++i) ...

#define MSG "Hello"

#define VARTYPE int
#define VARFMT "%d"

    VARTYPE var;
    printf("Your variable is " VARFMT, var);
```

C preprocessor replaces VARFMT with "%d"
C says that two string literals in a row will be concatenated

so this is equivalent to:

```
int var;
printf("Your variable is %d", var);
```

-> this would allow me to change the type of var and update all my format strings
-> of course, we still have to make sure our definitions are consistent

we can take advantage of conditional compilation to have different definitions of VARTYPE and VARFMT in different circumstances

conditional compilation: #if, #ifdef, #ifndef, #endif
-> these allow us to "turn off" or remove chunks of code

```
#ifdef LONGVAR

#define VARTYPE long
#define VARFMT "%ld"

#else

#define VARTYPE int
#define VARFMT "%d"
```

#endif

Now, we can use the macro LONGVAR to set the type and format code together

We can define LONGVAR using #define

#define LONGVAR

We can also define some macros on the command line when we call GCC

gcc -DLONGVAR

same as #define LONGVAR 1

gcc -DBUFSIZE=512

same as #define BUFSIZE 512

#ifdef "if defined"
#ifndef "if not defined"

#if BUFSIZE > 20

macros with arguments
----

idea: put a set of comma-separated arguments in parentheses after the macro name

#define square(X) ((X) * (X))

"X" is the argument
-> within the replacement text, it will be replaced

this will replace
square(a)

with
((a) * (a))

this will replace
square(x + y)

with
((x + y) * (x + y))

<- this is not a function call; it just looks like a function call

function calls happen when your program is running
macro substitution occurs before your program is compiled

arguments to functions are evaluated before the function is called
arguments to a macro are substituted as-is

```
#define isupper(C) ((C) >= 'A' && (C) <= 'Z')
```

<- safe, can compile efficiently (no function call at run-time)
<- but watch out if C is an expensive computation or has side-effects

```
char c;

isupper(++c);  <- will increment c twice!

    -> isupper(++c)
    -> ((++c) >= 'A' && (++c) <= 'Z')
```

```
#define log(X) log10(X)
```
<- quick and dirty rewrite of code to use a different logarithm function

```
#ifndef WRONGLOG
#define log(X) log10(X)
#endif
```
<- do the substitution unless the WRONGLOG macro is set

__FILE__ - replaced by the current file name
__LINE__ - replaced by the line in the source code

```
#define msg(S) printf("Message %s (%s:%d)\n", S, __FILE__, __LINE__)

    msg("I got here");
    ->
    printf("Message %s (%s:%d)", "I got here", "myfile.c", 1239);
```

Summary:
macros are fun and powerful
... but don't use them too much
easy to write obscure, baffling code

be liberal with parentheses!

separate compilation

Header files:
put function prototypes and type declarations in a single place
use #include to include header in all files using those functions/types

#include brings definitions into our source code

Linking
after compiling separate files, connect names to definitions

Separate compilation

have multiple .c and .h files
-> typically a .c file for each standalone concept or feature

separate compiling into two steps
regular compiling: turn .c file into .o file
linking: combine multiple .o files into an executable program

if we make a change in one .c file, we only need to recompile that file and then relink
-> don't need to recompile the entire program
-> this saves a lot of time for projects with 1000s of source files

managing separate compilation of hundreds/thousands of files is tedious
use make to do compilation
make detects which files need to be recompiled
-> only performs steps necessary to create the requested program

make is very general
-> not limited to compiling files
-> not restricted to any particular compiler or language

we use a "make file" to specify rules that say how to create/build/compile the thing that we want

e.g., we might have rules that say
how to create "demo"
what other files need to exist to create "demo"
how to create those files

by default, make will decide whether it needs to recreate something based on the modification times of files
if a file is older than the files it depends on, then it needs to be recreated

making make make with makefiles

Make is a tool that we can use to automate running programs
-> primarily to simplify compiling programs

You provide make a set of rules in a file called a "make file"
-> tells make how to create certain files / do certain tasks
-> traditionally a file named "Makefile"
    -> capital M means it appears earlier in the alphabetical listing
    -> make has a list of "default" file names that it will look for
    -> you can explicitly tell make which file to use as a make file

using make

```
$ make
```

    <- tells make to look at Makefile and update the first target

```
$ make some_target
$ make target1 target2 target3...
```

    <- tell make to update specified target(s)

```
$ make -B [targets...]
```

    <- tells make to update targets, ignoring modification dates

```
$ make -j N [targets...]
```

    <- tells make to run on N processors simultaneously (if possible)
    <- not necessarily useful for us

rules

```
[target name]: [zero or more dependencies]
	[recipe: shell commands]
```

NOTE: recipe must be indented using a single tab (\t) character

Example:

```
program: program.c
	gcc program.c -o program
```

This describes a rule to make "program"
- "program" depends on "program.c"
    - ie., if program.c changes, then program is out of date
- recipe says how to create program from program.c

- to run:
```
$ make program
```

How make treats this rule

- when we tell make to make program, it will check whether it needs to
update the file "program"

Process:
1. Recursively update all dependencies of the target
2. Check whether the target exists
3. If so, check whether the target is older than its dependencies
4. If target does not exist or is older than its dependencies (or we used -B)
make performs the recipe

Idea: only perform the recipe if we need to
- but the recursive check ensures that files get rebuilt as needed

Example:

```
demo: demo.o arraylist.o
	...
demo.o: demo.c arraylist.h
	...
arraylist.o: arraylist.c arraylist.h
	...
```

After we build demo, we will have
arraylist.h arraylist.c demo.c
older than
arraylist.o demo.o
older than
demo

I modify demo.c, so now it is newer than demo

demo.c < demo < demo.o

When I do "make demo", make will update demo
Recursively update demo.o
demo.o is older than demo.c, so we rebuild demo.o
Recursively update arraylist.o
arraylist.o is fresher than arraylist.c and arraylist.h, so do nothing
Now, demo.o is fresher than demo, so rebuild demo

Note that demo.o and arraylist.o both depend on arraylist.h
Why?

presumably both demo.c and arraylist.c include arraylist.h
changes to arraylist.h may require recompiling

-> make does not track dependencies automatically; you must specify
them yourself
(or use a program to track them)

Why might changing a header require recompilation?

Let's say I have

```
typedef int foo_t;
```

If I change this to

```
typedef unsigned int foo_t;
```

Any code doing math with a foo_t value may need to change
e.g., signed vs unsigned division instructions

Special cases:

Rules do not require dependencies

For example,

```
clean:
	rm -f list of files....
```

<- doesn't depend on anything, so no need for recursive check
<- doesn't create a file named "clean", so it will always execute
"pseudo-rule"

<- note that we can perform any command, not just compiling
<- we use -f to prevent rm complaining if we try to delete something that doesn't exist

Other non-compiling uses

```
test: demo
	./demo some_input
```

<- re/compile demo if needed (update target demo using rule)
<- call demo with specified argument(s)

Note:
make checks the exit status of the commands it performs
if a program does not return EXIT_SUCCESS, the rule fails

So if we "make test" and GCC reports an error when compiling "demo",
make will stop (i.e., not try to run "demo")

-> this is one of the reasons we have exit statuses in our programs
tell whoever ran the program whether the program succeeded

Make variables

VARIABLE_NAME = some text

<- declares a variable
<- variable names do not have to be all-caps, but this is common

Make will substitute $(VARIABLE_NAME) with the value of the variable

```
CC = gcc
CFLAGS = -g -Wall

demo: demo.o arraylist.o
	$(CC) $(CFLAGS) -o $@ $^
```

as though we had written
"gcc -g -Wall -o demo demo.o arraylist.o"

Make has a bunch of special variables
- $@ - the name of the target of that rule
- $^ - the dependency list for that rule
- $< - the first dependency for that rule

Why use these?
- save typing
- define rules in a more general way
- make changes in a centralized way (e.g., adjust CFLAGS)
- override variable declarations when we call make

```
make demo CC=clang
```

<- will compile using clang instead of gcc

- other fancy stuff when using multiple directories

-> but these are all optional; you are never required to use variables

Rule schemes
- fast way to define a bunch of rules at once

```
%.o: %.c
	gcc -c $<
```

This says: to make any file ending with .o, run this command;
it says that FILE.o depends on FILE.c
the command is "gcc -c FILE.c"

We can override this for specific files by writing an explicit rule

```
# general rule
%.o: %.c
	$(CC) -c $(CFLAGS) $<

# specific rule for one file
special.o: special.c
	$(CC) -c $(CFLAGS) -other-flag $<
```

Note: make has a default rule for %.o files that it will use if you don't
specify one

Why use $< instead of $^ ?

Recall that .o files can depend on .h files as well
We don't want to include the .h files in the argument to gcc

How can we specify .h files if we are using general rules?

We can declare additional dependencies with rules that don't have a recipe

```
%.o: %.c
	$(CC) -c $(CFLAGS) $<

demo.o: arraylist.h
arraylist.o: arraylist.h
```

-> now demo.o will depend on demo.c and arraylist.h

Make is not specific to any language

We could do something like this

```
%.class: %.java
	javac $<

# additional options may be required (I haven't used javac in a long time)
```

Note: target and dependencies can be paths

```
lib/foo.o: src/foo.c
	....
```

Try stuff out!

input and output
----

https://www.gnu.org/software/libc/manual/html_node/I_002fO-Overview.html

How do we communicate with the outside world from our program?

    print to screen
    read from keyboard
    read/write files
    send/receive network messages
    communicate with other processes

In Unix/C the answer to all of these are "files"

    in general, a file is a stream of bytes that we can read from and/or write to

    -> files may be actual files stored on a disk, but we reuse the abstract
        inferface for all I/O

This is not how older languages/operating systems worked

    Older file systems were based on "records"
        text files were weird/hard to work with
    different functions/system calls for different kinds of communication

In C/Unix, we use the same general model for all communication

    possibly with extra features for certain types
        e.g., we can "rewind" files or jump to specific places

We have two sets of functions for working with files in most C environments

- "streaming" functions specified by C standard
    obtained from <stdio.h>
    use FILE * (file pointer)
    function names usually start with f
        fopen, fclose, fread, fwrite, ...

- low-level system calls specified by Posix
    obtained from various places <unistd.h>, <fcntl.h>, and others
    use file descriptors (just an integer)
    open, close, read, write, ...

In this class, we will focus on the low-level system calls

In general, you will want to use the streaming functions most of the time

    - more portable (C standard library)
    - potentially fewer system calls

What is the operating system for?
-> provides an interface between programs and the hardware
    -> abstracts details of the hardware
-> isolates programs and users
    -> multiple programs can run without interfering with each other

therefore, program IO almost always happens via OS services ("system calls" "syscalls")

    -> instead of taking directly to the disk or display hardware,
      we ask the OS to send byte sequences appropriately

How system calls happen is different depending on the HW and OS
    -> C provides functions that wrap the syscall
    -> the functions themselves are HW/OS specific and may be written in assembly

Posix standardizes system calls for Unix-like operating systems
    there used to many, many proprietary Unix variants
    having a standard made it easier to write portable code
    -> nowadays, Unix is pretty much always Linux or BSD (including MacOS)
        -> plus a bunch of other software (eg. GNU utilities)

The Posix functions for file IO are open, close, read, write, &c.
-> central interface is the file descriptor
    -> this is a number that indicates which file you are talking about

Each process has a table of open files; the file descriptor is an index into this table
-> remember: files can be any stream of bytes
    -> e.g., we could read from a file on disk, or a terminal interface, or the network
      or other processes

#include <fcntl.h>   // "file control"
    provides open and close

int open(char *filename, int flags)

    filename is the name of a file or a path to the file
    flags is a bit-vector that says what features we want

    Must start with one of these
        O_RDONLY - open a file in read mode
        O_WRONLY - open a file in write mode
        O_RDWD   - open a file in read/write mode

    Use bit-wise or to add additional features

        O_APPEND - start with file pointer at the end of a file
        O_TRUNC   - delete contents of file
        O_CREAT   - create file if it does not exist (*special)

to open a file in append mode, we would say

```
int fd = open("log.txt", O_WRONLY|O_APPEND);
```

Use "man 2 open" to see a list of possible flags

most of time we will open in read mode

```
int fd = open("my_input.txt", O_RDONLY);
```

open returns -1 if it could not open the file
- file descriptors are always non-negative
- when open fails, it will set the global variable errno

traditionally, file descriptors 0, 1, and 2 are already open when your program starts
- 0 is standard input
- 1 is standard output
- 2 is standard error
- you don't open these yourself

Using O_CREAT adds a third parameter to open, so we can specify its permissions

int open(char *filename, int flags, mode_t mode)
- mode indicates what permissions the file should have
  - e.g., user/group/world readable/writeable/executable
- we will come back to this

Use close to close a file

```
int fd = open("my_file", O_RDONLY);
...
close(fd);
```

Reading from a file

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

```
size_t read(int file_descriptor, void *buffer, size_t buffer_size)
```

-> reads from the specified file
-> writes bytes starting at the address specified in buffer
-> reads up to buffer_size bytes
-> returns the number of bytes read
 -> returns 0 at end-of-file
 -> returns -1 and sets errno if something went wrong

```
// read from stdin

int bytes_read;
```

```
char buf[256];

bytes_read = read(0, buf, 256);
        // read UP TO 256 bytes from standard input (0)
                // we might read fewer than 256 bytes, but we definitely won't read more
        // write bytes to buf (an array of chars)
        // return the number of bytes successfully read (assigned to "bytes_read")
                // if we ask for more bytes than there are available, we get all of them
                // if only 100 bytes are available, then bytes_read will be 100
```

read does not read strings! If we want a terminator, we have to add it ourself

// writing to a file

ssize_t write(int file_descriptor, void *buffer, size_t count)

- -> writes to the specified file
- -> bytes are taken starting at address specified (buffer)
- -> number of bytes to write is specified (count)
- -> returns the number of bytes successfully written
    - -> returns -1 and sets errno if something went wrong
- -> note that the buffer can be any type!

```
// write to stdout

int bytes_written;
char buf[256] = "Hello!\n";

bytes_written = write(1, buf, strlen(buf));
        // write to stdout (file descriptor 1)
        // take bytes from buf
        // only write the bytes in the string, not including terminator
        // bytes_written will be assigned how much we were able to write
```

Q. why didn't we do this?

```
        write(1, buf, 256);
                -> this would write all the garbage data after the \0

        write(1, buf, strlen(buf) + 1);
                -> this would write the terminator
                -> ... but we don't normally want to do this
```

Remember: terminators are used in C strings, but files are not strings
    files can contain many \0 bytes, or none
    file length is tracked by file system

Using read and write with binary files
-> read and write work with any data

-> we take bytes directly from memory without interpretation

Writing an array of ints to a file

```
int a[] = {1,2,3,4};

int fd = open("my_data", O_WRONLY|O_TRUNC|O_CREAT, S_IRUSR);

write(fd, a, sizeof(int) * 4);
        // writes the array of ints to the file directly
        // does not print as text! does no interpretation!
        // the literal bytes from memory are written to the file
```

the file will be (say) 16 bytes,

on a little-endian machine (Intel) with 4-byte ints, it will contain (in hex):

    01 00 00 00 02 00 00 00 03 00 00 00 04 00 00 00

on a big-endian machine (ARM) with 4-byte ints, it will contain (in hex):

    00 00 00 01 00 00 00 02 00 00 00 03 00 00 00 04

this is why file transfer between different hardware can be challenging!

We can use read in the same way

```
int a[4];
int fd = open("my_data", O_RDONLY);
read(fd, a, 4 * sizeof(int));

        // reads 16 bytes from the file and writes the bytes to memory in a
        // no interpretation!
```

Typical pattern when reading a text file

- two loops
- outer loop calls read and writes to a char buffer
    - inner loop iterates through the bytes in the buffer
- outer loop ends when read returns 0 (EOF) or -1 (error)

```
// int bytes_read, length;
// char *buffer or char buffer[...]

bytes_read = read(my_file_desc, buffer, length);
while (bytes_read > 0) {
        for (i = 0; i < bytes_read; ++i) {
                // do something with buffer[i]
        }
        bytes_read = read(my_file_desc, buffer, length);
}
```

```
// note that we have to keep reading until we get 0
// don't want to assume how big the file is
// allocating a giant buffer is wasteful
```

Some people prefer not to duplicate the call to read

```
while ((bytes_read = read(my_file_desc, buffer, length)) > 0) {
    for (i = 0; i < bytes_read; ++i) {
        // do something with buffer[i]
    }
}
```

Recall that assignment is an expression that evaluates to the value assigned

```
(x = foo()) == y
    // calls foo(), assigns return value to x, and compares with y
```

Some people feel that putting the call to read in the while loop condition is confusing and makes it hard to see what is happening

------

File permissions (Unix/Linux specific)

We control access to a file by setting permissions bits

Three kinds of access
- r - read access
- w - write access
- x - execute access (for files) or content listing (for directories)

There are three groups of people we can give permissions to
- u - "user", or user account that "owns" the file
- g - "group", a group of accounts (excluding the owner)
- o - "other", all user accounts that are not the owner or group

3 * 3 = 9 permission bits, in groups of three

ls -l prints the permissions in the order

```
rwxrwxrwx
---       user permissions (owner)
   ---    group permissions
      --- other permissions (global)
```

Because these are in groups of 3, it is common to use octal to specify

```
4 -> 100 -> r--  read-only
6 -> 110 -> rw-  read/write
2 -> 010 -> -w-  write-only?
```

7 -> 111 -> rwx read/write/execute

600 -> rw------- user can read/write, no one else can read or write

Note: these are permissions for a file on disk
They are not directly related to O_RDONLY or O_WRONLY
-> those specify what operations the file descriptor supports

The permissions that a file has on disk restrict what modes you can open
a file in

Shell command: chmod
changes the permissions for a file

chmod 644 some_file
change permissions to rw-r--r--

chmod u+rw some_file
set the user read and user write bits to 1

Shell command: chown
change the owner (or group) for a file
(may be root-only)

When creating a file, we must specify the permissions to give it
this is the third argument to open
-> you must provide a mode when opening a file with O_CREAT
-> the compiler might not catch mistakes here, so be careful

Note that we can create a file in write-mode without giving the user
write permission!
But we cannot open an existing file in write-mode if the user does not
have write permission

O_TRUNC -> if the file exists, truncate its length to 0
O_APPEND -> starts writing from the end of the file

-> if we don't use O_TRUNC or O_APPEND, then we will start writing
from the beginning of the file and overwrite the existing data, but
leave data we don't overwrite in place

A common combination O_WRONLY|O_TRUNC|O_CREAT
open the file in write mode
delete its contents if it already exists
create it if it does not exist

Another O_WRONLY|O_CREAT|O_EXCL

open the file in write mode
create it if it does not exist
fail if it does exist (ie., return -1 and set errno)

Inodes and directory entries

Note: We are going to talk about the Unix file system; not all file systems work the same way, but most file systems have analogous concepts.

What is a file?
- in Unix, a sequence of bytes that is stored somewhere and we can refer to by name

How do we store a file on disk?
- naive approach: write data contiguously to disk (similar to heap)
    - problem: files often grow or shrink
        -> may not be room for a file to grow
        -> shrinking a file may lead to gaps between files
    - files can be deleted, this will lead to gaps
    -> disk-based systems do not work this way
        -> some tape-systems do, but these are rare nowadays

-> instead, we break files into parts ("blocks")
    -> each block is the same size
    -> blocks can be packed into the disk (like an array)
    -> large files will be made of multiple blocks
    -> now we need a way to indicate which blocks make up a file

In Unix, we distinguish data and metadata
    data - actual content of the file
    metadata - information about the file
        - how big it is
        - when it was created/modified/accessed
        - who has access
        - which blocks are used to store the data

-> the metadata for a file is stored in an "inode" ("i-node")
    -> Each inode has a unique number (ID)
    -> Each inode is the same size

Problem: how can we keep track of a variable amount of data using a fixed amount of metadata?
    -> we want the maximum file size to be very large
    -> we don't want the inode to be too big, because most files are small
    -> we want random-access to file contents

we could use a linked list, where each block refers to the next block in the file

- file if we only read sequentially
- but if we want to append to a file made of 1000 blocks, we have to walk through every block to find the end

We could just list all the blocks in the inode
-> this wastes space for small files
-> not helpful if we want to allow large files, but expect most files to be small

Unix file system uses a system of indirect accesses

For some N, the first N blocks are directly referenced in the inode
Next, we have indirect references
The inode refers to a block that contains block references

For example, let's say the inode contains 25 direct references
let's also say that a data block can hold 100 block IDs

-> the first 25 blocks are referred to directly by the inode
-> the next 100 blocks are given by the single indirect node
1 reference in inode gives us 100 indirect references
-> the next 10,000 blocks are given by the double indirect node
1 reference in inode gives us 100 blocks, each of which points to 100 blocks
-> the next 1,000,000 blocks are given by the triple indirect node
1 reference in inode
-> 100 references in block
-> each leads to 100 references
-> each pointing to 100 blocks

Summary: 28 entries in inode allows us to reference up to 1,010,125 blocks

The block's number in the file tells us how to find it
0.. 24 - direct reference
25.. 124 - single indirect (index 25 in inode)
125.. 10,124 - double indirect (index 26 in inode)
10,125..1,010,124 - triple indirect (index 27 in inode)

Note: 25 and 100 are just numbers I chose for this example
both will be larger for a real file system

-> fixed size for inode
-> large maximum size for file
-> minimal space usage for small files
-> constant-time access for blocks
-> later blocks in the file take longer to find, but we are limited to 3

E.g., to examine block 1,000,000, we only need to access the inode and three indirection blocks to get to the block
For a linked list, we would need to go through all 1,000,000 preceding blocks

This is a good balance between speed and flexibility
    -> no overhead for small files
    -> small overhead for large files

    -> essentially a lopsided, N-way tree

Why only triple indirection?
    No technical limitation here; we need to have a limit because inodes are
        fixed size
    This means we do have a maximum file size
        -> on modern file systems, the maximum file size is very, very big

Takeaway: fixed size metadata
    allow for very large files
    minimal overhead for small files
    fast random access

Aside: Non-sequential file access
    if we open a file in append mode (O_APPEND), then we will start writing
        from the end of the file
    lseek() lets us move the file cursor anywhere in the file
        i.e., we can skip ahead or go backward

Also of note:
    Unix-like filesystems use inodes to reference files
        we can refer to a file by its inode ID
        (the file system keeps track of how to find an inode given its ID)

    Notably, the file name is not part of the file metadata!
        -> file names are part of the directory listing

What is a directory?
    A special file that contains directory entries
        each entry contains some information, including
            - the name of the file
            - what type of file it is (regular file, directory, other)
                not related to extension, or the type of data in the file
                programs, source code, text files, photos, etc. are all regular files
            - its inode ID

The file system tracks the root directory /

    / contains its subdirectories & files
        each subdirectory contains its subdirectories and files

If we have a path like /user/foo/homework/hw1.txt, we can find its inode ID
    start from /

look up "user" in /
look up "foo" in /user
look up "homework" in /usr/foo
look up "hw1.txt" in /usr/foo/homework
get inode ID

We say that the name (or path) of a file links to its inode ID
the inode ID is the "true" name of the file
the path is the user-friendly alias

However:
Not every file is linked from the directory structure
Some files can be linked more than once
-> we can have multiple names for the same file

A file's inode tracks how many times it is linked
(i.e., how many names it has)
When we use rm to delete a file, we unlink the name from the file
if a file has no links, the file system deletes it

Use ln to create additional names for a file (sometimes called a hard link)

ln existing_file new_name

existing_file and new_name will both refer to the same inode
they are indistinguishable

If I then rename existing_file, the new name will still refer to the same
inode as new_name

Contrast with symbolic links

ln -s existing_name new_name

new_name is a new file that refers to the name "existing_name"

If I rename existing_name, the link breaks
If I then create a new file called existing_name, then new_name will refer to that

Stat

References (on iLab)
man 2 stat
man 7 inode

How can we get information about a file?

stat/fstat/lstat - gives us data from the inode

given a file name, is this file a regular file or a directory?
    how big is it?
    what is its creation/modification/access date?
    who can access it?

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

int isdir(char *name) {
        struct stat data;

        int err = stat(name, &data);

        // should confirm err == 0
        if (err) {
                perror(name);  // print error message
                return 0;
        }

        if (S_ISDIR(data.st_mode)) {
                // S_ISDIR macro is true if the st_mode says the file is a directory
                // S_ISREG macro is true if the st_mode says the file is a regular file

                return 1;
        }

        return 0;
}
```

assert

Do your programs check assumptions?

    If I have a function argument that should only be positive, do I check that?
    -> it's a good idea to do this!

    "impossible" things should never happen, so if they do happen you want
    to know about it
        -> it must be a bug, so you will want to fix it

rather than write a bunch of code, you can use assert to check for things
that should never happen

```
#include <assert.h>

{
        assert(x > 0);  // looks like a function
                // if its argument is false (0), it prints an error message and
                // halts your program
```

}

For example: a.out: assert.c:6: main: Assertion `1 == 0' failed

-> note that the error message includes the source file and line number
and the actual source code for the condition
-> handy!

-> assert() is actually a macro
it obtains the file and line number using __FILE__ and __LINE__
it obtains the source code by "stringifying" its argument

```
#define ensure(X) if (!(X)) { puts("PANIC! " #X); abort(); }
```

using #X says we want X as a string literal

```
ensure(x > 0)
```

->

```
if (!(x > 0)) { puts("PANIC! " "x > 0"); abort(); }
```

Remember that the compiler will concatenate adjacent string literals

Remember to check for errors! Even impossible errors!
close only fails in unusual circumstances,
but it might indicate a bug in your program, like closing a file twice
if it fails, you want to know about it

malloc almost never fails due to low memory,
but it will definitely fail if you accidentally give it a negative argument

malloc(-1) is the same as malloc(some enormous positive integer)

```
int *p = malloc(....);
assert(p);
    // some would argue this is inappropriate because we don't print the
    // specific reason malloc failed
```

Don't use assert for possible failures
e.g., there are many reasons why open might fail other than bugs
you should explicitly check for this

If you are using a function that sets errno, you can use perror to print an error
message

```
int fd = open(argv[1], O_RDONLY);
if (fd == -1) {
    perror(argv[1]); // will print argv[1] followed by a text description of the problem
```

```
        exit(EXIT_FAILURE);
    }

#define strict(X) do { \
        if ((X) == -1) { \
            perror(#X); \
            exit(EXIT_FAILURE); \
        } \
    } while (0)
```

we use a few tricks
-> \ at the end of the line continues the macro (convenient for readability)
-> we use do { ... } while (0) to swallow a semicolon

```
    strict(close(fd));
->
    do { ... } while(0);
```

-> compiler will realize that loop always runs exactly once and eliminate it

getting the line number as a string literal is tricky because of how macros are expanded

```
#define xstr(X) #X
#define str(X) xstr(X)

#define strict(X) do { \
        if ((X) == -1) { \
            perror(#X " @" __FILE__ ":" str(__LINE__)); \
            exit(EXIT_FAILURE); \
        } \
    } while (0)
```

Working with directories

What is a directory?
-> it is a special file that contains directory entries
- -> "special" because its inode has a flag saying "this is a directory"
- -> a directory entry links a name to an inode
  - -> each entry includes a name and an inode ID
  - -> the inode ID is the "true name" of the file

We use opendir to open a directory file and read its contents

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

DIR *opendir(char *path)

-> returns a pointer to an abstract DIR structure
-> returns NULL and sets errno on failure

int closedir(DIR *directory_pointer);

-> close the directory & deallocate struct
-> returns -1 and sets errno on failure

struct dirent *readdir(DIR *directory_pointer);

-> obtain the next entry from the directory
-> returns NULL if there are no further entries
-> each time we call it, we get a pointer to a struct containing data about the next directory entry
    -> this is managed by the library; do not attempt to free this pointer
    -> it is allowed for readdir to rewrite the data and return the same pointer each time
        in other words, we can only use the pointer we got back from readdir until the next call to readdir or closedir

Copied from man page for readdir:

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

-> illustrative, but some tricks are played (d_name may be longer than 256)

-> d_name is the name within the directory
    -> need to combine with directory name to get a complete path

<dir_name>/<file_name>

-> d_type is a char, but its actual values are implementation specific
    compare values to predefined macros

DT_REG - type of regular files
DT_DIR - type of directory files

directory entries are not the same as file descriptors
- directory entry is information about a file on disk (name, inode number)
- file descriptor is information about an open file
    (where it comes from, current position in file)

file descriptors are specific to a process (running program) directory entries are part of the file system

Note that there are always directory entries . and ..

. - another entry for the directory itself

.. - another entry for the parent directory

Paths on Unix don't do any special magic for . and ..

-> we literally just use the directory entries for them that always exist!

../../foo

this path follows the .. entry to get to the parent,
then follows its .. entry to get to the grandparent,
and then looks up the foo entry

if we think of the file system as a graph, then it is always cyclic
if it is a tree, then it is a tree where we can always follow links up
towards the root

Note that even the root, /, has a .. entry
it points back to the root

(Again, this is specific to Posix. Windows has its own rules.)

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

int main()
{
   DIR *dirp = opendir(".");  // open the current directory
   struct dirent *de;

   while ((de = readdir(dirp))) {
      //puts(de->d_name);
      printf("%lu %d %s\n",
         de->d_ino,
         de->d_type,
         de->d_name);
   }

   closedir(dirp); // should check for failure

   return EXIT_SUCCESS;
}
```

Exercise: recursively list all the children of a directory and its subdirectories
-> watch out for . and .. !

Blocking

Certain IO functions may not return immediately
-> if we are reading from a file, we may have to wait for data from the disk
-> if we are reading from a TTY, we have to wait until the user types something

This is called "blocking"
-> our program calls out to the OS for input, and the OS puts us on hold until the data becomes available

We expect functions like read and write to block in most circumstances

When does read block?
If we ask for data, and none currently available, but the file has not closed
If any data is available, read will return immediately, even if not all the bytes we asked for are present

read(fd, buf, 128);
<- if data is available, will copy up to 128 bytes to buf
<- if we are at the end of the file, returns 0
<- otherwise, it blocks until data becomes available

When we read from a file, we will usually get as much data as we asked for, except for the last read
-> This is because the OS caches the file system in memory, so data is usually waiting for us

When we read from a pipe, we will get data that is available, or block if we need to wait for the other program to write more

When we read from a TTY, we block until the user hits return/enter or ^D
this returns control to us and we get the data that was entered

A TTY will store the text you type in a "line buffer"
Hitting enter adds \n to the end of the buffer and sends it to the program
Hitting ^D sends the buffer to the program without adding any characters
-> thus, if we enter ^D at the start of the line, read will receive 0 bytes and return 0

TTYs are weird, because we can still get more data even after EOF
-> but relying on this can lead to unexpected behavior if stdin is not a TTY

We should assume that read only returns 0 at EOF and stop reading

buffering and syscalls

As we said, syscalls are more expensive than regular function calls
we require a "context switch" in order pass control to the OS
    -> changes permissions on the processor (user mode to supervisor mode)
    -> programs run in virtual memory, OS does not
    -> etc.

Not a huge burden, but it is slower than a regular function call

-> this is why read and write work with a buffer
    -> do a lot of work with a single system call

The major difference between the Posix file functions (that use file descriptors)
and the C file functions (that use FILE*), is that the C functions are buffered

when we call fopen, we get back a FILE *

what is FILE?
    this is a struct defined in stdio.h
    contents vary between compilers, but it generally includes

    - a file descriptor
    - a buffer (char array)
    - index of current byte in buffer
    - other info (e.g., are we at EOF)

On most Posix systems, fopen calls open, fread/fscanf/etc. use the buffer in FILE
    and call read when they need to refresh the buffer

    -> this is why fgetc is more efficient than calling read with a 1-char buffer

        fgetc reads a single character from a file
            -> but it only makes a syscall when the buffer is empty or completely read

```
while ((ch = fgetc(my_file)) != EOF) {
    // do something with ch
}
```

    makes fewer system calls than

```
while ((bytes_read = read(my_fd, &ch, 1)) > 0) {
    // do something with ch
}
```

On a Posix system, we can actually conver between file descriptors and FILE*

    fdopen -> create a FILE * for a file descriptor (add buffering!)

    fileno -> return the file descriptor used by a FILE *

Why is this useful?
    file descriptors can be used to access many things, not just files on disk

- standard input/output/error
- reading from devices in general
- communicating with other processes (e.g., pipes)
- communicating over a network (e.g., sockets)

The reason project 1 requires read/write is to give you a sense of why buffers are convenient

- also, it gives you practice for working with sockets

Read/write give us more control about when data is obtained from/sent to the OS

How big should your buffer be?

the tradeoff is memory use vs how often you call into the OS

- smaller buffer uses less space
- bigger buffer requires fewer syscalls

Common sizes: 64, 128, 256, 1024

but this is arbitrary

using make to run tests

Remember: we can have our recipe run any program

```
test: ww
	./ww 80 input.txt > output.txt
```

Now "make test" will (a) recompile ww if necessary, (b) wrap input.txt, and (c) save the output to "output.txt"

-> this still requires us to examine output.txt after the test

We can create a file with the expected output, e.g., "reference.txt"

We can use cmp or diff to compare the content of two files

cmp will find the first different byte between two files

diff will print out a detailed description of the differences

```
test: ww
	./ww 80 input.txt > output.txt
	cmp reference.text output.txt

test2: ww
	./ww 80 input.txt > output.txt
```

```
		diff reference.text output.txt

	test3: ww
		./ww 80 input.txt | cmp reference.text
```

test3 sends the output of ww directly to cmp without the need for an output file

cmp and diff use exit status to signal whether the files were the same or different
make always checks the exit status and reports recipe failure

if cmp gets one argument, it compares against standard input

use - as an argument to diff to have it read standard input

```
./program | diff reference -
```

pipe output from program to diff and compare against "reference"

We can have multiple tests in our Makefile and have a target that runs all of them

```
	tests: test1 test2 test3 test4

	test1: ww
		./ww 80 input1.txt | diff reference1.txt -
```

exit, _exit, abort

Our program ends when we return from main, value returned from main is the exit status

```
int main(...)_
{
	...
	return EXIT_SUCCESS;   // end process with exit status 0
}
```

But we have a few ways to terminate a process early

exit - terminates our process "normally"
exit status given by argument to exit
exit(EXIT_SUCCESS)
exit(EXIT_FAILURE)

- this will close files & flush FILE * buffers
    - e.g., anything we have written using FILE * that hasn't been sent to OS yet gets sent to OS
    - that is, make sure any pending printfs get to finish
- this will call any "exit handlers" we have registered

when we return from main, the run-time system calls exit next

int atexit(void (*function)(void));

- register a function as an "exit handler"
- when we call exit/return from main, the run-time will call any registered exit handlers

```
void cleanup();

...
{
    ...
    atexit(cleanup);  // register cleanup as an exit handler
        // should check return result to see if it succeeded
}
```

we can use this to allow our program to gracefully close network connections, etc., when it exits

there can be many exit handlers (up to ATEXIT_MAX, which is at least 32)

_exit() is just like exit(), but it does not call the exit handlers and might not flush buffers

```
void _exit(int status);
```

this is "immediate exit"
doesn't call our cleanup functions
argument sets the exit status

abort() is for "abnormal termination"

```
void abort();
```

essentially crashes your program with the SIGABRT signal
(more on signals later)

reports a non-zero exit status

does not do any cleanup
lets OS close files, etc.

does not call exit handlers
-> can be caught if we install a "signal handler" for SIGABRT

if something has gone very wrong, we maybe can't trust the contents of memory, so trying to clean up might cause further problems
-> so just abandon the process and let someone else clean up

For this class, exit() is sufficient, but _exit() and abort() are okay to use
-> be aware of the differences between them

Reminder about paths

-> the file names we give to open, opendir, fopen, etc. are really paths

A path is a sequence of names separated by slashes (/)
Paths starting with / are absolute
Other paths are relative
A relative path says how to reach a file, starting from the current directory (also called the working directory)

Eg., foo/bar/baz

means "baz" inside "bar" inside "foo" in the current directory

Our processes have a current working directory:
the directory the user was in when they started the program
that is, we inherit the working directory from the shell

-> thus, file names are interpreted relative to the directory we started in

How can we open files in another directory?

1. make a path: directory/filename
when reading from a directory, just append a slash and the file name to the name we used to open the directory
-> a little obnoxious, since strcat requires us to allocate space in advance
-> but eminently doable

There is a limit on path length, but this is not generally a problem
-> open will just return -1 if the name is too long
-> the max path length may be very long, so use dynamic (heap) allocation

2. use chdir to change the working directory
now all my filenames are relative to the new directory
no need for concatenating path names

int chdir(char *path);

returns 0 for success
on error, returns -1 and sets errno

--

/dev directory contains "files" that correspond to hardware devices

/dev/null - a device that just throws away its input
- commonly used as a way to throw out/ignore the output from a program

```
./program_with_lots_of_output > /dev/null
```

On Linux, /proc contains an entry for every running process

Processes

What is a "process"?
- for us, a process is a program that is currently executing

process = program + state

state includes contents of memory, registers, open files, OS data

In principle, we can have multiple processes running concurrently that all use the same program

The OS keeps track of each running process
On Unix, we have the Process Control Block
-> data structure used by the OS
- process ID (pid)
- what virtual memory address space we are using
- what files we have open
- environment information (e.g., current working directory)
- permissions, process user/group, etc.

Where do new processes come from?
In general, the OS has some mechanism
In Unix-like systems, new processes are created by fork()

pid_t fork();

pid_t is an integer type (width is OS-dependent)

fork clones or duplicates the current process
-> the original process is called the "parent"
-> the new process is called the "child"

The OS has a special program that starts first (possibly "init")
every other process is the child of some other process

When we call fork, the current process is duplicated
- both duplicates have the same program
- initially have the same memory contents / program state

- but the child gets a copy of the memory
- after fork returns, the parent and child may diverge

What fork returns will be different between the parent and child
-> parent receives the PID of the child
-> child receives 0
(0 is a valid PID, but it always identifies the root process, so it can never by the PID of a child process)

Typical usage

```
pid_t pid = fork();
if (pid < 0) {
	perror("fork");
	abort();
} else if (pid == 0) {
	puts("In child!");
} else {
	printf("In parent! Child is %d\n", pid);
}
```

Remember: both child and parent start with the same program in the same state
(aside from the return value from fork)
we can think of fork as returning twice
-> once to parent
-> once to child
both parent and child resume executing after the return from fork

Fork can fail if the OS (or the user) has too many processes
fork returns -1 and sets errno on failure

What happens when a process ends?
-> who collects our return code?

- Much of the data for our process is forgotten
- The process control block holds our exit status, in case our parent wants it

When a process ends, it becomes a "zombie"
A zombie process has died, but it's PCB still lives in the process list

The parent must call wait() to clean up the zombie process
wait is also how we get the exit status of the child

If a parent process ends before its child, the child becomes an "orphan"
An orphan process has no parent process to wait for it when it ends
-> OS will make sure the orphan is "adopted" by some process (usually init)
- the adopter will wait for the child to terminate

If the parent and child both terminate, the child becomes a "zombie orphan"
-> The OS will have some process adopt the child and wait

Only specialized processes can adopt orphans
- process do not necessarily adopt distant descendents
- there is no "grandparent" relationship

wait
----

pid_t wait(int *wstatus);

wait suspends a program (blocks) until one of its child processes terminates
It returns the PID of the terminated process
If wstatus is not NULL, it writes information about how the process exited
into wstatus
-> exit status
-> whether the process was terminated by a signal

wait (and related functions) are how we prevent orphans

any time we call fork, the parent should call wait

When should we call wait?
It depends on what we're doing

---

fork and exec

Where do processes come from?

Last time, we discussed how to create a process using fork

pid_t fork();

fork duplicates our process, or makes a copy of our process (*mostly)
the new process that is created is running the same program
and has the same contents of memory and registers, including the PC
meaning that the child will behave as though it had been the parent all along

the difference between the child and the parent:
in the parent, fork will return the PID of the child
in the child, fork will return 0

```
pid_t p = getpid();
pid_t c = fork();

// check for failure
if (c == -1) {
    perror("fork");
```

```
        abort();
    }

    if (c == 0) {
        // do the child thing
        printf("I am %d, my parent is %d\n", getpid(), p);
    } else {
        // do the parent thing
        printf("I am %d, my child is %d\n", p, c);

        ...
        wait(NULL);  // don't orphan our child!
    }
```

pid_t getpid(void)
    return PID of the current process

This by itself is not extremely useful
-> we can use this to take advantage of multiple processors
-> or split up work that is IO bound

-> this is called "multiprocessing" (because it involves multiple processes)

We can use functions like pipe to allow our processes to communicate

```
int pipe(int pipefd[2]);
```

-> creates two files (streams) and stores their file descriptors in the array

```
int fd[2];
pipe(fd);   // returns -1 on failure

    // anything written to fd[1] can be read from fd[0]
```

If we call pipe before fork, then both processes will have access to both ends
of the pipe

```
int fd[2];
pipe(fd);  // create a stream with two ends
    // fd[0] - read end
    // fd[1] - write end

pid_t child = fork();

if (child == 0) {
    close(fd[0]); // close the child's copy of the read end
```

```
    write(fd[1], "Hello!", 5);
    close(fd[1]);
    exit(EXIT_SUCCESS);  // stop here; don't do the stuff the parent will do
  }

  close(fd[1]); // close the parent's copy of the write end!
    // we won't get EOF on fd[0] until every copy of fd[1] has been closed

  char buf[100];
  int r;
  while ((r = read(fd[0], buf, 100)) > 0) {
    // do something
  }

  close(fd[0]);  // close read end
  wait(NULL);    // wait for child to exit (prevent zombie orphan)
```

-> for 2-way communication, we want two pipes
  -> having multiple processes writing to or reading from the same file can be unpredictable

But the main reason to use fork is to start a different program

1. use fork to spawn a new process
2. have the child process use exec to switch what program it is running

execl and execv are functions that change the current program
  -> the process stays the same, but its execution state is reset and its code changes to the new program

  -> this stops executing the current program and starts executing a different program
  -> most program attributes are preserved
    -> such as the list of open files
    -> this is also why standard input and standard output are shared with the shell (by default)
      that is, your program inherits files 0, 1, and 2 from the shell

-> this is the mechanism that the shell uses to start programs
-> this is how every program starts
  -> every program except init starts when its parent forks and execs

Both execl and execv specify the program file and the argument list

int execl(char *program, ... /* additional arguments followed by NULL */);

```
  execl("/bin/echo", "/bin/echo", "Hello", "world!", NULL);
    these will become  argv[0]  argv[1]   argv[2]
```

this starts the program /bin/echo and passes 3 arguments (argc = 3)
By convention, the first argument should be the same as the program file

execl takes multiple arguments, each of which is a (terminated) string
the last argument to execl must be NULL
(this is how execl knows to stop looking for additional arguments)

these populate argv

We can use execl to start any program (that we have permission to execute)
we can give execl any strings arguments, including spaces and special characters
(this does not go through the shell!)

memory tip: execl takes a list of arguments in its argument list

execv takes a vector (array) of arguments

int execv(char *program, char **argv)

argv is an array of pointers to terminated strings
the last entry in argv must be NULL

```
char *args[] = {"/bin/echo", "Hello", "world!", NULL};
execv("/bin/echo", args);

    // args will become argv in the new program (argc is derived from it)
```

Choice between execl and execv is just convenience; no other difference

Note: execl and execv replace the current running program!
These functions do not return (unless they failed)
There is no way to resume the current process

Like exit, exec does not return
But fork returns "twice"
-> so we can use fork to spawn a child and have the child exec the program we want

Typical scenario: run another program and wait for it to finish

```
pid_d child = fork();
    // FIXME check for -1

if (child == 0) {
    execv(program_path, args);

    // if we got here, then execv failed
    perror("exec");   // print a message indicating what went wrong
```

```
        abort();
    }

    int wstatus;
    wait(&wstatus);
        // FIXME should check return value

    // wait blocks until a child process halts
    // (wait returns -1 if there are no child processes or it had some other problem)
    // wait will write information about the halted process to wstatus

    if (WEXITSTATUS(wstatus) != 0) {
        // something went wrong with the child
        // WEXITSTATUS() is a macro that extracts the exit code
        // there is other information in wstatus, but we usually don't need it
    }
```

How does the shell do pipes and redirects?
- normally a process started from the shell gets the same stdin and stdout as the shell (e.g., the terminal)
- but if we use file redirection or a pipe, the process gets different stdin and stdout

- is this something we can do ourselves?
    - can we start a process and have its stdin read from a file?
    - can we start a process and read what it writes to stdout?

Recall:
    when we fork, the child gets copies of all the open file entries
        -> not copies of the files, just the file entry
        -> so the parent and child can read/write the same files
    When we exec, the new process will retain its open files (*usually)

If I want to start a process and read what it writes to stdout we can
1. use fork to spawn a child process
2. somehow change file descriptor 0
3. exec the program

int dup2(int oldfd, int newfd);
    this duplicates an open file descriptor
    if it succeeds, oldfd and newfd will refer to the same file
        -> if newfd is already open, it closes it

dup2(x, y);
    now x and y both refer to the same open file

specifically, y now refers to the same file as x

1. use pipe to create a two-ended stream
2. fork
3. in the child, use dup2 to change stdout to be the write end of the pipe
4. in the child, exec the new process
5. in the parent, close the write end of the pipe & read from the read end

```
// FIXME: should check for errors after all of these syscalls

int fd[2];
pid_t child;


pipe(fd);
child = fork();

if (child == 0) {
    close(fd[0]);
    dup2(fd[1], 1);   // make stdout the same as the write end of the pipe
    execv(my_prog, my_args);
}

// in parent
close(fd[1]);

// now we can read from fd[0] whatever my_prog wrote to standard output

...
close(fd[0]);
wait(...);   // always wait once per fork!
```

```
pid_t p1 = fork();    // spawns first child
pid_t p2 = fork();    // spawns second child (from parent) and third child (from first child)
```

-> results in 4 processes
    the original parent
        two children
            one child of the first child

parent
    p1 identifies first child
    p2 identifies second child

first child
    p1 is 0
    p2 identifies third child

second child
   p1 identifies first child
   p2 is 0

third child
   p1 is 0
   p2 is 0

```
   parent (pid 10)
      |
    p1 = fork() --------------------------> first child (pid 11)
      |                                    |
    p2 = fork() ----> second child (pid 12)    p2 = fork() -----> third child (pid 13)
      |               |                    |                 |
    p1: 11          p1: 11               p1: 0             p1: 0
    p2: 12          p2: 0                p2: 13            p2: 0
```

How can we have multiple processes on a single-processor system?

- "time sharing"; some method for switching between running processes
  - cooperative multitasking
    ("task" means "process" in this context)
    each process runs for a bit and then yields control to the OS
    the OS then resumes another process
    problem: uncooperative processes can monopolize the CPU

  - preemptive multitasking
    OS sets up a timer
    each process runs for a short period of time
    CPU interrupts program and returns control to OS
    OS lets the next process have a slice of time

essential difference is who controls when we switch processes
- which is better?
  - cooperative multitasking is vulnerable to bad programs and bugs
    one infinite loop can lock up the whole computer
  - preemptive multitasking requires more hardware support
    preemption may occur at awkward times
    e.g., a real-time system can't predict when it will be preempted

How does preemption occur?

-> Hardware interrupts or "traps"
   there are a bunch of related/similar ideas that have used different
   terms in different contexts, or used the same term in different ways

Basic idea: something happens where the CPU needs to respond to it immediately
  - e.g., data from an IO device arrives
  - run-time exception (division by zero, bad memory access)

- attempt to execute ill-formed/invalid instruction

If something happens, the CPU may interrupt the current process and transfer control to the OS
- current process state is saved
- control switches to OS code at a specified address (a "trap handler")
  -> trap handler will do something in response
  - copy data from IO device to a buffer in memory
  - terminate process with error condition
  - do nothing and resume process

For example, if we try to dereference a bad pointer (e.g., NULL)
    the CPU notices the attempt to read from an invalid address
    it switches to the trap handler for bad address errors
    the trap handler terminates our process with a SEGV signal

Typically, we (user programs) do not ever see traps or set trap handlers
    -> this is reserved for the OS

The OS can set alarms that will trap after some amount of time
    -> e.g, after 200 ms
    the trap handler can suspend the current process and have the scheduler
        resume another process

    -> thus, preemptive multitasking

    preemptive multitasking requires some hardware support, and is a bit more work than cooperative multitasking, but it is generally safer and more predictable

---

Signals

- mechanism for communicating with a running process
- signals are sent to a process (from OS or other processes or the same process)
- they start out as "pending"
- normally, after some short period, they are "delivered" to the process

- for each signal, we declare a "disposition"
  - block the signal (leaves it pending; may get delivered later if signal is unblock)
  - ignore the signal
  - terminate process
  - execute a signal handler

a signal handler is a function that will be called when the signal is delivered

We can declare our own signal handlers using signal

```
#include <signal.h>
```

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

manual for signal function: man 2 signal
manual for signals in general: man 7 signal

signal registers a signal handler
- There is a table somewhere that holds our process's disposition for each signal
- When the OS delivers a signal, it calls the appropriate signal handler

signal(int signal_number, sighandler_t signal_handler)

sighandler_t is usually a void function that takes an int
or it could be:
  SIG_DFL - the default handler for this signal
  SIG_IGN - ignore this signal

When our process starts, it has a default disposition for every signal
- what that will do depends on the specific signal
  - some terminate the process
  - some terminate the process and create a core file
  - some stop the process (but it can be resumed later)
  - some are ignored

man 7 signal lists the default behaviors

To register a signal handler:

```
sighandler_t prev = signal(SIGINT, interrupt_handler);
```

prev is the previous signal handler, or SIG_ERR

-> any time you register a signal, you should check for SIG_ERR

To ignore ^C

```
if (signal(SIGINT, SIG_IGN) == SIG_ERR) {
    // didn't work
}

// now we are immune to ^C
// not usually a good idea (makes it harder for user to stop the program)
// if we block SIGINT, then users will have to use SIGKILL to stop our program
//      SIGKILL is what kill -9  or kill -KILL sends
// we cannot set a handler for SIGKILL, so we can't do any cleanup if we
//    receive it
```

-> common technique is to intercept SIGINT, set a global variable, and then shut down cleanly

notes on signal handlers
-> signal handlers are functions that may be called from anywhere
- normally, the current function is interrupted and the signal handler is added to the call stack as though it had been called
- it is a function call that could happen at any time

-> it generally isn't a good idea to do a lot of work in a signal handler
- not all library functions are safe to call from a signal handler
  - e.g., you might get the signal in the middle of calling printf calling printf again may cause problems
- you may get another signal while you are executing the signal handler
- normally, a signal handler will not be interrupted by itself

signal does not behave consistently across Posix implementations
- the main difference has to do with what happens if you receive the same signal while the signal handler is running
  possibility 1: use the default handler for that signal
  possibility 2: block the signal until the handler completes
- portable code cannot assume which of these is being used
- GCC on the iLab uses #2

For portable code, use sigaction
- Posix (may not be available on non-Posix systems)
- more powerful and flexible
- more work to use

GCC Manual discussing signals
https://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html

Blocking signals
- in addition to signal handlers, our process has a "signal mask"
  - for each signal, the mask says that signal is blocked
  - a blocked signal stays pending; it does not get delivered
  - we can change the mask while we are running
  - if we unblock a signal, and there is a pending message for that signal, we receive it at that time

Waiting for signals

```
int pause(void);
```

pause suspends the current program until a signal is received
it returns the signal that was received, or -1 on error

e.g., if we didn't have sleep, we could implement our own using

```
alarm(some_time);  // set an alarm -> receive SIGALRM when it is up
pause();           // wait until a signal is received
```

-> be sure to override the default behavior for SIGALRM first!

Termination signals
Different signals are used to terminate our process in different circumstances
SIGHUP - "hang up"; sent if the shell that started our process ends
    (e.g., we closed the window or logged out)
SIGINT - "interrupt from keyboard" - user typed ^C
SIGTERM - "terminate", default signal for kill
SIGQUIT - "terminate and dump core"; used for debugging (type ^\)
SIGKILL - "kill", terminate without cleanup
    processes cannot handle or ignore SIGKILL; it always terminates

Stop and continue
SIGSTOP - stop signal; cannot be handled or ignored
SIGTSTP - "typed stop"; sent when user types ^Z; can be handled
SIGCONT - continue signal; sent when resuming a stopped process

Many other signals terminate by default
- usually because our program hit an error and can't safely proceed
  - division by zero
  - illegal memory access
  - malformed instruction
  - others (e.g., arithmetic errors)

- be careful when writing a handler for error condition
  - the error condition will still be there if the handler resumes
  - the only safe thing you can do is terminate the process or jump to somewhere
    else in the program (siglongjmp)

Aside: stopping and restarting processes in the shell

  Use ^Z to stop the current process
    Usually prints a message like: [1] Stopped your_program
    1 is the "job number"
      use jobs to get the current list of jobs
      processes started by the shell have job numbers
      all processes have PIDs
    Stopped is the current state of the process

  To restart the process, use fg or bg
  fg [job number]
    resumes process in the foreground
  bg [job number]
    resumes process in the background (concurrent with shell)

  Either give job number, or leave it out
    defaults to the most recent job

  To start a process in the background, put & at the end of a command

```
$ ./long_process_to_finish &
[1] Running  ./long_process_to_finish
```

    $

kill sends a signal to a process

    kill PID   - send SIGTERM to process PID
    kill -KILL PID - send SIGKILL to process PID

    Use %N to get the PID of job N

```
$ ./long_process &
[1] 10234
$ kill %1
$
[1]+ Done    ./long_process
```

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

volatile int signo = 0;
   // marked volatile because it may change asynchronously
   // that is, signals may be received at any time


// very simple signal handler
// note that it returns normally, so we can only use it with non-error signals
void handler(int signum)
{
   signo = signum;
}

// a very simple exit handler
void make_a_note(void)
{
   puts("We are in the exit handler");
}

int main(int argc, char **argv)
{
   // register an exit handler: make_a_note will be called after main returns
   atexit(make_a_note);

   // register some signal handlers
   // we can reuse handler because it will receive the signal
   signal(SIGHUP, handler);
   signal(SIGINT, handler);
   signal(SIGTERM, handler);
   signal(SIGCONT, handler);
```

```
    pause();  // stops the process until a signal is received
    // alternative:
    // while (signo == 0) { puts("Waiting"); sleep(1); }

    if (signo > 0) psignal(signo, "caught signal");

    return EXIT_SUCCESS;
}
```

Multithreading

We have discussed one method of multitasking:
    multiprocessing - running several processes simultaneously
        (either on separate processors or sharing time on a single processor)

    one limitation of this model is communication
        if two processes want to communicate, we can only do so by sending messages
            -> through the file system
            -> through sockets
            -> through shared pipes
            -> through signals (very limited!)
            -> parent can pass some data via arguments
            -> child can return some (limited!) data via exit status

    one advantage is safety
        separate processes have separate memory spaces
        can start and stop independently
        -> a bug in one process won't take other processes down

e.g., a server environment might start multiple processes
    child process does all the work
    parent process waits for the child process to crash, and restarts it
        -> "nanny" process

But sometimes we want more than this
    -> we might want to open/read from multiple files concurrently
        -> don't want to block the whole program while we wait
    -> we might have a complex computation that we need to perform (e.g., parse XML)
        -> we want our program to remain responsive
        -> we need a way to perform large tasks asynchronously

We want to be able to perform multiple tasks within a single process
    -> so we introduce threads and multithreading

a "thread" is an execution context with in process
    a single process with multiple threads is multithreaded

There are a few ways to do threads

- user threads / "green" threads
  -> language run-time or library allows us to switch between different tasks
  -> no involving the OS
- OS threads / kernel threads
  -> essentially multiple processes running in the same memory area
  -> each "thread" has its own process control block, processor context, stack
    -> all threads share process id

Threads behave like processes in some respects, but not all
exec changes the program running for the entire process (all threads are halted)
fork only duplicates the current thread
signals can be received by any thread
each thread has its own signal mask (which signals are blocked)
-> signals sent to a process will be delivered to any thread that is not blocking that signal
-> at most one thread will be delivered any particular signal
signal handlers are shared among threads

exit(), _exit(), and abort() terminate all threads in the process
returning from main(), all threads are terminated (because it calls exit())

man 7 pthreads lists some attributes that are thread-specific and some that are shared by all threads

So: how can we use threads?
In this course, we will discuss the Posix threading interface ("pthread")

type
pthread_t - thread ID (essentially an int)

We use pthread_create to start a new thread
-> but we need to indicate what that thread should do
-> instead of a fork-like model, we give pthread_create a specific function to execute

A "thread function" is a regular function

```
void *my_thread_function(void *arg);
```

- takes one argument, a pointer
- returns a pointer

```
err = pthread_create(&tid, NULL, my_thread_function, argptr);
```
<- calls my_thread_function "in the background" (asynchronously)
<- pthread_create returns immediately

compare with
```
p = my_thread_function(argptr)
```
<- calls my_thread_function synchronously
<- does not return until my_thread_function is finished

```
int   // returns 0 if successful, or an error code (does not set errno)
pthread_create(
   pthread_t *tid, // id of new thread will be written here
   pthread_attr_t *attrs,  // ptr to struct requesting features
      // or NULL to get the default attributes
   void *(*thread_function)(void *),
      // function that the thread will execute
   void *args  // argument to pass to thread_function
);
```

Why void*?
   This is how we write generic functions in C
   Using void * means we can pass (a pointer to) any value as an argument
      and we can return (a pointer to) any value
   Downside: no type checking; compiler will trust that what we do makes sense

Why have an argument at all?
   We want to start multiple threads with the same function
      -> passing an argument lets us send specific data to each thread

   We can just use NULL if we have nothing useful to pass to the thread

```
pthread_t tid;
err = pthread_create(&tid, NULL, thread_fun, NULL);
if (err != 0) { errno = err; perror("pthread_create"); exit(EXIT_FAILURE); }

// at this point, the thread has started and tid contains its thread id
```

But what about the return value?
   the thread function returns a void pointer, but who gets it?

How do we know when the thread has finished doing its task?

   pthread_join collects the return value of a specified thread

```
int   // returns 0 for success, error number on error
pthread_join(
   pthread_t tid,   // id of the thread we want to join (wait for)
   void **retval    // where to write the void * that the thread returned
                    // or NULL to ignore the return value
)
```

```
pthread_t tid;
struct arg_t *args = ....;
struct retval_t *ret;
```

```
    pthread_create(&tid, NULL, background_task, args);

    // at this point, the thread has started

    // do other things

    pthread_join(tid, &ret);

    // at this point, the thread has stopped and ret points to its return value
    // note: we passed &ret so that join could change what ret points to
```

create and join are analogous to fork and wait

when we join a thread
- join will block until the specified thread has ended
- join returns immediately if the thread has already ended

If we do not join a thread, no one will collect its return value
-> potential memory leak

-> As a rule, any thread we create must be joined
-> any thread can join a thread, not just its creator
-> but only one thread should join

```
// very simple example of splitting a task into multiple threads
// note: code to check errors not included

struct arg {
    int length;
    double *data;
};


// thread function
void *compute_sum(void *argptr)
{
    struct arg *args = (struct arg *) argptr;  // assume argptr has correct type

    int i;
    double sum = 0.0;
    for (i = 0; i < args->length; i++) {
        sum += args->data[i];
    }

    // must return a pointer
    double *retval = malloc(sizeof(double));

    *retval = sum;
```

```
    return retval;
}

// driver function

#define THREADS 5

void compute_sums(double *array, int length)
{
    pthread_t tids[THREADS];   // hold thread ids
    struct arg args[THREADS];  // hold arguments
    double *retval, sum = 0.0;
    int i, start, end;

    // initialize all arguments
    // e.g., we could break an array into equal-sized chunks
    end = 0;
    for (i = 0; i < THREADS; i++) {
        start = end;
        end = length * (i + 1) / THREADS;
        args[i].length = end - start;
        args[i].data = &array[start];
    }

    // start all threads
    for (i = 0; i < THREADS; i++) {
        pthread_create(&tids[i], NULL, compute_sum, &args[i]);
    }

    // we now have THREADS+1 threads running

    // wait for all threads to finish
    for (i = 0; i < THREADS; i++) {
        pthread_join(tids[i], &retval);
            // (only) current thread waits until tid[i] has finished
            // retval will point to the sum
        sum += *retval;
        free(retval);
    }
}
```

What do we need to do to use Pthreads?

In our source code

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

For GCC, add -pthread to command line when compiling
- this tells GCC to link against the PThread library
- this may tell the compiler to use threaded variants of some syntax/functions

How can we coordinate multiple threads?

- For simple programs, we may not need to coordinate
  - each thread works with its own data
  - arguments to the thread / responses from the thread handled with pthread_create and pthread_join

- We can use some facilities for message passing, such as pipes
  - e.g., one thread can use a pipe to send a stream of bytes to another thread

write and read are thread-safe and atomic
  - when writing to/reading from a file, no other thread will be able to use the file until my call is complete
  - if two threads try to write at the same time, one will wait until the other is finished
  - note that only single calls are atomic
    - if I call write twice, another thread might get a write in between

All the FILE* functions are also atomic
  I can call printf from multiple threads without problems
    E.g., each call to printf will complete before the next one can start

But these may not be sufficient for all purposes
  We may want more coordination than simply message passing
  Also, someone had to write printf and write to be thread-safe, but how?

Basic idea: mutual exclusion
  - mutual exclusion is a way to ensure that at most one thread has access to a resource at a time
  - this is the basis for all coordination between threads

```
int bank_balance = 1000;  // global variable

thread 1:
   bank_balance += 100;

thread 2:
   bank_balance -= 50;
```

What is value of bank_balance after both threads run?

  It should be 1050
  It might be 1100
  It might be 950

Problem: the threads used non-atomic operations
  atomic operations are either finished or have not started

non-atomic operations have multiple steps
-> so other threads may run in between steps

bank_balance += 100 is actually three steps
1. read value of bank_balance
2. add 100 to value
3. write new value to bank_balance

Thread 1	Thread 2
Read bank_balance (1000)	
add 100 (1100)	
	Read bank_balance (1000)
Write bank_balance (1100)	
	subtract 50 (950)
	Write bank_balance (950)

Problem arises because
1. increment/decrement are non-atomic operations
2. both threads could access bank_balance simultaneously

This is an example of a data race
-> the output that we get depends on which thread finishes first

To avoid problems, we must make the operation atomic or enforce mutual exclusion

Problem:
- we can't create atomic operations in software
  -> require hardware support to enforce atomic nature
- we can't enforce mutual exclusion without some atomic instructions

Solution:
CPU designs include one or more atomic operations that can be used to build mutual exclusion tools (e.g., locks)

Test-and-set
set a memory location and return the previous value

We can use test-and-set to build a lock

```
int lock; // global variable

void lock() {
    int prev = test_and_set(lock, 1);  // example; not a real function
        // sets lock to 1
        // prev has previous value of lock

    // if it was already locked, we need to wait until someone else unlocks
```

```
        while (prev == 1) {
            prev = test_and_set(lock, 1);
        }

        // once we get here, we know that we closed the lock

    }

        // only safe to call this if we hold the lock
    void unlock() {
        lock = 0;
    }
```

This is what is called a "spin lock"
- we have a loop that does not end until we are the thread that has locked the lock

Safe version of previous example

```
Thread 1:
    lock();
    bank_balance += 100;
    unlock();

Thread 2:
    lock();
    bank_balance -= 50;
    unlock();
```

This enforces mutual exclusion of bank_balance
thread 2 cannot interfere with thread 1, because lock() will not return until after thread 1 calls unlock

This lock/unlock pattern is called a "mutex" (short for "mutual exclusion")

Fetch-and-add
Similar idea, except we add the argument to the value
Still atomic

Compare-and-swap
The most generally useful primitive
we tell it the value we expect to see, and the new value
if the value is correct, it is changed; otherwise it is left alone

We can implement test-and-set and fetch-and-add using a finite number of compare-and-swaps (the reverse is not true)

These are part of the CPU's instruction set
-> the people who write the C standard libary and Pthread library use these

We use the functions provided by pthread.h

Use mutex to create a lock

pthread_mutex_t lock; // a struct or something (abstract)

pthread_mutex_init(&lock, NULL); // initialize lock
// must be called exactly once before the lock can be used

pthread_mutex_lock(&lock); // acquire lock; block until lock becomes available

pthread_mutex_unlock(&lock); // release lock

The example from before, using pthread functions

```
pthread_mutex_t balance_lock = PTHREAD_MUTEX_INITIALIZER;
   // global variable; already initialized


Thread 1:
   pthread_mutex_lock(&balance_lock);
   bank_balance += 100;
   pthread_mutex_unlock(&balance_lock);

Thread 2:
   pthread_mutex_lock(&balance_lock);
   bank_balance -= 50;
   pthread_mutex_unlock(&balance_lock);
```

The rule with a mutex, is that at most one thread can acquire the lock at a time
If a thread tries to lock the mutex, they have to wait

Note that mutex only guarantees mutual exclusion of the lock itself
Nothing stops me from writing code that accesses the resource without locking first

The second rule with a mutex is that only the thread that locked the lock can unlock it

```
int pthread_mutex_init(pthread_mutex_t *mut, pthread_mutex_attr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mut);
int pthread_mutex_unlock(pthread_mutex_t *mut);
int pthread_mutex_destroy(pthread_mutex_t *mut);
```

We always pass a pointer to the mutex object
- -> duplicating the object has undefined result
- -> we don't ever assign to a pthread_mutex_t

These all return 0 on success, non-0 on failure

You can put your mutex object in global space, the heap, or even the stack (if you are careful)

-> pthread_mutex_init does not allocate space

```
pthread_mutex_t *lock;
pthread_mutex_init(lock, NULL);   // undefined behavior! bad pointer! AAAAA!
```

We can use mutex to enforce mutual exclusion, but it only works if we use it correctly
-> mutex doesn't protect you from badly written or malicious code

Initialize once
lock before entering the mutually exclusive part of your code
unlock after exiting the mutually exclusive part of your code
Destroy when you no longer need the lock

---

coordination
----

mutex gives us mutual exclusion for "critical sections"
a critical section is a chunk of code that at most one thread can execute at a time

lock gives us exclusive access
when one thread has the lock, no other thread can acquire it
- threads wait (block) until the lock becomes available
call at start of critical section
unlock releases exclusive access
call at end of critical section
if any threads are waiting, one will wake up and acquire the lock

e.g., global variable with lock

```
pthread_mutex_t balance_lock = PTHREAD_MUTEX_INITIALIZER;
int balance;

{

    pthread_mutex_lock(&balance_lock);     // start critical section
        // we are the only thread that has balance_lock
    balance += deposit;
    pthread_mutex_unlock(&balance_lock);   // end critical section
        // now other threads can acquire balance_lock
}
```

```
// if we only read/write balance in critical sections, then we avoid
// some concurrency bugs (things will behave in a predictable way)
```

A mutex must be initialized exactly once, before you lock

```
int  // 0 for success, non-zero for failure
pthread_mutex_init(
    pthread_mutex_t *mut,        // address of mutex object
    pthread_mutex_attr_t *attr  // set additional attributes (or use NULL)
);

int  // 0 for success, non-zero for failure
pthread_mutex_destroy(
    pthread_mutex_t *mut         // address of mutex object
);



int // 0 for success, non-zero for failure
pthread_mutex_lock(
    pthread_mutex_t *mut   // lock we want to acquire
);

int // 0 for success, non-zero for failure
pthread_mutex_unlock(
    pthread_mutex_t *mut   // lock we want to release
);
```

Only the thread that holds the lock may unlock!
- by default, pthread_mutex_unlock does not check whether it is being called by the correct thread
- think of lock/unlock like braces
    - should not hold the lock for too long
    - make it obvious that you only unlock after lock succeeds

Should check the return value of lock and unlock for errors

conditions are always associated with a lock

pthread_condition_wait
releases the lock
suspends thread until the condition variable is signaled
reacquires the lock

pthread_condition_signal
wakes up one thread waiting for the condition variable
- does nothing if no threads are waiting

```
int  // 0 for success, non-zero for error
```

```
    pthread_cond_wait(
        pthread_cond_t *cond,  // condition variable to wait for
        pthread_mutex_t *mut   // lock to release/reacquire (must be held)
    );


    int
    pthread_cond_signal(
        pthread_cond_t *cond   // condition variable to signal
    );
```

Deadlock

Deadlock occurs when threads are blocked and cannot get unblocked
    -> program is stuck and cannot make progress

Simple example: two resources X and Y and two threads A and B

```
thread A:
    lock X
    lock Y
    do something
    unlock Y
    unlock X

thread B:
    lock Y
    lock X
    do something
    unlock X
    unlock Y
```

Danger scenario (not guaranteed to happen, but not guaranteed not to happen)

```
    A           B
    lock X
                lock Y
    lock Y
    (blocks)
                lock X
                (blocks)
```

Both threads are blocked
    A is waiting for B to release Y
    B is waiting for A to release X

    -> neither thread can advance until the other finishes
    -> thus, neither thread can release the resource

What is required to have deadlock?

1. mutual exclusion
  - it must be possible for a single thread to hold a resource and prevent other threads from obtaining it until they are finished
2. hold and wait
  - must be possible to block while holding exclusive access to something
  - e.g., any time we call printf while we hold a lock
3. no preemption
  - other threads cannot force a thread to give up exclusive access
4. circular wait
  - two or more threads are waiting for each other to give up a resource
    A waiting for B, B waiting for A
    A waiting for B, B waiting for C, C waiting for A
    etc.

Eliminating any of these is sufficient to prevent deadlock
- we can't really give up 1 or 2
- eliminating 3 is hard (what happens to the thread that got preempted?)
- most solutions focus on avoiding circular wait

A partial strategy
  always acquire resources in a fixed order
    e.g., if we need both X and Y, always get them in the same order
    -> prevents the simple scenario described above
  avoid holding multiple locks whenever possible
  -> have a priority ordering of locks
    e.g., it's okay to get exclusive access to stdout if you hold lock X,
      but not vice versa

In general, detecting deadlock in code is hard
- protect yourself by avoiding complicated interactions between threads as much as you can
- if deadlock is possible, it is a bug and should be fixed

-> no scheme can find all deadlocks
-> write carefully and think about possible scenarios

How can we tell if a program is deadlocked?
- no general method
  - how can we tell whether the program is blocked forever or for a long time?
- if you understand how your program works, you can get insight into what is happening

When working on project II, ask yourself what possible ways can your threads run?
- a context switch may happen at any time
- your program should be okay even if the switch happens at the worst possible time

"Monitors" are data structures/interfaces that can avoid or detect deadlock (relating

to their own use)
    - we will talk more about these later

Barriers

A way of enforcing a different kind of coordination between threads

A barrier has a specific number
    threads can wait at a barrier
    once the specified number of threads are waiting, all the threads resume

https://man7.org/linux/man-pages/man3/pthread_barrier_init.3p.html
https://man7.org/linux/man-pages/man3/pthread_barrier_wait.3p.html

pthread_barrier_t

```
int   // 0 for success, non-zero for error
pthread_barrier_init(
    pthread_barrier_t *barrier,    // barrier to initialize
    pthread_barrierattr_t *attr,  // configuration options, or NULL for defaults
    unsigned count
)

int  // 0 for success, non-zero for error
pthread_barrier_destroy(
    pthread_barrier_t *barrier
)

int   // 0 or PTHREAD_BARRIER_SERIAL_THREAD for success, anything else for error
pthread_barrier_wait(
    pthread_barrier_t *barrier   // barrier to wait at
)
```

    - exactly one thread gets PTHREAD_BARRIER_SERIAL_THREAD
        - unspecified which thread gets it
    - every other thread gets 0

Another way to create a "rendezvous"
    i.e., coordinate threads to make sure they reach a point before they continue

Example
    - we have N worker threads
    - each worker thread has to set up some data before it can start working
    - all the threads must be ready before any of them can start

We could do this with a mutex and condition variable
    - keep track of threads that have finished setting up

- last thread broadcasts to all the other threads

But this is what barriers are for
- main thread creates barrier for N threads

```
pthread_barrier_init(&worker_bar, NULL, N);
for (i = 0; i < N; ++i)
    pthread_create(&tid[i], NULL, worker, &arg[i]);
```

```
void *worker(void *arg)
{
    //  do set up stuff

    err = pthread_barrier_wait(&worker_bar);
    if (err) ...

    // do coordinated work
    // we won't get past the barrier until all the worker threads have finished
    // setting up
}
```

Semaphores
----

optional reading:

The Little Book of Semaphores
https://greenteapress.com/wp/semaphores/

The "original" synchronization mechanism
- all other mechanisms can be made using one or more semaphores
- very general, so compiler/library/runtime cannot optimize as much

Two basic operations
- several names, none of which are completely intuitive
- original names: P and V
  (may be derived from Dutch railway terminology)

Idea: we have an integer
- threads can (safely) increase it or decrease it
- a thread that tries to decrease it below zero blocks until some other thread increases it

Operations
- create / initialize
  set initial number
- P / wait / decrease
  reduce integer, or wait until it becomes non-negative
- V / post / increase

increase integer

We can use these as a mutex

```
pthread_mutex_init    -  sem_init 1
pthread_mutex_lock    -  sem_wait   (reduce by 1)
pthread_mutex_unlock  -  sem_post   (increase by 1)
```

sem X initially 1

```
thread A            thread B
wait X
   X <- 0
   does not block

                 wait X
                 |   X already 0
                 |   blocks
                                                  |
...                                               |
                                                  |
post X                                            |
   X <- 1           -> now wait can finish
                    X <- 0


                 ...

                 post X
                    X <- 1
```

"Binary" semaphore: value is always 0 or 1
General semaphore: value can be any non-negative integer

We can use a semaphore to simulate a mutex
- -> mutex is more restricted
  - -> only the thread that locked can unlock
- -> semaphore is more general
  - -> any thread can post at any time

Turnstile pattern -> wait followed by post

```
wait X
post X
   <- we pass through this immediately if X > 0
```

If some thread waits (reducing X to 0), then no other thread can pass through
the semaphore

What about general semaphores?

- we can think of these as representing the amount of resources available
    - wait claims a resource
    - post releases a resource

bounded queue can be done with 3 semaphores, or 2 semaphores + 1 mutex

Queue:

```
int array[QSIZE];
sem_t open;            // spaces available to write
sem_t used;            // items available to read
    // open + used == QSIZE
pthread_mutex_t mut;   // gives us mutual exclusion (could use another semaphore)
unsigned head;
unsigned count;
```

enqueue

```
sem_wait(open);    // claim one open space, or wait until one is available

lock(mut);
    i = (head + count) % QSIZE;
    array[i] = item;
unlock(mut);

sem_post(used);   // indicate one more item is ready to be dequeued
```

dequeue

```
sem_wait(used);  // claim one item in queue, or wait until one is available

lock(mut);
    ret = array[head];
    head = (head + 1) % QSIZE;
unlock(mut);

sem_post(open);  // indicate one more open space is available
```

unbounded stack

```
struct node *head;  // initially NULL
sem_t available;   // number of items available (initially 0)
```

```
    sem_t lock;        // mutex (initially 1) (could use pthread_mutex_t instead)

push
    struct node *new = malloc(sizeof(struct node));

    sem_wait(lock);
        new->next = head;
        head = new;
    sem_post(lock);

    sem_post(available);   // indicate that another item is available

pop
    sem_wait(available);  // claim an item or wait until one is available

    sem_wait(lock);
        head = head->next;
            // NOTE: we don't need to check whether head is NULL; why?
    sem_post(lock);



sem_t   // semaphore type (a struct; do not duplicate)

int  // 0 for success, -1 (and set errno) for failure
sem_init(
    sem_t *sem,
    int pshared,         // whether or not it is shared between processes
                         // always 0 unless you are using shared memory regions
    unsigned int value   // initial value of semaphore
);

int  // 0 for success, -1 (and set errno) for failure
sem_destroy(sem_t *sem);

int  //   0 for success, -1 (and set errno) for failure
sem_wait(sem_t *sem);

int  // 0 for success, -1 (and set errno) for failure
sem_post(sem_t *sem);


// named semaphores exist outside an individual process
//   essentially files that contain an integer, with atomic increment/decrement
// not much call for these, but useful to coordinate multiple processes
// man 7 sem_overview explains what sorts of names you may use

// open an existing named (persistent, multi-process) semaphore
sem_t *
sem_open(
    char *path,  // essentially a file name
```

```
    int oflag    // same as flags to open(), excluding O_CREAT
);


// open or create named semaphore
sem_t *
sem_open(
    char *path,         // name of semaphore
    int oflag,          // should include O_CREAT, optionally O_EXCL
    mode_t mode,        // permissions (if creating new)
    unsigned int value  // initial value (if creating new)
);

int  // 0 for success, -1 (and set errno) for failure
sem_close(sem_t *sem);
```

Command-line tricks

"file globs"

* -> matches any sequence of zero or more characters
If we use * in a command, the shell will look for files matching this pattern

*.txt -> replaced with the names of all files that end in .txt
foo*bar -> replaced with the names of all files that begin with foo and end with bar

? -> matches one character (wildcard)

    section??.txt -> matches section01.txt and section99.txt,
        but not section1.txt or section100.txt

By default, globs work in one directory (the working directory)
With /, we search in different directories

```
*.c   <- matches all C source files in the current directory
src/*.c  <- all C source files in the subdirectory src
../*.c   <- all C source files in the parent directory

*/*.c   <- all C source files in any subdirectory
../*/*.c   <- all C source files in any subdirectory of the parent directory

*.c
*/*.c
*/*/*.c

../src/module*/*.c
/usr/include/lib*/*.h
```

The shell replaces globs with lists of file names

If we type this
mv *.h include

The shell replaces it with
mv foo.h bar.h baz.h include

mv *.c *.bak <- sadly not possible
<- mv never sees the globs, so it can't tell what we want to do

Shell scripts

Just bunch of shell commands in a (text) file
Mark as a executable
begin with #! followed by a path to the interpreter

Executing a shell script is just as if you had entered the commands yourself

```
---
#!/bin/bash

cp *.h backups
cp *.c backups
---
```

./mybackup.sh

Useful Unix commands

cat
- "concatenate"
- reads one or more files and prints to stdout

cat section*.txt
- combine all files whose names match the pattern and print

cat section*.txt > output
- combine all files and write to a file named "output"

more
- like cat, but pauses each time the screen fills up
- related program: less
  - does the same thing, but is fancier

- can give multiple arguments, or no arguments to read from stdin

-> we can pipe the output of a program to more, and let more present it one screenful at a time

ps -ef <- dumps information about all running processes
ps -ef | more <- dumps the same information, one screen at a time

head [-number of lines] [files...]
- prints the first few lines of a file (or stdin)
- can optionally specify the number of lines (e.g. -20 to get 20 lines)

tail [-number of lines] [files...]
- prints the last few lines

Recall: ls -ltr lists all files in reverse chronological order
    ls -ltr | tail   <- only prints the last few
    ls -ltr | tail -1   <- only print the most recently modified file

file [files...]
- guess what kind of data a file contains
- uses a variety of methods, including looking for format markers
  and doing statistical analysis
- purely for your assistance; no programs depend on this output
- does not look at file names (extensions mean nothing)

wc [options] [files...]
- "word count"
- counts the number of characters, words, and lines in a file or files
- can request only certain counts (e.g., -l for just the number of lines)
- reads from stdin if no arguments

sort [files...]
- sorts lines alphabetically
  - concatenates all input files, and/or reads from stdin

uniq
- reads its input, but skips any line that is the same as the previous line
- read from multiple files and/or stdin
- use -c to print the number of times a line is repeated

grep [options] [pattern] [files...]

- multi-file text search using regular expressions
  - look for lines matching the pattern and print them
- can read from multiple files and/or stdin
  -> can use grep to search the output of a program
    ps -ef | grep lab
    ps -ef | grep -v root

regular expression syntax
  * repeat zero or more times
  + repeat one or more times
  . wildcard
  [abcd] match any listed character
  [a-z]   match any listed character in the given range
  ^ match start of line

$ match end of line

[0-9]+ match any sequence of decimal digits

-> not the same syntax as file globs!

-v negates the grep (print the lines that do not match)
-c counts the number of lines that matched (per file)
-n prints file name and line number before the match

-> many options! so many!

cmp [file1] [file2]
    says whether two files are the same
        EXIT_SUCCESS and no output if the same
        EXIT_FAILURE and some output if different

diff [file1] [file2]
    compare two files and print all lines that are different

    -> frequently used to compare versions of a source file

    -> can use this to compare the output of a program to its expected output

        ./prog | diff expected -

    - many options to control what is considered a difference and how it is reported

worth looking into: sed, awk

ps
    - lists running processes
    - use -e to get all processes
    - many options to get more details, such as -f

top
    - lists all running processes, sorted by CPU use
    - updates the screen: live listing

Networks & inter-process communication

broadly: how do we send a message from one computer to another?
    - more precisely, from one process to another

there are many ways for processes on the same computer to communicate
    -> file system
    -> pipes
    -> use the OS to pass messages
    -> shared memory

usually require both processes to be on the same computer
    frequently, one process must be the parent of another

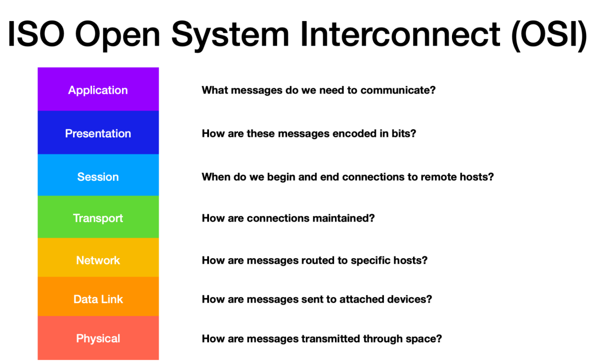
We want communication between processes
- running on different devices (no shared memory, disks, etc.)
- started independently of each other

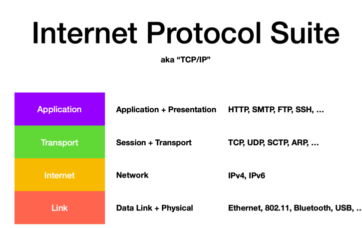
Questions
- how is the communication organized
  - send individual messages?
  - stream of bytes?
- how are messages transferred from one process to the other?
- how do the processes identify each other?

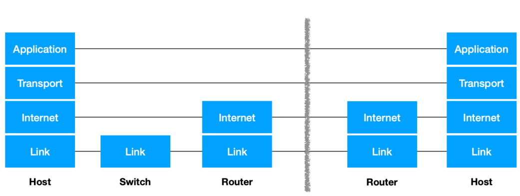
Any networking system must be able to answer these questions
- there have been many networking systems that have had different answers

Typical designs are layered
    different subsystems have different areas of responsibility
    standard interfaces used to access these systems
        -> sockets are one such interface

socket interface is very general
    -> we can use the same interface for many different kinds of network
    -> a typical program will only use a small number of networks (e.g., one)







Questions for networking

- how do we identify the hosts/processes communicating
- how do messages get from their source to their destination
- how do different machines send messages without interfering

Each level answers this question differently

Link level
- typical examples: Ethernet, Wi-Fi (802.11)
    - less typical examples: ATM, Token ring, AppleTalk, Novell
    - Bluetooth can also be considered link-level

Ethernet
- addresses: MAC (must be unique within a network)
- packet based
    - communication based on packets
    - each packet identifies the machine it is intended for
- "best effort" delivery
    - packets can get lost
- does not have "quality of service" guarantees
    - no way to reserve time on the network

Hub and spoke model

Each host on a link connects to a hub
-> one hub, many hosts
The hub forwards every message to every host
Hosts ignore messages not intended for them
-> not secure at all; Ethernet assumes all hosts on the network are trusted

We can connect hubs to other hubs to increase the size of our network
-> this does not scale well to large networks

Switch

optimized hub
-> only sends messages to some hosts
-> more expensive/complicated than a hub
divides link into sub-links
-> learns which MAC addresses are present in which sub-link
messages are only sent to the sub-link that contains the destination

Routers - network-level connections

-> a host on a link that connects to an outside network
-> a connection point between two links
-> can even connect different link types
-> wi-fi to ethernet
-> ethernet to DSL/cable modem/etc.
-> etc.
-> behaves a lot like a switch

Hub vs switch - same level, but have different complexity
Switch vs router - different levels (link vs network)

Internet level

- identifies hosts using IP addresses
  - IPv4 address is 4 bytes (32 bits)
    - usually written in "dotted quad" form: each byte written in decimal separated by periods (eg., 127.0.0.1)
  - IPv6 address is 16 bytes (128 bits)
- most IP addresses refer to specific hosts
  - some hosts may have multiple IP addresses (rare)
  - some IP addresses have special meaning

- IP addresses are obtained in blocks with a common prefix
  - these blocks are assigned to organizations by various international groups
  - ultimately, ICANN is responsible for distributing IP addresses

IP addresses with a common prefix controlled by a single organization are called a "subnet"
We can identify a subnet by writing a slash and the number of bits after

the IP address

a.b.c.d/p
the first p bits identify the subnet
the last 32-p bits identify a specific host within the subnet

33.54.120.240/16
33.54.x.x <- subnet identifier / subnet mask

Routers use subnet masks to simplify routing tables
-> all IP addresses in the same subnet will be routed the same way
"classless internet routing"

IPv4 is still the dominant Internet standard
2^32 possible addresses (roughly 10^9)
-> less than the population of the earth
-> not enough in a world where everyone has â‰¥1 Internet device

IPv6 is the intended solution
-> increases number of addresses to 2^128 (approx. 10^36)
-> has not been widely adopted

What we actually did was introduce network address translation (NAT)
share a single IP address between multiple devices
devices "behind" the NAT use a range of "local"/"private" IP addresses
-> addresses that are not globally unique / cannot be routed
NAT device translates addresses in messages intended for local hosts

NAT effectively live at the Transport layer
may have to keep track of multiple on-going connections and forward packets to the appropriate local device

-> it is hard to run a server behind a NAT
the NAT must know which local device will handle a connection request
-> non-connection-based protocols (UDP) require special knowlege on the NAT

As far as the larger Internet can tell, all devices behind the NAT are
the same host

-> NAT is similar to a firewall, but is not the same thing
-> a "firewall" is something that blocks certain communications
-> typically exist at the router level
-> block attempts to connect to specific services

Transport layer - process to process
- TCP and UDP: processes are identified by host address and port number

44.55.66.77:80

indicates port 80 on host 44.55.66.77

- specific services are associated with port numbers
  - certain port numbers are standard (e.g., HTTP is 80)
  - ports above 5000 are usually free for personal use

-> port numbers are mapped to processes within a host
-> each port is used by a single process
-> cannot have two processes on a single port
-> attempting to "bind" a port that is already in use will fail

-> nothing in TCP or UDP requires any specific port to be used for any
specific purpose
-> the "standard" ports are just suggestions, but going along with
conventions leads to fewer surprises

Typical TCP scenario
client connects to server at well-known port number
client is given an arbitrary port number at its end

Both ends are identified by host address + port number

How do we actually open a connection?
-> socket interface (originally for BSD Unix, now part of Posix)

#include <sys/socket.h>

socket - creates an entry in the file table that we can use to connect over
a network

```
int   // returns a file descriptor, or -1 on failure (sets errno)
socket(
    int domain,     // what kind of network/protocol family
                    // typical: AF_INET, AF_INET6
```

```
        int type,       // what are the semantics of the socket
                        // SOCK_STREAM  - this is a streaming socket (TCP)
                        // SOCK_DGRAM   - this is a datagram socket (UDP)
        int protocol    // identify the protocol, if more than one have the same
                        // semantics (almost always 0)
    );
```

connect - establish a streaming connection to a remote host
- only used with connection-oriented protocols (TCP)

```
    int    // 0 for success, -1 for failure
    connect(
        int socket,
        struct sockaddr *address,  // e.g., IP address + port
        socklen_t address_size     // size (in bytes) of *address
    );
```

-> struct sockaddr is a lie; essentially acts a void pointer
each domain has its own socket address struct (e.g., sockaddr_inet)
the pointer we pass has to be to one of those
-> we specify the size, because different domains have different-sized addresses
IPv6 address larger than IPv4

How do we get the struct sockaddr for our specific domain?

We can construct it manually, but doing so is difficult
-> we need to be concerned about big- vs little-endian data
-> IP addresses and port numbers must be given in "network order" (big-endian)
hton() converts host-order integers to network order

-> we also need to look up the address of the remote host
hosts are usually identified by domain name (candle.cs.rutgers.edu)
domain name service (DNS) maps domain names to specific IP addresses

solution:
getaddrinfo()
-> we specify domain name or ip address + service name or port number
-> it gives us the sockaddr with everything set up
-> it is a little cumbersome, but it is the recommended way to do this

I will post some sample code with comments explaining how to use getaddrinfo

```
#include <netdb.h>

    struct addrinfo {
        int         ai_flags;
        int         ai_family;    // used with socket
        int         ai_socktype;  // used with socket
        int         ai_protocol;  // used with socket
```

```
        socklen_t        ai_addrlen;   // used with bind/connect
        struct sockaddr *ai_addr;      // used with bind/connect
        char            *ai_canonname;
        struct addrinfo *ai_next;
};
```

```
int    // 0 for success, non-zero for failure
getaddrinfo(
        char *host,            // e.g., domain name or IP address (dotted quad)
        char *service,         // e.g., port number or service name
        struct addrinfo *hints,  // additional information (narrows down what we get)
        struct addrinfo **list   // output var: list will point to a linked list
                                 // of struct addrinfo nodes
);
```

linked list is used in case there are multiple ways to connect to the remote host (e.g., if both IPv4 and IPv6 are available)

typical use: iterate through list until we successfully connect or bind

use freeaddrinfo() to deallocate the list once you are done with it

Basics (see sample code for a more complete version)

To connect to a remote host using TCP

```
int sock, err;
struct addrinfo hints, *info;

// initialize hints
memset(&hints, 0, sizeof(struct addrinfo));  // initialize all bytes to 0
hints.ai_family   = AF_UNSPEC;    // allow multiple domains (AF_INET and AF_INET6)
hints.ai_socktype = SOCK_STREAM;  // we want a streaming connection


err = getaddrinfo(remote_host, service, &hints, &info);
        // getaddrinfo("www.rutgers.edu", "http", &hints, &info)
        // getaddrinfo("candle.cs.rutgers.edu", "22", &hints, &info)
if (err != 0) ...

// now info points to a linked list of addresses

// create socket
sock = socket(info->ai_family, info->ai_socktype, info->ai_protocol);
if (sock < 0) ...

// connect to remote host
err = connect(sock, info->ai_addr, info->ai_addrlen);
if (err) ...
```

```
freeaddrinfo(info);

write(sock, "Hello\n", 6);
```

To accept incoming connections

```
int listener, connection, err;
struct addrinfo hints, *info;
struct sockaddr_storage remote_addr;  // as big as the largest supported address struct
socklen_t remote_addrlen;

// initialize hints
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family   = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hitns.ai_flags    = AI_PASSIVE;   // indicate we are going to use listen()

err = getaddrinfo(NULL, service, &hints, &info);
        // NULL addresss means we want a port on our own host
if (err != 0) ...

listener = socket(info->ai_family, info->ai_socktype, info->ai_protocol);
if (listener < 0) ...

// associate this socket with a port
err = bind(listener, info->ai_addr, info->ai_addrlen);
if (err != 0) ...

// set up queue of incoming connection requests
err = listen(listener, queue_length);
if (err != 0) ...

freeaddrinfo(info);

// wait for an incoming connection request
connection = accept(listener, (struct sockaddr *) &remote_addr, &remote_addrlen);
                        // accept(listener, NULL, NULL)
if (connection < 0) ...

        // use getnameinfo() to convert remote_addr back to human-readable strings

read(connection, buf, buflen);
write(connection, response, responselen);
```

Opening a connection:

getaddrinfo() to get a list of addrinfo structs

for each struct,
  create a socket using the provided fields

ai_family, ai_socktype, ai_protocol
socket()
attempt to connect to the remote host/service
ai_addr, ai_addrlen
connect()
- returns 0 for success

if connect() succeeds, then we can use the socket as a file descriptor

read()
write()

more powerful variants
recv()
send()

Note: TCP connections are "full duplex"
two streams:
client sends bytes to server
server sends bytes to client

To open a socket and wait for incoming connection requests

```
char *port = ...;
struct addrinfo hints, *info_list, *info;
memset(&hints, 0, sizeof(struct_addrinfo)); // set all bytes to 0
hints.ai_family = AF_UNSPEC;     // we want IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // we want a TCP connection
hints.ai_flags = AI_PASSIVE;     // we will want to listen

error = getaddrinfo(NULL, port, &hints, &info_list);
   // NULL because we want a port on this host
   // port specifies what port we want (e.g., "5050")
   // info_list will point to the head of the linked list of results

for each addrinfo struct
   listener = socket(info->ai_family, info->ai_socktype, info->ai_protocol);
      // create our listening socket

   error = bind(listener, info->ai_addr, info->ai_addrlen);
      // associates the socket with the specified port
      // will fail if port is unavailable or in use

   error = listen(listener, queue_length);
      // set up socket to accept incoming connections
      // queue_length is mostly arbitrary (e.g., using 5 or 8 is usually okay)
```

if we succeeded in binding and listening to the socket, we can wait
for incoming connection requests

    connection = accept(listener, NULL, NULL);
        // blocks until a remote host (client) tries to connect to our port (using TCP)
        // connection is a new file descriptor/socket
        //   it is specific to this connection

    once we have accepted a connection, we use read() and write() and
        eventually close()

    to get the next incoming connection request, we have to call accept() again

read() and write() work with sockets similarly to how they work with files
-> you need to be a little more careful about blocking

    recall:
        we give read a buffer and a requested (maximum) number of bytes

            bytes = read(connection, buffer, BUFFER_SIZE);

        bytes will contain the actual number of bytes read from the socket
        or 0 if the socket has closed
        or -1 if something went wrong

    read blocks if no data is currently available

    the network stack maintains a buffer of bytes that have arrived but
    have not been read

    when we call read, we get as much data as is available (up to our maximum)
        if we request 10 bytes, but only 2 bytes have arrived,
        read gives us two bytes

    if no bytes are available, read blocks until data arrives or the connection
        is closed (by the remote host)

NOTE WELL:
    TCP gives us a stream of bytes, not messages
    there is no guarantee that I will get a complete message in a single read
    there is no guarantee that a single read will contain only one message

From TCP's perspective, a session with our client is two uninterrupted streams

stream from client to server is
    "GET\n3\nday\nSET\n11\nday\nSunday\nGET\n6\na\nb c\n"

challenge:
    server and client need to respond to messages as they arrive
    * server won't send response until it has a complete request
    * client may not send next request until it gets a response

if we are not careful, the client or server may block on a call to read()

while the other party is waiting for more information

This is why the Project III protocol specifies the message length and includes an end-of-message sequence (the final \n)

-> server must not try to read past the end of a message
-> may wait forever, because client might not send more data until it gets a response

You can use read()/write() for your server

or, we can use C's formatted I/O

```
FILE *fp = fdopen(file_descriptor, mode);
   // fdopen() creates a FILE struct for an existing file descriptor
   //    works for files, pipes, sockets, etc.
   // fp refers to the same file as file_descriptor
   // now we use fprintf(), fscanf(), getc(), fputc(), etc.
   // fclose() will close the underlying file descriptor
```

but, we need to read and write this socket
using files in read-write mode is tricky

solution: use dup() to create a second file descriptor for the socket

```
FILE *fin = fdopen(dup(connection), "r");  // copy socket & open in read mode
FILE *fout = fdopen(connection "w")  // open in write mode
```

We can use getc() to get individual bytes from the socket

```
int c = getc(fin);
if (c == EOF) ....
```

We can use fscanf() to read and parse integers

```
fields = fscanf(fin, "%d", &len);
if (fields != 1) ...
```

We can use fprintf() to write to our socket

```
fprintf(fout, "GET\n%d\n%s\n", strlen(key)+1, key);
```

When we are done, we fclose() the FILEs

```
fclose(fin);
fclose(fout);
```

If you don't want to mess around with fdopen(),
    it is also okay to just call read() and request 1 byte at a time

    bytes = read(connection, &some_char, 1);

In POSIX C, we have two families of IO operations

Posix operations / syscalls / non-buffered IO

    read() and write() (also send() and recv())

    - low-level calls
    - provide the same interface for files, sockets, pipes, etc
    - precise control
        - the data I send using write() is sent to the OS / no longer my responsibility
        - when I receive data using read(), I can set a specific maximum

    -> no built-in formatted IO
        printing a decimal integer requires allocating a local buffer
        and either writing a function or using sprintf()

C operations / buffered IO

    fread(), fwrite(), getc(), putc()
    fprintf(), fscanf()

    these maintain a local buffer
        data written using putc() and fprintf() may not get sent immediately
            to the OS

        when we call getc() or fscanf(), the library may prefetch data and
            store it in the buffer

        -> less control about when data is sent and received
        -> potentially fewer system calls (= better performance)
            -> calling getc() in a loop is far more efficient than calling
                read(fd, &ch, 1)

    If you want to use buffered IO with a socket, you will probably want
        separate read and write buffers

    FILE *fp = fdopen(socket, "r+")
        // create a FILE for a socket in read/write mode
        // only has one buffer, shared by reading and writing operations
            // after we write, we must fflush() before reading
            // after we read, must reset the file pointer before writing

    Instead, when using buffered IO on a socket, duplicate the socket and
        create two separate FILEs

int sock2 = dup(sock);
// should confirm sock2 != -1
FILE *fin = fdopen(sock, "r");
FILE *fout = fdopen(sock2, "w");
// should confirm fin != NULL and fout != NULL

when you call fclose(), it calls close() on the file descriptor

fclose(fin);
fclose(fout);

fscanf(fin, "%d" &len);
// reads bytes until the first non-digit

Note: sockets do not have all the features of files
-> we can't skip around; no way to skip forward or back
-> we don't necessarily know how many bytes we have written

recall: writing using buffered IO does not necessarily send data to the OS immediately

fflush(fout);   // sends anything in the buffer to the OS

fprintf(fout, "OKG\n%d\n%s\n", strlen(value)+1, value);
// puts data into the buffer
fflush(fout);
// sends buffer contents to OS -> to client

Sockets do not behave exactly like files
read() with files usually gives us all the data we want
read() with sockets gives us all the data currently available
-> may be less than we asked for
-> we have no control over how quickly data arrives from the other party
-> if the connection is bad, we could get data 1-byte at a time!

write() with files pretty much always writes all the bytes given
write() with sockets usually does, but might not
-> other party may have closed the connection
-> we could have been interrupted by a signal

-> for maximum safety, any time we call write() we need to check how much was actually written, and possibly call write() again to rewrite the remainder

char *buf;
int buflen, bytes, written = 0;

```
    while (written < buflen) {
        bytes = write(fd, buf + written, buflen - written);
        if (bytes < 1) { some sort of error handling }

        written += bytes;
    }
```

-> we don't usually need to do this, because write() usually writes everything
-> we should still check the return value in case we were interrupted by a signal, or the connection closed/file became unwritable

Fuzzy thinking about sockets
* our protocol is described in terms of messages
    clients sends a request
    server sends a response

* the TCP model gives us two streams of bytes
    TCP does not guarantee that we will get a whole message at once
    TCP does not guarantee a break between messages

When we read, we don't want to read too much
- if the client sent another request without waiting for our response, we could read part of the second message
    -> now we have to hold onto it until we finish dealing with the first one

- if we ask for more bytes than the client sent, but the client is waiting for our response before it sends any more, then we deadlock
        - server is waiting for the client
        - client is waiting for server
    - buggy client sends message that is too short
    - buggy server asks for more bytes than it should

    - we defend against server bugs by writing good code
    - we defend against short messages from the client by looking for the terminating newline

Multithreading in servers
----

echos.c, and your Project III, start a thread for each connection request

```
loop {

    struct arg_t args = malloc(sizeof(struct arg_t));

    connection_fd = accept(listening_fd, NULL, NULL);
        // block until a remote host tries to connect
```

```
    if (connection_fd < 0) ...

    args->fd = connection_fd;


    err = pthread_create(&worker_id, NULL, worker_function, args);
        // worker_fun will eventually close connection_fd and free args
    if (err) ...

    pthread_detach(worker_id);
        // thread will clean up after itself; no return value
        // the main thread doesn't need to remember how many threads are
        // running, and does not need to join them
}
```

As written, the only way to stop this server is to terminate it with
   SIGINT, SIGTERM, SIGKILL, etc
   -> server shuts down immediately, closes all sockets, terminates all
      threads

How can we make this cleaner?
- we can use a signal handler to catch SIGINT and/or SIGKILL, etc.
- many blocking system calls will return early if you receive a signal
   while you are blocked
   - they may also be set to auto-resume by default

One possible strategy
   - install a signal handler
   - have some flag that indicates whether the signal has been received
   - accept() will return -1 if it is interrupted by a signal
   - have the main loop exit if the signal was received

```
    while (running) {
        fd = accept(listener_fd, &addr, &addrlen);
        if (fd < 0) continue;

        .. start up thread to handle connection
    }

    // we don't get here until after the signal arrives
```

A few points to consider
- when we return from main(), the whole process stops and all threads are
   terminated
- if we call pthread_exit(), only that thread ends
   - if the main thread exits, the process won't terminate until every
      thread is finished

- any exit handlers will get called after the last thread finishes
  e.g.,
  atexit(cleanup_data);

- signals are sent to any thread that is not blocking the signal
  - each thread has its own signal mask (= which signals are blocked)
  - all threads have the same signal dispositions (SIG_IGN, SIG_DEF, functions)
  - accept() will only break out of the block if the main thread gets the
    signal
  - we need to arrange things so that only the main thread receives the
    signals of interest
  - when a thread is created, it inherits its parent's signal mask
  - so:
    block SIGINT
    spawn child thread
    unblock SIGINT