Data representation 2: Object representation

Abstract machine and hardware

Programming involves turning an idea into hardware instructions. This transformation happens in multiple steps, some you control and some controlled by other programs!

First you have an *idea*, like "I want to make a bouncy ball iPhone game." The computer can't (yet) understand that idea. So you transform the idea into a *program*, written in some *programming language*. This process is called *programming*.

A C++ program actually runs on an *abstract machine*. The behavior of this machine is defined by the C++ standard, a technical document. This document is supposed to be so precisely written as to have an exact mathematical meaning, defining exactly how every C++ program behaves. But the document can't run programs!

C++ programs are meant to run on *hardware*, and the hardware determines what behavior we see. Mapping abstract machine behavior to instructions on real hardware is the task of the C++ **compiler** (and the standard library and operating system). A C++ compiler is correct if and only if it translates each correct program to instructions that simulate the expected behavior of the abstract machine.

This same rough series of transformations happens for any programming language, although some languages use *interpreters* rather than compilers.

Objects

The C++ abstract machine concerns the construction and modification of *objects*. An object is a region of memory that contains a value, such as the integer 12. (Specifically, "a region of data storage in the execution environment, the contents of which can represent values".) Consider:

```
char global_ch = 65;
const char const_global_ch = 66;

void f() {
    char local_ch = 67;
    char* allocated_ch = new char(68)
    // C-style: `allocated_ch = (char*) malloc(sizeof(char)); *allocated_ch = 68;`
}
```

There are five objects here:

- `global_ch`
- `const_global_ch`
- `local_ch`
- `allocated_ch`
- the anonymous memory allocated by `new char` and accessed by `*allocated_ch`

Objects never overlap: the C abstract machine requires that each of these objects occupies distinct memory.

Each object has a *lifetime*, which is called *storage duration* by the standard. There are three different kinds of lifetime.

- **static** lifetime: The object lasts as long as the program runs. (Example: `global_ch`, `const_global_ch`)

- **automatic** lifetime: The compiler allocates and destroys the object automatically. (`local_ch`, `allocated_ch`)
- **dynamic** lifetime: The programmer allocates and destroys the object explicitly. (`*allocated_ch`)

An object can have many names. For example, here, `local` and `*ptr` refer to the same object:

```
void f() {
    int local = 1;
    int* ptr = &local;
}
```

The different names for an object are sometimes called *aliases*.

What happens when an object is uninitialized? The answer depends on its lifetime.

- static lifetime (e.g., `int global;` at file scope): The object is initialized to 0.
- automatic or dynamic lifetime (e.g., `int local;` in a function, or `int* ptr = new int`): The object is *uninitialized* and reading the object's value before it is assigned causes undefined behavior.

Objects with dynamic lifetime aren't easy to use correctly. Dynamic lifetime causes many serious problems in C programs, including memory leaks, use-after-free, double-free, and so forth (more on these Thursday). Those serious problems cause undefined behavior and play a "disastrously central role" in "our ongoing computer security nightmare".

But dynamic lifetime is critically important. Only with dynamic lifetime can you construct an object whose size isn't known at compile time, or construct an object that outlives its creating function.

Memory layout

How does a hardware machine implement the three kinds of lifetime? We can use a program to find out. (See cs61-lectures/datarep2/mexplore.cc.)

Hardware implements C objects using *memory* (so called because it remembers object values). At a high level, a memory is a modifiable array of *M* **bytes**, where a byte is a number between 0 and 255 inclusive. That means that, for any number *a* between 0 and *M*–1, we can:

- Write a byte at address *a*.
- Read the byte at address *a* (obtaining the most-recently-written value).

The number *a* is called an **address**, and since every memory address corresponds to a byte, this is a *byte-addressable memory*.

On old machines, such as old Macintoshes (pre-OS X), C programs worked directly with this kind of memory. It was a disaster: an incorrect program could overwrite memory belonging to any other running program. Modern machines avoid this problem; we'll see how in unit 4.

The compiler and operating system work together to put objects at different addresses. A program's *address space* (which is the range of addresses accessible to a program) divides into regions called **segments**. The most important ones are:

- Code (aka text). Contains instructions and constant static objects; unmodifiable; static lifetime.
- Data. Modifiable; static lifetime.
- Heap. Modifiable; dynamic lifetime.

- Stack. Modifiable; automatic lifetime.

Data layout

Memory stores bytes, but the C abstract machine refers to values of many types, some of which don't fit in a single byte. The compiler, hardware, and standard together define how objects map to bytes. Each object uses a contiguous range of addresses (and thus bytes).

Since C is designed to help software interface with hardware devices, the C standard is transparent about how objects are stored. A C program can ask how big an object is using the `sizeof` keyword. `sizeof(T)` returns the number of bytes in the representation of an object of type `T`, and `sizeof(x)` returns the size of object `x`. The result of `sizeof` is a value of type `size_t`, which is an unsigned integer type large enough to hold any representable size. On 64-bit architectures, such as x86-64 (our focus in this course), `size_t` can hold numbers between 0 and 2^{64}–1.

Unsigned integer representation

A bit is the fundamental unit of digital information: it's either 0 or 1.

C++ manages memory in units of bytes—8 contiguous bits, that together can represent numbers between 0 and 255. C's unit for a byte is `char`: the abstract machine says a byte is stored in `char`. That means an `unsigned char` holds values [0, 255] inclusive.

> The C++ standard actually doesn't *require* that a byte hold 8 bits, and on some crazy machines from decades ago, bytes could hold *nine* bits! (!?)

Other integer types can hold more values. On x86-64, an `unsigned short` can hold values [0, 65535] inclusive. An `unsigned int` can hold values [0, 4294967295] inclusive. And an `unsigned long` can hold values [0, 18446744073709551615] inclusive.

The abstract machine doesn't specify how large integers are stored in memory—it's the compiler and hardware's job to make that choice. But modern computers make one of two basic choices, called **little endian** and **big endian**. Here's how it works.

- Write the large integer in hexadecimal format, including all leading zeros. For example, the `unsigned int` value 65534 would be written `0x0000FFFE`. There will be twice as many hexadecimal digits as `sizeof(TYPE)`.
- Both little-endian and big-endian mode divide the integer into its component bytes, which are its digits in base 256. In our example, they are, from most to least significant, 0x00, 0x00, 0xFF, and 0xFE.
- In *little endian* representation, the bytes are stored in memory from *least to most* significant. If our example was stored at address 0x30, we would have:

```
0x30: 0xFE    0x31: 0xFF    0x32: 0x00    0x33: 0x00
```

- In *big endian* representation, the bytes are stored in memory from *most to least* significant, which is the reverse order.

```
0x30: 0x00    0x31: 0x00    0x32: 0xFF    0x33: 0xFE
```

x86-64 uses little endian order. The Internet's fundamental protocols, such as IP and TCP, use big endian order, which is therefore also called "network" byte order.

Signed integer representation

But how are signed integers represented in memory? What bit patterns should represent negative numbers?

The best representation for signed integers (and the choice made by x86-64) is **two's complement**. Two's complement representation is based on this principle: *Addition and subtraction of signed integers shall use the **same instructions** as addition and subtraction of unsigned integers.*

To see what this means, let's think about what `-x` should mean when `x` is an unsigned integer. Wait, negative unsigned?! This isn't an oxymoron because C++ uses *modular arithmetic* for unsigned integers: the result of an arithmetic operation on unsigned values is always taken modulo 2^B, where B is the number of bits in the unsigned value type. Thus, on x86-64,

```
unsigned a = 0xFFFFFFFFU; // = 2^32 - 1
unsigned b = 0x00000002U;
assert(a + b == 0x00000001U); // True because 2^32 - 1 + 2 = 1 (mod 2^32)!
```

`-x` is simply the number that, when added to `x`, yields 0 (mod 2^B). For example, when `unsigned x = 0xFFFFFFFFU`, then `-x == 1U`, since `x + -x` equals zero (mod 2^{32}).

To obtain `-x`, we flip all the bits in `x` (an operation written `~x`) and then add 1. To see why, consider the bit representations. What is `x + (~x + 1)`? Well, $(\sim x)_i$ is 1 whenever x_i is 0, and vice versa. That means that *every bit* of `x + ~x` is 1 (there are no carries), and `x + ~x` is the largest unsigned integer, with value 2^B-1. If we add 1 to this, we get 2^B. Which is 0 (mod 2^B)! The highest "carry" bit is dropped, leaving zero.

Two's complement arithmetic uses half of the unsigned integer representations for negative numbers. A two's-complement signed integer with B bits has the following values:

- If the most-significant bit is 1, the represented number is negative. Specifically, the represented number is `-(~x + 1)`, where the outer negative sign is mathematical negation (not computer arithmetic).
- If every bit is 0, the represented number is 0.
- If the most-significant but is 0 but some other bit is 1, the represented number is positive.

The most significant bit is also called the "sign bit."

Another way to think about two's-complement is that, for B-bit integers, the most-significant bit has place value 2^{B-1} in unsigned arithmetic and **negative** 2^{B-1} in signed arithmetic. All other bits have the same place values in both kinds of arithmetic.

The two's-complement bit pattern for `x + y` is the same whether `x` and `y` are considered as signed or unsigned values. For example, in 4-bit arithmetic, 5 has representation `0b0101`, while the representation `0b1100` represents 12 if unsigned and –4 if signed (`~0b1100 + 1` = `0b0011 + 1` == 4). Let's add those bit patterns and see what we get:

```
  0b0101
+ 0b1100
--------
 0b10001 == 0b0001 (mod 2^4)
```

Note that this is the right answer for *both signed and unsigned arithmetic*: 5 + 12 = 17 = 1 (mod 16), and 5 + -4 = 1.

Subtraction and multiplication also produce the same results for unsigned arithmetic and signed two's-complement arithmetic. (For instance, 5 * 12 = 60 = 12 (mod 16), and 5 * -4 = -20 = -4 (mod 16).) This is *not* true of division. (Consider dividing the 4-bit representation `0b1110` by 2. In signed arithmetic, `0b1110` represents -2, so `0b1110/2 == 0b1111` (-1); but in unsigned arithmetic, `0b1110` is 14, so `0b1110/2 == 0b0111` (7).) And, of course, it is *not* true of comparison. In signed 4-bit arithmetic, `0b1110 < 0`, but in unsigned 4-bit arithmetic, `0b1110 > 0`. This means that a C compiler for a two's-complement machine can use a single `add` instruction for either signed or unsigned numbers, but it must generate different instruction patterns for signed and unsigned division (or less-than, or greater-than).

There are a couple quirks with C signed arithmetic. First, in two's complement, *there are more negative numbers than positive numbers*. A representation with sign bit is 1, but every other bit 0, has no positive counterpart at the same bit width: for this number, `-x == x`. (In 4-bit arithmetic, `-0b1000 == ~0b1000 + 1 == 0b0111 + 1 == 0b1000`.) Second, and far worse: *arithmetic overflow on signed integers is undefined behavior*.

Data representation 3: Layout

Last lecture discussed how objects are represented in memory, for a couple important kinds of objects (integers of various sizes, signed and unsigned). This lecture concerns *layout*: the ways that compilers and operating systems place *multiple* objects in relationship to one another. The abstract machine defines certain aspects of layout; others are left up to the compiler and runtime and operating system.

Segments

One aspect of layout is that objects are segregated into different address ranges based on lifetime. These ranges are called *segments*. The compiler decides on a segment for each object based on its lifetime. The linker then groups all the program's objects by segment (so, for instance, global variables from different compiler runs are grouped together into a single segment). Finally, when a program runs, the operating system loads the segments into memory. (The stack and heap segments grow on demand.)

Object declaration (C program text)	**Lifetime (abstract machine)**	**Segment (executable location in Linux)**	**Example address range (runtime location in x86-64 Linux)**
Constant global	Static	Code (aka Text)	0x400000 ($\approx 1 \times 2^{22}$)
Global	Static	Data	0x600000 ($\approx 1.5 \times 2^{22}$)
Local	Automatic	Stack	0x7fff448d0000 ($\approx 2^{25} \times 2^{22}$)
Anonymous, returned by `new`	Dynamic	Heap	0x1a00000 ($\approx 8 \times 2^{22}$)

Constant global data and global data have the same lifetime, but are stored in different segments. The operating system uses different segments so it can prevent the program from modifying constants. It marks the code segment, which contains functions (instructions) and constant global data, as read-only, and any attempt to modify code-segment memory causes a crash (a "Segmentation violation").

An executable is normally at least as big as the static-lifetime data (the code and data segments together). Since all that data must be in memory for the entire lifetime of the program, it's written to disk and then loaded by the OS before the program starts running. There is an exception, however: the "bss" segment is used to hold modifiable static-lifetime data with initial value zero. Such data is common, since all static-lifetime data is initialized to zero unless otherwise specified in the program text. Rather than *storing* a bunch of zeros in the object files and executable, the compiler and linker simply track the *location and size* of all zero-initialized global data. The operating system sets this memory to zero during the program load process. Clearing memory is faster than loading data from disk, so this optimization saves both time (the program loads faster) and space (the executable is smaller).

Compiler layout

The compiler has complete freedom to pick locations for objects, subject to the abstract machine's constraints —most importantly, that each object occupies disjoint memory from any other object that's active at the same time. For instance, consider this program:

```
void f() {
    int i1 = 0;
    int i2 = 1;
    int i3 = 2;
    char c1 = 3;
    char c2 = 4;
    char c3 = 5;
    ...
}
```

On Linux, GCC will put all these variables into the stack segment, which we can see using `hexdump`. But it can put them in the stack segment *in any order*, as we can see by reordering the declarations (try declaration order `i1`, `c1`, `i2`, `c2`, `c3`), by changing optimization levels, or by adding different scopes (braces). The abstract machine gives the programmer no guarantees about how object addresses relate. In fact, the compiler may *move objects around* during execution, as long as it ensures that the program behaves according to the abstract machine. Modern optimizing compilers often do this, particularly for automatic objects.

But what order does the compiler choose? With optimization disabled, the compiler appears to lay out objects in decreasing order by declaration, so the first declared variable in the function has the highest address. With optimization enabled, the compiler follows roughly the same guideline, but it also rearranges objects by type—for instance, it tends to group `char`s together—and it can reuse space if different variables in the same function have disjoint lifetimes. The optimizing compiler tends to use less space for the same set of variables.

Collections: Abstract machine layout

The C++ programming language offers several *collection* mechanisms for grouping subobjects together into new kinds of object. The collections are structs, arrays, and unions. (Classes are a kind of struct.) Although the compiler can lay out different objects however it likes relative to one another, the *abstract machine* defines how subobjects are laid out inside a collection. This is important, because it lets C/C++ programs exchange messages with hardware and even with programs written in other languages. (Messages can be exchanged only when both parties agree on layout. C/C++'s rules let a C program match any known layout.)

The sizes and alignments for user-defined types—arrays, structs, and unions—are derived from a couple simple rules or principles. Here they are. The first rule applies to all types.

1. First-member rule. The address of the first member of a collection equals the address of the collection.

Thus, the address of an array is the same as the address of its first element. The address of a struct is the same as the address of the first member of the struct.

The next three rules depend on the class of collection. Every C abstract machine enforces these rules.

2. Struct rule. The second and subsequent members of a struct are laid out in order, with no overlap, subject to alignment constraints.

3. Array rule. The address of the `i`th element of an array of type `T` is `ADDRESSOF(array) + i * sizeof(T)`.

4. Union rule. All members of a union share the address of the union.

> In C, every struct follows the struct rule, but in C++, only *simple* structs follow the rule. Complicated structs, such as structs with some `public` and some `private` members, or structs with `virtual` functions, can be laid out however the compiler chooses. The typical situation is that C++ compilers for a machine architecture (e.g., "Linux x86-64") will all agree on a layout procedure for complicated structs. This allows code compiled by different compilers to interoperate.

Alignment

Repeated executions of programs like `./mexplore` show that the C compiler and library restricts the addresses at which some kinds of data appear. In particular, the address of every `int` value is always a multiple of 4, whether it's located on the stack (automatic lifetime), the data segment (static lifetime), or the heap (dynamic lifetime).

A bunch of observations will show you these rules:

Type	Size	Address restriction
`char` (`signed char`, `unsigned char`)	1	No restriction
`short` (`unsigned short`)	2	Multiple of 2
`int` (`unsigned int`)	4	Multiple of 4
`long` (`unsigned long`)	8	Multiple of 8
`float`	4	Multiple of 4
`double`	8	Multiple of 8
`T*`	8	Multiple of 8

These are the **alignment restrictions** for an x86-64 Linux machine.

> These restrictions hold for most x86-64 operating systems, except that on Windows, the `long` type has size and alignment 4. (The `long long` type has size and alignment 8 on all x86-64 operating systems.)

Just like every type has a size, every type has an alignment. The alignment of a type `T` is a number `a`≥1 such that the address of every object of type `T` must be a multiple of `a`. Every object with type `T` has size `sizeof(T)`—it occupies `sizeof(T)` contiguous bytes of memory; and has alignment `alignof(T)`—the address of its first byte is a multiple of `alignof(T)`. You can also say `sizeof(x)` and `alignof(x)` where `x` is the name of an object or another expression.

Alignment restrictions can make hardware simpler, and therefore faster. For instance, consider cache blocks. CPUs access memory through a transparent hardware cache. Data moves from primary memory, or RAM (which is large—a couple gigabytes on most laptops—and uses cheaper, slower technology) to the cache in units of 64 or 128 bytes. Those units are always aligned: on a machine with 128-byte cache blocks, the bytes with memory addresses [127, 128, 129, 130] live in two different cache blocks (with addresses [0, 127] and [128, 255]). But the 4 bytes with addresses [4n, 4n+1, 4n+2, 4n+3] *always live in the same cache block.* (This is true for any small power of two: the 8 bytes with addresses [8n,…,8n+7] always live in the same cache block.) In general, it's often possible to make a system faster by leveraging restrictions—and here, the CPU hardware can load data faster when it can assume that the data lives in exactly one cache line.

The compiler, library, and operating system all work together to enforce alignment restrictions.

On x86-64 Linux, `alignof(T) == sizeof(T)` for all fundamental types (the types built in to C: integer types, floating point types, and pointers). But this isn't always true; on x86-32 Linux, `double` has size 8 but alignment 4.

It's possible to construct user-defined types of arbitrary size, but the largest alignment required by a machine is fixed for that machine. C++ lets you find the maximum alignment for a machine with `alignof(std::max_align_t)`; on x86-64, this is 16, the alignment of the type `long double` (and the alignment of some less-commonly-used SIMD "vector" types).

Alignment rules

Two more rules and we can reason about how collection sizes and alignments interact.

Every C++ abstract machine enforces

5. Malloc rule. Any non-null pointer returned by `malloc` has alignment appropriate for *any* type. In other words, assuming the allocated size is adequate, the pointer returned from `malloc` can safely be cast to `T*` for any `T`.

Oddly, this holds even for small allocations. The C++ standard (the abstract machine) requires that `malloc(1)` return a pointer whose alignment is appropriate for any type, including types that don't fit.

The last rule is *not* required by the abstract machine, but it's how sizes and alignments on our machines work:

6. Minimum rule. The sizes and alignments of user-defined types, and the offsets of struct members, are minimized within the constraints of the other rules.

The minimum rule, and the sizes and alignments of basic types, are defined by the x86-64 Linux "ABI"—its Application Binary Interface. This specification standardizes how x86-64 Linux C compilers should behave, and lets users mix and match compilers without problems.

Consequences of the size and alignment rules

From these rules we can derive some interesting consequences.

First, **the size of every type is a multiple of its alignment**.

To see why, consider an array with two elements. By the array rule, these elements have addresses `a` and `a+sizeof(T)`, where `a` is the address of the array. Both of these addresses contain a `T`, so they are both a multiple of `alignof(T)`. That means `sizeof(T)` is also a multiple of `alignof(T)`.

We can also **characterize the sizes and alignments of different collections**.

- The size of an array of `N` elements of type `T` is `N * sizeof(T)`: the sum of the sizes of its elements. The alignment of the array is `alignof(T)`.
- The size of a union is the maximum of the sizes of its components (because the union can only hold one component at a time). Its alignment is also the maximum of the alignments of its components.
- The size of a struct is at least as big as the sum of the sizes of its components. Its alignment is the maximum of the alignments of its components.

Thus, the alignment of every collection equals the maximum of the alignments of its components.

> It's also true that the alignment equals the least common multiple of the alignments of its components. You might have thought lcm was a better answer, but the max is the same as the lcm for every architecture that matters, because all fundamental alignments are powers of two.

The size of a struct might be *larger* than the sum of the sizes of its components, because of alignment constraints. Since the compiler *must* lay out struct components in order, and it *must* obey the components' alignment constraints, and it *must* ensure different components occupy disjoint addresses, it *must* sometimes introduce extra space in structs. Here's an example: the struct will have 3 bytes of **padding** after `char c`, to ensure that `int i2` has the correct alignment.

```
struct twelve_bytes {
    int i1;
    char c;
    int i2;
};
```

Thanks to padding, reordering struct components can sometimes reduce the total size of a struct.

The rules also imply that **the offset of any struct member**—which is the difference between the address of the member and the address of the containing struct—**is a multiple of the member's alignment**.

To see why, consider a struct `s` with member `m` at offset `o`. The malloc rule says that any pointer returned from `malloc` is correctly aligned for `s`. Every pointer returned from `malloc` is maximally aligned, equalling `16*x` for some integer `x`. The struct rule says that the address of `m`, which is `16*x + o`, is correctly aligned. That means that `16*x + o = alignof(m)*y` for some integer `y`. Divide both sides by `a = alignof(m)` and you see that `16*x/a + o/a = y`. But `16/a` is an integer—the maximum alignment is a multiple of every alignment—so `16*x/a` is an integer. We can conclude that `o/a` must also be an integer!

Finally, we can also derive the necessity for padding at the end of structs. (How?)

Pointer representation

We distinguish *pointers*, which are concepts in the C abstract machine, from *addresses*, which are hardware concepts. A pointer combines an address and a type.

The memory representation of a pointer is the same as the memory representation of its address, so a pointer with address 0x1347810A is stored the same way as the integer with the same value.

The C abstract machine defines an unsigned integer type `uintptr_t` that can hold any address. (You have to `#include <inttypes.h>` or `<cinttypes>` to get the definition.) On most machines, including x86-64, `uintptr_t` is the same as `unsigned long`. Casts between pointer types and `uintptr_t` are information preserving, so this assertion will never fail:

```
void* ptr = malloc(...);
uintptr_t addr = (uintptr_t) ptr;
void* ptr2 = (void*) addr;
assert(ptr == ptr2);
```

Since it is a 64-bit architecture, the size of an x86-64 address is 64 bits (8 bytes). That's also the size of x86-64 pointers.

Compiler hijinks

In C++, most dynamic memory allocation uses special language operators, `new` and `delete`, rather than library functions.

Though this seems more complex than the library-function style, it has advantages. A C compiler cannot tell what `malloc` and `free` do (especially when they are redefined to debugging versions, as in the problem set), so a C compiler cannot necessarily optimize calls to `malloc` and `free` away. But the C++ compiler may assume that *all* uses of `new` and `delete` follow the rules laid down by the abstract machine. That means that if the compiler can prove that an allocation is unnecessary or unused, it is free to remove that allocation!

For example, we compiled this program in the problem set environment (based on `test003.cc`):

```
int main() {
    char* ptrs[10];
    for (int i = 0; i < 10; ++i) {
        ptrs[i] = new char[i + 1];
    }
    for (int i = 0; i < 5; ++i) {
        delete[] ptrs[i];
    }
    m61_printstatistics();
}
```

The optimizing C++ compiler removes all calls to `new` and `delete`, leaving only the call to `m61_printstatistics()`! (For instance, try `objdump -d testXXX` to look at the compiled x86-64 instructions.) This is valid because the compiler is explicitly allowed to eliminate unused allocations, and here, since the `ptrs` variable is local and doesn't escape `main`, *all* allocations are unused. The C compiler cannot perform this useful transformation. (But the C compiler can do other cool things, such as unroll the loops.)

Data representation 4: Pointers and undefined behavior

Arrays and pointers

One of C's more interesting choices is that it explicitly relates pointers and arrays. Although arrays are laid out in memory in a specific way, they generally behave like pointers when they are used. This property probably arose from C's desire to explicitly model memory as an array of bytes, and it has beautiful and confounding effects.

We've already seen one of these effects. The `hexdump` function has this *signature* (arguments and return type):

```
void hexdump(const void* ptr, size_t size);
```

But we can just pass an array as argument to `hexdump`:

```
char c[10];
hexdump(c, sizeof(c));
```

When used in an expression like this—here, as an argument—the array magically changes into a pointer to its first element. The above call has the same meaning as this:

```
hexdump(&c[0], 10 * sizeof(c[0]));
```

C programmers transition between arrays and pointers very naturally.

A confounding effect is that unlike all other types, in C arrays are passed to and returned from functions *by reference* rather than by value. C is a call-by-value language except for arrays. This means that all function arguments and return values are copied, so that parameter modifications inside a function do not affect the objects passed by the caller—except for arrays. For instance:

```
void f(int a[2]) {
    a[0] = 1;
}
int main() {
    int x[2] = {100, 101};
    f(x);
    printf("%d\n", x[0]); // prints 1!
}
```

If you don't like this behavior, you can get around it by using a struct or a C++ `std::array`.

```
#include <array>
struct array1 { int a[2]; };
void f1(array1 arg) {
    arg.a[0] = 1;
}
void f2(std::array<int, 2> a) {
    a[0] = 1;
}
int main() {
    array1 x = {{100, 101}};
    f1(x);
    printf("%d\n", x.a[0]); // prints 100
    std::array<int, 2> x2 = {100, 101};
    f2(x2);
    printf("%d\n", x2[0]); // prints 100
}
```

Pointer arithmetic

C++ extends the logic of this array–pointer correspondence to support *arithmetic* on pointers as well.

Pointer arithmetic rule. In the C abstract machine, arithmetic on pointers produces the same result as arithmetic on the corresponding array indexes.

Specifically, consider an array `T a[n]` and pointers `T* p1 = &a[i]` and `T* p2 = &a[j]`. Then:

1. **Equality**: `p1 == p2` if and only if (iff) `p1` and `p2` point to the same address, which happens iff `i == j`.
2. **Inequality**: Similarly, `p1 != p2` iff `i != j`.
3. **Less-than**: `p1 < p2` iff `i < j`.
4. Also, `p1 <= p2` iff `i <= j`; and `p1 > p2` iff `i > j`; and `p1 >= p2` iff `i >= j`.
5. **Pointer difference**: What should `p1 - p2` mean? Using array indexes as the basis, `p1 - p2 == i - j`. (But the type of the difference is always `ptrdiff_t`, which on x86-64 is `long`, the signed version of `size_t`.)

6. **Addition**: `p1 + k` (where `k` is an integer type) equals the pointer `&a[i + k]`. (`k + p1` returns the same thing.)
7. **Subtraction**: `p1 - k` equals `&a[i - k]`.
8. **Increment and decrement**: `++p1` means `p1 = p1 + 1`, which means `p1 = &a[i + 1]`. Similarly, `--p1` means `p1 = &a[i - 1]`. (There are also postfix versions, `p1++` and `p1--`, but C++ style prefers the prefix versions.)

No other arithmetic operations on pointers are allowed. You can't multiply pointers, for example. (You can multiply *addresses* by casting the pointers to the address type, `uintptr_t`—so `(uintptr_t) p1 * (uintptr_t) p2`—but why would you?)

From pointers to iterators

Let's write a function that can sum all the integers in an array.

```
int sum(int a[], int size) {
    int sum = 0;
    for (int i = 0; i != size; ++i) {
        sum += a[i];
    }
    return sum;
}
```

This function can compute the sum of the elements of any `int` array. But because of the pointer–array relationship, its `a` argument is really a *pointer*. That allows us to call it with *subarrays* as well as with whole arrays. For instance:

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int s1 = sum(a, 10);       // 45
int s2 = sum(&a[0], 10);   // same as s1
int s3 = sum(&a[1], 5);    // sums s[1]...s[5], computing 15
int s4 = sum(a + 1, 5);    // same as s3
```

This way of thinking about arrays naturally leads to a style that avoids sizes entirely, using instead a *sentinel* or *boundary* argument that defines the *end* of the interesting part of the array.

```
int sum(int* first, int* last) {
    int sum = 0;
    while (first != last) {
        sum += *first;
        ++first;
    }
    return sum;
}
```

These expressions compute the same sums as the above:

```
int s1 = sum(a, a + 10);
int s2 = sum(&a[0], &a[0] + 10);
int s3 = sum(&a[1], &a[1] + 5);
int s4 = sum(a + 1, a + 6);
```

Note that the data from `first` to `last` forms a *half-open range*. iIn mathematical notation, we care about elements in the range `[first, last)`: the element pointed to by `first` is included (if it exists), but the element pointed to by `last` is not. Half-open ranges give us a simple and clear way to describe *empty* ranges, such as zero-element arrays: if `first == last`, then the range is empty.

Note that given a ten-element array `a`, the pointer `a + 10` can be formed and compared, but *must not* be dereferenced—the element `a[10]` does not exist. The C/C++ abstract machines allow users to form pointers to the "one-past-the-end" boundary elements of arrays, but users must not dereference such pointers.

So in C, two pointers naturally express a range of an array. The C++ standard template library, or STL, brilliantly *abstracts* this pointer notion to allow two *iterators*, which are pointer-like objects, to express a range of *any* standard data structure—an array, a vector, a hash table, a balanced tree, whatever. This version of `sum` works for any container of `int`s; notice how little it changed:

```
template <typename It>
int sum(It first, It last) {
    int sum = 0;
    while (first != last) {
        sum += *first;
        ++first;
    }
    return sum;
}
```

Some example uses:

```
std::set<int> set_of_ints;
int s1 = sum(set_of_ints.begin(), set_of_ints.end());
std::list<int> linked_list_of_ints;
int s2 = sum(linked_list_of_ints.begin(), linked_list_of_ints.end());
```

Addresses vs. pointers

What's the difference between these expressions? (Again, `a` is an array of type `T`, and `p1 == &a[i]` and `p2 == &a[j]`.)

```
ptrdiff_t d1 = p1 - p2;
ptrdiff_t d2 = (uintptr_t) p1 - (uintptr_t) p2;
```

The first expression is defined analogously to index arithmetic, so `d1 == i - j`. But the second expression performs the arithmetic on *the addresses* corresponding to those pointers. We will expect `d2` to equal `sizeof(T) * d1`. Always be aware of which kind of arithmetic you're using. Generally arithmetic on *pointers* should *not* involve `sizeof`, since the `sizeof` is included automatically according to the abstract machine; but arithmetic on *addresses* almost always *should* involve `sizeof`.

Undefined behavior

Although C++ is a low-level language, the abstract machine is surprisingly strict about which pointers may be formed and how they can be used. Violate the rules and you're in hell because you have invoked the dreaded **undefined behavior**.

Given an array `a[N]` of `N` elements of type `T`:

- Forming a pointer `&a[i]` (or `a + i`) with `0 ≤ i ≤ N` is safe.
- Forming a pointer `&a[i]` with `i < 0` or `i > N` causes undefined behavior.
- Dereferencing a pointer `&a[i]` with `0 ≤ i < N` is safe.
- Dereferencing a pointer `&a[i]` with `i < 0` or `i ≥ N` causes undefined behavior.

(For the purposes of these rules, objects that are not arrays count as single-element arrays. So given `T x`, we can safely form `&x` and `&x + 1` and dereference `&x`.)

What "undefined behavior" means is horrible. A program that executes undefined behavior is erroneous. But the compiler need not catch the error. In fact, the abstract machine says **anything goes**: undefined behavior is "behavior … for which this International Standard imposes no requirements." "Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message)." Other possible behaviors include allowing hackers from the moon to steal all of a program's data, take it over, and force it to delete the hard drive on which it is running. Once undefined behavior executes, a program may do anything, including making demons fly out of the programmer's nose.

Pointer arithmetic, and even pointer comparisons, are also affected by undefined behavior. It's undefined to go beyond and array's bounds using pointer arithmetic. And pointers may be compared for equality or inequality even if they point to different arrays or objects, but if you try to compare different arrays via less-than, like this:

```
int a[10];
int b[10];
if (a < b + 10) ...
```

that causes undefined behavior.

> If you really want to compare pointers that might be to different arrays—for instance, you're writing a hash function for arbitrary pointers—cast them to `uintptr_t` first.

Undefined behavior and optimization

A program that causes undefined behavior **is not a C++ program**. The abstract machine says that a C++ program, by definition, is a program whose behavior is always defined. The C++ compiler is allowed to assume that its input is a C++ program. (Obviously!) So the compiler can assume that its input program will never cause undefined behavior. Thus, since undefined behavior is "impossible," if the compiler can prove that a condition would cause undefined behavior later, it can assume that condition will never occur.

Consider this program:

```
char* x = /* some value */;
assert(x + 1 > x);
printf("x = %p, x + 1 = %p\n", x, x + 1);
```

If we supply a value equal to `(char*) -1`, we're likely to see output like this:

```
x = 0xffffffffffffffff, x + 1 = 0
```

with no assertion failure! But that's an apparently impossible result. The printout can only happen if `x + 1 > x` (otherwise, the assertion will fail and stop the printout). But `x + 1`, which equals `0`, is **less than** `x`, which is the largest 8-byte value!

The impossible happens because of undefined behavior reasoning. When the compiler sees an expression like `x + 1 > x` (with `x` a pointer), it can reason this way:

- "Ah, `x + 1`. This must be a pointer into the same array as `x` (or it might be a boundary pointer just past that array, or just past the non-array object `x`). This must be so because forming any other pointer would cause undefined behavior.
- "The pointer comparison is the same as an index comparison. `x + 1 > x` means the same thing as `&x[1] > &x[0]`. But that holds iff `1 > 0`.
- "In my infinite wisdom, I know that `1 > 0`. Thus `x + 1 > x` always holds, and the assertion will never fail.
- "My job is to make this code run fast. The fastest code is code that's not there. This assertion will never fail —might as well remove it!"

Integer undefined behavior

Arithmetic on signed integers also has important undefined behaviors. Signed integer arithmetic must never overflow. That is, the compiler may assume that the *mathematical* result of any signed arithmetic operation, such as `x + y` (with `x` and `y` both `int`), can be represented inside the relevant type. It causes undefined behavior, therefore, to add 1 to the maximum positive integer. (The `ubexplore.cc` program demonstrates how this can produce impossible results, as with pointers.)

Arithmetic on *unsigned* integers is much safer with respect to undefined behavior. Unsigned integers are defined to perform arithmetic modulo their size. This means that if you add 1 to the maximum positive *unsigned* integer, the result will always be zero.

Dividing an integer by zero causes undefined behavior whether or not the integer is signed.

Sanitizers

Sanitizers, which in our makefiles are turned on by supplying `SAN=1`, can catch many undefined behaviors as soon as they happen. Sanitizers are built in to the compiler itself; a sanitizer involves cooperation between the compiler and the language runtime. This has the major performance advantage that the compiler introduces exactly the required checks, and the optimizer can then use its normal analyses to remove redundant checks.

That said, undefined behavior checking can still be slow. Undefined behavior allows compilers to make assumptions about input values, and those assumptions can directly translate to faster code. Turning on undefined behavior checking can make some benchmark programs run 30% slower [link].

Data representation 5: Undefined behavior, bitwise operations, arena allocation

Signed integer undefined behavior

File `cs61-lectures/datarep5/ubexplore2.cc` contains the following program.

```
int main(int argc, const char *argv[]) {
    assert(argc >= 3);
    int n1 = strtol(argv[1], nullptr, 0);
    int n2 = strtol(argv[2], nullptr, 0);

    for (int i = n1; i <= n2; ++i) {
        printf("%d\n", i);
    }
}
```

What will be printed if we run the program with `./ubexplore2 0x7ffffffe 0x7fffffff`?

`0x7fffffff` is the largest positive value can be represented by type `int`. Adding one to this value yields `0x80000000`. In two's complement representation this is the smallest negative number represented by type `int`.

Assuming that the program behaves this way, then the loop exit condition `i > n2` can never be met, and the program should run (and print out numbers) forever.

However, if we run the optimized version of the program, it prints only two numbers and exits:

```
2147483646
2147483647
```

The unoptimized program does print forever and never exits.

What's going on here? We need to look at the compiled assembly of the program with and without optimization (via `objdump -S`). (*NB: The description that follows is more correct than the description from lecture!*)

The unoptimized version basically looks like this:

```
1. compare i and n2...            (mov -0x1c(%rbp),%eax; cmp -0x18(%rbp),%eax)
2.   and exit if i is greater     (jg <end of function>)
3. otherwise, print i             (... callq ...)
4. increment i                    (mov -0x1c(%rbp),%eax; add $0x1,%eax;
                                   mov %eax,-0x1c(%rbp))
5. and go back to step 1          (jmp <step 1>)
```

This is a pretty direct translation of the loop.

The optimized version, though, does it differently. As always, the optimizer has its own ideas. (Your compiler may produce different results!)

```
1. compare i and n2...            (cmp %r14d,%ebx)
2.   and exit if i is greater     (jg <end of function>)
3. otherwise, set tmp = n2 + 1    (lea 0x1(%rax),%ebp)
4. print i                        (... callq ...)
5. increment i                    (add $0x1,%ebx)
6. compare i and tmp...           (cmp %ebp,%ebx)
7.   and go to step 4 if unequal  (jne <step 4>)
```

The compiler changed the source's *less than or equal to* comparison, `i <= n2`, into a *not equal to* comparison in the executable, `i != n2 + 1` (in both cases using signed computer arithmetic, i.e., modulo 2^{32})! The comparison `i <= n2` will *always* return true when `n2 == 0x7FFFFFFF`, the maximum signed integer, so the loop goes on forever. But the `i != n2 + 1` comparison *does not* always return true when `n2 == 0x7FFFFFFF`: when `i` wraps around to `0x80000000` (the smallest negative integer), then `i` equals `n2 + 1` (which also wrapped), and the loop stops.

Why did the compiler make this transformation? In the original loop, the step-6 jump is immediately followed by another comparison and jump in steps 1 and 2. The processor jumps all over the place, which can confuse its prediction circuitry and slow down performance. In the transformed loop, the step-7 jump is *never* followed by a comparison and jump; instead, step 7 goes back to step 4, which always prints the current number. This more streamlined control flow is easier for the processor to make fast.

But the streamlined control flow is only a valid substitution under the assumption that *the addition `n2 + 1` never overflows*. Luckily (sort of), signed arithmetic overflow causes undefined behavior, so the compiler is totally justified in making that assumption!

> Programs based on `ubexplore2` have demonstrated undefined behavior differences for years, even as the precise reasons why have changed. In some earlier compilers, we found that the optimizer just upgraded the `int`s to `long`s—arithmetic on `long`s is just as fast on x86-64 as arithmetic on `int`s, since x86-64 is a 64-bit architecture, and sometimes using `long`s for everything lets the compiler avoid conversions back and forth. The `ubexplore2l` program demonstrates this form of transformation: since the loop variable is added to a `long` counter, the compiler opportunistically upgrades `i` to `long` as well. This transformation is also only valid under the assumption that `i + 1` will not overflow—which it can't, because of undefined behavior.

Using `unsigned` type prevents all this undefined behavior, because arithmetic overflow on unsigned integers is *well defined* in C/C++. The `ubexplore2u.cc` file uses an unsigned loop index and comparison, and `./ubexplore2u` and `./ubexplore2u.noopt` behave exactly the same (though you have to give arguments like `./ubexplore2u 0xfffffffe 0xffffffff` to see the overflow).

Computer arithmetic and bitwise operations

Basic bitwise operators

Computers offer not only the usual arithmetic operators like `+` and `-`, but also a set of *bitwise* operators. The basic ones are `&` (and), `|` (or), `^` (xor/exclusive or), and the unary operator `~` (complement). In truth table form:

& (and)	**0**	**1**
0	0	0

& (and)	0	1
1	0	1

| (or)	0	1
0	0	1
1	1	1

^ (xor)	0	1
0	0	1
1	1	0

~ (complement)	
0	1
1	0

In C or C++, these operators work on integers. But they work bitwise: the result of an operation is determined by applying the operation independently at each bit position. Here's how to compute 12 & 4 in 4-bit unsigned arithmetic:

```
   12 == 0b 1 1 0 0
^   4 == 0b 0 1 0 0
--------------------
         0b 0 1 0 0 == 4
```

These basic bitwise operators simplify certain important arithmetics. For example, `(x & (x - 1)) == 0` tests whether `x` is zero or a power of 2.

Negation of signed integers can also be expressed using a bitwise operator: `-x == ~x + 1`. This is in fact how we define two's complement representation. We can verify that `x` and `(-x)` does add up to zero under this representation:

```
x + (-x) == (x + ~x) + 1
         == 0b 1111... + 1
         == 0
```

Bitwise "and" (`&`) can help with modular arithmetic. For example, `x % 32 == (x & 31)`. We essentially "mask off", or clear, higher order bits to do modulo-powers-of-2 arithmetics. This works in any base. For example, in decimal, the fastest way to compute `x % 100` is to take just the two least significant digits of `x`.

Bitwise shift of unsigned integer

`x << i` appends `i` zero bits starting at the least significant bit of `x`. High order bits that don't fit in the integer are thrown out. For example, assuming 4-bit unsigned integers

```
0b 1101 << 2 == 0b 0100
```

Similarly, `x >> i` appends `i` zero bits at the most significant end of `x`. Lower bits are thrown out.

```
0b 1101 >> 2 == 0b 0011
```

Bitwise shift helps with division and multiplication. For example:

```
x / 64 == x >> 6

x * 64 == x << 6
```

A modern compiler can optimize `y = x * 66` into `y = (x << 6) + (x << 1)`.

Bitwise operations also allows us to treat bits within an integer separately. This can be useful for "options".

For example, when we call a function to open a file, we have a lot of options:

- Open for reading?
- Open for writing?
- Read from the end?
- Optimize for writing?

We have a lot of true/false options.

One bad way to implement this is to have this function take a bunch of arguments -- one argument for each option. This makes the function call look like this:

```
open_file(..., true, false, ...)
```

The long list of arguments slows down the function call, and one can also easily lose track of the meaning of the individual true/false values passed in.

A cheaper way to achieve this is to use a single integer to represent all the options. Have each option defined as a power of 2, and simply `|` (or) them together and pass them as a single integer.

```
#define O_READ 1
#define O_WRITE 2

open_file(..., O_READ | O_WRITE); // setting both O_READ and O_WRITE flags
```

> Flags are usually defined as powers of 2 so we set one bit at a time for each flag. It is less common but still possible to define a combination flag that is not a power of 2, so that it sets multiple bits in one go.

Arena allocation

File `cs61-lectures/datarep5/membench.cc` contains a memory allocation benchmark. The core of the benchmark looks like this

```
void benchmark() {
    // allocate a new memory arena for this thread.
    // An "arena" is an object that encapsulates a set of memory allocations.
    // Arenas can capture allocation statistics and improve speed.
    memnode_arena* arena = memnode_arena_new();

    // Allocate 4096 memnodes.
    memnode* m[4096];
    for (int i = 0; i != 4096; ++i) {
        m[i] = memnode_alloc(arena);
    }

    // `noperations` times, free a memnode and then allocate another one.
    for (unsigned i = 0; i != noperations; ++i) {
        unsigned pos = i % 4096;
        memnode_free(arena, m[pos]);
        m[pos] = memnode_alloc(arena);
    }

    // Free the remaining memnodes and the arena.
    for (int i = 0; i != 4096; ++i) {
        memnode_free(arena, m[i]);
    }
    memnode_arena_free(arena);
}
```

The benchmark tests the performance of `memnode_alloc()` and `memnode_free()` allocator functions. It allocates 4096 `memnode` objects, then free-and-then-allocates them for `noperations` times, and then frees all of them.

We notice that by the end of the function, all dynamically allocated data are freed. Can we take advantage of this property to speed up allocation/deallocation?

We only allocate `memnode`s, and all `memnode`s are of the same size, so we don't need metadata that keeps track of the size of each allocation. Furthermore, since all dynamically allocated data are freed at the end of the function, for each individual `memnode_free()` call we don't really need to return memory to the system allocator. We can simply reuse these memory during the function and returns all memory to the system at once when the function exits.

If we run the benchmark with 100000000 allocation, and use the system `malloc()`, `free()` functions to implement the `memnode` allocator, the benchmark finishes in 0.908 seconds.

Our alternative implementation of the allocator can finish in 0.355 seconds, beating the heavily optimized system allocator by a factor of 3. We will reveal how we achieved this in the next lecture.

Data representation 6: Arena allocation

We continue our exploration with the memnode allocation benchmark introduced from the last lecture.

File `cs61-lectures/datarep6/mb-malloc.cc` contains a version of the benchmark using the system `new` and `delete` operators.

```
unsigned long memnode_benchmark(unsigned noperations, unsigned step) {
    assert(step % 2 == 1);  // `step` must be odd
    long counter = 0;

    // Allocate 4096 memnodes.
    const unsigned nnodes = 4096;
    memnode* m[nnodes];
    for (unsigned i = 0; i != nnodes; ++i) {
        m[i] = new memnode;
        m[i]->file = "datarep/mb-filename.cc";
        m[i]->line = counter;
        ++counter;
    }

    // Replace one `noperations` times.
    for (unsigned i = 0; i != noperations; ++i) {
        unsigned pos = (i * step) % nnodes;
        delete m[pos];

        m[pos] = new memnode;
        m[pos]->file = "datarep/mb-filename.cc";
        m[pos]->line = counter;
        ++counter;
    }

    // Compute a statistic and free them.
    unsigned long result = 0;
    for (unsigned i = 0; i != nnodes; ++i) {
        result += m[i]->line;
        delete m[i];
    }

    return result;
}
```

In this function we allocate an array of 4096 pointers to `memnode`s, which occupy $2^3*2^{12}=2^{15}$ bytes on the stack. We then allocate 4096 `memnode`s. Our `memnode` is defined like this:

```
struct memnode {
    std::string file;
    unsigned line;
};
```

Each `memnode` contains a `std::string` object and an unsigned integer. Each `std::string` object internally contains a pointer points to an character array in the heap. Therefore, every time we create a new `memnode`, we need 2 allocations: one to allocate the memnode itself, and another one performed internally by the `std::string` object when we initialize/assign a string value to it.

Every time we deallocate a `memnode` by calling `delete`, we also delete the `std::string` object, and the string object knows that it should deallocate the heap character array it internally maintains. So there are also 2 deallocations occuring each time we free a memnode.

> We make the benchmark to return a seemingly meaningless `result` to prevent an aggressive compiler from optimizing everything away. We also use this result to make sure our subsequent optimizations to the allocator are correct by generating the same result.

This version of the benchmark, using the system allocator, finishes in 0.335 seconds. Not bad at all.

Spoilers alert: We can do 15x better than this.

1st optimization: `std::string`

We only deal with one file name, namely "datarep/mb-filename.cc", which is constant throughout the program for all `memnode`s. It's also a string literal, which means it as a constant string has a static life time. Why can't we just simply use a `const char*` in place of the `std::string` and let the pointer point to the static constant string? This saves us the internal allocation/deallocation performed by `std::string` every time we initialize/delete a string.

The fix is easy, we simply change the `memnode` definition:

```
struct memnode {
    const char* file;
    unsigned line;
};
```

This version of the benchmark now finishes in 0.143 seconds, a 2x improvement over the original benchmark. This 2x improvement is consistent with a 2x reduction in numbers of allocation/deallocation mentioned earlier.

> You may ask why people still use `std::string` if it involves an additional allocation and is slower than `const char*`, as shown in this benchmark. `std::string` is much more flexible in that it also deals data that doesn't have static life time, such as input from a user or data the program receives over the network. In short, when the program deals with strings that are not constant, heap data is likely to be very useful, and `std::string` provides facilities to conveniently handle on-heap data.

2nd optimization: the system allocator

We still use the system allocator to allocate/deallocate `memnode`s. The system allocator is a general-purpose allocator, which means it must handle allocation requests of all sizes. Such general-purpose designs usually comes with a compromise for performance. Since we are only `memnode`s, which are fairly small objects (and all have the same size), we can build a special- purpose allocator just for them.

File `cs61-lectures/datarep6/mb-arena-01.cc` contains a version of the benchmark using an *arena allocator*.

```
unsigned long memnode_benchmark(unsigned noperations, unsigned step) {
    assert(step % 2 == 1);  // `step` must be odd
    long counter = 0;
    memnode_arena arena;

    // Allocate 4096 memnodes.
    const unsigned nnodes = 4096;
    memnode* m[nnodes];
    for (unsigned i = 0; i != nnodes; ++i) {
        m[i] = arena.allocate();
        m[i]->file = "datarep/mb-filename.cc";
        m[i]->line = counter;
        ++counter;
    }

    // Replace one `noperations` times.
    for (unsigned i = 0; i != noperations; ++i) {
        unsigned pos = (i * step) % nnodes;
        arena.deallocate(m[pos]);

        m[pos] = arena.allocate();
        m[pos]->file = "datarep/mb-filename.cc";
        m[pos]->line = counter;
        ++counter;
    }

    // Compute a statistic and free them.
    unsigned long result = 0;
    for (unsigned i = 0; i != nnodes; ++i) {
        result += m[i]->line;
    }

    return result;
}
```

Compare to the previous version of the benchmark, in this version, instead of calling `new` and `delete`, we use `arena.allocate()` and `arena.deallocate()` to allocate and free `memnode`s. Our `arena` object (with type `memnode_area`) is our special-purpose allocator for our `memnode`s.

This is how we implement the `memnode_arena` allocator:

```
struct memnode_arena {
    std::vector<memnode*> free_list;

    memnode* allocate() {
        memnode* n;
        if (free_list.empty()) {
            n = new memnode;
        } else {
            n = free_list.back();
            free_list.pop_back();
        }
        return n;
    }

    void deallocate(memnode* n) {
        free_list.push_back(n);
    }
};
```

This allocator maintains a free list (a C++ `vector`) of freed `memnode`s. `allocate()` simply pops a `memnode` off the free list if there is any, and `deallocate()` simply puts the `memnode` on the free list. This free list serves as a buffer between the system allocator and the benchmark function, so that the system allocator is invoked less frequently. In fact, in the benchmark, the system allocator is only invoked for 4096 times when it initializes the pointer array. That's a huge reduction because all 10-million "recycle" operations in the middle now doesn't involve the system allocator.

With this special-purpose allocator we can finish the benchmark in 0.057 seconds, another 2.5x improvement.

However this allocator now leaks memory: it never actually calls `delete`! Let's fix this by letting it also keep track of all allocated `memnode`s. The modified definition of `memnode_arena` now looks like this:

```
struct memnode_arena {
    std::vector<memnode*> allocated;
    std::vector<memnode*> free_list;

    ~memnode_arena() {
        destroy_all();
    }

    void destroy_all() {
        for (auto a : allocated) {
            delete a;
        }
    }

    memnode* allocate() {
        memnode* n;
        if (free_list.empty()) {
            n = new memnode;
            allocated.push_back(n);
        } else {
            n = free_list.back();
            free_list.pop_back();
        }
        return n;
    }

    void deallocate(memnode* n) {
        free_list.push_back(n);
    }
};
```

With the updated allocator we simply need to invoke `arena.destroy_all()` at the end of the function to fix the memory leak. And we don't even need to invoke this method manually! We can use the C++ destructor for the `memnode_arena` struct, defined as `~memnode_arena()` in the code above, which is automatically called when our `arena` object goes out of scope. We simply make the destructor invoke the `destroy_all()` method, and we are all set.

Fixing the leak doesn't appear to affect performance at all. This is because the overhead added by tracking the `allocated` list and calling `delete` only affects our initial allocation the 4096 `memnode*` pointers in the array plus at the very end when we clean up. These 8192 additional operations is a relative small number compared to the 10 million recycle operations, so the added overhead is hardly noticeable.

Spoiler alert: We can improve this by another factor of 2.

3rd optimization: `std::vector`

In our special purpose allocator `memnode_arena`, we maintain an allocated list and a free list both using C++ `std::vector`s. `std::vector`s are dynamic arrays, and like `std::string` they involve an additional level of indirection and stores the actual array in the heap. We don't access the allocated list during the "recycling" part of the benchmark (which takes bulk of the benchmark time, as we showed earlier), so the allocated list is probably not our bottleneck. We however, add and remove elements from the free list for each recycle operation, and the indirection introduced by the `std::vector` here may actually be our bottleneck. Let's find out.

Instead of using a `std::vector`, we could use a linked list of all free `memnode`s for the actual free list. We will need to include some extra metadata in the memnode to store pointers for this linked list. However, unlike in the debugging allocator pset, in a free list we don't need to store this metadata in addition to actual `memnode` data: the `memnode` is free, and not in use, so we can use reuse its memory, using a union:

```
union freeable_memnode {
    memnode n;
    freeable_memnode* next_free;
};
```

We then maintain the free list like this:

```
struct memnode_arena {
    std::vector<freeable_memnode*> allocated_groups;
    freeable_memnode* free_list;

    ...

    memnode* allocate() {
        if (!free_list) {
            refresh_free_list();
        }
        freeable_memnode* fn = free_list;
        free_list = fn->next_free;
        return &fn->n;
    }

    void deallocate(memnode* n) {
        freeable_memnode* fn = (freeable_memnode*) n;
        fn->next_free = free_list;
        free_list = fn;
    }

    ...

};
```

Compared to the `std::vector` free list, this free list we always directly points to an available `memnode` when it is not empty (`free_list !=nullptr`), without going through any indirection. In the `std::vector` free list one would first have to go into the heap to access the actual array containing pointers to free `memnode`s, and then access the `memnode` itself.

With this change we can now finish the benchmark under 0.3 seconds! Another 2x improvement over the previous one!

Compared to the benchmark with the system allocator (which finished in 0.335 seconds), we managed to achieve a speedup of nearly 15x with arena allocation.

Assembly 1: Basics

Registers

Registers are the fastest kind of memory available in the machine. x86-64 has 14 general-purpose registers and several special-purpose registers. This table gives all the basic registers, with special-purpose registers highlighted in yellow. You'll notice different naming conventions, a side effect of the long history of the x86 architecture (the 8086 was first released in 1978).

Full register name	32-bit (bits 0–31)	16-bit (bits 0–15)	8-bit low (bits 0–7)	8-bit high (bits 8–15)	**Use in calling convention**	**Callee-saved?**
General-purpose registers:						
%rax	%eax	%ax	%al	%ah	Return value (accumulator)	No
%rbx	%ebx	%bx	%bl	%bh	–	**Yes**
%rcx	%ecx	%cx	%cl	%ch	4th function argument	No
%rdx	%edx	%dx	%dl	%dh	3rd function argument	No
%rsi	%esi	%si	%sil	–	2nd function argument	No
%rdi	%edi	%di	%dil	–	1st function argument	No
%r8	%r8d	%r8w	%r8b	–	5th function argument	No
%r9	%r9d	%r9w	%r9b	–	6th function argument	No
%r10	%r10d	%r10w	%r10b	–	–	No
%r11	%r11d	%r11w	%r11b	–	–	No
%r12	%r12d	%r12w	%r12b	–	–	**Yes**
%r13	%r13d	%r13w	%r13b	–	–	**Yes**
%r14	%r14d	%r14w	%r14b	–	–	**Yes**
%r15	%r15d	%r15w	%r15b	–	–	**Yes**
Special-purpose registers:						
%rsp	%esp	%sp	%spl	–	Stack pointer	**Yes**
%rbp	%ebp	%bp	%bpl	–	Base pointer (general-purpose in some compiler modes)	**Yes**
%rip	%eip	%ip	–	–	Instruction pointer (Program counter; called $pc in GDB)	*
%rflags	%eflags	%flags	–	–	Flags and condition codes	No

Note that unlike *primary* memory (which is what we think of when we discuss memory in a C/C++ program), registers have no addresses! There is no address value that, if cast to a pointer and dereferenced, would return the contents of the `%rax` register. Registers live in a separate world from the memory whose contents are partially prescribed by the C abstract machine.

The %rbp register has a special purpose: it points to the bottom of the current function's stack frame, and local variables are often accessed relative to its value. However, when optimization is on, the compiler may determine that all local variables can be stored in registers. This frees up %rbp for use as another general-purpose register.

The relationship between different register bit widths is a little weird.

1. Loading a value into a 32-bit register name sets the *upper* 32 bits of the register to zero. Thus, after `movl $-1, %eax`, the `%rax` register has value 0x00000000FFFFFFFF.
2. Loading a value into a 16- or 8-bit register name *leaves all other bits unchanged*.

There are special instructions for loading signed and unsigned 8-, 16-, and 32-bit quantities into registers, recognizable by instruction suffixes. For instance, `movzbl` moves an 8-bit quantity (a **b**yte) into a 32-bit register (a **l**ongword) with **z**ero extension; `movslq` moves a 32-bit quantity (**l**ongword) into a 64-bit register (**q**uadword) with **s**ign extension. There's no need for `movzlq` (why?).

Instruction format

The basic kinds of assembly instructions are:

1. **Computation.** These instructions perform computation on values, typically values stored in registers. Most have zero or one *source operands* and one *source/destination operand*, with the source operand coming first. For example, the instruction `addq %rax, %rbx` performs the computation `%rbx := %rbx + %rax`.
2. **Data movement.** These instructions move data between registers and memory. Almost all have one source operand and one destination operand; the source operand comes first.
3. **Control flow.** Normally the CPU executes instructions in sequence. Control flow instructions change the instruction pointer in other ways. There are unconditional branches (the instruction pointer is set to a new value), conditional branches (the instruction pointer is set to a new value if a condition is true), and function call and return instructions.

(We use the "AT&T syntax" for x86-64 assembly. For the "Intel syntax," which you can find in online documentation from Intel, see the Aside in CS:APP3e §3.3, p177, or Wikipedia, or other online resources. AT&T syntax is distinguished by several features, but especially by the use of percent signs for registers. Sadly, the Intel syntax puts destination registers *before* source registers.)

Some instructions appear to combine computation and data movement. For example, given the C code `int* ip; ... ++(*ip);` the compiler might generate `incl (%rax)` rather than `movl (%rax), %ebx; incl %ebx; movl %ebx, (%rax)`. However, the processor actually divides these complex instructions into tiny, simpler, invisible instructions called microcode, because the simpler instructions can be made to execute faster. The complex `incl` instruction actually runs in three phases: data movement, then computation, then data movement. This matters when we introduce parallelism.

Directives

Assembly generated by a compiler contains instructions as well as *labels* and *directives*. Labels look like `labelname:` or `labelnumber:`; directives look like `.directivename arguments`. Labels are markers in the generated assembly, used to compute addresses. We usually see them used in control flow instructions, as in `jmp L3` ("jump to L3"). Directives are instructions to the assembler; for instance, the `.globl L` instruction says "label `L` is globally visible in the executable", `.align` sets the alignment of the following data, `.long` puts a number in the output, and `.text` and `.data` define the current segment.

We also frequently look at assembly that is *disassembled* from executable instructions by GDB, `objdump -d`, or `objdump -S`. This output looks different from compiler-generated assembly: in disassembled instructions, there are no intermediate labels or directives. This is because the labels and directives disappear during the process of generating executable instructions.

For instance, here is some compiler-generated assembly:

```
    .globl  _Z1fiii
    .type   _Z1fiii, @function
_Z1fiii:
.LFB0:
    cmpl    %edx, %esi
    je  .L3
    movl    %esi, %eax
    ret
.L3:
    movl    %edi, %eax
    ret
.LFE0:
    .size   _Z1fiii, .-_Z1fiii
```

And a disassembly of the same function, from an object file:

```
0000000000000000 <_Z1fiii>:
   0:   39 d6                   cmp    %edx,%esi
   2:   74 03                   je     7 <_Z1fiii+0x7>
   4:   89 f0                   mov    %esi,%eax
   6:   c3                      retq
   7:   89 f8                   mov    %edi,%eax
   9:   c3                      retq
```

Everything but the instructions is removed, and the helpful `.L3` label has been replaced with an actual address. The function appears to be located at address 0. This is just a placeholder; the final address is assigned by the linking process, when a final executable is created.

Finally, here is some disassembly from an executable:

```
0000000000400517 <_Z1fiii>:
  400517:   39 d6                   cmp    %edx,%esi
  400519:   74 03                   je     40051e <_Z1fiii+0x7>
  40051b:   89 f0                   mov    %esi,%eax
  40051d:   c3                      retq
  40051e:   89 f8                   mov    %edi,%eax
  400520:   c3                      retq
```

The instructions are the same, but the addresses are different. (Other compiler flags would generate different addresses.)

Address modes

Most instruction operands use the following syntax for values. (See also CS:APP3e Figure 3.3 in §3.4.1, p181.)

Type	Example syntax	Value used
Register	`%rbp`	Contents of `%rbp`
Immediate	`$0x4`	0x4
Memory	`0x4`	Value stored at address
	`symbol_name`	Value stored in global `symbol_name`
	`symbol_name(%rip)`	`%rip`-relative addressing for global (see below)
	`(%rax)`	Value stored at address in `%rax`
	`0x4(%rax)`	Value stored at address `%rax + 4`
	`(%rax,%rbx)`	Value stored at address `%rax + %rbx`
	`(%rax,%rbx,4)`	Value stored at address `%rax + %rbx*4`
	`0x18(%rax,%rbx,4)`	Value stored at address `%rax + 0x18 + %rbx*4`

The full form of a memory operand is `offset(base,index,scale)`, which refers to the address `offset + base + index*scale`. In `0x18(%rax,%rbx,4)`, `%rax` is the base, `0x18` the offset, `%rbx` the index, and `4` the scale. The offset (if used) must be a constant and the base (if used) must be a register; the scale must be either 1, 2, 4, or 8. The default offset, base, and index are 0, and the default scale is 1.

`symbol_name`s are found only in compiler-generated assembly; disassembly uses raw addresses (`0x601030`) or `%rip`-relative offsets (`0x200bf2(%rip)`).

Jumps and function call instructions use different syntax : `*`, rather than `()`, represents indirection.

Type	Example syntax	Address used
Register	`*%rax`	Contents of `%rax`
Immediate	`.L3`	Address of `.L3` (compiler-generated assembly)
	`400410` or `0x400410`	Given address
Memory	`*0x200b96(%rip)`	Value stored at address `%rip + 0x200b96`
	`*(%r12,%rbp,8)`	Other address modes accepted

Address computations

The `base(offset,index,scale)` form compactly expresses many array-style address computations. It's typically used with a `mov`-type instruction to dereference memory. However, the compiler can use that form to compute addresses, thanks to the `lea` (Load Effective Address) instruction.

For instance, in `movl 0x18(%rax,%rbx,4), %ecx`, the address `%rax + 0x18 + %rbx*4` is computed, then immediately dereferenced: the 4-byte value located there is loaded into `%ecx`. In `leaq 0x18(%rax,%rbx,4), %rcx`, the same address is computed, but it is *not* dereferenced. Instead, the *computed address* is moved into register `%rcx`.

Thanks to `lea`, the compiler will also prefer the `base(offset,index,scale)` form over `add` and `mov` for certain computations on integers. For example, this instruction:

```
leaq (%rax,%rbx,2), %rcx
```

performs the function `%rcx := %rax + 2 * %rbx`, but in one instruction, rather than three (`movq %rax, %rcx; addq %rbx, %rcx; addq %rbx, %rcx`).

`%rip`-relative addressing

x86-64 code often refers to globals using **%rip-relative** addressing: a global variable named `a` is referenced as `a(%rip)` rather than `a`.

This style of reference supports *position-independent code* (PIC), a security feature. It specifically supports *position-independent executables* (PIEs), which are programs that work independently of where their code is loaded into memory.

To run a conventional program, the operating system loads the program's instructions into memory *at a fixed address that's the same every time*, then starts executing the program at its first instruction. This works great, and runs the program in a predictable execution environment (the addresses of functions and global variables are the same every time). Unfortunately, the very predictability of this environment makes the program easier to attack.

In a position-independent executable, the operating system loads the program at *varying* locations: every time it runs, the program's functions and global variables have different addresses. This makes the program harder to attack (though not impossible).

Program startup performance matters, so the operating system doesn't recompile the program with different addresses each time. Instead, the compiler does most of the work in advance by using *relative addressing*.

When the operating system loads a PIE, it picks a starting point and loads all instructions and globals relative to that starting point. The PIE's instructions never refer to global variables using direct addressing: you'll never see `movl global_int, %eax`. Globals are referenced *relatively* instead, using deltas relative to the next `%rip`: `movl global_int(%rip), %eax`. These relative addresses work great independent of starting point! For instance, consider an instruction located at (starting-point + 0x80) that loads a variable `g` located at (starting-point + 0x1000) into `%rax`. In a non-PIE, the instruction might be written `movq 0x400080, %rax` (in compiler output, `movq g, %rax`); but this relies on `g` having a fixed address. In a PIE, the instruction might be written `movq 0xf80(%rip), %rax` (in compiler output, `movq g(%rip), %rax`), which works out beautifully no matter the starting point.

If the starting point is…	The instruction is at…	`g` is at…	With delta…
0x400000	0x400080	0x401000	0xF80
0x404000	0x404080	0x405000	0xF80
0x4003F0	0x400470	0x4013F0	0xF80

Assembly 2: Calling convention

Calling convention

A **calling convention** governs how functions on a particular architecture and operating system interact. This includes rules about includes how function arguments are placed, where return values go, what registers functions may use, how they may allocate local variables, and so forth. Calling conventions ensure that functions compiled by different compilers can interoperate, and they ensure that operating systems can run code from different programming languages and compilers. Some aspects of a calling convention are derived from the instruction set itself, but some are conventional, meaning decided upon by people (for instance, at a convention).

Calling conventions constrain both *callers* and *callees*. A caller is a function that calls another function; a callee is a function that was called. The currently-executing function is a callee, but not a caller.

For concreteness, we learn the x86-64 calling conventions for Linux. These conventions are shared by many OSes, including MacOS (but not Windows), and are officially called the "System V AMD64 ABI."

The official specification: AMD64 ABI

Argument passing and stack frames

One set of calling convention rules governs how function arguments and return values are passed. On x86-64 Linux, the first six function arguments are passed in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`, respectively. The seventh and subsequent arguments are passed on the stack, about which more below. The return value is passed in register `%rax`.

The full rules more complex than this. You can read them in the AMD64 ABI, section 3.2.3, but they're quite detailed. Some highlights:

1. A structure argument that fits in a single machine word (64 bits/8 bytes) is passed in a single register.

   Example: `struct small { char a1, a2; }`

2. A structure that fits in two to four machine words (16–32 bytes) is passed in sequential registers, as if it were multiple arguments.

   Example: `struct medium { long a1, a2; }`

3. A structure that's larger than four machine words is always passed on the stack.

   Example: `struct large { long a, b, c, d, e, f, g; }`

4. Floating point arguments are generally passed in special registers, the "SSE registers," that we don't discuss further.

5. If the return value takes more than eight bytes, then the *caller* reserves space for the return value, and passes the *address* of that space as the first argument of the function. The callee will fill in that space when it returns.

Writing small programs to demonstrate these rules is a pleasant exercise; for example:

```
struct small { char a1, a2; };
int f(small s) {
    return s.a1 + 2 * s.a2;
}
```

compiles to:

```
movl %edi, %eax           # copy argument to %eax
movsbl %dil, %edi         # %edi := sign-extension of lowest byte of argument (s.a1)
movsbl %ah, %eax          # %eax := sign-extension of 2nd byte of argument (s.a2)
movsbl %al, %eax
leal (%rdi,%rax,2), %eax  # %eax := %edi + 2 * %eax
ret
```

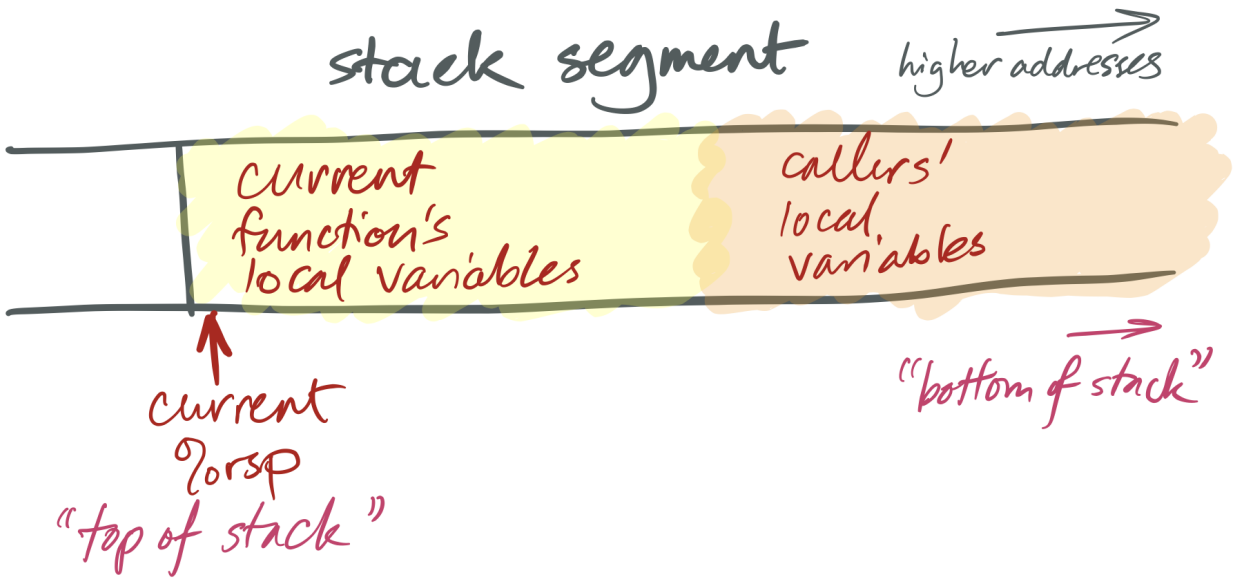
Stack

Recall that the stack is a segment of memory used to store objects with automatic lifetime. Typical stack addresses on x86-64 look like `0x7ffd'9f10'4f58`—that is, close to 2^{47}.

The stack is named after a data structure, which was sort of named after pancakes. Stack data structures support at least three operations: **push** adds a new element to the "top" of the stack; **pop** removes the top element, showing whatever was underneath; and **top** accesses the top element. Note what's missing: the data structure does not allow access to elements other than the top. (Which is sort of how stacks of pancakes work.) This restriction can speed up stack implementations.

Like a stack data structure, the stack memory segment is only accessed from the top. The currently running function accesses *its* local variables; the function's caller, grand-caller, great-grand-caller, and so forth are dormant until the currently running function returns.

x86-64 stacks look like this:



The x86-64 `%rsp` register is a special-purpose register that defines the current "stack pointer." This holds the address of the current top of the stack. On x86-64, as on many architectures, stacks grow *down*: a "push" operation adds space for more automatic-lifetime objects by moving the stack pointer left, to a numerically-

smaller address, and a "pop" operation recycles space by moving the stack pointer right, to a numerically-larger address. This means that, considered numerically, the "top" of the stack has a smaller address than the "bottom."

This is built in to the architecture by the operation of instructions like `pushq`, `popq`, `call`, and `ret`. A `push` instruction pushes a value onto the stack. This both modifies the stack pointer (making it smaller) and modifies the stack segment (by moving data there). For instance, the instruction `pushq X` means:

```
subq $8, %rsp
movq X, (%rsp)
```

And `popq X` undoes the effect of `pushq X`. It means:

```
movq (%rsp), X
addq $8, %rsp
```

`X` can be a register or a memory reference.

The portion of the stack reserved for a function is called that function's **stack frame**. Stack frames are aligned: x86-64 requires that each stack frame be a multiple of 16 bytes, and when a `callq` instruction begins execution, the `%rsp` register must be 16-byte aligned. This means that every function's entry `%rsp` address will be 8 bytes off a multiple of 16.

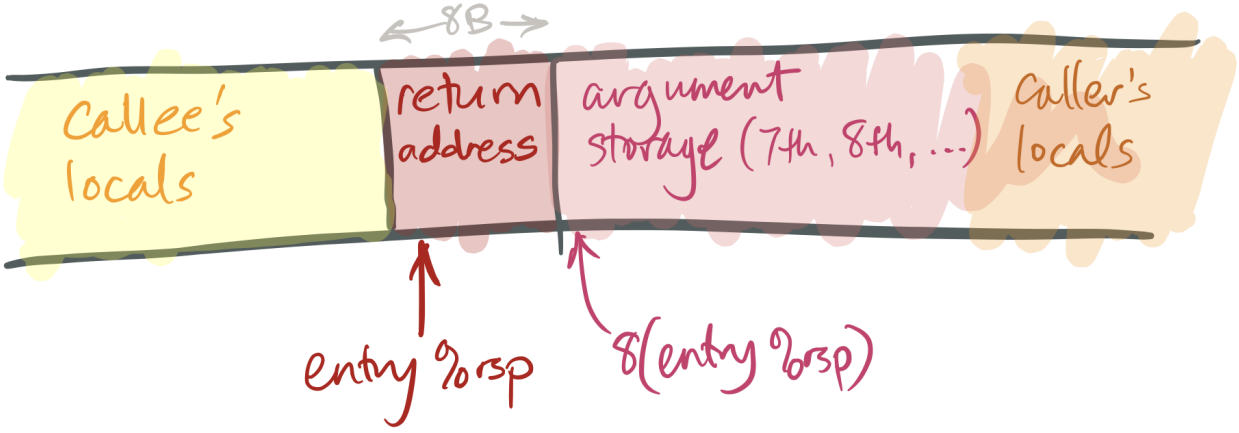
Return address and entry and exit sequence

The steps required to call a function are sometimes called the *entry sequence* and the steps required to return are called the *exit sequence*. Both caller and callee have responsibilities in each sequence.

To prepare for a function call, the caller performs the following tasks in its entry sequence.

1. The caller stores the first six arguments in the corresponding registers.
2. If the callee takes more than six arguments, or if some of its arguments are large, the caller must store the surplus arguments on its stack frame. It stores these in increasing order, so that the 7th argument has a smaller address than the 8th argument, and so forth. The 7th argument must be stored at `(%rsp)` (that is, the top of the stack) when the caller executes its `callq` instruction.
3. The caller saves any caller-saved registers (see below).
4. The caller executes `callq FUNCTION`. This has an effect like `pushq $NEXT_INSTRUCTION; jmp FUNCTION` (or, equivalently, `subq $8, %rsp; movq $NEXT_INSTRUCTION, (%rsp); jmp FUNCTION`), where `NEXT_INSTRUCTION` is the address of the instruction immediately following `callq`.

This leaves a stack like this:



To return from a function:

1. The callee places its return value in `%rax`.
2. The callee restores the stack pointer to its value at entry ("entry `%rsp`"), if necessary.
3. The callee executes the `retq` instruction. This has an effect like `popq %rip`, which removes the return address from the stack and jumps to that address.
4. The caller then cleans up any space it prepared for arguments and restores caller-saved registers if necessary.

Particularly simple callees don't need to do much more than return, but most callees will perform more tasks, such as allocating space for local variables and calling functions themselves.

Callee-saved registers and caller-saved registers

The calling convention gives callers and callees certain guarantees and responsibilities about the values of registers across function calls. Function implementations may expect these guarantees to hold, and must work to fulfill their responsibilities.

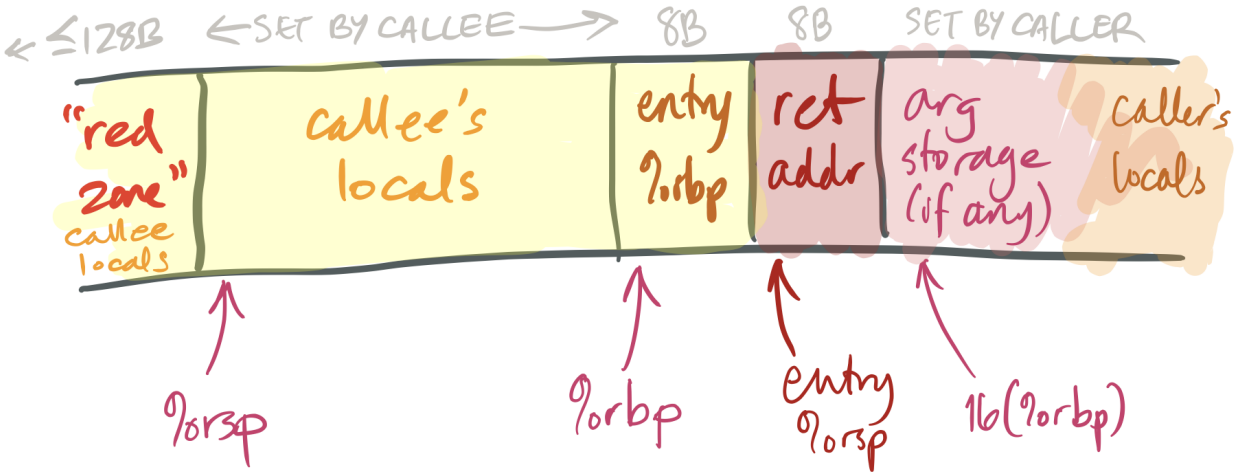
The most important responsibility is that certain registers' values *must be preserved across function calls*. A callee may use these registers, but if it changes them, it must restore them to their original values before returning. These registers are called **callee-saved registers**. All other registers are **caller-saved**.

Callers can simply use callee-saved registers across function calls; in this sense they behave like C++ local variables. Caller-saved registers behave differently: if a caller wants to preserve the value of a caller-saved register across a function call, the caller must explicitly save it before the `callq` and restore it when the function resumes.

On x86-64 Linux, `%rbp`, `%rbx`, `%r12`, `%r13`, `%r14`, and `%r15` are callee-saved, as (sort of) are `%rsp` and `%rip`. The other registers are caller-saved.

Base pointer (frame pointer)

The `%rbp` register is called the *base pointer* (and sometimes the *frame pointer*). For simple functions, an optimizing compiler generally treats this like any other callee-saved general-purpose register. However, for more complex functions, `%rbp` is used in a specific pattern that facilitates debugging. It works like this:



1. The first instruction executed on function entry is `pushq %rbp`. This saves the caller's value for `%rbp` into the callee's stack. (Since `%rbp` is callee-saved, the callee must save it.)
2. The second instruction is `movq %rsp, %rbp`. This saves the current stack pointer in `%rbp` (so `%rbp` = entry `%rsp` - 8).

   This adjusted value of `%rbp` is the callee's "frame pointer." The callee will not change this value until it returns. The frame pointer provides a stable reference point for local variables and caller arguments. (Complex functions may need a stable reference point because they reserve varying amounts of space for calling different functions.)

   Note, also, that the value stored at `(%rbp)` is the *caller's* `%rbp`, and the value stored at `8(%rbp)` is the return address. This information can be used to trace backwards through callers' stack frames by functions such as debuggers.
3. The function ends with `movq %rbp, %rsp; popq %rbp; retq`, or, equivalently, `leave; retq`. This sequence restores the caller's `%rbp` and entry `%rsp` before returning.

Stack size and red zone

Functions execute fast because allocating space within a function is simply a matter of decrementing `%rsp`. This is much cheaper than a call to `malloc` or `new`! But making this work takes a lot of machinery. We'll see this in more detail later; but in brief: The operating system knows that `%rsp` points to the stack, so if a function accesses nonexistent memory near `%rsp`, the OS assumes it's for the stack and transparently allocates new memory there.

So how can a program "run out of stack"? The operating system puts a limit on each function's stack, and if `%rsp` gets too low, the program segmentation faults.

The diagram above also shows a nice feature of the x86-64 architecture, namely the **red zone**. This is a small area *above* the stack pointer (that is, at lower addresses than `%rsp`) that can be used by the currently-running function for local variables. The red zone is nice because it can be used without mucking around with the stack pointer; for small functions `push` and `pop` instructions end up taking time.

Branches

The processor typically executes instructions in sequence, incrementing `%rip` each time. Deviations from sequential instruction execution, such as function calls, are called **control flow transfers**.

Function calls aren't the only kind of control flow transfer. A *branch* instruction jumps to a new instruction without saving a return address on the stack.

Branches come in two flavors, unconditional and conditional. The `jmp` or `j` instruction executes an unconditional branch (like a `goto`). All other branch instructions are conditional: they only branch if some condition holds. That condition is represented by **condition flags** that are set as a side effect of every arithmetic operation.

Arithmetic instructions change part of the `%rflags` register as a side effect of their operation. The most often used flags are:

- **ZF** (zero flag): set iff the result was zero.
- **SF** (sign flag): set iff the most significant bit (the sign bit) of the result was one (i.e., the result was negative if considered as a signed integer).
- **CF** (carry flag): set iff the result overflowed when considered as unsigned (i.e., the result was greater than 2^W-1).
- **OF** (overflow flag): set iff the result overflowed when considered as signed (i.e., the result was greater than $2^{W-1}-1$ or less than -2^{W-1}).

Although some instructions let you load specific flags into registers (e.g., `setz`; see CS:APP3e §3.6.2, p203), code more often accesses them via conditional jump or conditional move instructions.

Instruction	Mnemonic	C example	Flags
j (jmp)	Jump	`break;`	(Unconditional)
je (jz)	Jump if equal (zero)	`if (x == y)`	ZF
jne (jnz)	Jump if not equal (nonzero)	`if (x != y)`	!ZF
jg (jnle)	Jump if greater	`if (x > y)`, signed	!ZF && !(SF ^ OF)
jge (jnl)	Jump if greater or equal	`if (x >= y)`, signed	!(SF ^ OF)
jl (jnge)	Jump if less	`if (x < y)`, signed	SF ^ OF
jle (jng)	Jump if less or equal	`if (x <= y)`, signed	(SF ^ OF) || ZF
ja (jnbe)	Jump if above	`if (x > y)`, unsigned	!CF && !ZF
jae (jnb)	Jump if above or equal	`if (x >= y)`, unsigned	!CF
jb (jnae)	Jump if below	`if (x < y)`, unsigned	CF
jbe (jna)	Jump if below or equal	`if (x <= y)`, unsigned	CF || ZF
js	Jump if sign bit	`if (x < 0)`, signed	SF
jns	Jump if not sign bit	`if (x >= 0)`, signed	!SF
jc	Jump if carry bit	N/A	CF
jnc	Jump if not carry bit	N/A	!CF
jo	Jump if overflow bit	N/A	OF
jno	Jump if not overflow bit	N/A	!OF

The `test` and `cmp` instructions are frequently seen before a conditional branch. These operations perform arithmetic but throw away the result, except for condition codes. `test` performs binary-and, `cmp` performs subtraction.

`cmp` is hard to grasp: remember that `subq %rax, %rbx` performs `%rbx := %rbx - %rax`—the source/destination operand is on the left. So `cmpq %rax, %rbx` evaluates `%rbx - %rax`. The sequence `cmpq %rax, %rbx; jg L` will jump to label `L` if and only if `%rbx` is greater than `%rax` (signed).

The weird-looking instruction `testq %rax, %rax`, or more generally `testq REG, SAMEREG`, is used to load the condition flags appropriately for a single register. For example, the bitwise-and of `%rax` and `%rax` is zero if and only if `%rax` is zero, so `testq %rax, %rax; je L` jumps to `L` if and only if `%rax` is zero.

Sidebar: C++ data structures

C++ compilers and data structure implementations have been designed to avoid the so-called *abstraction penalty*, which is when convenient data structures compile to more and more-expensive instructions than simple, raw memory accesses. When this works, it works quite well; for example, this:

```
long f(std::vector<int>& v) {
    long sum = 0;
    for (auto& i : v) {
        sum += i;
    }
    return sum;
}
```

compiles to this, a very tight loop similar to the C version:

```
        movq    (%rdi), %rax
        movq    8(%rdi), %rcx
        cmpq    %rcx, %rax
        je      .L4
        movq    %rax, %rdx
        addq    $4, %rax
        subq    %rax, %rcx
        andq    $-4, %rcx
        addq    %rax, %rcx
        movl    $0, %eax
.L3:
        movslq  (%rdx), %rsi
        addq    %rsi, %rax
        addq    $4, %rdx
        cmpq    %rcx, %rdx
        jne     .L3
        rep ret
.L4:
        movl    $0, %eax
        ret
```

We can also use this output to infer some aspects of `std::vector`'s implementation. It looks like:

- The first element of a `std::vector` structure is a pointer to the first element of the vector;
- The elements are stored in memory in a simple array;
- The second element of a `std::vector` structure is a pointer to *one-past-the-end* of the elements of the vector (i.e., if the vector is empty, the first and second elements of the structure have the same value).

Assembly 3: Optimizations and assembly

Compiler optimizations

Argument elision

A compiler may decide to elide (or remove) certain operations setting up function call arguments, if it can decide that the registers containing these arguments will hold the correct value before the function call takes place. Let's see an example of a function disassembled function `f` in `f31.s`:

```
subq    $8, %rsp
call    _Z1gi@PLT
addq    $8, %rsp
addl    $1, %eax
ret
```

This function calls another function `g`, adds 1 to `g`'s return value, and returns that value.

It is possible that the function has the following definition in C++:

```
int f() {
    return 1 + g();
}
```

However, the actual definition of `f` in `f31.cc` is:

```
int f(x) {
    return 1 + g(x);
}
```

The compiler realizes that the argument to function `g`, which is passed via register `%rdi`, already has the right value when `g` is called, so it doesn't bother doing anything about it. This is one example of numerous optimizations a compiler can perform to reduce the size of generated code.

Inlining

A compiler may also copy the body of function to its call site, instead of doing an explicit function call, when it decides that the overhead of performing a function call outweights the overhead of doing this copy. For example, if we have a function `g` defined as `g(x) = 2 + x`, and `f` is defined as `f(x) = 1 + g(x)`, then the compiler may actually generate `f(x)` as simply `3 + x`, without inserting any `call` instructions. In assembly terms, function `g` will look like

```
leal    2(%rdi), %eax
ret
```

and `f` will simply be

```
leal    3(%rdi), %eax
ret
```

Tail call elimination

Let's look at another example in `f32.s`:

```
addl    $1, %edi
jmp _Z1gi@PLT
```

This function doesn't even contain a `ret` instruction! What is going on? Let's take a look at the actual definition of `f`, in `f32.cc`:

```
int f(int x) {
    return g(x + 1);
}
```

Note that the call to function `g` is the last operation in function `f`, and the return value of `f` is just the return value of the invocation of `g`. In this case the compiler can perform a *tail call elimination*: instead of calling `g` explicitly, it can simply jump to `g` and have `g` return to the same address that `f` would have returned to.

> A tail call elimination may occur if a function (caller) ends with another function call (callee) and performs no cleanup once the callee returns. In this case the caller and simply jump to the callee, instead of doing an explicit call.

Loop unrolling

Before we jump into loop unrolling, let's take a small excursion into an aspect of calling conventions called caller/callee-saved registers. This will help us under the sample program in `f33.s` better.

Calling conventions: caller/callee-saved registers

Let's look at the function definition in `f33.s`:

```
    pushq   %r12
    pushq   %rbp
    pushq   %rbx
    testl   %edi, %edi
    je  .L4
    movl    %edi, %r12d
    movl    $0, %ebx
    movl    $0, %ebp
.L3:
    movl    %ebx, %edi
    call    _Z1gj@PLT
    addl    %eax, %ebp
    addl    $1, %ebx
    cmpl    %ebx, %r12d
    jne .L3
.L1:
    movl    %ebp, %eax
    popq    %rbx
    popq    %rbp
    popq    %r12
    ret
.L4:
    movl    %edi, %ebp
    jmp .L1
```

From the assembly we can tell that the backwards jump to `.L3` is likely a loop. The loop index is in `%ebx` and the loop bound is in `%r12d`. Note that upon entry to the function we first moved the value `%rdi` to `%r12d`. This is necessary because in the loop `f` calls `g`, and `%rdi` is used to pass arguments to `g`, so we must move its value to a different register to used it as the loop bound (this case `%r12`). But there is more to this: the compiler also needs to ensure that this register's value is preserved across function calls. Calling conventions dictate that certain registers always exhibit this property, and they are called **callee-saved registers**. If a register is callee-saved, then the caller doesn't have to save its value before entering a function call.

We note that upon entry to the function, `f` saved a bunch of registers by pushing them to the stack: `%r12`, `%rbp`, `%rbx`. It is because all these registers are callee-saved registers, and `f` uses them during the function call. In general, the following registers in x86_64 are callee-saved:
`%rbx`, `%r12`–`%r15`, `%rbp`, `%rsp` (`%rip`)

All the other registers are **caller-saved**, which means the callee doesn't have to preserve their values. If the caller wants to reuse values in these registers across function calls, it will have to explicitly save and restore these registers. In general, the following registers in x86_64 are caller-saved:
`%rax`, `%rcx`, `%rdx`, `%r8`–`%r11`

Now let's get back to loop unrolling. Let us a look at the program in `f34.s`:

```
    testl   %edi, %edi
    je  .L7
    leal    -1(%rdi), %eax
    cmpl    $7, %eax
    jbe .L8
    pxor    %xmm0, %xmm0
    movl    %edi, %edx
    xorl    %eax, %eax
    movdqa  .LC0(%rip), %xmm1
    shrl    $2, %edx
    movdqa  .LC1(%rip), %xmm2
.L5:
    addl    $1, %eax
    paddd   %xmm1, %xmm0
    paddd   %xmm2, %xmm1
    cmpl    %edx, %eax
    jb  .L5
    movdqa  %xmm0, %xmm1
    movl    %edi, %edx
    andl    $-4, %edx
    psrldq  $8, %xmm1
    paddd   %xmm1, %xmm0
    movdqa  %xmm0, %xmm1
    cmpl    %edx, %edi
    psrldq  $4, %xmm1
    paddd   %xmm1, %xmm0
    movd    %xmm0, %eax
    je  .L10
.L3:
    leal    1(%rdx), %ecx
    addl    %edx, %eax
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    2(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    3(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    4(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    5(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    leal    6(%rdx), %ecx
    cmpl    %ecx, %edi
    je  .L1
    addl    %ecx, %eax
    addl    $7, %edx
```

```
    leal    (%rax,%rdx), %ecx
    cmpl    %edx, %edi
    cmovne  %ecx, %eax
    ret
.L7:
    xorl    %eax, %eax
.L1:
    rep ret
.L10:
    rep ret
.L8:
    xorl    %edx, %edx
    xorl    %eax, %eax
    jmp .L3
```

Wow this looks long and repetitive! Especially the section under label `.L3`! If we take a look at the original function definition in `f34.cc`, we will find that it's almost the same as `f33.cc`, except that in `f34.cc` we know the definition of `g` as well. With knowledge of what `g` does the compiler's optimizer decides that unrolling the loop into 7-increment batches results in faster code.

Code like this can become difficult to understand, especially when the compiler begins to use more advanced registers reserved for vector operations. We can fine-tune the optimizer to disable certain optimizations. For example, we can use the `-mno-sse -fno-unroll-loops` compiler options to disable the use of SSE registers and loop unrolling. The resulting code, in `f35.s`, for the same function definitions in `f34.cc`, becomes much easier to understand:

```
    testl   %edi, %edi
    je  .L4
    xorl    %edx, %edx
    xorl    %eax, %eax
.L3:
    addl    %edx, %eax
    addl    $1, %edx
    cmpl    %edx, %edi
    jne .L3
    rep ret
.L4:
    xorl    %eax, %eax
    ret
```

Note that the compiler still performed inlining to eliminate function `g`.

Optimizing recursive functions

Let's look at the following recursive function in `f36.cc`:

```
int f(int x) {
    if (x > 0) {
        return x * f(x - 1);
    } else {
        return 0;
    }
}
```

At the first glance it may seem that the function returns factorial of `x`. But it actually returns 0. Despite it doing a series of multiplications, in the end it always multiplies the whole result with 0, which produces 0.

When we compile this function to assembly without much optimization, we see the expensive computation occurring:

```
    movl    $0, %eax
    testl   %edi, %edi
    jg  .L8
    rep ret
.L8:
    pushq   %rbx
    movl    %edi, %ebx
    leal    -1(%rdi), %edi
    call    _Z1fi
    imull   %ebx, %eax
    popq    %rbx
    ret
```

In `f37.cc` there is an actual factorial function:

```
int f(int x) {
    if (x > 0) {
        return x * f(x - 1);
    } else {
        return 1;
    }
}
```

If we compile this function using level-2 optimization (`-O2`), we get the following assembly:

```
    testl   %edi, %edi
    movl    $1, %eax
    jle .L4
.L3:
    imull   %edi, %eax
    subl    $1, %edi
    jne .L3
    rep ret
.L4:
    rep ret
```

There is no `call` instructions again! The compiler has transformed the recursive function into a loop.

If we revisit our "fake" factorial function that always returns 0, and compile it with `-O2`, we see yet more evidence of compiler's deep understanding of our program:

```
xorl    %eax, %eax
ret
```

Optimizing arithmetic operations

The assembly code in `f39.s` looks like this:

```
leal    (%rdi,%rdi,2), %eax
leal    (%rdi,%rax,4), %eax
ret
```

It looks like some rather complex address computations! The first `leal` instruction basically loads `%eax` with value `3*%rdi` (or `%rdi + 2*%rdi`). The second `leal` multiplies the previous result by another 4, and adds another `%rdi` to it. So what it actually does is `3*%rdi*4 + %rdi`, or simply `13*%rdi`. This is also revealed in the function name in `f39.s`.

The compiler choose to use `leal` instructions instead of an explicit multiply because the two `leal` instructions actually take less space.

Assembly 4: Buffer overflows

The `storage1/attackme.cc` file contains a particularly dangerous kind of undefined behavior, a **buffer overflow**. In a buffer overflow, an untrusted input is transferred into a *buffer*, such as an array of characters on the stack, without checking to see whether the buffer is big enough for the untrusted input.

Historically buffer overflows have caused some of the worst, and most consequential, C and C++ security holes known. They are particularly bad when the untrusted input derives from the network: then breaking into a network can be as simple as sending a packet.

The buffer overflow in `attackme.cc` derives from a **checksum function**. Our simple `checksum` takes in a pointer to the buffer, then copies that buffer to a local variable, `buf`, and processes the copy. Unfortunately, `checksum` doesn't verify that the input fits its buffer, and if the user supplies too big an argument, `checksum`'s buffer will overflow!

Buffer overflows cause undefined behavior—it's illegal to access memory outside any object. Undefined behavior is always bad, but some is worse than others; and this particular buffer overflow allows the attacker to completely own the victim program, causing it to execute *any shell command*!

The attack works as follows.

- The function's entry sequence allocates local variable space with `subq $112, %rsp`. Examining the assembly we can infer that `buf` is stored at this `%rsp`.
- The function's return address is stored at `112(%rsp)`, which is `buf.c + 112`.
- Any input `arg` with `strlen(arg) > 99` will overflow the buffer. (The 100th character is the null character that ends the string). But `arg`s with 100 to 111 characters won't cause problems, because the remaining 12 bytes of local variable space are unused (they're present to ensure stack frame alignment).
- `arg`s of 112 or more characters, however, will run into the stack slot reserved for `checksum`'s return address! An attacker can put any value they want in that location, as long as the value contains no null characters (since `checksum`'s buffer copy stops when it encounters the end of the input string).
- Overflowing the return address slot causes harm when `checksum` returns. Examining the state during the exit sequence, we see that the immediately previous instructions load `%rdi` with `buf` and call `finish_checksum`. But what a coincidence—`finish_checksum` does not modify `%rdi`! Thus, when `checksum` returns, `%rdi` will be loaded with the address of `buf`.
- Our attacker therefore supplies a carefully-chosen string that overwrites `checksum`'s return address with *the address of the `run_shell` function*. This will cause `run_shell` to run the programs defined in the initial portion of the string!

Thus, the attacker has taken control of the victim by **returning** to an unexpected library function, with carefully-chosen arguments. This attack is called a **return-to-libc attack**.

> A version of `checksum`'s copy-to-aligned-buffer technique is actually useful in practice, but real code would use a smaller buffer, processed one slice at a time.

Defending against return-to-libc attacks and buffer overflows

Return-to-libc attacks used to be pretty trivial to find and execute, but recent years have seen large improvements in the robustness of typical C and C++ programs to attack. Here are just a few of the defenses we had to disable for the simple `attackme` attack to work.

Register arguments: In older architectures, function arguments were *all* passed on the stack. In those architectures attackers could easily determine not only the *function* returned to, but also *its arguments* (a longer buffer overflow would overwrite the stack region interpreted by the "return-callee" as arguments). `attackme` only works because `finish_checksum` happens not to modify its argument.

Modern operating systems and architectures support **position-independent code** and **position-independent executables**. These features make programs work independent of specific addresses for functions, and the operating system can put a program's functions at different addresses each time the program runs. When a program is position-independent, the address of the attacker's desired target function can't be predicted, so the attacker has to get lucky. (The x86-64 designers were smart to add `%rip`-relative addressing, since that's what enables efficient position-independent executables!)

Finally, modern compilers use a technique called **stack protection** or **stack canaries** to detect buffer overflows and stop `retq` from returning to a corrupted address. This is a super useful technique.

> It's called a "canary" in reference to the phrase "canary in a coal mine".

1. The operating system reserves a special tiny memory segment when the program starts running. This tiny memory segment is used for special purposes, such as **thread-local storage**. Although this segment can be addressed normally—it exists in memory—the operating system also ensures it can be accessed through special instruction formats like this:

   ```
   movq %fs:0x28, %rax
   ```

   The `%fs:` prefix tells the processor that `0x28` (a memory offset) is to be measured relative to the start of the thread-local storage segment.

2. A specific memory word in thread-local storage is reserved for a canary value. The operating system sets this value to a random word when starting the program.

3. The compiler adds code to function entry and exit sequences that use this value. Specifically, something like this is added to entry:

   ```
   movq %fs:0x28, REG    # load true canary to register
   pushq REG             # push canary to stack
   ```

   And something like this is added to exit:

   ```
   popq REG              # pop canary from stack
   xorq %fs:0x28, REG    # xor with true canary
   jne fail              # fail if they differ
   ```

   where the `fail` branch calls a library-provided function such as `__stack_chk_fail`.

The pushed canary is located between the function's buffers (its local variables) and the function's return address. Given that position, any buffer overflow that reaches the return address will also overwrite the canary! And the attacker can't easily guess the canary or overwrite the true canary's memory location, since both are random.

If the stack canary and the true canary don't match, the function can infer that it executed some undefined behavior. Maybe the return address was corrupted and maybe it wasn't; either way, executing undefined behavior means the program is broken, and it is safe to exit immediately.

These techniques, and others like them, are incredibly useful, important, fun, and good to understand. But they don't make C programming safe: attackers are persistent and creative. (Check out return-oriented programming.) Memory-unsafe languages like C and C++ make programming inherently risky; only programs with no undefined behavior are safe. (And they're only safe if the underlying libraries and operating system have no undefined behavior either!) Good C and C++ programmers need a healthy respect for—one might say fear of—their tools. Code cleanly, use standard libraries where possible, and test extensively under sanitizers to catch bugs like buffer overflows.

Or listen to cranky people like Trevor Jim, who says that "C is bad and you should feel bad if you don't say it is bad" and also "Legacy C/C++ code is a nuclear waste nightmare that will make you WannaCry".

On checksums

A **checksum** is a short, usually fixed-length summary of a data buffer. Checksums have two important properties:

1. If two buffers contain the same data (the same bytes in the same order), then their checksums **must** equal. (This property is mandatory.)
2. If two buffers contain *different* data, then their checksums **should not** be equal. (This property is optional: it is OK for distinct data to have equal checksums, though in a good checksum function, this should be rare.)

Checksums have many uses, but they are often used to detect errors in data transcription. For example, most network technologies use checksums to detect link errors (like bursts of noise). To send data over a network, the sender first computes a checksum of the data. Then it transmits both data and checksum over the possibly-noisy network channel. The receiver computes its own checksum of the received data, and compares that with the sender's checksum. If the checksums match, that's a good sign that the data was transmitted correctly. If the checksums don't match, then the data can't be trusted and the receiver throws it away.

Most checksum functions map from a large domain to a smaller range—for instance, from the set of all variable-length buffers to the set of 64-bit numbers. Such functions *must* sometimes map unequal data to equal checksums, so some errors must go undetected; but a good checksum function detects common corruption errors all, or almost all, the time. For example, some checksum functions can *always* detect a single flipped bit in the buffer.

The requirements on checksums are the same as the requirements for hash codes: equal buffers have equal checksums (hash codes), and distinct buffers should have distinct checksums (hash codes). A good hash code can be a good checksum and vice versa.

The most important characteristics of a checksum function are speed and strength. Fast checksums are inexpensive to compute, but easy to break, meaning that many errors in the data aren't detected by the checksum function. Widely-used fast checksum functions include the IPv4/TCP/UDP checksum and cyclic redundancy checks; our `checksum` function is a lot like the IPv4 checksum. Strong checksums, also known as cryptographic hash functions, are hard to break. Ideally they are *infeasible* to break, meaning that no one knows a procedure for finding two buffers that have the same checksum. Widely-used strong checksum functions

include SHA-256, SHA-512, and SHA-3. There are also many formerly-strong checksum functions, such as SHA-1, that have been broken—meaning that procedures are known for finding two buffers with the same checksum—but are still stronger in practice than fast checksums.

References: Checksums on Wikipedia; CS 225 explores some related theory.

Storage 1: Caches

First part of lecture: Buffer overflow

Caches

A **cache** is **a small amount of fast storage used to speed up access to larger, slower storage**.

A cache basically works by storing **copies** of data whose primary home is on slower storage. A program can access data faster when a it's located "nearby," in fast storage.

Caches abound in computer systems; sometimes it seems like caching is the only performance-improving idea in systems. Processors have caches for primary memory. The operating system uses most of primary memory as a cache for disks and other stable storage devices. Running programs reserve some of their private memory to cache the operating system's cache of the disk.

When a program processes a file, the actual computation is done using registers, which are caching values from the processor cache, which is caching data from the running program's memory, which is caching data from the operating system, which is caching data from the disk. And modern disks contain caches inside their hardware too!

Caches also abound in everyday life. Imagine how life would differ without, say, food storage—if every time you felt hungry you had to walk to a farm and eat a carrot you pulled out of the dirt. Your whole day would be occupied with finding and eating food! Instead, your refrigerator (or your dorm's refrigerator) acts as a cache for your neighborhood grocery store, and that grocery store acts as a cache for all the food producers worldwide. Caching food is an important aspect of complex animals' biology too. Your stomach and intestines and fat cells act as caches for the energy stored in the food you eat, allowing you to spend time doing things other than eating. And your colon and bladder act as caches for the waste products you produce, allowing you to travel far from any toilet.

(Link not about caches: Anna's Since Magic Show Hooray!)

File I/O

The problem set focuses on **file I/O**—that is, input and output ("I" and "O") for files. You're likely familiar with files already. The `stdin`, `stdout`, and `stderr` objects in C are references to files, as are C++'s `std::cin`, `std::cout`, and `std::cerr`. C library functions like `fopen`, `fread`, `fprintf`, `fscanf`, and `fclose` all perform operations on files. These library routines are part of a package called **standard I/O** or **stdio** for short. They are implemented in terms of lower-level abstractions; the Linux/Mac OS X versions are called file descriptors, which are accessed using system calls such as `open`, `read`, and `write`. Although file descriptors and system calls are often hidden from programmers by stdio, we can also access them directly.

We used simple I/O benchmark programs to evaluate the real-world impact of caches. Specifically, we ran many programs that write data to disk—my laptop's SSD drive, via my virtual machine. These programs accomplish the same goal, but in several different ways.

- **`w01-syncbyte`** writes the file (1) one byte at a time, (2) using direct system calls (`write`), (3) **synchronously** to the disk.

"Synchronously" means that each `write` operation completes only when the data has made its way through all layers of caching out to the true disk hardware. (At least that's what it *should* mean!) We request synchronous writes by opening the file with the `O_SYNC` option to `open`.

On my laptop, `w01-syncbyte` can write about 2,400 bytes per second (it varies).

- **`w03-byte`** writes the file (1) one byte at a time, (2) using direct system calls (`write`), (3) **asynchronously**—in other words, allowing the operating system to use its caches.

  On my laptop, `w03-byte` can write about 870,000 bytes per second—360x more than `w01-syncbyte`!

- **`w05-stdiobyte`** writes the file (1) one byte at a time, (2) using stdio library function calls (`fwrite`), (3) asynchronously.

  On my laptop, `w05-stdiobyte` can write about 52,000,000 bytes per second—22,000x more than `w01-syncbyte`!

Even though all three programs write one byte at a time, caching makes the fastest one 22,000x faster. The `w05-stdiobyte` program uses two distinct layers of software cache: a **stdio cache**, managed by the stdio library inside the program, uses caching to reduce the cost of system calls; and the **buffer cache**, managed by the operating system inside the kernel, uses caching to reduce the cost of disk access. The combination of these is 22,000x.

We can get even faster, though, if we write more than one byte at a time, which is sort of like implementing an *application-level* cache for the data we plan to write.

- **`w02-syncblock`** writes the file (1) **512** bytes at a time, (2) using system calls, (3) synchronously to the disk. On my laptop, it can write about 1,300,000 bytes per second.
- **`w04-block`** writes the file (1) 512 bytes at a time, (2) using system calls, (3) asynchronously. On my laptop, it can write about 200,000,000 bytes per second.
- **`w06-stdioblock`** writes the file (1) 512 bytes at a time, (2) using stdio, (3) asynchronously. On my laptop, it can write about 270,000,000 bytes per second—112,500x more than `w01-syncbyte`!

Synchronous writes in depth

What kinds of I/O properties make caches necessary?

We looked first at `w01-syncbyte`. Here's what happens at a high level when that application writes bytes to the file.

1. The application makes a `write` system call asking the operating system to write a byte.
2. The operating system performs this write to its cache. But it also observes that the written file descriptor requires synchronous writes, so it begins the process of writing to the drive—in my case, an SSD.
3. This drive **cannot** write a single byte at a time! Most storage technologies other than primary memory read and write data in contiguous blocks, typically 4,096 or 8,192 bytes big. So for every 1-byte application write, the operating system must write an additional 4,095 surrounding bytes, just because the disk forces it to!

   That 4,096-byte write is also slow, because it induces physical changes. The job of a disk drive is to preserve data in a **stable** fashion, meaning that the data will outlast accidents like power loss. On today's technology, the physical changes that make data stable take more time than simple memory writes.

But that's not the worst part. SSD drives have a strange feature where data can be *written* in 4KiB or 8KiB units, but *erasing* already-written data requires much larger units—typically **64KiB** or even 256KiB big! The large erase, plus other SSD features like "wear leveling," may cause the OS and drive to move all kinds of data around: our 1-byte application write can cause 256KiB or more traffic to the disk. This explosion is called **write amplification**.

4. So the system call only returns after 4–256KiB of data is written to the disk.

Buffer cache

We can speed up this process using caches, and the first cache we "turned on" was the **buffer cache**. This is a cache **in memory** of file data managed by the operating system **kernel**. (The kernel is the operating system program that runs with full privilege over the operation of the machine. The kernel responds to system calls from application programs and makes calls out to hardware as required.)

In today's machines and operating systems, the buffer cache ends up occupying much of the machine's memory. It can hold data for many files, and for many drives and sources of storage, simultaneously.

Here's what happens at a high level when `w03-byte` writes bytes to the file.

1. The application makes a `write` system call asking the operating system to write a byte (as before).
2. The operating system performs this write to the buffer cache.
3. **The operating system immediately returns to the application!**

That is, the application gets control back before the data is written to disk. The data *is* written to disk eventually, either after some time goes by or because the buffer cache runs out of room, but such later writes can be overlapped with other processing.

Stdio cache

But `w03-byte` is still much slower than it needs to be, because it performs many expensive operations—namely **system calls**. These operating system invocations are slow because they require the processor to save its state, switch contexts to the kernel, process arguments, and then switch back. Another layer of caching, the **stdio cache**, can get rid of these system calls. The stdio cache is a cache in **application memory** for the buffer cache (which is in kernel memory).

Here's what happens at a high level when `w05-stdiobyte` writes bytes to the file.

1. The application makes an `fwrite` library function call asking the stdio library to write a byte.
2. The library performs this write to its cache and **immediately returns to the application**.

That is, the application gets control back even before a system call is made. The data is written out of the stdio cache using a system call eventually, either when the stdio cache runs out of room, it is explicitly flushed, or the program exits.

Storage 2: Cache model

Cost of storage

We think a lot in computer science about costs: time cost, space cost, memory efficiency. And these costs fundamentally shape the kinds of solutions we build. But *financial* costs also shape the systems we build—in some ways more fundamentally—and the costs of storage technologies have changed at least as dramatically as their capacities and speeds.

This table gives the price per megabyte of different storage technology, in price per megabyte (2010 dollars), up to 2018.

Year	Memory (DRAM)	Flash/SSD	Hard disk
~1955	$411,000,000		$6,230
1970	$734,000.00		$260.00
1990	$148.20		$5.45
2003	$0.09	$0.305	$0.00132
2010	$0.019	$0.00244	$0.000073
2018	$0.0059	$0.00015	$0.000020

The same costs *relative* to the cost of a hard disk in ~2018:

Year	Memory	Flash/SSD	Hard disk
~1955	20,500,000,000,000		312,000,000
1970	36,700,000,000		13,000,000
1990	7,400,000		270,000
2003	4,100	15,200	6.6
2010	950	122	3.6
2018	29.5	7.5	1

These are initial purchase costs and don't include maintenance or power. DRAM uses more energy than hard disks, which use more than flash/SSD. (Prices from here and here. $1.00 in 1955 had "the same purchasing power" as $9.45 in 2018 dollars.)

There are many things to notice in this table, including the relative changes in prices over time. When flash memory was initially introduced, it was 2300x more expensive per megabyte than hard disks. That has dropped substantially; and that drop explains why large amounts of fast SSD memory is so ubiquitous on laptops and phones.

Performance metrics

Latency measures the time it takes to complete a request. Latency is bad: lower numbers are better.

Throughput (also called **bandwidth**) measures the number of operations that can be completed per unit of time. Throughput is good: higher numbers are better.

In storage, latency is typically measured in seconds and throughput in operations per second, or in bytes per second or megabytes per second (assuming the operations transfer different numbers of bytes).

The best storage media have **low latency** and **high throughput**: it takes very little time to complete a request, and many units of data can be transferred per second.

Latency and throughput are not simple inverses: a system can have high latency *and* high throughput. A classic example of this is "101Net," a high-bandwidth, high-latency network that has many advantages over the internet. Rather than send terabytes of data over the network, which is expensive and can take forever, just put some disk drives in a car and drive it down the freeway to the destination data center. The high information density of disk drives makes this high-latency, high-throughput channel hard to beat in terms of throughput and expense for very large data transfers. Amazon Web Services offers physical-disk transport; you can mail them a disk using the Snowball service, or if you have around 100 petabytes of storage (that's roughly 10^{17} bytes!!?!?!?!?!) you can send them **a shipping container of disks on a truck**. Look, here it is.

"Successful *startups*, given the way they collect data, have exabytes of data," says the man introducing the truck, so the truck is filled with 10^{17} bytes of tracking data about how long it took you to favorite your friend's Pokemon fashions or whatever and we are not not living in hell.

(101Net is a San Francisco version of the generic term sneakernet. XKCD took it on.)

Stable and volatile storage

Computer storage technologies can be either stable or volatile.

Stable storage is robust to power outages. If the power is cut to a stable storage device, the stored data doesn't change.

Volatile storage is not robust to power outages. If the power is cut to a volatile storage device, the stored data is corrupted and usually considered totally lost.

Registers, SRAM (static random-access memory—the kind of memory used for processor caches), and DRAM (dynamic random-access memory—the kind of memory used for primary memory) are volatile. Disk drives, including hard disks, SSDs, and other related media are stable.

Aside: Stable DRAM?

However, even DRAM is more stable than one might guess. If someone wants to steal information off your laptop's memory, they might grab the laptop, cool the memory chips to a low temperature, break open the laptop and transplant the chips to another machine, and read the data there.

Here's a video showing DRAM degrading after a power cut, using the Mona Lisa as a source image.

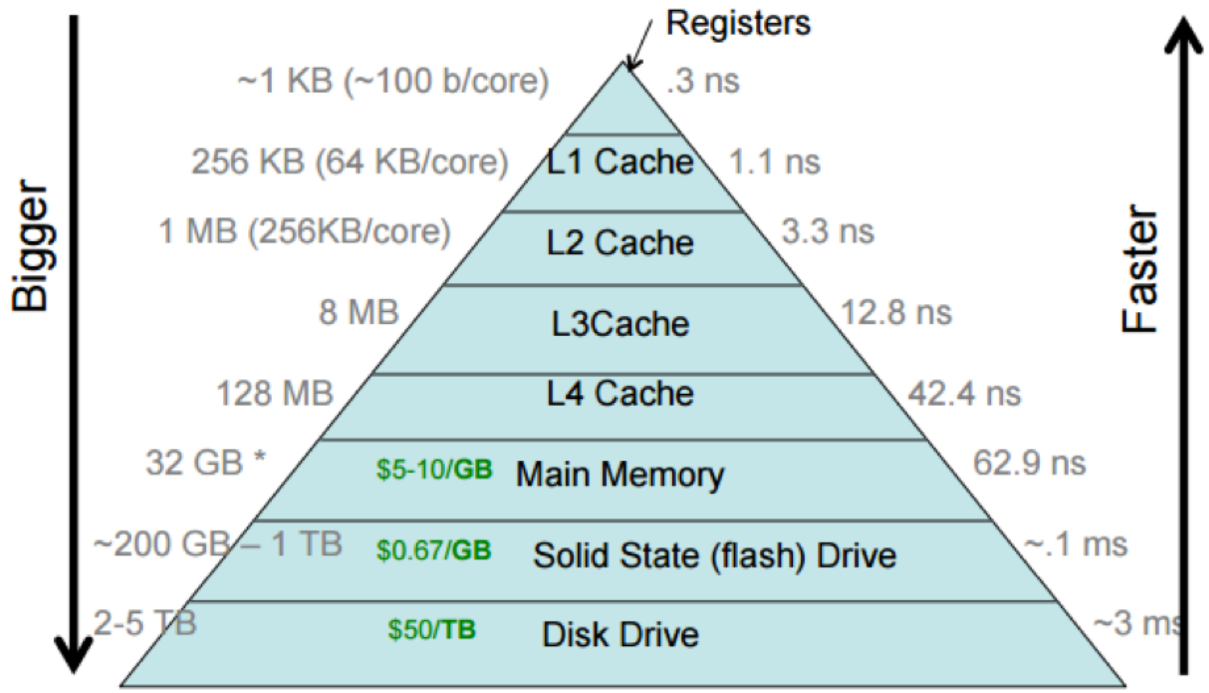
Storage performance

This table lists the storage technologies in a typical computer, starting from the smallest and fastest and ending at the largest and slowest.

Storage type	Capacity	Latency	Throughput (random access)	Throughput (sequential)
Registers	~30 (1KB)	0.5 ns	16 GB/sec (2×10^9 accesses/sec)	
SRAM (processor caches)	9MB	4 ns	1.6 GB/sec (2×10^8 accesses/sec)	
DRAM (main memory)	8GB	70 ns	100 GB/sec	
SSD (stable storage)	512GB	60 µs	550 MB/sec	
Hard disk	2–5TB	4–13 ms	1 MB/sec	200 MB/sec

The performance characteristics of hard disks are particularly interesting. Hard disks can access data in **sequential** access patterns much faster than in **random** access patterns. (More on access patterns) This is due to the costs associated with moving the mechanical "head" of the disk to the correct position on the spinning platter. There are two such costs: the **seek time** is the time it takes to move the head to the distance from the center of the disk, and the **rotational latency** is the time it takes for the desired data to spin to underneath the head. Hard disks have much higher sequential throughput because once the head is in the right position, neither of these costs apply: sequential regions of the disk can be read as the disk spins. This is just like how a record player plays continuous sound as the record spins underneath the needle. A typical recent disk might rotate at 7200 RPM and have ~9ms average seek time, ~4ms average rotational latency, and 210MB/s maximum sustained transfer rate. (*CS:APP3e*)

These technologies are frequently called the **storage hierarchy**, and shown this way:



The storage hierarchy shows the processor caches divided into multiple **levels**, with the L1 cache (sometimes pronounced "level-one cache") closer to the processor than the L4 cache. This reflects how processor caches are actually laid out, but we often think of a processor cache as a single unit.

Different computers have different sizes and access costs for these hierarchy levels; the ones above are typical, but here are some more, based on those for my desktop iMac, which has four cores (i.e., four independent processors): ~1KB of registers; ~9MB of processor cache; 8GB primary memory; 512GB SSD; and 2TB hard disk. The processor cache divides into three levels. There are 128KB of L1 cache, divided into four 32KB components; each L1 cache is accessed only by a single core, which makes accessing it faster, since there's no need to arbitrate between cores. There are 512KB of L2 cache, divided into two 256KB components (one per "socket", or pair of cores). And there are 8MB of L3 cache shared by all cores.

And distributed systems have yet lower levels. For instance, Google and Facebook have petabytes or exabytes of data, distributed among many computers. You can think of this as another storage layer, **networked storage**, that is typically slower to access than HDDs.

Each layer in the storage hierarchy acts as a cache for the following layer.

Latency Numbers Every Programmer Should Know

Cache model

We describe a generic cache as follows.

Fast storage is divided into **slots**.

- Each slot is capable of holding data from slow storage.
- A slot is **empty** if it contains no data. It is **full** if it contains data from slow storage.
- Each slot has a name (for instance, we might refer to "the first slot" in a cache).

Slow storage is divided into **blocks**.

- Each block can fit in a slot.
- Each block has an **address**.
- If a cache slot (in fast storage) is full, meaning it holds data from a block (in slow storage), it must also know the address of that block.

There are some terms that are used in the contexts of specific levels in the hierarchy; for instance, a **cache line** is a slot in the processor cache (or, sometimes, a block in memory).

Read caches must respond to user requests for data at particular addresses. On each access, a cache typically checks whether the specified block is already loaded into a slot. If it is, the cache returns that data; otherwise, the cache first loads the block into some slot, then returns the data from the slot.

A cache access is called a **hit** if the data is already loaded into a cache slot, and a **miss** otherwise. Cache hits are good because they are cheap. Cache misses are bad: they incur both the cost of accessing the cache and the cost of accessing the slower storage. We want most accesses to hit. That is, we want a high **hit rate**, where the hit rate is the fraction of accesses that hit.

Eviction policies and reference strings

When a cache becomes full, we may need to evict data currently in the cache to accommodate newly required data. The policy the cache uses to decide which existing slot to vacate (or evict) is called the **eviction policy**.

We can reason about different eviction policies using sequences of addresses called **reference strings**. A reference string represents the sequence of blocks being accessed by the cache's user. For instance, the reference string "1 2 3 4 5 6 7 8 9" represents nine sequential accesses.

For small numbers of slots, we can draw how a cache responds to a reference string for a given number of slots and eviction policy. Let's take the reference string "1 2 3 4 1 2 5 1 2 3 4 5", and an *oracle* eviction policy that makes the best possible eviction choice at every step. Here's how that might look. The cache slots are labeled with Greek letters; the location of an access shows what slot was used to fill that access; and misses are circled.

	1	2	3	4	1	2	5	1	2	3	4	5
α	①				1			1		③		
β		②				2			2		④	

	1	2	3	4	1	2	5	1	2	3	4	5
γ			③	④			⑤					5

Optimal, 3 slots, hit rate 5/12

> Some notes on this diagram style: At a given point in time (a given column), read the cache contents by looking at the rightmost numbers in or before that column. The least-recently-used slot has the longest distance to the most recent reference before the column. And the least-recently-*loaded* slot has the longest distance to the most recent *miss* before the column (the most recent circled number).

The hit rate is 5/12.

What the oracle's doing is the optimal cache eviction policy: **Bélády's optimal algorithm**, named after its discoverer, László Bélády (the article). This algorithm selects for eviction *a slot whose block is used farthest in the future* (there might be multiple such slots). It is possible to prove that Bélády's algorithm is optimal: no algorithm can do better.

Unfortunately, it is possible to use the optimal algorithm only when the complete sequence of future accesses is known. This is rarely possible, so operating systems use other algorithms instead.

LRU

The most common algorithms are variants of **Least Recently Used (LRU) eviction**, which selects for eviction a slot that was least recently used in the reference string. Here's LRU working on our reference string and a three-slot cache.

	1	2	3	4	1	2	5	1	2	3	4	5
α	①			④			⑤			③		
β		②			①			1			④	
γ			③			②			2			⑤

LRU, 3 slots, hit rate 2/12

LRU assumes **locality of reference**: if we see a reference, then it's likely we will see another reference to the same location soon.

FIFO

Another simple-to-specify algorithm is **First-In First-Out (FIFO) eviction**, also called **round-robin**, which selects for eviction the slot that was least recently *evicted*. Here's FIFO on our reference string.

	1	2	3	4	1	2	5	1	2	3	4	5
α	①			④			⑤					5
β		②			①			1		③		
γ			③			②			2		④	

FIFO, 3 slots, hit rate 3/12

Larger caches

We normally expect that bigger caches work better. If a cache has more slots, it should have a higher hit rate. And this is generally true—but not always! Here we raise the number of slots to 4 and evaluate all three algorithms.

	1	2	3	4	1	2	5	1	2	3	4	5
α	①				1			1			④	
β		②				2			2			
γ			③							3		
δ				④			⑤					5

Optimal, 4 slots, hit rate 6/12

	1	2	3	4	1	2	5	1	2	3	4	5
α	①				1			1				⑤
β		②				2			2			
γ			③				⑤				④	
δ				④						③		

LRU, 4 slots, hit rate 4/12

	1	2	3	4	1	2	5	1	2	3	4	5
α	①				1		⑤				④	
β		②				2		①				⑤
γ			③						②			
δ				④						③		

FIFO, 4 slots, hit rate 2/12

The optimal and LRU algorithms' hit rates improved, but FIFO's hit rate dropped!

This is called **Bélády's anomaly** (the same Bélády): for some eviction algorithms, hit rate can decrease with increasing numbers of slots. FIFO suffers from the anomaly. But even some simple algorithms do not: LRU is anomaly-proof.

> The argument is pretty simple: Adding a slot to an LRU cache will never turn a hit into a miss, because if an *N*-slot LRU cache and an (*N*+1)-slot LRU cache process the same reference string, the contents of the *N*-slot cache are a subset of the contents of the (*N*+1)-slot cache at every step. This is because for all *N*, an LRU cache with *N* slots always contains the *N most*-recently-used blocks—and the *N* most-recently-used blocks are always a subset of the *N*+1 most-recently-used blocks.

Implementing eviction policies

The cheapest policy to implement is FIFO, because it doesn't keep track of any facts about the reference string (the least-recently-evicted slot is independent of reference string). LRU requires the cache to remember when slots were accessed. This can be implemented precisely, by tracking every access, or imprecisely, with sampling procedures of various kinds. Bélády's algorithm can only be implemented if the eviction policy knows future accesses, which requires additional machinery—either explicit information from cache users or some sort of profiling information.

Cache associativity

Let's a consider a cache with *S* total slots for *B* total blocks (of slow storage).

- In a **fully-associative cache**, any block can fit in any slot.
- In a **set-associative cache**, the block with address *A* can fit in a subset of the slots called *C*(*A*). For example, a cache might reserve even-numbered slots for blocks with even-numbered addresses.
- In a **direct-mapped cache**, each block can fit in *exactly one* slot. For example, a cache might require that the block with address *A* go into slot number (*A* mod *S*).

Fully-associative and direct-mapped caches are actually special cases of set-associative caches. In a fully-associative cache, *C*(*A*) always equals the whole set of slots. In a direct-mapped cache, *C*(*A*) is always a singleton.

Another special case is a **single-slot cache**. In a single-slot cache, *S* = 1: there is only one slot. A single-slot cache is kind of both direct-mapped and fully-associative. The stdio cache is a single-slot cache.

Single-slot caches and unit size

You might think that single-slot caches are useless unless the input reference string has lots of repeated blocks. But this isn't so because caches can also change the *size* of data requested. This effectively turns access patterns with few repeated accesses into access patterns with many repeated accesses, for example if the cache's user requests one byte at a time but the cache's single slot can contain 4,096 bytes. (More in section notes)

Consider reading, for example. If the user only requests 1 byte at a time, and the cache reads 4096 bytes at once, then every subsequent read for the next 4095 bytes from the application can be easily fulfilled from the cache, without reaching out to the file system.

On the other hand, if the user reads *more* than 4,096 bytes at once, then a naively-implemented stdio cache might have unfortunate performance. It might read 4,096 bytes at a time, rather than in a single go, thus making more system calls than necessary.

Storage 3: Stdio cache

Request costs

A simple conceptual formula lets us reason about the cost of an I/O request. We write:

$$C = NU + R$$

where

- *C* is the overall cost of the request;
- *R* is **per-request cost**: the cost of any request, no matter how much data is involved;
- *U* is **per-unit cost**: the cost per unit of data requested (for example, the cost per byte read or written); and
- *N* is the **number of units** in the request.

Different storage technologies have different balances between per-request and per-unit costs. The costs can be measured in seconds, to evaluate a request's latency, or money, to evaluate its expense.

High-throughput, high-latency requests, such as 101Net or Sneakernet, have high per-request costs but very low per-unit costs.

Caches are useful when they lower the request costs seen by their users.

Cache benefits: How caches improve performance

We can break down the ways I/O caches improve application performance further into some basic components. Caches support many conceptually different optimizations. The basic ones are **prefetching**, **batching**, and **eliminating redundant writes**.

Prefetching

Prefetching is the primary optimization caches offer read requests.

To prefetch, a cache loads data from slower storage *before* the application needs it. Prefetching fills the cache with data that has not been requested yet, but that the cache believes will be accessed soon. If the cache predicts accesses effectively, then most read requests will hit and be serviced quickly.

For example, when an application opens a disk file for reading, the operating system might prefetch the file, or part of the file, into the buffer cache. This essentially predicts that the application will soon read the file—a pretty good assumption!

A cache that cannot prefetch offers little benefit for read requests. If prefetching doesn't work, then every access requires the cache to load data from slower storage; this can only be slower than having the application access slower storage directly, bypassing the cache.

Batching

Batching is an optimization for both writes and reads. Batching improves throughput, but not latency.

To batch, a cache changes a large number of small requests into a smaller number of larger requests. That is, the cache absorbs many requests with small *N* (small numbers of units per request), and emits fewer requests with larger *N*. For example, a stdio cache might combine many small write requests into a single write system call with more bytes.

Batching is most effective when per-request cost *R* is high, because it makes fewer requests. (If you're driving down the 101, you might as well take two disks.) It requires that the slower storage medium supports different sizes of request.

Write coalescing

Write coalescing, also called eliminating redundant writes, is an optimization for write requests.

If the same address is written multiple times, only the last-written value counts. To coalesce writes, the cache delays sending a write request to slower storage on the assumption that the data will be overwritten again.

Write coalescing is related to batching, but it works even if per-request cost is low, because it also reduces per-unit cost by writing fewer units.

Write coalescing is extremely useful for processor caches. Consider the following C code:

```
extern int x;
for (x = 0; x < 10000; ++x) {
    ... [assume this doesn't access x] ...
}
```

Without a processor cache, each write to x would cause an expensive write access to primary memory. The processor cache makes this much faster by eliminating these redundant writes; it can make this code behave something like the following:

```
extern int x;
int register_x;
for (register_x = 0; register_x < 10000; ++register_x) {
    ...
}
x = register_x;
```

Now all the redundant writes are coalesced into one.

Write coalescing is also extremely useful for disks because, as we saw in Storage 1, disks' minimum write sizes can cause redundant writes. The operating system's buffer cache eliminates most of these redundant writes.

Costs

Each of these strategies has costs as well as benefits.

Write coalescing can have a **correctness** cost (or consequence): data in the cache may be more up to date (contain later values) than the underlying slower storage. This matters for volatile caches running on top of stable storage, such as the buffer cache and stdio caches. Primary memory is volatile; if some of your writes are hanging out in volatile caches, they can be lost if the computer loses power at an inopportune time.

Prefetching has an **opportunity** cost. Prefetching data that isn't actually needed later can be expensive on its own (it wastes time and energy), and it can crowd out more useful data from the cache.

Cache coherence

Caches aim to give their users better performance without changing their users' view of underlying storage. Caches aren't allowed to make up data: a read cache should never return bogus data not present in the underlying storage, and a write cache should never scribble garbage into the underlying storage. But some caches do change what their users can see. This involves a property called **coherence**.

A **coherent cache** always provides its read users with the most up-to-date view of underlying storage. A coherent cache provides the same semantics as accessing the underlying storage directly.

An **incoherent cache** *does not* provide its read users with the most up-to-date view of underlying storage. In particular, an incoherent cache can return stale data that a concurrent write has changed.

The `coherence.cc` program in `cs61-lectures/storage3` demonstrates that the stdio cache is incoherent: stdio does not update its cache when the underlying file is changed.

The buffer cache *is* coherent, however. Every access to the disk is made through the operating system kernel, through system calls, and an important part of the kernel's job is to ensure that the buffer cache is kept coherent —that write system calls made by one application are visible to all other applications after they are made.

The processor cache is also coherent (up to the limitations of the machine's memory model, an advanced topic).

The stdio application interface does give applications some control over its coherence. Applications can request that stdio not cache a given file with `setvbuf(3)`. Applications can also mark a file's cache as invalid with `fflush(3)`, causing written data to be flushed to the operating system with a system call and causing prefetched data to be read again.

Random-access files, streams, and file positions

One of the Unix operating system's big ideas was the unification of two different kinds of I/O into a single "file descriptor" abstraction. These kinds of I/O are called **random-access files** and **streams**. Metaphorically, a random-access file is like a book and a stream is like time itself.

A random-access file has a finite length, called its *size*. It is possible to efficiently skip around inside the file to read or write at any position, so random-access files are *seekable*. It is also often possible to map the file into memory (see below). A random-access file is like an array of bytes.

A **stream** is a possibly-infinite sequence of bytes. It is *not* possible to skip around inside a stream. A stream is more like a *queue* of bytes: a stream writer can only add more bytes to the end of the stream, and a stream reader can only read and remove bytes from the beginning of the stream. Streams are not seekable or mappable. The output of a program is generally a stream. (We used `yes` as an example.)

Some previous operating systems offered fundamentally different abstractions for different kinds of file and stream. Unix (like some other OSes) observed that many operations on random-access files and streams can have essentially similar semantics if random-access files are given an explicit feature called the **file position**. This is a file offset, maintained by the kernel, that defines the next byte to be read or written.

When a random-access file is opened, its file position is set to 0, the beginning of a file. A `read` system call that reads *N* bytes advances the file position by *N*, and similarly for `write`. An explicit `seek` system call is used to change the file position. When the file position reaches the end of the file, `read` will return `0`.

Streams don't have file positions—the `seek` system call will return `-1` on streams. Instead, a `read` system call consumes the next *N* bytes from the stream, and a `write` system call adds *N* bytes to the end of the stream. When the `read` system call reaches the end of the stream (which only happens after the write end is closed), it will return `0`.

What's important here is that a sequence of `read` system calls will return *the same results* given a random-access file or a stream with the same contents. And similarly, a sequence of `write` system calls will produce the same results whether they are directed to a random-access file or a stream. Programs that simply read their inputs sequentially and write their outputs sequentially—which are most programs!—work identically on files and streams. This makes it easy to build very flexible programs that work in all kinds of situations.

(But special system calls that work on specific file offsets are useful for random-access files, and now they exist: see `pread(2)` and `pwrite(2)`.)

File descriptor notes

Read caches

The `r*` programs in `cs61-lectures/storage3` demonstrate different mechanisms for reading files. We looked at several:

- **`r01-directsector`** reads the file (1) 512 bytes at a time (this unit is called a **sector**), (2) using system calls (`read`), (3) **synchronously** from the disk. The `O_DIRECT` flag disables the buffer cache, forcing synchronous reads; when `O_DIRECT` is on, the application can only read in units of 512 bytes.

  On my laptop, `r01-directsector` can read about 7,000,000 bytes per second.

- **`r02-sector`** reads the file (1) 512 bytes at a time, (2) using system calls, (3) asynchronously (allowing the operating system to use the buffer cache).

  On my laptop, `r02-sector` can read somewhere around 360,000,000 bytes per second (with a lot of variability).

- **`r07-stdioblock`** reads the file (1) 512 bytes at a time, (2) using stdio, (3) asynchronously.

  On my laptop, `r07-stdioblock` can read somewhere around 480,000,000 bytes per second (with a lot of variability).

- **`r03-byte`** reads the file (1) one byte at a time, (2) using system calls, (3) asynchronously.

  On my laptop, `r03-byte` can read around 2,400,000 bytes per second.

- **`r05-stdiobyte`** reads the file (1) one byte at a time, (2) using stdio, (3) asynchronously.

  On my laptop, `r05-stdiobyte` can read around 260,000,000 bytes per second.

These numbers give us a lot of information about relative costs of different operations. Reading direct from disk is clearly much faster than writing direct to disk. (Of course this might also be impacted by the virtual machine on which we run.) And this also gives us some feeling for the cost of system calls: reading a byte at a time is about 150x slower than reading 512 bytes at a time. Assuming we knew the cost of the rest of the `r*` programs (i.e., the costs of running the loop and occasionally printing statistics, which are about the same for all programs), then this information would let us deduce the *R* and *U* components of system call costs.

Reverse reading

Some `r*` programs feature different access patterns. We looked at:

- **`r08-revbyte`** reads the file (0) **in reverse**, (1) one byte at a time, (2) using system calls, (3) asynchronously. On my laptop, it reads about 1,400,000 bytes per second.
- **`r09-stdiorevbyte`** reads the file (0) in reverse, (1) one byte at a time, (2) using stdio, (3) asynchronously. On my laptop, it reads about 3,500,000 bytes per second—more than twice as much!

We used strace to examine the system calls made by `r09-stdiorevbyte`. We discovered the stdio uses an **aligned** cache for reads. This means that, for reads, the stdio cache always aligns its cache so that the first offset in the cache is a multiple of the cache size, which is 4,096 bytes. This aligned cache is quite effective for forward reads and for reverse reads; in both cases, stdio's prefetching works.

Stdio doesn't work great for all access patterns. For example, for random one-byte reads distributed through in a large file, stdio will each time read 4,096 bytes, only one of which is likely to be useful. This incurs more per-unit costs than simply accessing the bytes directly using one-byte system calls.

The stdio cache is aligned for reads, but is it for writes? Check out the `strace` results for `w10-stdiorevbyte` to find out. You can do better than stdio here, at the cost of some complexity.

Memory mapping

Why is the stdio cache only 4,096 bytes? One could make it bigger, but very large stdio caches can cause problems by crowding out other data. Imagine an operating system where many programs are reading the same large file. (This isn't crazy: shared library files, such as the C++ library, are large and read simultaneously by most or all programs.) If these programs each have independent copies of the file cached, those redundant copies would mean there's less space available for the buffer cache and for other useful data.

The neat idea of **memory-mapped I/O** allows applications to cache files without redundancy by **directly accessing the operating system's buffer cache**.

The relevant system calls are `mmap(2)` and `munmap(2)`. `mmap` tells the operating system to place a region of a file into the application's heap. But unlike with `read`, this placement involves no copying. Instead, the operating system plugs the relevant part of the buffer cache into that part of the application's heap! The application is now sharing part of the buffer cache. The application can "read" or "write" the file simply by accessing that heap region.

- **`r16-mmapbyte`** reads the file (1) one byte at a time, (2) using `mmap`, (3) asynchronously (the operating system takes care of prefetching, batching, and, for writes, write coalescing). On my laptop, it reads about 350,000,000 bytes per second—more even than stdio!

Memory mapping is very powerful, but it has limitations. Streams cannot be memory mapped: memory mapping only works for random-access files (and not even for *all* random-access files). Memory mapping is a little more dangerous; if your program has an error and modifies memory inappropriately, that might now corrupt a disk file. (If your program has such an error, it suffers from undefined behavior and could corrupt the file anyway, but memory mapping does make corruption slightly more likely.) Finally, memory mapping has far nastier error behavior. If the operating system encounters a disk error, such as "disk full", then a `write` system call will return −1, which gives the program a chance to clean up. For a memory-mapped file, on the other hand, the program will observe a segmentation fault.

Advice

Memory-mapped I/O also offers applications an interesting interface for telling the operating system about future accesses to a file, which, if used judiciously, could let the operating system implement Bélády's optimal algorithm for cache eviction. The relevant system call is `madvise(2)`. `madvise` takes the address of a portion of a memory-mapped file, and some *advice*, such as `MADV_WILLNEED` (the application will need this data soon—prefetch it!) or `MADV_SEQUENTIAL` (the application will read this data sequentially—plan ahead for that!). For very large files and unusual access patterns, `madvise` can lead to nice performance improvements.

More recently, a similar interface has been introduced for *regular* system call I/O, `posix_fadvise(2)`. (Mac OS X does not support this interface.)

Storage 4: Consistency

Clean and dirty slots

Data in a read cache is sometimes in sync with underlying storage, and sometimes—if the cache is incoherent—that data is older than underlying storage.

Conversely, sometimes data in a *write* cache is *newer* than underlying storage. This can happen if the write cache is coalescing writes or batching, and therefore contains data written by its user that have not been written yet to the underlying slower storage.

A slot that contains data newer than underlying storage is called **dirty**. A slot containing data that is *not* newer than underlying storage is called **clean**.

For an example of both incoherent caches and dirty slots, consider the following steps involving a single file, `f.txt`, that's opened twice by stdio, once for reading and once for writing. (Either the same program called `fopen` twice on the same file, or two different programs called `fopen` on the same file.) The first column shows the buffer cache's view of the file (which is coherent with the underlying disk). The other two show the stdio caches' views of the file. The file initially contains all "A"s.

	Buffer cache	**Stdio cache f1**	**Stdio cache f2**
1. Initial state of `f.txt`	"AAAAAAAAAA"		
2. `f1 = fopen("f.txt", "r")`	"AAAAAAAAAA"	(empty)	
3. `fgetc(f1)`	"AAAAAAAAAA"	"AAAAAAAAAA"	
4. `f2 = fopen("f.txt", "w")`	"AAAAAAAAAA"	"AAAAAAAAAA"	(empty)
5. `fprintf(f2, "BB…")`	"AAAAAAAAAA"	"AAAAAAAAAA"	"BBBBBBBBBB"
6. `fflush(f2)`	"BBBBBBBBBB"	"AAAAAAAAAA"	(empty)

At step 5, after `fprintf` but before `fflush`, f2's stdio cache is dirty: it contains newer data than underlying storage. At step 6, after `fflush`, f1's stdio cache demonstrates incoherence: it contains older data than underlying storage. At steps 2–4, though, f1's stdio cache is clean and coherent.

A cache that never contains dirty slots is called a **write-through cache**. Write-through caches can't perform write coalescing or batching for writes: every write to the cache goes immediately "through" to underlying storage. Such caches are mostly useful for reads. A cache that can contain dirty slots is called a **write-back cache**.

System call atomicity

Unix file system system calls, such as `read` and `write`, should have **atomic effect**. Atomicity is a correctness property that concerns **concurrency**—the behavior of a system when multiple computations are happening at the same time. (For example, multiple programs have the file open at the same time.)

An operation is **atomic**, or has **atomic effect**, if it always behaves as if it executes without interruption, at one precise moment in time, with no other operation happening. Atomicity is good because it makes complex behavior much easier to understand.

The standards that govern Unix say `read`s and `write`s should have atomic effect. It is the operating system kernel's job to ensure this atomic effect, perhaps by preventing different programs from read or writing to the same file at the same time.

Unfortunately for our narrative, experiment shows that on Linux many `write` system calls do not have atomic effect, meaning Linux has bugs. But `write`s made in "append mode" (`open(… O_APPEND)` or `fopen(…, "a")`) **do** have atomic effect. In this mode, which is frequently used for log files, `write`s are always placed at the *end* of the open file, after all other data.

Kernel 1: Robustness and safety

What is the kernel?

Kernel is the OS software running with full privilege over machine operations. Its jobs include:

- Arbitrates among other programs
- Provides safe and convenient access to hardware

User land, the virtual environment provided by the kernel where user programs run: happy fun-time villege, sun is shining and everything looks perfect.

Kernel land: scary.

Evil program

Let's look at the following simple program, which in early days can bring down entire computer systems:

```
int main() {
    while (1) {
    }
}
```

This simple program compiles down to only 3 instructions

```
5fa:    push    $rbp
5fb:    mov     %rsp,%rbp
5fe:    jmp     5fe
```

When running this program in a modern OS, like Ubuntu Linux, macOS, and Windows, this program can't really do any damage. Even though the program appears stuck in an infinite loop, our computer is still responsive, and we can do other tasks running on the same OS just fine. We can even kill the problematic program by hitting Ctrl+C in the terminal. You may take these behaviors for granted, but a huge amount of effort went into the design and engineering of both OS software and processor hardware to enable these.

Protected control transfer

Recall what we saw how the `stdio` library functions invoke system calls like `read()` and `write()`. We saw in assembly that a `syscall` instruction was invoked, and it only returns after the system call was finished. This `syscall` instruction is a key interface via which user processes can interact with the kernel. It implements a form of **Protected Control Transfer** -- it transfers control of the processor to the kernel in a safe and limited way.

> Protected control transfer is safe because a process can only enter kernel at well-specified entry points. The process can't just jump to random code reside within the kernel.

Every process's address space contains a portion (usually the higher half of the address space) reserved for the kernel, and many kernel code reside there. One can write a program that attempts to jump some of these instructions in the kernel:

```
int main() {
    unsigned long kernel_insn = 0xffffffff80000100;
    void (*f)() = (void (*)())kernel_insn;
    f();
}
```

This program will crash with a segmentation fault, because a user process is not allowed to access anything reserved for the kernel directly. A user process can only invoke the kernel at specific entry points by using the `syscall` instruction.

Q: Why must we only allow control transfer to kernel at specific points?

Answer: The kernel code executes in an environment that's privileged and unprotected, which means it has total and complete control over the machine. Preserving the integrity of the kernel's control flow is therefore extremely important, since we don't want any process on a computer to be able to execute abitrary kernel code in privelieged mode. A breach of this limitation can result in losing control over the machine to a malicous or misbehaving program.

The limitation and restriction guaranteed by protected control transfer is implemented by both the OS software and the hardware. This hardened interface between the user land and the kernel land is the conerstone of security in modern computer systems.

Memory isolation

The hardened interface between user and kernel lands not only introduces restrictions. They also make several useful features possible, like presenting different "views" of memory to different programs. Let's take a look at the program in `storage4/r16-mmapbyte.cc` as an example.

The "guts" of the program is shown below, which counts number of bytes in an `mmap`ped file based on their values modulo 16:

```
size_t n = 0;
while (n < size) {
    memcpy(buf, &file_data[n], 1);
    n += 1;
    if (n % PRINT_FREQUENCY == 0) {
        report(n, tstamp() - start);
    }
    histogram[(unsigned char) buf[0] % 16] += 1;
}
```

If we run this program, we will see output like

```
0: 51199999
6:        1
```

This program accesses the file using `mmap`, a form of memory-mapped I/O. We can examine the region of memory used for I/O on the file by printing out the memory addresses of the buffer (`file_data`) returned by `mmap`.

We notice that the `file_data` buffer is located in the heap and changes every time the program runs.

> It was shown in lecture that during a particular run of the program, the buffer `file_data` located at address `0x7fae4adc3000`. It looked like a stack address, but an actual stack address, as shown by printing out the address of the `file_data` pointer variable itself, should be around `0x7ffdfebf07a8`, which is about 319 GB above the location of the `file_data` buffer. So the buffer was actually in the heap.

We can suspend the histogram computation program, launch another program writing to the same file using memory-mapped I/O. We observe that the memory-mapped file buffer is located at a different address from the file buffer in the suspended program. However, the memory appears truly shared, because all data that was written by one program gets reflected in the other program's histogram computation once we resume it.

How can two programs seemingly access the exact same piece of memory at different memory addresses?

This is an effect of another important feature of the hardened interface between the kernel and the user: **memory isolation**. Memory isolation provides:

- Kernel memory is isolated from user programs (except as allowed)
- User program memory is isolated from each other (except as allowed, as shown in the `mmap` example above)

Without explicit sharing, all user programs' memories are completely isolated from each other.

Process vs. program

We've talked about programs and process, as if they are equivalent. In precise terms, a program is just any piece of code that can be executed. It can be a file that sits on a disk, or it can be the image of an application that's currently being executed. Process specifics refers to programs that are currently being executed on a system. In general, in the operating system context, when we say programs or processes we refer to user-level software running under the protection and isolation of the kernel, although technically the kernel is also a program.

Kernel 2: Process isolation and virtual memory

Process isolation

General principle: processes interact only as allowed by OS policy.

What it means:

- No memory clobbering!
- Fair* sharing of machine resources!
- Protection of processes from each other, of kernel for processes.

Recall that the **kernel** is the part of the OS that runs with full machine privilege. There exists a hardened interface between the user and kernel.

Things in the user land: shell, emacs, browser
Things in the kernel land: the buffer cache, access to hardware like disk and keyboard

But processes for sure can access these hardware, or the buffer cache. These accesses are provided via the kernel by system calls.

The kernel also provides mechanisms other than system calls to enable controlled resource sharing with user-level processes. For example, access to memory, probably one of the most important piece of resources in computing, is not shared by system calls -- a process need not involve system calls to access memory. A process appears to have direct access to memory.

Virtualization and abstraction

Virtualization and abstraction are ways to provide a protected version of the interface.

In virtualization, the protected and unprotected (unprivileged and privileged) interfaces look the same.
Example: memory. Typically virtualization requires specialized hardware support.

In abstraction, the protected interface looks different from privileged interface.
Example: the file system. Implementating abstractions usually do not require special hardware support.

Q: How to choose between virtualization and abstraction when designing an interface?

Answer: For interfaces that are very frequently accessed, like memory, prefers virtualization because specialized hardware support makes them efficient. For higher-level interfaces that are less frequently used, abstraction is better because it doesn't need specialized hardware and usually one implementation works on many different hardware.

Interrupts

Recall the evil program from the last lecture, the infinite loop attack. The infinite loop attack dominates the resource of time, and we need to impose some limitation on this resource to deflect this attack.

Resource: Time
Need: Limit time in a process
Solution: Add a timeout!

We need a clock in the computer, and we do. It's called a **timer**. The timer hardware is configured to generate an "alarm" every millisecond. But what happens when an alarm goes off? Who should control the policy? The kernel! This means when the alarm goes off, the kernel needs to take control. So whatever program the machine was running before, once the alarm goes off, the program gets *interrupted* and the kernel gets to run. This kind of control transfer from a user process to the kernel is called an **interrupt**.

> An interrupt is a hardware initiated event that suspends normal processing, saves the current processor state, and transfers control to the kernel. The interrupt saves enough state such that the suspended execution can be resumed from right where it left off, as if it was never suspended.

The kernel then has the freedom to choose the next step to take. It can resume execution of the suspended program, or it can pick another process to run. This is how kernel implements the time sharing policy.

Interrupts can be disabled using the `cli` instruction. For this reason, this instruction is kernel-only, and cannot be accessed from a user-level process. There are many such "dangerous" instructions that can only be accessed by the kernel, and such restrictions are enforced by the processor hardware. But how does the processor know whether or not it was the kernel that executed a dangerous instruction? How does it know what's currently executing, the kernel or the user process?

On x86, the *current privilege level* (CPL) in which the processor operates is stored in the `%cs` register. The least significant 2 bits of the `%cs` register represents the privilege level:

```
0:     Kernel (privileged)
1,2,3: Unprivileged
```

In most cases only level 0 and 3 are used. Dangerous instructions mentioned above can only be executed in level 0.

> The `%cs` register, like any other register, can be written to or read from using `mov` and `pop` instructions. Instructions that change the value of this register, and therefore changing the privilege level, are also dangerous instructions and can only be accessed by the kernel.

When the system starts up, the processor is at privilege level 0. The kernel then runs and starts user-level processes in privilege level 3. The kernel's special privilege is established by the fact that it was the first program that gets to run once the hardware boots up.

Virtual memory

Virtual memory is a hardware mechanism used to isolate process memory, both from the kernel and from other processes. What it means is that an instruction attempting to move data from kernel memory or other processes' memory should not succeed.

Most modern architectures provide such guarantee through a mechanism called the **page table**.

Page table

Page table can be thought of a filter, through which a process accesses memory. Every process has its own page table, and the process accesses memory through its page table. We will study a simple version of a page table before moving on to introducing the real x86 page table.

Basic requirements of a page table:

1. Distinguish access by privilege
   - U (Unprivileged) bit: this part of the filter is OK for unprivileged access
2. Distinguish writes from reads
   - W (Writable) bit: this part of the filter is OK for writes
3. It's also useful to simply have some part of memory be untouchable (not present)
   - P (Present) bit: this part of the filter is OK to access
4. Allows arbitrary rearrangement (or aliasing) of memory

Our first page table assumes a 6-bit architecture, where pages are 8 bytes long. The following is a memory address in this architecture:

3 bits	**3 bits**
index	offset

Every address is divided into two parts: *page index* and *page offset*. The architecture has only 8 pages in total, each identified by an *index*. Within a page, there are 8 addresses, and each address can be referred to using the *offset*.

At a high level, the page table can be thought of as implementing the following mapping:

PT(virtual address, access type) --> physical address OR fault

The hardware performs the lookup in this mapping every time a memory access occurs.

In our 6-bit architecture, the lookup proceeds as follows (with virtual address `va` and access type `at`):

- Start from physical address `%cr3` (location of the page table)
- Access physical memory at `%cr3[va >> 3]` (`va >> 3` is the page index)
- Check access type, maybe fault, or return `%cr3[va >> 3] | (va & 7)` as the physical address

The page table, which is just a single page in this architecture, stores *page table entries* (PTEs). Each page table entry contains information about the physical address and permission bits we mentioned above:

3 bits	**1**	**1**	**1**
higher 3 bits of physical address	U	W	P

When a memory access is issued by a different process, the `%cr3` register will hold a different value, and this look up with start from a different location (with a different page table). Accesses to the same virtual address in different processes can end up in different physical addresses because the mapping is different.

Kernel 3: x86-64 page tables and WeensyOS

Page Tables

As we've seen before, the goal of this unit is to provide a layer of indirection between addresses and physical memory, which lets each process (which is an unprivileged entity) have a different view of memory. We achieve this goal by mapping virtual pages to physical pages.

Page tables are what let us actually convert a virtual address to physical address. A page table is like an array of pointers.

Single level page table

Let's first consider a single level page table. Here, a virtual address contains one index into the page table and an offset that is always 12 bits. When we use that index to find the corresponding entry in the page table page, we get a physical page address. The offset from the virtual address tells us the offset into the physical page.

Question: Why is the offset 12 bits?

Answer: We define the size of a page to be 4096 = 2^12 bytes. We want to be able to index into any byte of a destination physical page. So, we need 12 bits to represent every possible offset.

Question: Why is a single level not good enough?

Answer: Because we would need 2^39 bytes of data, which is too much memory! 2^39 = 2^36 * 2^3. The 2^36 comes from the maximum value the index can represent (an address is 64 bits and we reserve 12 bits for the offset, so we have 36 bits remaining for the index). The 2^3 comes from an address being 2^3 = 8 bytes.

x86-64 page tables

As we just saw, a single level page table takes up a lot of space! We can save some space by using multiple levels. We can think of a multi-level page table structure as a tree. Multiple levels leads to less space because when we look up the physical page for a given virtual address, we may visit a branch of the tree that tells us there's actually no valid physical page for us to access. In that case, we just stop searching. So, multiple levels means we can have a sparse tree.

Consequences for virtual addresses

The x86-64 architecture uses 4 levels. This is reflected in the structure of a virtual address:

63-48	47-39	38-29	29-21	20-12	11-0
	L4	L3	L2	L1	OFFSET

In x86-64 virtual address has 64 bits, but only the first 48 bits are *meaningful*. We have 9 bits to index into each page table level, and 12 bits for the offset. This means 16 bits are left over and unused.

Question: Why does each index get 9 bits?

Answer: Because the size of one page is 2^12 bytes, and each page table page entry is an address which has 2^3 bytes. $2^{12/2}3$ = 2^9 entries per page. We want to be able to index into any given entry, which means we need 9 bits.

%cr3

We store the physical address of the top level (L4) page table in a special register: `%cr3`.

Question: Why does `%cr3` store a physical address, when every other register stores a virtual address?

Answer: Page tables are used to convert virtual addresses to physical addresses, so if we stored our top level page table address as a *virtual address* then we wouldn't know how to convert it to a physical address!

The lookup process

A successful lookup (finding a physical address from a virtual address) goes as follows:

1. Use `%cr3` and the L4 index from the virtual address to get the L3 page table address
2. Use the L3 page table address and the L3 index to get a L2 page table address
3. Use the L2 page table address and the L2 index to get a L1 page table address
4. Use the L1 page table address and the L1 index to get the destination physical page
5. Use the destination physical page and the offset to get actual physical address within that destination physical page

Flags

Each entry in a page table is an address with the following structure:

63	62-48	47-12	11-3	2	1	0
NX		Physical address		U	W	P

Bits 0-2 contain the P (**p**resent), W (**w**ritable), and U (**u**ser-accessible) flags. However, we can also have other flags, like the NX (**n**on-e**x**ecutable stack) bit, which prevents us from executing instructions on the stack. This is important for preventing buffer overflows!

Again, page table entries don't have an offset because we used the physical addresses they store to find the start of another page table page or a destination physical page. We use the bits in a virtual address to access specific locations in those pages.

WeensyOS

The goals of pset 4 are to add process isolation, and implement the `fork` and `exit` system calls. In the handout code, there is no process isolation because each process has the same page table!

We run WeensyOS in QEMU, which emulates hardware for an x86-64 architecture.

Kernel and user addresses

The kernel (designated as `K`) lives in addresses that start with `0x40000`, while processes live in the upper addresses of virtual memory, starting at `0x10000`.

We also have hardware pages (designated as `R`) in the middle of the physical address space. This is because Bill Gates once said "no one should ever need more than 640K of memory", and processors would give us 1 MB of memory. The hardware lives in the upper portion of that memory (between 640 K and 1 MB).

The hardware includes one page marked as `C` for the CGA console. The console is an instance of memory mapped I/O. The console is not memory, but behaves like it; we can write output to the console by writing values in memory

Control transfer

We want to be able to switch into the kernel from user space and vice versa. All of these entry/exit points are defined in `k-exception.s`. You won't need to modify this file, but you should understand what it does.

When we have an exception (e.g. a timer interrupt or a segfault), or the user makes a syscall, we need to switch into the kernel. First, we save processor state by saving each register. The process's `%rsp` is pre-saved on the kernel stack. Then, we jump into a specific place in the kernel, which is determined by the entry code in `k-exception.s`.

When we want to give control back to the user process, we simply restore the registers we already saved.

Kernel 4: Protection and isolation

System calls and protected control transfer

The following is a list of actions that will occur, either by the process or by the kernel, once a process invokes a system call:

1. The process sets up arguments of the system call in registers, according to the system call's calling convention.
2. The process invokes the `syscall` instruction, which initiates a protected control transfer to the kernel.
3. The kernel starts executing from a pre-defined entry point, and executes a *handler* of the system call.
4. The kernel finishes processing the system call, and it picks another process to resume execution.

In the user space, a process has a system call "wrapper", which is a stub function that sets up the necessary state for executing a system call, and actually invokes the `syscall` instruction to transfer control to the kernel. Let's take look at the following system call, `sys_getsysname`, and this is its wrapper in the user space, defined in `process.hh`:

```
inline int sys_getsysname(char* buf) {
    register uintptr_t rax asm("rax") = SYSCALL_GETSYSNAME;
    asm volatile ("syscall"
                  : "+a" (rax), "+D" (buf)
                  :
                  : "cc", "rcx", "rdx", "rsi",
                    "r8", "r9", "r10", "r11", "memory");
    return rax;
}
```

> The list of arguments `cc`, `rcx`, ... at the very end of the inline assembly tells the compiler that all these registers will be destroyed by the system call. Because system call is different from a normal function call, the compiler needs to be explicitly informed about its calling convention.

The kernel's handler of this system call is located in `kernel.cc`, within the `syscall()` function. The relevant part is shown below:

```
case SYSCALL_GETSYSNAME: {
    const char* osname = "DemoOS 61.61";
    char* buf = (char*) current->regs.reg_rdi;
    strcpy(buf, osname);
    return 0;
}
```

It appears that the kernel is getting an argument of the system call, which is the buffer where to put the system call name, from `current->regs.reg_rdi`. Let's break it down how it works:

- Each process has a *process descriptor* structure, maintained by the kernel.
- When a `syscall` instruction is invoked, the kernel takes over, and it can access the process's process descriptor via pointer `current`.
- The process descriptor maintains many information about the process, including the register state of the process right before the protected control transfer occurred. Some work is done by the kernel to copy all that information to the process descriptor (in this case, all register values are copied to a struct called `reg` within the process descriptor), more on that later.

- Since register `%rdi` points to the buffer before `syscall` is executed (normal x86 calling convention), the kernel can access its value using `current->reg.reg_rdi`.

Let's explain in detail how the kernel saved all the information to the process descriptor after a `syscall` instruction gets invoked. The kernel doesn't start executing the `syscall()` function directly once the protected control transfer occurs. The actual entry point for `syscall` instructions is defined in `k-exception.S`, line 135:

```
syscall_entry:
    movq %rsp, KERNEL_STACK_TOP - 16 // save entry %rsp to kernel stack
    movq $KERNEL_STACK_TOP, %rsp     // change to kernel stack

    // structure used by `iret`:
    pushq $(SEGSEL_APP_DATA + 3)   // %ss
    subq $8, %rsp                  // skip saved %rsp
    pushq %r11                     // %rflags
    pushq $(SEGSEL_APP_CODE + 3)   // %cs
    pushq %rcx                     // %rip

    // other registers:
    subq $8, %rsp                  // error code unused
    pushq $-1                      // reg_intno
    pushq %gs
    pushq %fs
    pushq %r15 // callee saved
    pushq %r14 // callee saved
    pushq %r13 // callee saved
    pushq %r12 // callee saved
    subq $8, %rsp                  // %r11 clobbered by `syscall`
    pushq %r10
    pushq %r9
    pushq %r8
    pushq %rdi
    pushq %rsi
    pushq %rbp // callee saved
    pushq %rbx // callee saved
    pushq %rdx
    subq $8, %rsp                  // %rcx clobbered by `syscall`
    pushq %rax

    // load kernel page table
    movq $kernel_pagetable, %rax
    movq %rax, %cr3

    // call syscall()
    movq %rsp, %rdi
    call _Z7syscallP8regstate

    // load process page table
    movq current, %rcx
    movq (%rcx), %rcx
    movq %rcx, %cr3

    // skip over other registers
    addq $(8 * 19), %rsp

    // return to process
    iretq
```

We can see that the kernel eventually calls the `syscall()` function before returning to the process, but a lot of setup work is done before the function call. Here is a high-level summary about what these setups are about:

- Save and set `%rsp` so that the kernel starts using its own stack, instead of the process' stack.

- Set up a structure, on the kernel stack, used by the `iretq` instruction to return to the process after the system call finishes.
- Push all other user registers to the kernel stack.
- Up until this point the kernel is running using the process's page table, switch the hardware (by setting `%cr3`) to use the kernel page table.
- Call the C++ function `syscall()`, using the register structure we just saved on the stack as argument.
- Restore to the process's page table, and return to the process.

In the `syscall()` function we copy the register structure passed in to the `regs` field in the process descriptor.

When a system call "returns", you can think of it as that all register values in `current->regs` gets copied (or restored) to the actual processor's registers before the process resumes execution.

> System call handing is a bit like programming with exceptions. If you have experience with exceptions in another programming language, it may help you understand system calls. Every time a new system call occurs and the handler gets executed, the handler is not aware of any prior invocations of the same handler, unless mediated by some other state explicitly managed by the handler. The entire kernel stack gets "blown away" once a system call "returns". You can also think of it as a event-driven programming model.

Types of protected control transfers

- Interrupts: caused by non-CPU hardware (e.g. timer)
- Traps: caused by software intentionally (syscall)
- Faults: caused by software error

> It's worth pointing out that all these control transfers are administered by the x86 CPU.

When a process "returns" from a protected control transfer, kernel restores its `%rip` register to point to:

- Interrupts: the next instruction that hasn't been executed yet.
- Traps: the next instruction following the trap.
- Faults: the problematic instruction causing the fault.

TL;DR: If a process enters the kernel because of an interrupt or trap, then once it resumes execution it will pick up from where it left off. If a process enters the kernel because of a fault, then it will retry the faulty instruction if it resumes.

Q: How can the kernel, in the system call handler, simply overwrite register values from the process? Won't that mess up the process's state and cause it to crash?

Answer: System calls are *traps* (also called *synchronized events*), which means they are explicitly invoked by the process, and the process expects a control transfer to occur and respects the calling convention of such control transfers. This is in contrast to interrupts and faults where such control transfers occur without the process's knowledge. In the case of system calls, calling convention designates certain registers to be overwritten by the kernel to convey information regarding results of the system call (`%rax`, for example hold return value of the system call), so the kernel are free to overwriting these registers. In fact, without using these registers, it becomes rather difficult to convey results of a system call unless the kernel exposes parts of its own memory to the process. It is worth noting though that the kernel can't just overwrite process registers arbitrarily.

Let's take a look at an example of an interrupt next, the timer interrupt (`kernel.cc:241`):

```
case INT_TIMER:
    ++ticks;
    schedule();
    break;      /* will not be reached */
```

The code above is located in an exception handler, which is similar to the system call handler in terms of how it saves the process's state to the process descriptor. We see it simply increments a `ticks` variable, and calls the `schedule()` function, which picks another process to execute on the processor. Note that in this interrupt handling code no modifications were made to the process's register state. The process does not expect a timer interrupt to occur so we had better make them transparent to the process. By not modifying any registers we achieve this goal.

The `sys_yeild` system call, which is similar to the timer interrupt, has the following relevant code in the system call handler:

```
case SYSCALL_YIELD:
    current->regs.reg_rax = 0;
    schedule();             // does not return
```

Here we can modify the process's `%rax` state because again, it is a system call, and the process expects the occurrence of a control transfer and the overwriting of value in `%rax` by the kernel.

So, we have a super well-isolated operating system, called DemoOS, and there is *absolutely nothing* a program can do to take over the machine.

Absolutely nothing.

Or, is there?

Alice and Eve

We now look at two programs written by two rivals, Alice and Eve. They will be running on DemoOS.

This is Alice, in `p-alice.cc`:

```
#include "process.hh"
#include "lib.hh"

void process_main() {
    char buf[128];
    sys_getsysname(buf);
    app_printf(1, "Welcome to %s\n", buf);

    unsigned i = 0;
    while (1) {
        ++i;
        if (i % 512 == 0) {
            app_printf(1, "Hi, I'm Alice! #%d\n", i / 512);
        }
        sys_yield();
    }
}
```

And this is Eve, in `p-eve.cc`:

```
#include "process.hh"
#include "lib.hh"

void process_main() {
    unsigned i = 0;

    while (1) {
        ++i;
        if (i % 512 == 0) {
            app_printf(0, "Hi, I'm Eve! #%d\n", i / 512);
        }
        sys_yield();
    }
}
```

We can see both Alice and Eve contain infinite loops, but programs are being nice! By explicitly invoking the `sys_yield()` system call, they are "yielding" (i.e. letting another process to run) precious CPU time to each other so that both of them can make progress.

If our DemoOS is as wonderful as we claimed, there is *absolutely nothing* Eve can do to prevent Alice from running, and vice versa. The two always get about equal share of CPU resources.

Attack 1: No yielding any more

If Eve stops calling `sys_yeild()`, then Eve no longer actively yields the CPU to any other program, and it takes over the entire machine.

This problem occurs because timer interrupt is properly configured. After initializing the timer interrupt, Eve no longer gets to take over the machine. Alice gets chance to run again, although Eve still visibly uses more resources from the machine by not yielding. This strictly speaking doesn't provide fairness, but it is a reasonable policy an OS may choose to implement.

Attack 2: Disabling interrupts

Eve disables interrupts by using the `cli` instruction. Alice again gets no CPU time after Eve successfully executes this instructions.

We fix it by disallowing processes to control interrupts. Now Eve is not able to execute `cli`, or it will crash.

> When Eve attempts to execute the no-longer-allowed `cli` instruction, the hardware generates a fault and transfers control to the kernel. We made the kernel handle this fault as an exception, and it falls under case `INT_GPF` , or general protection fault. This is a catch-all default type of fault the hardware throws when the reason of the the error doesn't fall under any of the more specific types of fault.

Attack 3: I crash, you crash (divide by zero)

Eve then changes its program to contain a divide by zero error. When that instruction hits DemoOS crashes and nobody gets to run.

As it turns out divide by zero error triggers another hardware exception that was not handled by us. We don't really want to list all the exceptions one-by- one and write code to handle them all, because for many of them we do the exact same thing: kill the faulting process. We may be tempted to just add this code in the `default` case: for all unexpected exceptions, just kill the process.

We need to be careful here though, as we should really only do this if an exception stems from problems in user code. If the kernel throws an exception, it usually indicates a serious bug in the kernel and it's a bad idea to carry on with life as if it never happened. This can be done by adding a simple check under the `default` case:

```
default:
    if (regs->reg_cs & 3 != 0) {
        current->state = P_BROKEN;
    } else {
        panic("Unexpected exception %d!\n", regs->reg_intno);
    }
    break;
```

> The least significant 2 bits of `%cs` register stores the privilege level the processor is running at before the fault occurred.

We can also have more fun with Eve. Imagine that we don't crash Eve's program when a divide-by-zero occurred, but to confuse her. We can do this by handling the divide-by-zero exception this way:

```
case INT_DIVIDE:
    current->regs.reg_rax = 61;
    current->regs.reg_rip += 2;
    break;
```

As per x86 specification, this should be enough to convince Eve that anything divides by zero is always 61. (The specification of the `idiv` instruction says that the quotient of the division is stored in `%rax`). We also incremented Eve's `%rip` because divide by zero is a fault, and `%rip` saved by the kernel will point to the faulty instruction, which is the division instruction. Without changing `%rip` Eve would re-execute the `idiv` instruction once it resumes. We move past the divide instruction by adding 2 to `%rip` (the `idiv` instruction is 2-byte-long). This shows the control and power the kernel has over a process.

Attack 4: Jump-to-kernel

Eve now examines the kernel assembly and finds out that the `syscall` entry point is located at address `0x40ac6`. She then made her program write two magical bytes to that location: `0xeb 0xfe`. The two bytes form an evil instruction that jumps to itself: another infinite loop attack! Now whenever a system call is made, the kernel enters an infinite loop, and the machine hangs.

> Infinite loops in the kernel is particularly disastrous because the kernel usually runs with interrupts disabled.

This attack can succeed because DemoOS doesn't properly implement kernel memory isolation -- a user process can access (read and write) any kernel memory that's mapped in the process's address space. We isolate the kernel by setting the proper permissions for kernel memory:

```
for (vmiter it(kernel_pagetable, 0);
     it.va() < MEMSIZE_PHYSICAL;
     it += PAGESIZE) {
    // Don't set the U bit, except for the console page
    if (it.va() != (uintptr_t) console) {
        it.map(it.pa(), PTE_P | PTE_W);
    }
}
```

After adding protection for kernel memory, Eve crashes after attempting to overwrite the `syscall` entry point, and Alice gets to run till the end.

Epilogue

Process isolation is still far from achieved in DemoOS. There are several kinds of attacks Eve can still perform against Alice. Just to name a couple:

1. Eve can clobber Alice's memory
2. Fork bomb...

Thanks William for playing Eve from MIT.

Kernel 5: Confused deputy attack, scheduling, and process management

Confused deputy attack

A *confused deputy attack* occurs when the attacker has low privilege, but the attacker convinced a privileged deputy to complete the attack on its behalf.

In the context of operating systems, a process is unprivileged, the kernel has full privilege and acts as a privileged deputy by handling system calls. A confused deputy attack may occur if the process, by invoking system calls, can somehow convince the kernel to execute a privileged attack.

Certain system calls are vulnerable to such attacks, but others are not. Very simple system calls like `sys_getpid` usually aren't susceptible to these attacks because they don't change the state of the kernel or other processes at all -- it simply copies the current process's `pid` value to its `%rax` register, and that's it. It's so simple that there is little room for bad things to happen.

Recall that at the end of last lecture, Eve tried to overwrite the `syscall` entry point in kernel memory with an infinite loop. We fixed this by denying access to kernel memory from the user level. Now Eve cannot just directly write to kernel memory any more, but is it possible to for Eve to convince the kernel to write malicious data/code to kernel memory?

Let's take a look at `sys_getsysname`. Could it be used by Eve to perpetrate a confused deputy attack? What if Eve simply does the following:

```
char* syscall_entry_addr = (char*)0x40ac6;
sys_getsysname(syscall_entry_addr);
```

Eve actually manages to crash the entire OS by adding just these 2 lines of code! The kernel did not perform any sanity checks before writing the string containing the OS name to the user-supplied buffer. In this case the buffer happens to point to the `syscall` entry point in kernel memory, and the kernel happily overwrote it. **A successful confused deputy attack!**

> It's worth pointing out that Eve never directly wrote to any kernel memory. The kernel overwrote part of its own memory because the buffer passed by Eve points to kernel memory.

> You may wonder why would a buffer in Eve's address space point to a critical component in the kernel's address space. It is true that when the kernel performs the string copy, it uses Eve's page table to perform address translation (for the destination of the copy). However, note that in DemoOS (and WeensyOS), the part of the page table mapping below `PROC_START_ADDR` (where the `syscall` entry point is located) is shared among all processes as well as the kernel. So this problematic address Eve passes to the kernel will translate into the same physical address, in kernel memory, regardless of which page table (including the kernel's page table) is used for address translation. Additionally, because processes and the kernel also share the same physical pages for mappings below `PROC_START_ADDR`, overwriting instructions there has a global effect, meaning that all process's (and the kernel's) `syscall` entry point code will get corrupted.

Eve is still not satisfied, because this attack blows away all processes in the system, including Eve itself. Eve would like a more targeted attack against *only* Alice. Eve could try, via a confused deputy attack, to corrupt Alice's register file in its process descriptor.

With information about kernel memory layout and Alice's PID, it is possible for Eve to figure out where Alice's process descriptor is located in memory. Eve then once again uses the `sys_getsysname()` system call to corrupt Alice's process descriptor, and after the system call is executed, Alice crashes and only Eve gets to run on the system.

Confused deputy attack can be far more devious and do way more damage than demonstrated. For example, if a malicious process can somehow convince the kernel to turn on certain bits in its page table, it could suddenly gain access to kernel memory and even inspect/change other process's memory. Instead of simply causing the victim to crash, it can passive monitor and steal information from the victim or do more.

To prevent such attacks, the kernel should always perform checks on user-supplied inputs before acting on them. In this example of `sys_getsysname()`, we should make sure that the buffer address supplied by the user is mapped as user-accessible in the process's page table. A safer version of the `sys_getsysname()` handler should look like the following:

```
case SYSCALL_GETSYSNAME: {
    const char* osname = "DemoOS 61.61";
    char* buf = (char*) current->regs.reg_rdi;

    // Check that the entire span of the buffer is mapped
    // as user-accessible
    size_t len = strlen(osname) + 1;
    size_t i = 0;
    for (vmiter it(current, (uintptr_t) buf);
         i < len;
         it += 1, ++i) {
        if (!it.user()) {
            return -1;
        } else {
            // Performs the actual copy
            // Note that the destination of the copy is
            // address-translated using the process's
            // page table.
            *((char*) it.pa()) = osname[i];
        }
    }

    return 0;
}
```

We briefly note here that `sys_page_alloc()` has a similar vulnerability. Eve could request to map a new physical page to the virtual page containing the `syscall` entry point instructions in its address space. It only affects Eve's page table and has no global effect, but it grants Eve control over its `syscall` entry point, and Eve can put arbitrary code there. The next time Eve executes a system call, the inserted code within the newly mapped physical page will run in privileged mode, and it can easily compromise the entire machine. Note that in this case no direct corruption of kernel memory occurred, but a confused deputy attack can still take place.

Fairer scheduling

Recall previously when Eve executes an infinite loop attack, and we deployed defense against it by turning on timer interrupts. Alice still gets to run, but it only gets a small fraction of CPU time. This is because Alice is being "nice" by constantly yielding, but Eve is not being nice and we rely on the timer to take it off the CPU. So what happens is that Eve gets to run a full timer interval, and then Alice simply prints out a message, before yielding back to Eve again.

How can measure niceness (or greediness) of processes, and take it into account when scheduling them, such that the greedy process gets to run less often?

Measure greediness

Signals the kernels are getting about "greediness" of a process come from timer interrupts and yield system calls.

- Yielding is a sign of niceness
- Interrupt is a sign of greediness

We can imagine we maintain a counter per process for its greediness. Negative greediness means the process is nice, positive means the process is greedy. Every time we receive a yield call, we decrement the counter. Every time we receive a timer interrupt, we increment the counter. The counters should also have saturating maximum and minimum values so that this greediness measure is reasonably bounded. The scheduler then picks the process with the least greediness to run in every scheduler tick. The following code snippets show how we may implement this policy in the DemoOS kernel.

In the yield system call handler:

```
case SYSCALL_YIELD:
    if (current->greediness > -100) {
        --current->greediness;
    }
    current->regs.reg_rax = 0;
    schedule();
    break;
```

In the timer interrupt handler:

```
case INT_TIMER:
    if (current->greediness < 100) {
        current->greediness += 5;
    }
    ++ticks;
    schedule();
    break;
```

In the scheduler (`schedule()`):

```
void schedule() {
    while (true) {
        int least_greedy_pid = -1;
        int least_greediness = 100000;
        for (int pid = 1; pid < NPROC; ++pid) {
            if (ptable[pid].state = P_RUNNABLE
                && ptable[pid].greediness < least_greediness) {
                least_greedy_pid = pid;
                least_greediness = ptable[pid].greediness;
            }
        }

        // Run least_greedy_pid if exists
        if (least_greedy_pid > 0) {
            run(&ptable[least_greedy_pid]);
        }
    }
}
```

After we make these changes, Alice gets much more CPU time and Eve gets penalized for being too greedy.

> There are many different scheduling policies, and this very simplistic scheduling algorithm follows what we call a *strict priority* scheduling policy. In this case it is possible that Eve (the greedier process, therefore with strictly lower priority in our system) can get starved and never gets to run again. In a real system we often need more sophisticated scheduling schemes to avoid starvation. Randomness can solve starvation in many cases.

Process management

We are now moving towards a new unit exploring how an operating system can manage processes so that we as users can do useful things with them. Topics in this unit include process life cycle management and communications with and/or between processes.

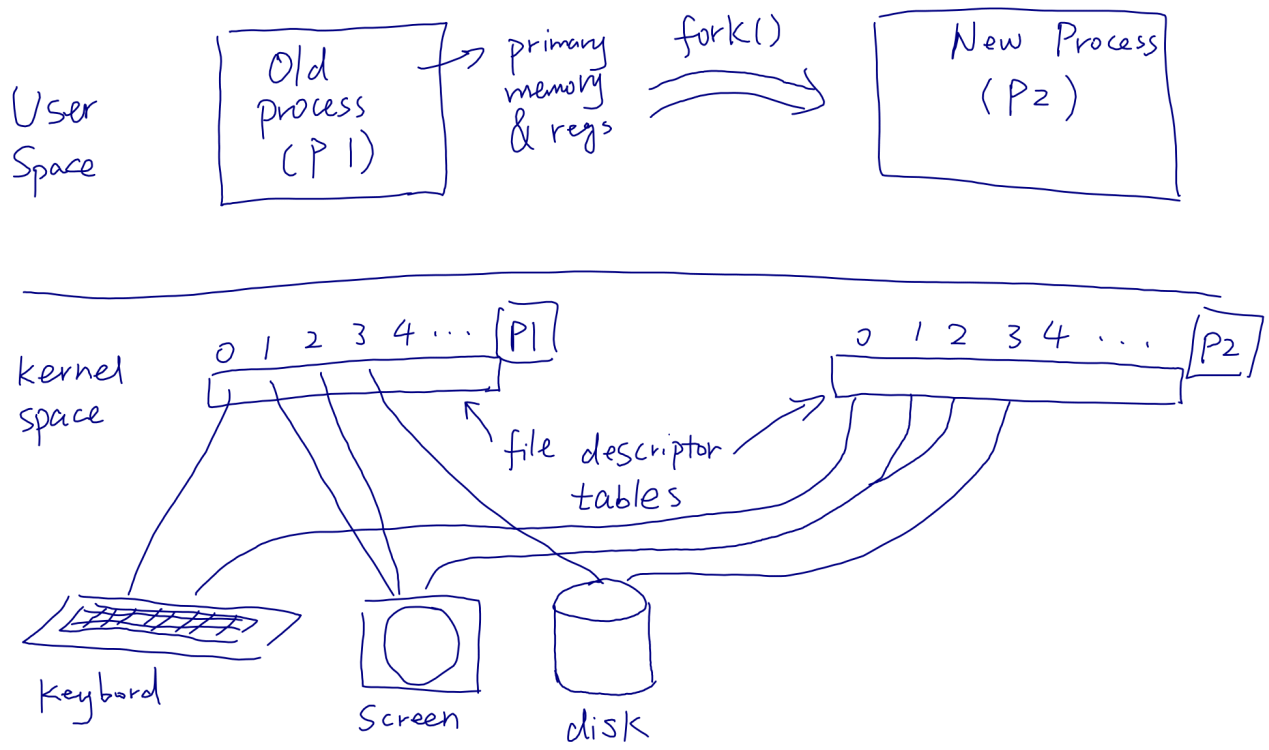
Process creation: the `fork()` system call

A new process can be created by a running process via invocation of the `fork()` system call.

- The new process has a copy of the old process's state (memory and register file)
- The new process and the old process shares the abstract hardware (copies I/O descriptors)
- `fork()` returns 0 to the new process, and returns the new process's ID to the old process invoking `fork()`

The last bullet point above is how one may programmatically distinguish the new process from the old one after `fork()` returns.

> When a process calls `fork()`, it also creates a copy of its file descriptor table in the newly created process. The file descriptors in the new process point to the same underlying devices as they do in the old process. The effect of a `fork()` is illustrated below:


User Space
Old process (P1)
primary memory & regs
fork()
New Process (P2)
kernel space
0 1 2 3 4 ... P1
0 1 2 3 4 ... P2
file descriptor tables
Keyboard
Screen
disk

Shell 1: Process control

This unit is about the system calls for

- creating
- managing
- managing process environments
- coordinating

of processes.

"Screwing together programs like garden hose"

The UNIX philosophy: Simple programs achieve simple tasks, but a complex task can be accomplished by running many simple programs in coordination. The same program may even be reused multiple times in the same task. The operating system should enable this coordination scheme without much hassle.

There are costs and benefits of running multiple small programs instead of a big, complicated program for a certain task.

Costs: In some programming environments, there is a one-time, constant, but very high cost of running a program, regardless of the complexity of the program itself. The Java runtime is such an example, where memory consumption can be high for even very simple programs. In some interpreted languages like Python and JavaScript, running a program may involve starting up the interpreter, which can incur a high cost because these interpreters are very complex programs themselves.

Benefits: Different programs run in isolated environments, so an error in one program responsible for a certain task would be self-contained to that program only. This makes it easy to isolate failures in the system. The same program may also be reused for different stages of the task, reducing the code base required to perform the task. The main benefit, as described above, is what we call the *modularity* benefit.

Modularity: A programming system's property of dividing itself into self-contained modules.

A module ideally have a simple and well-defined interface, so that it's easy to use. The best modules also have deep and good implementations of the interfaces so that they perform their tasks efficiently.

"Say 'word count' one more time"

We are going to visit an article, *Programming pearls: A literate program*, about Turing Award winner and Stanford professor Donald Ervin Knuth. As one of the most respected voices in the computer science community, Prof. Knuth was the creator of the famous TeX typesetting system (which LaTeX is derived from). He was very good at creating incredibly complex software that also just appears to be 100% correct.

The columnist who wrote the article, Prof. Jon Bentley, once posed the following problem to Prof. Knuth:

> Given a text file and an integer K, print the K most common words in the file (and the number of their occurrences) in decreasing frequency.

You can read Prof. Knuth's solution in the article, which involves some rigorous type definitions and a fascinating data structure. In short, it's very complicated.

Prof. Bentley proposed a UNIX shell script that achieves the same goal (adapted to modern shell syntax) with reasonably good performance:

```
#!/bin/bash

# script: shell_count.sh

tr -cs A-Za-z'' | \
tr A-Z a-z | \
sort | \
uniq -c | \
sort -rn | \
head -n ${1} # ${1} refers to the first parameter of the script, in this case the integer K
```

What the script does is to spawn a series of programs (6 in total in this case) and to connect them to form a *pipeline*. The 6 programs run *in parallel* and consume each others' outputs in the pipe line to perform the described task. Let's see what it does:

- `tr -cs A-Za-z''`: First "translator" program, delete all characters that are not A-Z or a-z.
- `tr A-Z a-z`: Second translator program, transforms all upper case letters to lower case.
- `sort`: Lexicographically sorts the input by line.
- `uniq -c`: De-duplicates input by line (only neighboring duplicate lines are removed). The `-c` option also puts a duplicate count at the beginning of each line in the output.
- `sort -rn`: Numerically sorts the output in reverse order by line.
- `head -n ${1}`: Takes the first `K` lines of the input as the final output.

We can use this script (assuming it's called `shell_count.sh`) in a UNIX shell command line environment as shown below:

```
# to show 10 most frequent words and their counts in input.txt
$ cat input.txt | ./shell_count.sh 10
```

The simplicity of the shell script demonstrates the benefit of having modular coordination among small programs, or benefit of the UNIX programming philosophy in general.

> Read further: Prof. Bentley's previous column, *Programming pearls: Literate programming*.

Controlling processes

In order to support this style of coordinated task solving with multiple programs we need the capability to easily control and manage multiple independent programs. There is one program whose only purpose is to provide such capacity -- **the shell**.

The shell uses a number of system calls to perform these process coordination tasks. We will discuss them one at a time.

Process creation: `fork()`

Recall that the `fork()` system calls creates a new process. The new process has a copy of the old process's state (at the time of the system call). Let's consider the following program:

```
int main() {
    pid_t p1 = fork();
    assert(p1 >= 0);

    printf("Hello from pid %d\n", getpid());
}
```

Running this program gives outputs like this:

```
Hello from pid 35931
Hello from pid 35932
```

This is not surprising at all. We may recognize that the process with the lower `pid` is the process that calls `fork()` (the *parent* process), and the one with the higher `pid` is the newly created process (the *child* process).

If we run the program multiple times though, sometimes we also see the following outputs:

```
Hello from pid 35933
<shell prompt> $ Hello from pid 35934
```

In this case a shell prompt gets printed before the child gets to print out its message. This indicates that the shell only waits for the parent to finish before printing out a prompt and therefore can accept new commands. When a shell "waits" for a process (like the parent, in this case) in this way, we call that the process executes *in the foreground* within the shell.

We know from the last lecture that `fork()` performs the following:

- Makes a copy of the parent process.
- On success, `fork()` returns 0 to the child, and returns the child's `pid` to the parent.
- On failure, `fork()` returns -1 to the parent.

Here is `fork2.cc`:

```
int main() {
    pid_t p1 = fork();
    assert(p1 >= 0);

    pid_t p2 = fork();
    assert(p2 >= 0);

    printf("Hello from pid %d\n", getpid());
}
```

Running this program gives us outputs like this:

```
Hello from pid 73285
Hello from pid 73286
Hello from pid 73288
<shell prompt> $ Hello from pid 73287
```

We have 4 processes running in this case, and the shell again only waits for the parent to finish. We can visualize it like:

```
                    1st fork()
                   /          \
              (p1!=0)          (p1==0)
                |                 |
           2nd fork()          2nd fork()
           /        \          /        \
       (p2!=0)  (p2==0)    (p2!=0)  (p2==0)
          |        |          |        |
       parent   child2     child1  grand child
```

Or we can draw a process hierarchy diagram (shown on the board during lecture):

```
                      (proc0)
                      /     \
                     /       \
                 1st fork  2nd fork
                   /           \
                  /             \
              (proc1)          (proc2)
                /
               /
           2nd fork
             /
            /
         (proc3)

proc0: parent
proc1: child1
proc2: child2
proc3: grand child
```

Let's take a look at another program in `forkmix-syscall.cc`:

```
int main() {
    pid_t p1 = fork();
    assert(p1 >= 0);

    const char* test;
    if (p1 == 0) {
        text = "BABY\n";
    } else {
        text = "mama\n";
    }

    for (int i = 0; i != 1000000; ++i) {
        ssize_t s = write(STDOUT_FILENO, text, strlen(text));
        assert(s == (ssize_t) strlen(text));
    }
}
```

Running the program prints out interleaving lines of `BABY` and `mama` to the screen. Sometimes the interleaving is perfect (we see exactly one `BABY` followed by exactly one `mama`, and so on), sometimes we see consecutive printouts of `BABY` or `mama`. It does seem that every line is either `BABY` or `mama`.

We can verify this using the shell knowledge we just learned. Running

```
./forkmix-syscall | sort | uniq
```

generates exactly 2 lines (`BABY` and `mama`), so our assumption seems correct. It is guaranteed that every line is either `BABY` or `mama`. This is because of the atomicity guarantee of the `write()` system call.

Let's now look at a very similar program in `forkmix-stdio.cc`. The program is almost identical to `forkmix-syscall.cc` except that it uses stdio's `fputs()` instead of the `write()` system call to write out data.

When we run this program, first we observe that it is much faster (because fewer system calls!), second the output doesn't look quite right this time. When we again run the `sort-uniq` pipeline against its output, we get much weirder results like lines with `maBABY`, `mamaBABY`, `maY`, and so on. So looks like even characters are now interleaved! Why?

Recall that stdio uses a internal user-level cache that's 4096 bytes large. Once the cache is filled the whole 4KB block is written using a single system call, which is atomic. The problem is that the strings we are writing, `"BABY\n"` and `"mama\n"`, are actually 5 bytes long. 4096 is not divisible by 5, so at cache line boundaries we can have "left-over" characters that gives us this behavior. It's also worth noting here that the two processes here (parent and child) do not share the same stdio cache, because the cache is in the process's own isolated address space.

Now let's look at our last `forkmix` program, `forkmix-file.cc`. Again the program is similar, except that it now opens a file (all using stdio) and writes to the file, instead of the screen (stdout).

Note the following few lines in the program:

```
...
FILE* f = fopen(argv[1], "w");

pid_t p1 = fork();
assert(p1 >= 0);
...
```

So the program opened the file before calling `fork()`. This shouldn't give us any surprising results compared to `forkmix-stdio`. But what if we called `fork()` *before* opening the file?

Once we move the `fopen()` call after the fork, we observe that the output file now contains only half of the expected number of characters.

The reason for this effect is because when we call `fopen()` after the `fork()`, the parent and the child will get two separate offsets into the same file (maintained by the kernel). The two processes then essentially race with each other to fill up the file. The process that writes slower will overwrite what's already been written in the file at that offset. Eventually the length of the file will only be the number of characters written by either process, because that's the final value of the two file offset values in both the parent and the child.

(diagrams to follow)

Shell 2: Process creation and interprocess communication

`fork()` recap

Let's do a quick exercise to remind us of what `fork()` does. Especially after creating a "copy" of a process, what's copied and what's not copied. Take a look at `shell3/fork3.cc`:

```
int main() {
    printf("Hello from initial pid %d\n", getpid());

    pid_t p1 = fork();
    assert(p1 >= 0);

    pid_t p2 = fork();
    assert(p2 >= 0);

    printf("Hello from final pid %d\n", getpid());
}
```

Question: How many lines of output would you expect to see when you run the program?

> 5 lines. The first `printf()` prints one line, only in the parent, and then the second `printf()` will run four times, one in each process (parent + 2 children + 1 grand child).
>
> Hide solution

Question: How many lines of output would you expect if we run the program and redirect its output to a file (using `./fork3 > f`)?

> We actually see 8 lines of output in the file. What's going on??
>
> Note that we are using `printf()`, which is a stdio library function and not a system call. So there is caching going on. After the first `printf()` is called, the output only gets written to a buffer but not the actual file descriptor. The buffer is in user-space memory and will get duplicated after `fork()` is invoked. Therefore every child process has two lines of output in its stdio buffer by the end of its execution, and a total of 8 lines get written to the file in the end.
>
> Recall that this buffering only occurs when `stdout` is being redirected to a file. When operating on the console, stdio flushes the buffer after each new line character, making it behaving like a system call. That's why we don't see this effect when running the program in the console without I/O redirection.
>
> We can avoid this behavior by calling `flush(stdout)` after the first `printf()` call.
>
> Hide solution

Running a new program

The UNIX way: `fork-and-exec` style

There is a family of system calls in UNIX that executes a new program. The system call we will discuss here is `execv()`. At some point you may want to use other system calls in the `exec` syscall family. You can use `man exec` to find more information about them.

The `execv` system call (and all system calls in the `exec` family) performs the following:

- Blow away the current process's virtual address space
- Begin executing the specified program in the current process

Note that `execv` does not "spawn" a process. It destroys the current process. Therefore it's common to use `execv` in conjunction with `fork`: we first use `fork()` to create a child process, and then use `execv()` to run a new program inside the child.

Let's look at the program in `shell3/myecho.cc`:

```
int main(int argc, char* argv[]) {
    fprintf(stderr, "Myecho running in pid %d\n", getpid());
    for (int i = 0; i != argc; ++i) {
        fprintf(stderr, "Arg %d: \"%s\"\n", i, argv[i]);
    }
}
```

It's a simple program that prints out its `pid` and content in its `argv[]`.

We will now run this program using the `execv()` system call. The "launcher" program where we call `execv` is in `forkmyecho.cc`:

```
int main() {
    const char* args[] = {
        "./myecho", // argv[0] is the string used to execute the program
        "Hello!",
        "Myecho should print these",
        "arguments.",
        nullptr
    };

    pid_t p = fork();

    if (p == 0) {
        fprintf(stderr, "About to exec myecho from pid %d\n", getpid());

        int r = execv("./myecho", (char**) args);

        fprintf(stderr, "Finished execing myecho from pid %d; status %d\n",
                getpid(), r);
    } else {
        fprintf(stderr, "Child pid %d should exec myecho\n", p);
    }
}
```

The goal of the launcher program is to run `myecho` with the arguments shown in the `args[]` array. We need to pass these arguments to the `execv` system call. In the child process created by `fork()` we call `execv` to run the `myecho` program.

> `execv` and `execvp` system calls take an array of C strings as the second parameter, which are arguments to run the specified program with. Note that everything here is in C: the array is a C array, and the strings are C strings. The array must be terminated by a `nullptr` as a C array contains no length information. You will need to set up this data structure yourself (converting from the C++ counterparts provided in the handout code) in the shell problem set.

Running `forkecho` gives us outputs like the following:

```
Child pid 78462 should exec myecho
About to exec myecho from pid 78462
<shell prompt> $ Myecho running in pid 78462
Arg 0: "./myecho"
Arg 1: "Hello!"
Arg 2: "Myecho should print these"
Arg 3: "arguments."
```

We notice that the line "Finished execing myecho from pid..." never gets printed. The `fprintf` call printing this message takes place after the `execv` system call. If the `execv` call is successful, the process's address space at the time of the call gets blown way so anything after `execv` won't execute at all. Another way to think about it is that if the `execv` system call succeeds, then the system call never returns.

Alternative interface: `posix_spawn`

Calling `fork()` and `execv()` in succession to run a process may appear counter-intuitive and even inefficient. Imagine a complex program with gigabytes of virtual address space mapped and it wants to creates a new process. What's the point of copying the big virtual address space of the current program if all we are going to do is just to throw everything away and start anew?

These are valid concerns regarding the UNIX style of process management. Modern Linux systems provide an alternative system call, called `posix_spawn()`, which creates a new process without copying the address space or destroying the current process. A new program gets "spawned" in a new process and the `pid` of the new process is returned via one of the passed-by-reference arguments. Non-UNIX operating systems like Windows also uses this style of process creation.

The program in `spawnmyecho.cc` shows how to use the alternative interface to run a new program:

```
int main() {
    const char* args[] = {
        "./myecho", // argv[0] is the string used to execute the program
        "Hello!",
        "Myecho should print these",
        "arguments.",
        nullptr
    };

    fprintf(stderr, "About to spawn myecho from pid %d\n", getpid());

    pid_t p;
    int r = posix_spawn(&p, "./myecho", nullptr, nullptr,
                        (char**) args, nullptr);

    assert(r == 0);
    fprintf(stderr, "Child pid %d should run myecho\n", p);
}
```

Note that `posix_spawn()` takes many more arguments than `execv()`. This has something to do with the managing the *environment* within which the new process to be run.

In the fork-and-exec style of process creation, `fork()` copies the current process's environment, and `execv()` preserves the environment. The explicit gap between `fork()` and `execv()` provides us a natural window where we can set up and tweak the environment for the child process as needed, using the parent process's environment as a starting point.

With an interface like `posix_spawn()`, however, this aforementioned window no longer exists and we need to supply more information directly to the system call itself. We can take a look at `posix_spawn`'s manual page to find out what these extra `nullptr` arguments are about, and they are quite complicated. This teaches an interesting lesson in API design: performance and usability of an API, in many cases, are a pair of trade-offs. It can take some very careful studies and several rounds of retrogressions to settle on an interface design that's both efficient and user-friendly.

> The debate of which style of process creation is better has never settled. Modern UNIX operating systems inherited the fork-and-exec style from the original UNIX, where `fork()` turned out extremely easy to implement. Modern UNIX systems can execute `fork()` very efficiently without actually performing any substantial copying (using copy-on-write optimization) until necessary. For these reasons, in practice, the performance of the fork-and-exec style is not a common concern.

Running `execv()` without `fork()`

Finally let's take a look at `runmyecho.cc`:

```
int main() {
    const char* args[] = {
        "./myecho", // argv[0] is the string used to execute the program
        "Hello!",
        "Myecho should print these",
        "arguments.",
        nullptr
    };
    fprintf(stderr, "About to exec myecho from pid %d\n", getpid());

    int r = execv("./myecho", (char**) args);

    fprintf(stderr, "Finished execing myecho from pid %d; status %d\n",
            getpid(), r);
}
```

This program now invokes `execv()` directly, without `fork`-ing a child first. The new program (`myecho`) will print out the same `pid` as the original program. `execv()` blows away the old program, but it does not change the `pid`, because no new processes gets created. The new program runs inside the same process after the old program gets destroyed.

Note on a common mistake

It's sometimes tempting to write the following code when using the fork-and-exec style of process creation:

```
... // set up

pid_t p = fork();

if (p == 0) {
    ... // set up environment
    execv(...);
}

... // do things are parent
```

Note that the code executes assuming it's the parent is outside of the if block. It appears correct because a successful execution of `execv` blows away the current program, so the unconditional code following the if block with `execv` in the child will never execute. It is, however, not okay to assume that `execv` will always succeed (the same can be said with any system call). If the `execv()` call failed, the rest of the program will continue execute in the child, and the child can mistake itself as the parent and run into some serious logic errors. It is therefore always recommended to explicitly terminate the child (e.g. by calling `exit()`) if `execv` returns an error.

Interprocess communication

Processes operates in isolated address spaces. What if you want processes to talk to each other? After all the entire UNIX programming paradigm relies on programs being able to easily pass along information among themselves.

One way processes can communication with each other is through the file system. Two processes can agree on a file (by name) which they will use for communication. One process then can write to the file, and another process reads from the file. It is possible, but file systems are not exactly built for this purpose. UNIX provides a plethora of specific mechanisms for interprocess communication (IPC).

Simplest form of IPC: exit detection

It's useful for a parent to detect whether/when the child process has exited. The system call to detect a process exit is called `waitpid`. Let's look at `waitdemo.cc` for an example.

```
int main() {
    fprintf(stderr, "Hello from parent pid %d\n", getpid());

    // Start a child
    pid_t p1 = fork();
    assert(p1 >= 0);
    if (p1 == 0) {
        usleep(500000);
        fprintf(stderr, "Goodbye from child pid %d\n", getpid());
        exit(0);
    }
    double start_time = tstamp();

    // Wait for the child and print its status
    int status;
    pid_t exited_pid = waitpid(p1, &status, 0);
    assert(exited_pid == p1);

    if (WIFEXITED(status)) {
        fprintf(stderr, "Child exited with status %d after %g sec\n",
                WEXITSTATUS(status), tstamp() - start_time);
    } else {
        fprintf(stderr, "Child exited abnormally [%x]\n", status);
    }
}
```

The program does the following:

- Creates a child.
- The child sleeps for half a second, prints out a message, and exits.
- The parent waits for the child to finish, and prints out a message based on the child's exit status.

> See section notes for on exit status.

The interesting line in the program is the call to `waitpid()` in the parent. Note the last argument to `waitpid()`, 0, which tells the system call to *block* until the child exits. This makes the parent not runnable after calling `waitpid()` until the child exists. Blocking, as opposed to *polling*, can be a more efficient way to programmatically "wait for things to happen". It is a paradigm we will see over again in the course.

The effect of the `waitpid()` system call is that the parent will not print out the "Child exited..." message until after the child exits. The two processes are effectively *synchronized* in this way.

Exit detection communicates very little information between processes. It essentially only communicates the exit status of the program exiting. The fact that it can only deliver the communication after one program has already exited further restricts the types of actions the listening process can take after hearing from the communication. Clearly we would like a richer communication mechanism between processes. If only we can create some sort of channel between two processes which allows them to exchange arbitrary data.

Stream communication: pipes

UNIX operating systems provide a stream communication mechanism called "pipes". Pipes can be created using the `pipe()` system call. Each pipe has 2 user-facing file descriptors, corresponding to the *read end* and the *write end* of the pipe.

The signature of the `pipe()` system call looks like this:

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

A successful call creates 2 file descriptors, placed in array `pfd`:

- `pfd[0]`: read end of the pipe
- `pfd[1]`: write end of the pipe

> Useful mnemonic to remember which one is the read end:
>
> - 0 is the value of `STDIN_FILENO`, 1 is the value of `STDOUT_FILENO`
> - Program reads from stdin and writes to stdout
> - `pfd[0]` is the read end (input end), `pfd[1]` is the write end (output end)

Data written to `pfd[1]` can be read from `pfd[0]`. Hence the name, pipe.

> The read end of the pipe can't be written, and the write end of the pipe can't be read. Attempting to read/write to the wrong end of the pipe will result in a system call error (the `read()` or `write()` call will return -1).

Let's look at a concrete example in `selfpipe.cc`:

```
int main() {
    int pfd[2];
    int r = pipe(pfd);
    assert(r == 0);

    char wbuf[BUFSIZ];
    sprintf(wbuf, "Hello from pid %d\n", getpid());

    ssize_t n = write(pfd[1], wbuf, strlen(wbuf));
    assert(n == (ssize_t) strlen(wbuf));

    char rbuf[BUFSIZ];
    n = read(pfd[0], rbuf, BUFSIZ);
    assert(n >= 0);
    rbuf[n] = 0;

    assert(strcmp(wbuf, rbuf) == 0);
    printf("Wrote %s", wbuf);
    printf("Read %s", rbuf);
}
```

In this program we create a pipe, write to the pipe, and then read from the pipe. We then assert that the string we get out of the pipe is the same string we wrote into the pipe. We do everything all within the same process.

Question: Where does the data go after the write but before the read from the pipe?

The data doesn't live in the process's address space! It actually goes into the buffer cache, which is in the kernel address space.

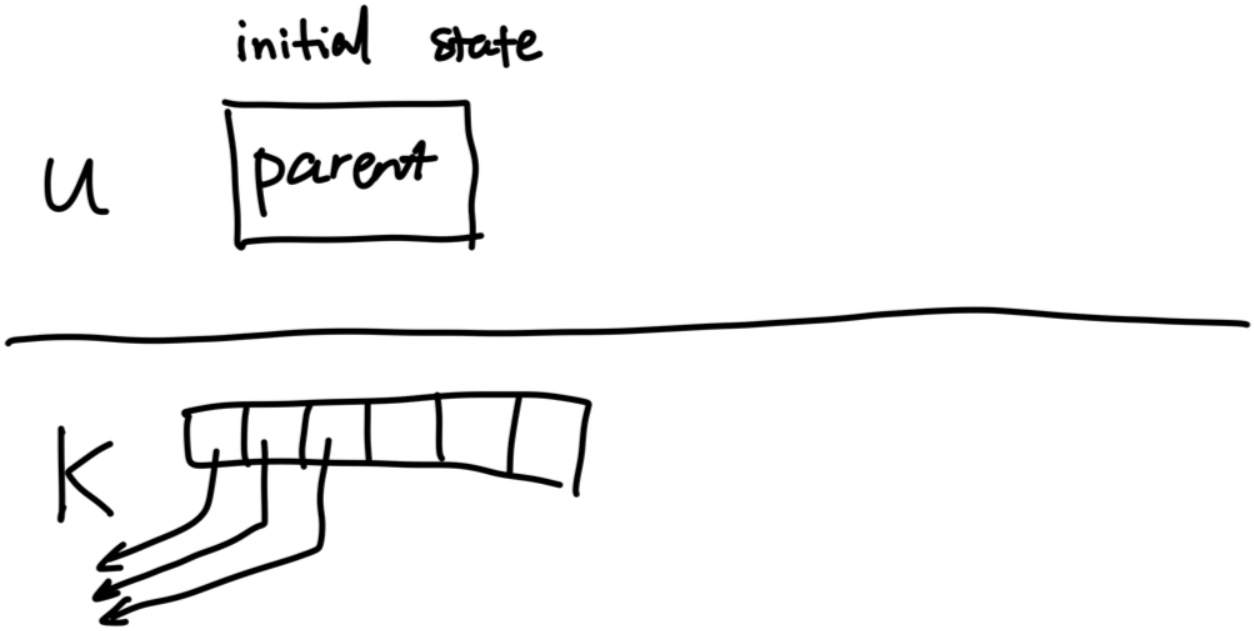
The `read()` system call blocks when reading from a stream file descriptor that doesn't have any data to be read. Pipe file descriptors are stream file descriptors, so reading from an empty pipe will block. `write()` calls to a pipe when the buffer is full (because reader the not consuming quickly enough) will also block. A `read()` from a pipe returns `EOF` if all write ends of a pipe is closed. A pipe can have multiple read ends and write ends, as we will show below.

So far we've only seen pipe functioning within the same process. Since the pipe lives in the kernel, it can also be used to pass data between processes. Let's take a look at `childpipe.cc` as an example:

```
int main() {
    int pipefd[2];
    int r = pipe(pipefd);
    assert(r == 0);

    pid_t p1 = fork();
    assert(p1 >= 0);

    if (p1 == 0) {
        const char* message = "Hello, mama!\n";
        ssize_t nw = write(pipefd[1], message, strlen(message));
        assert(nw == (ssize_t) strlen(message));
        exit(0);
    }

    FILE* f = fdopen(pipefd[0], "r");
    while (!feof(f)) {
        char buf[BUFSIZ];
        if (fgets(buf, BUFSIZ, f) != nullptr) {
            printf("I got a message! It was “%s”\n", buf);
        }
    }
    printf("No more messages :(\n");
    fclose(f);
}
```

Again we use `fork()` to create a child process, but before that we created a pipe first. The `fork()` duplicates the two pipe file descriptors in the child, but note that the pipe itself is not duplicated (because the pipe doesn't live in the process's address space). The child then writes a message to the pipe, and the same message can be read from the parent. Interprocess communication!

Note that in the scenario above we have 4 file descriptors associated with the pipe, because `fork()` duplicates the file descriptors corresponding to two ends of a pipe. The pipe in this case has 2 read ends and 2 write ends.



The program doesn't exactly behave as expected, because the parent never receives an end of file (EOF) while reading, so it hangs after printing out the message from the child. This is because there always exists a write end of the pipe in the parent itself that never gets closed.

In order for the program to work, we need to close the write end of the pipe in the parent, after the fork:

```
...
pid_t p1 = fork();
assert(p1 >= 0);

if (p1 == 0) {
    ... // child code
}

close(pipefd[1]); // close the write end in the parent
FILE* f = fdopen(pipefd[0], "r");
...
```

Pipe in a shell

Recall how we connect programs into "pipelines" using a shell:

```
a | b
```

This syntax means we create a pipe between a and b, and then let a write its stdout to the pipe, and let b read its stdin from the pipe. This gives us the effect of a passing its output to be consumed by b as input.

The shell can build up a pipeline as follows:

Start in the parent (shell) process

1. Create a pipe using `pipe()`;
2. `fork()` off child process for `a`;

Now in the child process

3. Close `pfd[0]` in child process;
4. Connect `pfd[1]` to `STDOUT_FILENO` using `dup2()`;
5. Close `pfd[1]`;
6. Run `a` using `execv()` in the child;

Now back in the parent

7. Back in the parent (shell), close `pfd[1]`;
8. `fork()` off another child process for `b`;

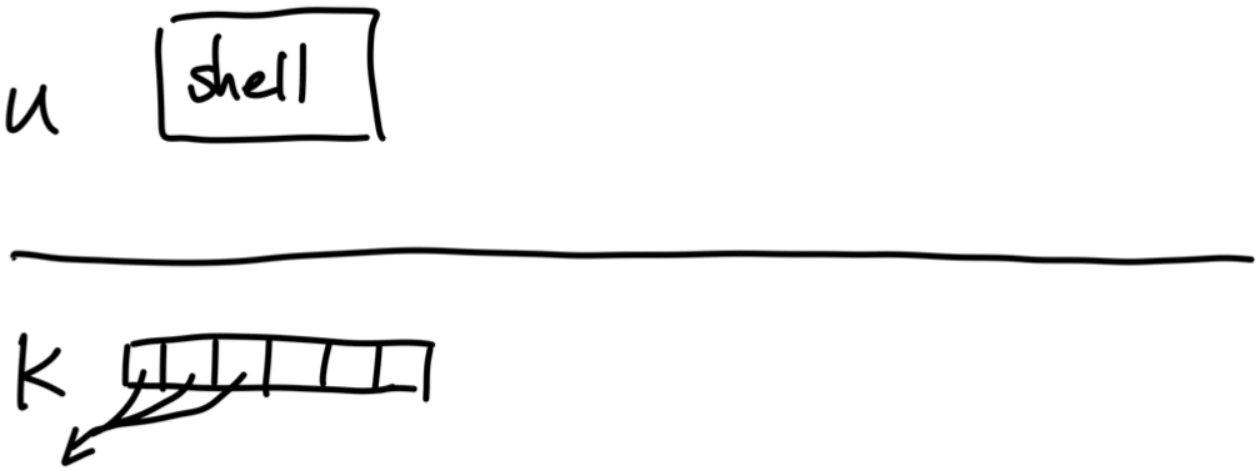
Now in the child process

9. Connect `pfd[0]` to `STDIN_FILENO` using `dup2()`;
10. Close `pfd[0]`;
11. Run `b` using `execv()` in the child;

Back in the parent

12. Close `pfd[0]` in the parent.

> We can close the pipe file descriptor after `dup2()` because `dup2()` makes the two file descriptors point to the same kernel object. UNIX lacks a "rename" feature for file descriptors so we need to manually invoke `close()` after the `dup2()` to achieve effective "rename" a file descriptor.


U
shell
K

Shell 3: Sieve of Eratosthenes, polling vs. blocking

Blocking pipe

We've mentioned in the previous lecture that I/O operations on pipe file descriptors have *blocking* semantics when the operation cannot be immediately satisfied. This can occur when the buffer for the pipe is full (when writing), or when there is no data ready to be read from the pipe (when reading). If any of these situations occur, the corresponding I/O operation will block and the current process will be suspended until the I/O operation can be satisfied or encounters an error.

The blocking semantics is useful in that it simplify *congestion control* within the pipe channel. If two processes at the two ends of the pipe generate and consume data at different rates, the faster process will automatically block (suspend) to wait for the slower process to catch up, and the application does not need to program such waiting explicitly. It provides an illusion of a reliable stream channel between two processes in that no data can ever be lost in the pipe due to a full buffer.

The Linux kernel implements fixed-size buffers for pipes. It is merely an implementation choice and a different kernel (or even a different flavor of the Linux kernel) can have different implementations. We can find out the size of these pipe buffers using the blocking behavior of pipe file descriptors: we create a new pipe, and simply keep writing to it one byte at a time. Every time we successfully wrote one byte, we know it has been placed in the buffer because there is no one reading from this pipe. We print the number of bytes written so far every time to wrote one byte to the pipe successfully. The program will eventually stop printing numbers to the screen (because `write()` blocks once there is no more space in the buffer), and the last number printed to the screen should indicate the size of the pipe buffer. The program in `pipesizer.cc` does exactly this:

```
int main() {
    int pipefd[2];
    int r = pipe(pipefd);
    assert(r >= 0);

    size_t x = 0;
    while (1) {
        ssize_t nw = write(pipefd[1], "!", 1);
        assert(nw == 1);
        ++x;
        printf("%zu\n", x);
    }
}
```

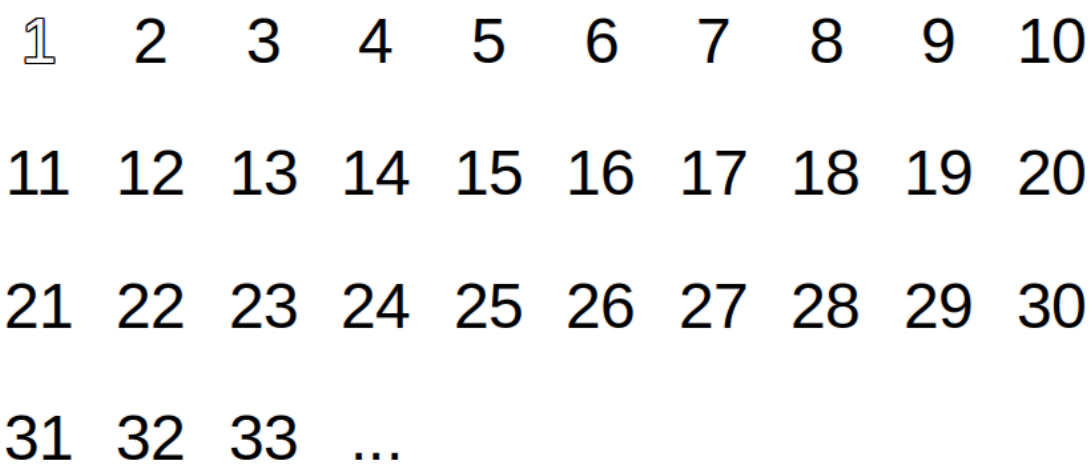
It will show that in our Ubuntu VM the size of a pipe buffer is 65536 bytes (64 KB).

Sieve of Eratosthenes and pipes

The sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to an arbitrary limit. The algorithm outputs prime numbers by filtering out, in multiple stages, multiples of known prime numbers. The algorithm works as follows:

1. Start with a sequence of integers from 2 up to the given limit;
2. Take the first number off sequence and outputs it as a prime *p*;
3. Filter out all multiples of *p* in the sequence;
4. Repeat from step 2 using the filtered sequence;

The following is a demonstration of using the sieve of Eratosthenes to find all prime numbers up to 33:



The sieve of Eratosthenes algorithm can be easily translated into a dynamically extending UNIX pipeline. Every time the algorithm outputs a new prime number, a new stage is appended to the pipeline to filter out all multiples of that prime. The following animation illustrates how the pipeline grows as the algorithm proceeds. Each `fm` program contained in a pipeline stage refers to the `filtermultiple` program, which is a simple program that reads a sequence of integers from its stdin, , filters out all multiples of the given argument, and writes the filtered sequence to stdout:

```
Seq 2 1000000
```

How can we write such a program? The tricky part lies with dynamically extending the pipeline every time a new prime is generated.

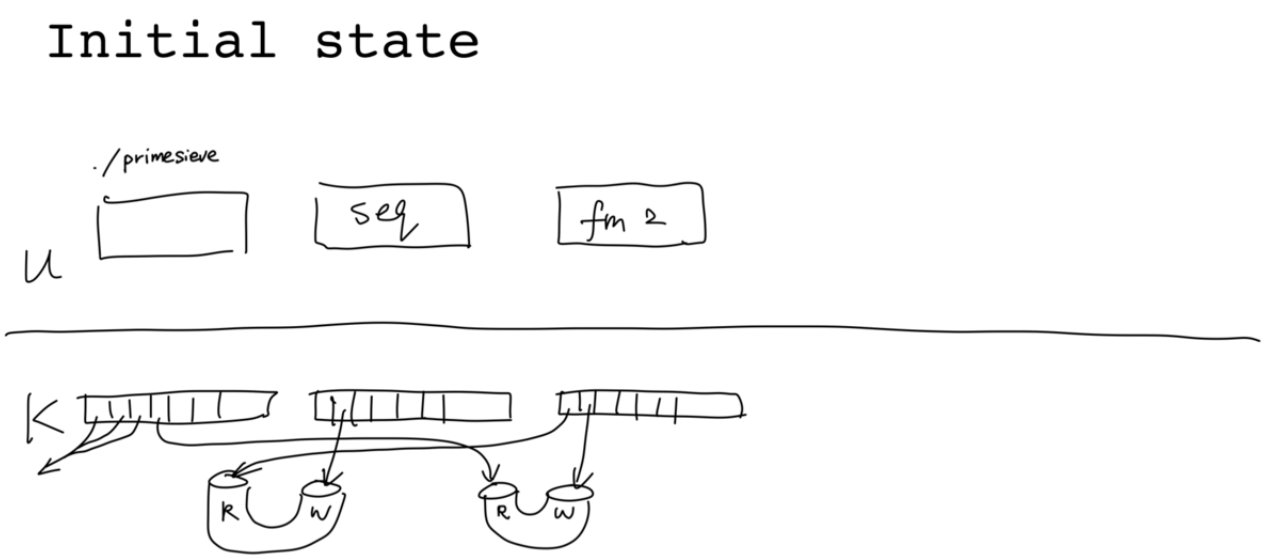
Adding a pipeline stage

There are several high-level goals we need to achieve when extending a pipeline. We would like a `filtermultiple` program to be running in the added pipeline stage, and we would like it to be reading from the output from the previous stage. There also exist a parent process (our `primesieve` program), which is the main coordinator program responsible for the creation and management of all `filtermultiple` programs in the pipeline. We also need to pay attention to *pipe hygiene*, which means by the end of our pipeline extension all pipes should look clean: each end of any pipe is only associated with one file descriptor from any process.

More concretely, adding a pipeline stage involves the following 8 steps ("ps:" means running as the parent `./primesieve` process; "c:" means running as child process). Assuming at the initial stage the parent reads the first element of the stream from the read end of a pipe via file descriptor 3:

1. ps: call `pipe()`, assuming returned file descriptors 4 (read end) and 5 (write end)
2. ps: call `fork()`
3. ps: `close(5)` (write end of the new pipe not used by parent)
4. ps: `close(3)` (read end of the old pipe no longer used by parent)
5. c: call `dup2(5, 1)`, redirect stdout to write end of the new pipe
6. c: call `dup2(3, 0)`, redirect stdin from read end of the old pipe
7. c: call `close()` on fds 3, 4 ,5 (pipe hygiene)
8. c: call `execvp()`, start running `./filtermultiple`

The process is illustrated below (with slightly more steps because of the sequence we close "unhygienic" pipe file descriptors):



Synchronization: polling and blocking

Now we transition to the last unit of the course: parallelism and synchronization.

There are 2 ways to "wait for things" in a process: polling and blocking:

- Polling: The process waits for an event while remaining runnable.
- Blocking: The process waits for an event while not runnable.

> Polling is sometimes also referred to as non-blocking. Strictly speaking, there is a slight conceptual distinction between the two: non-blocking refers to the property of a single interface call, whereas polling refers to action of repeatedly querying a non-blocking interface until some condition is met.

Consider that we need to wait for the following condition (called *timed wait*):

> Wait for the earliest of:
>
> - Child to exit
> - 0.75 seconds to elapse

How can we wait this kind of conditions using a combination of blocking and polling system calls?

Since part of our condition is to wait for the child to exit, maybe `waitpid()` jumps to mind. However, the `waitpid()` described in prior lectures operates in *blocking mode*, and there is no way to specify a 0.75-second "timeout" value. So if the child doesn't exit within 0.75 seconds we miss our condition.

There actually exists a version of the `waitpid()` system call that operates in *polling mode*. We can configure which mode `waitpid()` should operate in using the last parameter (called *options*) of the call:

- `waitpid(pid, status, 0)`: blocking mode
- `waitpid(pid, status, WNOHANG)`: polling (non-blocking) mode
  - In polling mode, `waitpid()` returns 0 immediately if the process has not exited yet.

We can implement this compounded waiting using these 2 versions of `waitpid()`. The relevant code is in `timedwait-poll.cc`:

```
int p1 = ... // p1 is pid of the child
int status;
pid_t exited_pid = 0;

while (tstamp() - start_time < 0.75 && exited_pid == 0) {
    exited_pid = waitpid(p1, &status, WNOHANG);
    assert(exited_pid == 0 || exited_pid == p1);
}
```

This is a form of "busy-waiting": the process keeps running and querying about the exit status of the child before the condition is met. This consumes CPU resources and therefore power. Ideally we would like a "waiting" process to be taken off the CPU until the condition it's waiting on becomes satisfied. In order for a process to yield the CPU resources in this way some blocking needs to be introduced. What we also need is to unblock at the right time. If only the blocked process can receive a "signal" when, for example, 0.75 seconds has elapsed or when the child has exited.

Signals

Signals are user-level abstraction of interrupts. A signal has the following effects:

1. Interrupts normal process execution.
2. Interrupts *blocked* system calls, making them immediately return -1 with `errno` set to `EINTR`.

We have in fact all used signals to manage processes before. When we hit `CTRL+C` on the keyboard to kill a running program, we are actually telling the operating system to send a signal (`SIGINT` in this case) to the program. The default behavior of the `SIGINT` signal is to kill the process.

Handling signals

The process can define a *signal handler* to override the default behavior of some signals. (This is why `CTRL+C` doesn't work in some programs as you would expect. In gdb and many text editors you will see this behavior.)

We use a signal handler to help us with the timed wait task. Let's see the following code in `timedwait-block.cc`:

```
void signal_handler(int signal) {
    (void) signal;
}

int main(int, char** argv) {
    fprintf(stderr, "Hello from %s parent pid %d\n", argv[0], getpid());

    // Demand that SIGCHLD interrupt system calls
    int r = set_signal_handler(SIGCHLD, signal_handler);
    assert(r >= 0);

    // Start a child
    pid_t p1 = fork();
    assert(p1 >= 0);
    if (p1 == 0) {
        usleep(500000);
        fprintf(stderr, "Goodbye from %s child pid %d\n", argv[0], getpid());
        exit(0);
    }

    double start_time = tstamp();

    // Wait for the child and print its status
    r = usleep(750000);
    if (r == -1 && errno == EINTR) {
        fprintf(stderr, "%s parent interrupted by signal after %g sec\n",
                argv[0], tstamp() - start_time);
    }

    ...
}
```

By registering a signal handler, we demand that all outstanding blocked system calls be interrupted when the parent receives the `SIGCHLD` signal. The `SIGCHLD` signal is delivered to the process when any of its children exits. The default behavior of `SIGCHLD` is to ignore the signal, so we need to explicitly define and register a handler to override the default behavior.

> For descriptions and list of all signals supported by the Linux kernel, use manual page `man 7 signal`.

The waiting part in the parent now becomes just this, without any polling loop:

```
    // Wait for the child and print its status
    r = usleep(750000);
    if (r == -1 && errno == EINTR) {
        fprintf(stderr, "%s parent interrupted by signal after %g sec\n",
                argv[0], tstamp() - start_time);
    }
```

The parent simply goes to sleep for 0.75 seconds. If the child exited before the 0.75-second deadline, the `SIGCHLD` signal will cause the `usleep()` system call to unblock with `errno == EINTR`. This way the parent does no busy waiting and will only be blocked for at most 0.75 seconds.

Race conditions in signals

The code above looks correct and behaves as expected in our tests. It is really correct though? What if we make child exit right away, instead of waiting for half a second before exiting? If the child exited so fast that the SIGCHLD signal arrived before the parent even started calling `usleep()`, the parent can simply "miss" the signal. In this case the parent would wait for the full 0.75 seconds even though the child exited immediately (within milliseconds). This is not what we want!

The program has what we call a *race condition*. A race condition is a bug that's dependent on scheduling ordering. These bugs are particularly difficult to deal with because they may not manifest themselves unless specific scheduling conditions are met.

Perhaps we can eliminate the race condition by having the signal handler actually do something. We modify the signal handler to set a flag variable when a `SIGCHLD` signal is received. The modified program is in `timedwait-blockvar.cc`.

```
static volatile sig_atomic_t got_signal;

void signal_handler(int signal) {
    (void) signal;
    got_signal = 1;
}

int main(int, char** argv) {
    ...

    // Wait for the child and print its status
    if (!got_signal) {
        r = usleep(750000);
    }

    ...
}
```

We changed the waiting logic in the parent so that it only goes to sleep after making sure that the flag variable is not set (meaning the child has not exited yet). This should solve our problem!

Does it though?

What if the `SIGCHLD` signal arrives right after we check the flag variable in the `if` conditional, but before we invoke the `usleep()` system call?

Again, there still exists a window, however small it is, for the parent to miss the signal.

We actually already learned the mechanism needed to reliably eliminate this race condition. It's called, wait for it, **a pipe**. We will show how to use pipes to solve this race condition in the next lecture.

Synchronization 1: Signals, race condition, threads

Note on zombie processes

This is relevant to the problem set. When a process forks a child, the child eventually exits and will have a exit status. The child process then enters a "zombie state": the process no longer exits, but the kernel still keeps around its pid and its exit status, waiting for `waitpid()` to be called on the process. Zombie processes consume kernel resources and we should avoid having zombies lying around whenever possible.

The trick for avoiding zombie processes is to call `waitpid()` at least once for each child. Invoking `waitpid()` with -1 as the pid argument will check on an exit status of an arbitrary child.

Signals

Recall that one very important functionality of an operating system is to provide user-accessible abstractions of the underlying hardware features that are inconvenient or unsafe for a user process to access directly. UNIX-like systems abstract interrupts (as hardware feature) to user-space signals.

Comparisons between interrupts and signals:

Interrupts	Signals
Can happen at any time, can be blocked	Can happen any any time, can be blocked
Protected control transfer, saves machine state as part of kernel state	Previous machine state was saved by kernel to ptable
Kernel invokes interrupt handler in kernel mode on the kernel stack	Kernel invokes signal handler in user mode on a signal stack

We use the following system calls to manipulate signals:

- `sigaction(sig, handler, oldhandler)`: installs a signal handler
- `kill(pid, sig)`: sends a signal

We use `sigaction()` to install a signal handler, which should be a *very simple* function that runs when the signal gets delivered. Since the signal can arrive at any time between 2 instructions, the signal handler should really be kept at minimum complexity and it definitely should not call user space library functions like `malloc()`. Imagine that a signal may arrive in between 2 instructions in the `malloc()` routine itself, and the internal state and data structure used by `malloc()` may not be consistent. Typical things we do in a signal handler is setting a flag variable or checking the value of some other variable, and we leave the complex task to be triggered by these flag variables in the main process.

> For a list of functions that *can* be safely used in signal handlers, see manual page `man 7 signal-safety`.

Solving the race condition...

Recall from the last lecture where we were trying to implement the timed wait. Our plan was to have a signal (SIGCHLD) unblock the `usleep()` system call in the parent if the child exited within 0.75 seconds. Our solution has a condition: there is small window during which if the signal gets delivered, the parent would miss the effect

of the signal and overlook the fact that the child has already exited. Before the end of the last lecture we hinted that the problem can be reliably solved using a pipe.

... with pipe

Take a look at code in `shell4/timedwait-selfpipe.cc` for how we could do it. The relevant portion of the code is pasted below.

```
int signalpipe[2];

void signal_handler(int signal) {
    (void) signal;
    ssize_t r = write(signalpipe[1], "!", 1);
    assert(r == 1);
}

int main(int, char** argv) {
    ...

    // Wait for 0.75 sec, or until a byte is written to `signalpipe`,
    // whichever happens first
    struct timeval timeout = { 0, 750000 };
    fd_set fds;
    FD_SET(signalpipe[0], &fds);
    r = select(signalpipe[0] + 1, &fds, nullptr, nullptr, &timeout);

    ...
}
```

We simply write one character into the pipe in the signal handler (it is fine to invoke system calls in signal handlers), and then we use the `select()` system call in the parent to wait for the read end of the pipe to become ready, with a 0.75-second timeout value.

> Some notes on `select()`. The `select()` system call is very commonly seen in even-driven programs (like a server). It has the following signature:
>
> ```
> select (
>     nfds,     // largest fd value to monitor + 1
>     read_set,
>     write_set,
>     exp_set,
>     timeout
> )
> ```
>
> `read_set` and `write_set` are `fd_set` objects.
>
> The system call will return at the earliest of:
>
> - Some fd in `read_set` becomes readable (meaning that `read(fd)` will not block);
> - Some fd in `write_set` becomes writable (`write(fd)` will not block);
> - Timeout elapses;
> - A signal is delivered;

This is indeed one of the few correct ways to implement timed wait, eliminating all potential race conditions.

> It may be surprising that we need to involve so many machinery to solve this seemingly simple problem. The race condition we discussed in this example is a very common occurrence at all levels of concurrent systems design. It is so important that there is a name for it, and it's called the *sleep-wakeup race condition*. As we learn later in this unit, there are synchronization mechanisms designed specifically to address sleep-wakeup race conditions.

... with `signalfd`

There is a system call, `signalfd()`, that automatically builds a signal-pipe-like vehicle for us to handle these race conditions. Instead of explicitly creating a pipe with 2 ends and define a signal handler that explicitly writes to the pipe, we simply tell the operating to create a file descriptor from which information about a signal can be read once a signal arrives. Code for doing this is in `shell4/timedwait-signalfd.cc`. Note that there is no definition of a signal handler any more. The relevant code is shown below:

```
int main(int, char** argv) {
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    int r = sigprocmask(SIG_BLOCK, &mask, nullptr);
    assert(r == 0);
    int sigfd = signalfd(-1, &mask, SFD_CLOEXEC);
    assert(sigfd >= 0);

    ...

    // Wait for 0.75 sec, or until something is written to `sigfd`,
    // whichever happens first
    struct timeval timeout = { 0, 750000 };
    fd_set fds;
    FD_SET(sigfd, &fds);
    r = select(sigfd + 1, &fds, nullptr, nullptr, &timeout);

    ...
}
```

Threads

Modern day computers usually have multiple processors (CPUs) built in. The operating system provides a user-level abstraction of multiple processors call *threads*. Here is the comparison table of threads (as an abstraction) and multiple processors (as a hardware feature):

Multiple processors	Threads
One primary memory	One virtual address space
Multiple sets of registers and logical computations	Multiple sets of registers, stacks, and logical computations

A process can contain multiple threads. All threads within the same process share the same virtual address space and file descriptor table. Each thread will however have its own set of registers and stack. The processes we have looked at so far all have a single thread running. For multi-threaded processes the kernel will need to store a set of registers for each thread, rather than for each process, to maintain the full process state.

Let's look at how to use threads using our example program in `incr-basic.cc`:

```
void threadfunc(unsigned* x) {
    // This is a correct way to increment a shared variable!
    // ... OR IS IT?!?!?!?!?!?!?!?!?!
    for (int i = 0; i != 10000000; ++i) {
        *x += 1;
    }
}

int main() {
    std::thread th[4];
    unsigned n = 0;
    for (int i = 0; i != 4; ++i) {
        th[i] = std::thread(threadfunc, &n);
    }
    for (int i = 0; i != 4; ++i) {
        th[i].join();
    }
    printf("%u\n", n);
}
```

In this code we run the function `threadfunc()` in parallel in 4 threads. The `std::thread::join()` function makes the main thread block until the thread upon which the `join()` is called finishes execution. So the final value of `n` will be printed by the main thread after all 4 threads finish.

In each thread we increment a shared variable 10 million times. There are 4 threads incrementing in total and the variable starts at zero. What should the final value of the variable be after all incrementing threads finish?

40 million seems like a reasonable answer, but by running the program we observe that sometimes it prints out values like 30 million or 20 million. What's going on?

First we noticed that the compiler can optimize away the loop in `threadfunc()`. It recognizes that the loop is simply incrementing the shared variable by an aggregate of 10 million, so it will transform the loop into a single `addl` instruction with immediate value 10 million.

Secondly, there is a race condition in the `addl` instruction itself! Up until this point in the course, we've been thinking x86 instructions as being indivisible and atomic. In fact, they are not, and their lack of atomicity shows up in a multi-processor environment.

Inside the processor hardware, the `addl $10000000, (%rdi)` is actually implemented as 3 separate "micro-op" instructions:

1. `movl (%rdi), %temp` (load)
2. `addl $10000000, %temp` (add)
3. `movl %temp, (%rdi)` (store)

Imagine 2 threads executing this `addl` instruction at the same time (concurrently). Each thread loads the same value of `(%rdi)` from memory, then adds 10 million to it in their own separate temporary registers, and then write the same value back to `(%rdi)` in memory. The last write to memory by each thread will overwrite each other with the same value, and one increment by 10 million will essentially be lost.

This is the behavior of running 2 increments on the same variable in x86 assembly. In the C/C++ abstract machine, accessing shared memory from different threads without proper synchronization is ***undefined behavior***, unless all accesses are reads.

> Re-running this experiment with compiler optimizations turned off (so the loop does not get optimized away), we will get a result like 15285711, where roughly 2/3 of the updates are lost.

There are 2 ways to synchronize shared memory accesses in C++. We will describe a low-level approach, using C++'s `std::atomic` library, first, before introducing a higher level and more general way of performing synchronization.

Atomics

`incr-atomic.cc` implements synchronized shared-memory access using C++ atomics. Relevant code in `threadfunc()` is shown below.

```
void threadfunc(std::atomic<unsigned>* x) {
    for (int i = 0; i != 10000000; ++i) {
        x->fetch_add(1);
        // `*x += 1` and `(*x)++` also work!
    }
}
```

C++'s atomics library implements atomic additions using an x86's atomic instruction. When we use `objdump` to inspect the assembly of `threadfunc()`, we see an `lock addl ...` instruction instead of just `addl ...`. The `lock` prefix of the `addl` instruction asks the processor the hold on to the cache line with the shared variable (or in Intel terms, lock the memory bus) until the entire `addl` instruction has completed.

C++ atomics and `lock`-prefixed instructions only work with word-sized variables that do not span multiple cache lines. A more general way to perform synchronized access to arbitrary data in memory is called a mutex (short for mutual exclusion), and we will cover it more in the next lecture.

Synchronization 2: Mutex, bounded buffers

Recall that in the last lecture we showed that updating shared memory variables in concurrent threads requires synchronization, otherwise some update may get lost because of the interleaving of operations in different threads. Synchronized updates can be achieved by using the C++ `std::atomic` template library, which automatically translates the `addl` instruction used to perform the update into a `lock`-prefixed instruction that behaves atomically.

C++'s `std::atomic` library is powerful (and also great progress in standardization of atomic operations), but it works only on integers that are word-sized or below. To synchronize more complex objects in memory or perform more complex synchronization tasks, we need abstractions called *synchronization objects*.

Synchronization objects are types whose methods can be used to achieve synchronization and atomicity on normal (non-`std::atomic`-wrapped) objects. Synchronization objects provides various abstract properties that simplify programming multi-threaded access to shared data. The most common synchronization object is called a *mutex*, which provides the *mutual exclusion* property.

Mutual exclusion

Mutual exclusion means that at most one thread accesses the shared data at a time.

In our multi-threaded `incr-basic.cc` example from the last lecture, the code does not work because more than one thread can access the shared variable at a time. The code would behave correctly if the mutual exclusion policy is enforced. We can use a `mutex` object to enforce mutual exclusion (`incr-mutex.cc`). In this example it has the same effect as wrapping `*x` in a `std::atomic` template:

```
std::mutex mutex;

void threadfunc(unsigned* x) {
    for (int i = 0; i != 10000000; ++i) {
        mutex.lock();
        *x += 1;
        mutex.unlock();
    }
}
```

The mutex has an internal state (denoted by `state`), which can be either *locked* or *unlocked*. The semantics of a mutex object is as follows:

- Upon initialization, `state = unlocked`.
- `mutex::lock()` method: waits until `state` becomes `unlocked`, and then *atomically* sets `state = locked`. Note the two steps shall complete in one atomic operation.
- `mutex::unlock()` method: asserts that `state == locked`, then sets `state = unlocked`.

The mutual exclusion policy is enforced in the code region between the `lock()` and `unlock()` invocations. We call this region the *critical region*.

Implementing a mutex

Let's now think about how we may implement such a mutex object.

Does the following work?

```
struct mutex {
    static constexpr int unlocked = 0;
    static constexpr int locked = 1;

    int state = unlocked;

    void lock() {
        while (state == locked) {}
        state = locked;
    }

    void unlock() {
        state = unlocked;
    }
};
```

No! Imagine that two threads calls `lock()` on an unlocked mutex at the same time. They both observe that `state == unlocked`, skip the while loop, and set `state = locked`, and then return. Now both threads think they have the lock and proceeds to the critical region, all at the same time! Not Good!

In order to properly implement a mutex we need atomic read-modify-write operation. `std::atomic` provides these operations in the form of `operator++`and `operator--` operators. The following mutex implementation should be correct:

```
struct mutex {
    std::atomic<int> state = 0;

    void lock() {
        while (++state != 1) {
            --state;
        }
    }

    unlock() {
        --state;
    }
};
```

Note that the `unlock()` method performs an atomic decrement operation, instead of simply writing 0. Simply storing 0 to the mutex state is incorrect because of if this store occurred between the `++state` and `--state` steps in the `while` loop in the `lock()` method, `state` becomes negative and the `while` loop can never exit.

With the help of mutex, we can build objects that support synchronized accesses with more complex semantics.

Bounded buffers

A bounded buffer is a synchronized object that supports the following operations:

- `read(buf, n)`: reads up to `n` chars from the bounded buffer to `buf`;
- `write(buf, n)`: writes up to `n` chars into the bounded buffer from `buf`.

Bounded buffer is the abstraction used to implement pipes and non-seekable stdio caches.

Unsynchronized buffer

Let's first look at a bounded buffer implementation `bbuffer-basic.cc`, which does not perform any synchronization.

```
struct bbuffer {
    static constexpr size_t bcapacity = 128;
    char bbuf_[bcapacity];
    size_t bpos_ = 0;
    size_t blen_ = 0;
    bool write_closed_ = false;

    ssize_t read(char* buf, size_t sz);
    ssize_t write(const char* buf, size_t sz);
    void shutdown_write();
};

ssize_t bbuffer::write(const char* buf, size_t sz) {
    assert(!this->write_closed_);
    size_t pos = 0;
    while (pos < sz && this->blen_ < bcapacity) {
        size_t bindex = (this->bpos_ + this->blen_) % bcapacity;
        size_t bspace = std::min(bcapacity - bindex, bcapacity - this->blen_);
        size_t n = std::min(sz - pos, bspace);
        memcpy(&this->bbuf_[bindex], &buf[pos], n);
        this->blen_ += n;
        pos += n;
    }
    if (pos == 0 && sz > 0) {
        return -1;  // try again
    } else {
        return pos;
    }
}

ssize_t bbuffer::read(char* buf, size_t sz) {
    size_t pos = 0;
    while (pos < sz && this->blen_ > 0) {
        size_t bspace = std::min(this->blen_, bcapacity - this->bpos_);
        size_t n = std::min(sz - pos, bspace);
        memcpy(&buf[pos], &this->bbuf_[this->bpos_], n);
        this->bpos_ = (this->bpos_ + n) % bcapacity;
        this->blen_ -= n;
        pos += n;
    }
    if (pos == 0 && sz > 0 && !this->write_closed_) {
        return -1;  // try again
    } else {
        return pos;
    }
}
```

It implements what we call *circular buffer*. Every time we read a character out of the buffer, we increment `bpos_`, which is the index of the next character to be read in the buffer. Whenever we write to the buffer, we increment `blen_`, which is the number of bytes currently occupied in the buffer. The buffer is circular, which is why all

index arithmetic is modulo the total capacity of the buffer.

When there is just one thread accessing the buffer, it works perfectly fine. But does it work when multiple threads are using the buffer at the same time? In our test program in `bbuffer-basic.cc`, we have one reader thread reading from the buffer and a second writer thread writing to the buffer. We would expect everything written to the buffer by the writer thread to show up exactly as it was written once read out by the reader thread.

It does not work, because there is no synchronization over the internal state of the bounded buffer. `bbuffer::read()` and `bbuffer::write()` both modify internal state of the `bbuffer` object (most critically `bpos_` and `blen_`), and such accesses require synchronization to work correctly in a multi- threaded environment. One way to fix this is to wrap the function bodies of the `read()` and `write()` methods in critical regions, using a mutex.

A correct version of a synchronized bounded buffer can be found in `bbuffer-mutex.cc`. Key differences from the unsynchronized version are highlighted below:

```
struct bbuffer {
    ...

    std::mutex mutex_;

    ...
};

ssize_t bbuffer::write(const char* buf, size_t sz) {
    this->mutex_.lock();

    ...

    this->mutex_.unlock();
    if (pos == 0 && sz > 0) {
        return -1;  // try again
    } else {
        return pos;
    }
}

ssize_t bbuffer::read(char* buf, size_t sz) {
    this->mutex_.lock();

    ...

    this->mutex_.unlock();
    if (pos == 0 && sz > 0 && !this->write_closed_) {
        return -1;  // try again
    } else {
        return pos;
    }
}
```

This correctly implements a synchronized bounded buffer. Simply wrapping accesses to shared state within critical sections using a mutex is the easiest and probably also the most common way to make complex in-memory objects synchronized (or thread-safe).

Using one mutex for the entire object is what we called *coarse-grained* synchronization. It is correct, but it also limits concurrency. Under this scheme, whenever there is a writer writing to the buffer, the reader can't acquire the lock and will have to wait until the writer finishes. The reader and the writer can never truly read and write to the buffer at the same time. We will show in the next lecture that we can achieve more concurrency by using a different synchronization object.

Synchronization 3: Mutexes, condition variables, and compare-exchange

We continue from where the previous lecture left off -- implementing a synchronized bounded buffer.

Recall that a bounded buffer is a circular buffer that can hold up to `CAP` characters, where `CAP` is the capacity of the buffer. The bounded buffer supports following opeartions:

- `read(buf, sz)`:
  - Reads up to `sz` characters from the bounded buffer into `buf`
  - Removes characters read from the bounded buffer
  - If the bounded buffer is empty, it should lock until it becomes nonempty or closed
  - Returns the number of characters read
- `write(buf, sz)`:
  - Writes up to `sz` characters, from `buf` to the end of bounded buffer
  - Writes up to bounded buffer capacity `CAP` characters
  - If bounded buffer becomes full, it should block until buffer becomes nonfull
  - Returns the number of characters written

Example: Assuming we have a bounded buffer object `bbuf` with `CAP=4`.

```
bbuf.write("ABCDE", 5); // returns 4
bbuf.read(buf, 3);      // returns 3 ("ABC")
bbuf.read(buf, 3);      // returns 1 ("D")
bbuf.read(buf, 3);      // blocks
```

The bounded buffer preserves two important properties:

- Every character written can be read exactly once
- The order in which characters are read is the same order in which they are written

Each bounded buffer operation should also be *atomic*, just like `read()` and `write()` system calls.

Review the bounded buffer implementation in `synch2/bbuffer-basic.cc` to see how the circular buffer operates. This version of the bounded buffer does not yet perform synchronization so it only works with a single thread. Instead of blocking the buffer returns -1 whenever blocking should occur. We need to fix it in the synchronized version.

To help us a design a synchronized bounded buffer, we need help from something we call the *Fundamental Law of Synchronization*.

Fundamental Law of Synchronization: If two threads simultaneously access an object in memory, then both accesses *must* be reads. Otherwise, the program invokes undefined behavior.

> C++ `std::atomic`-wrapped variables do not fall under this rule. They can be read and written concurrently in different threads and behavior is well-defined by the C++ memory consistency model.

Now let's look at the definition of the bounded buffer:

```
struct bbuffer {
    static constexpr size_t bcapacity = 128;
    char bbuf_[bcapacity];
    size_t bpos_ = 0;
    size_t blen_ = 0;
    bool write_closed_ = false;

    ...
};
```

The bounded buffer's internal state, `bbuf_`, `bpos_`, `blen_`, and `write_closed_` are both modified and read by `read()` and `write()` methods. Local variables defined within these methods are not shared. So we need to carry out synchronization on shared variables (internal state of the buffer), but not on local variables.

Mutex synchronization

We can make the bounded buffer synchronized by using `std::mutex` objects. We can associate a mutex with the internal state of the buffer, and only access these internal state when the mutex is locked.

The `bounded_buffer::write()` method in `synch2/bbuffer-mutex.cc` is implemented with mutex synchronization. Note that we added a definition of a mutex to the `bbuffer` struct definition, and we are only accessing the internal state of the buffer within the region between `this->mutex_.lock()` and `this->mutex_.unlock()`, which the the time period when the thread locks the mutex.

> The association between the mutex and the state it protects is rather arbitrary. These mutexes are also called "advisory locks", as their association with the state they protect are not enforced in any way by the runtime system, and must be taken care of by the programmer. Their effectiveness solely relies on the program following protocols associating the mutex with the protected state. In other words, if a mutex is not used correctly, there is no guarantee that the underlying state is being properly protected.

Mutex pitfalls

Now consider the following code, which move the mutex `lock()`/`unlock()` pair to inside the `while` loop. We still have just one `lock()` and one `unlock()` in our code. Is it correct?

```
ssize_t bbuffer::write(const char* buf, size_t sz) {
    assert(!this->write_closed_);
    size_t pos = 0;
    while (pos < sz && this->blen_ < bcapacity) {
        this->mutex_.lock();
        size_t bindex = (this->bpos_ + this->blen_) % bcapacity;
        this->bbuf_[bindex] = buf[pos];
        ++this->blen_;
        ++pos;
        this->mutex_.unlock();
    }
    ...
}
```

The code is incorrect because `this->blen_` is not protected by the mutex, but it should be.

What about the following code -- is it correct?

```
ssize_t bbuffer::write(const char* buf, size_t sz) {
    this->mutex_.lock();
    assert(!this->write_closed_);
    size_t pos = 0;
    while (pos < sz && this->blen_ < bcapacity) {
        this->mutex_.lock();
        size_t bindex = (this->bpos_ + this->blen_) % bcapacity;
        this->bbuf_[bindex] = buf[pos];
        ++this->blen_;
        ++pos;
        this->mutex_.unlock();
    }
    ...
}
```

It's also wrong! Upon entering the `while` loop for the first time, the mutex is already locked, and we are trying to lock it again. Trying to lock a mutex multiple times in the same thread causes the second lock attempt to block indefinitely.

So what if we do this:

```
ssize_t bbuffer::write(const char* buf, size_t sz) {
    this->mutex_.lock();
    assert(!this->write_closed_);
    size_t pos = 0;
    while (pos < sz && this->blen_ < bcapacity) {
        this->mutex_.unlock();
        this->mutex_.lock();
        size_t bindex = (this->bpos_ + this->blen_) % bcapacity;
        this->bbuf_[bindex] = buf[pos];
        ++this->blen_;
        ++pos;
        this->mutex_.unlock();
        this->mutex_.lock();
    }
    ...
}
```

Now everything is protected, right? NO! This is also incorrect and in many ways much worse than the two previous cases.

Although `this->blen_` is now seemingly protected by the mutex, it is being protected within a different region from the region where the rest of the buffer state (`bbuf_`, `bpos_`) is protected. Further more, the mutex is unlocked at the end of every iteration of the `while` loop. This means that when two threads call the `write()` method concurrently, the lock can bounce between the two threads and the characters written by the threads can be interleaved, violating the atomicity requirement of the `write()` method.

C++ mutex patterns

We may find it very common that we write some synchronized function where we need to lock the mutex first, and then unlock it before the function returns. Doing it repeatedly can be tedious. Also, if the function can return at multiple points, it is possible to forget a unlock statement before a return, resulting errors. C++ has a pattern

to help us deal with these problems and simplify programming: a *scoped lock*.

We use scoped locks to simplify programming of the bounded buffer in `synch2/bbuffer-scoped.cc`. The `write()` method now looks like the following:

```
ssize_t bbuffer::write(const char* buf, size_t sz) {
    std::unique_lock<std::mutex> guard(this->mutex_);
    assert(!this->write_closed_);
    size_t pos = 0;
    while (pos < sz && this->blen_ < bcapacity) {
        size_t bindex = (this->bpos_ + this->blen_) % bcapacity;
        this->bbuf_[bindex] = buf[pos];
        ++this->blen_;
        ++pos;
    }
    ...
}
```

Note that in the first line of the function body we declared a `std::unique_lock` object, which is a scoped lock that locks the mutex for the scope of the function. Upon initialization of the `std::unique_lock` object, the mutex is automatically locked, and when this object goes out of scope, the mutex is automatically unlocked. These special scoped lock objects lock and unlock mutexes in their constructors and destructors to achieve this effect. This design pattern is also called *Resource Acquisition is Initialization* (or RAII), and is a common pattern in software engineering in general. The use of RAII simplify coding and also avoids certain programming errors.

You should try to leverage these scoped lock objects for your synchronization tasks in the problem set as well.

Condition variables

So far we implemented everything in the spec of the bounded buffer except **blocking**. It turns out mutex alone is not enough to implement this feature -- we need another synchronization object. In standard C++ this object is called a *condition variable*.

A condition variable supports the following operations:

- `wait(std::unique_lock& lock)`: In one atomic step, it unlocks the lock, blocks until another thread calls `notify_all()`. It also relocks the lock before returning (waking up).
- `notify_all()`: Wakes up all threads blocked by calling `wait()`.

Condition variables are designed to avoid the sleep-wakeup race conditions we briefly visited when we discussed signal handlers. The atomicity of the unlocking and the blocking guarantees that a thread that blocks can't miss a `notify_all()` message.

Logically, the writer to the bounded buffer should block when the buffer becomes full, and should unblock when the buffer becomes nonfull again. Let's create a condition variable, called `nonfull_`, in the bounded buffer, just under the mutex. Note that we conveniently named the condition variable after the condition under which the function should unblock. It will make code easier to read later on. The `write()` method implements blocking is in `synch2/bbuffer-cond.cc`. It looks like the following:

```
ssize_t bbuffer::write(const char* buf, size_t sz) {
    std::unique_lock<std::mutex> guard(this->mutex_);
    assert(!this->write_closed_);
    while (this->blen_ == bcapacity) {  // #1
        this->nonfull_.wait(guard);
    }
    size_t pos = 0;
    while (pos < sz && this->blen_ < bcapacity) {
        size_t bindex = (this->bpos_ + this->blen_) % bcapacity;
        this->bbuf_[bindex] = buf[pos];
        ++this->blen_;
        ++pos;
    }
    ...
```

The new code at #1 implements blocking until the condition is met. This is a pattern when using condition variables: *the condition variable's `wait()` function is almost always called in a `while` loop, and the loop tests the condition in which the function must block.*

On the other hand, `notify_all()` should be called whenever some changes we made might turn the unblocking condition true. In our scenario, this means we must call `notify_all()` in the `read()` method, which takes characters out of the buffer and can potentially unblock the writer, as shown in the inserted code #2 below:

```
ssize_t bbuffer::read(char* buf, size_t sz) {
    std::unique_lock<std::mutex> guard(this->mutex_);
    ...
    while (pos < sz && this->blen_ > 0) {
        buf[pos] = this->bbuf_[this->bpos_];
        this->bpos_ = (this->bpos_ + 1) % bcapacity;
        --this->blen_;
        ++pos;
    }
    if (pos > 0) {                    // #2
        this->nonfull_.notify_all();
    }
```

Note that we only put `notify_all()` in a `if` but put the `wait()` inside a `while` loop. Why is it necessary to have `wait()` in a `while` loop?

`wait()` is almost always used in a loop because of what we call *spurious* wakeups. Since `notify_all()` wakes up all threads blocking on a certain `wait()` call, by the time when a particular blocking thread locks the mutex and gets to run, it's possible that some other blocking thread has already unblocked, made some progress, and changed the unblocking condition back to false. For this reason, a "woken-up" must revalidate the unblocking condition before proceeding further, and if the unblocking condition is not met it must go back to blocking. The `while` loop achieves exactly this.

Implementing a mutex with a single bit

Last time we showed that we can implement a busy-waiting mutex using an atomic counter. Now we show that a mutex can also be implemented using just a single bit, with some special atomic machine instructions.

A busy-waiting mutex (also called a spin lock) can be implemented as follows:

```
struct mutex {
    std::atomic<int> spinlock;

    void lock() {
        while (spinlock.swap(1) == 1) {}
    }

    void unlock() {
        spinlock.store(0);
    }
};
```

The `spinlock.swap()` method is an atomic swap method, which in one atomic step stores the specified value to the atomic `spinlock` variable and returns the old value of the variable.

It works because `lock()` will not return unless `spinlock` previously contains value 0 (which means unlocked). In that case it will atomically stores value 1 (which means locked) to `spinlock` and prevents other `lock()` calls from returning, hence ensuring mutual exclusion. While it spin-waits, it simply swaps `spinlock`'s old value 1 with 1, effectly leaving the lock untouched. Please take a moment to appreciate how this simple construct correctly implements mutual exclusion.

x86-64 provides this atomic swap functionality via the `lock xchg` assembly instruction. We have shown that it is all we need to implement a mutex with just one bit. x86-64 provides a more powerful atomic instruction that further opens the possibility of what can be done atomically in modern processors. The instruction is called a *compare-exchange*, or `lock cmpxchg`. It is powerful enough to implement atomic swap, add, subtract, multiplication, square root, and many other things you can think of.

The behavior of the instruction is defined as follows:

```
// Everything in one atomic step
int compare_exchange(int* object, int expected, int desired) {
    if (*object == expected) {
        *object = desired;
        return expected;
    } else {
        return *object;
    }
}
```

This instruction is also accessible as the `this->compare_exchange_strong()` member method for C++ `std::atomic` type objects. Instead of returning an old value, it returns a boolean indicating whether the exchange was successful.

For example, we can use the compare-exchange instruction to atomically add 7 to any integer:

```
void add7(int* x) {
    int expected = *x;
    while (compare_exchange(x, expected, expected + 7)
            != expected) {
        expected = *x;
    }
}
```

Synchronization 4: Networking and Synchronization

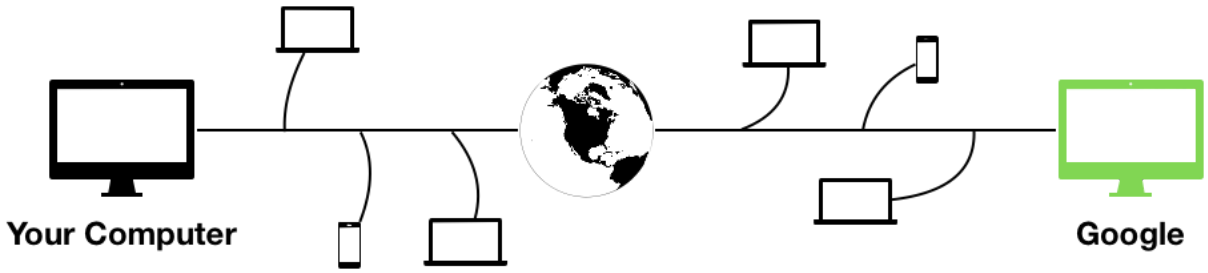
Networking

We all use Internet, a type of networking in modern computer systems.

Networking is the OS abstraction for computers to communicate with one another.

It's difficult to think about computer networks without thinking about the underlying infrastructure powering them. In the old days of telephony networks, engineers and telephone operators relied on *circuit switching* to manage connections for telephone calls, meaning that each telephone connection occupies a physical dedicated phone line. Circuit switching was widely used over a long period of time, even during early days of modern digital computing. Circuit switching significantly underutilized resources in that an idle connection (e.g. periods in a phone conversation when neither party was actively saying anything) must also keep a phone line occupied. Extremely complex circuit switching systems were built as telephone systems expanded, but circuit switching itself is inherently not scalable.

Modern computer networks use *packet switching* such that computers do not rely on dedicated direct connections to communicate. The physical connections between computers are shared, and the network carries individual packets, instead of full connections. The concept of a connection now becomes an abstraction, implemented by layers of software protocols responsible for transmitting/processing packets and presented to the application software as a stream connection by the operating system. Instead of having direct, dedicated connections to computers over the network you want to talk to, modern computer networks look more like the following, where a lot of the physical infrastructure is widely shared among tens of millions of other computers making up the Internet:



Thanks to packet switching and the extensive sharing of the physical infrastructure it enables, Internet becomes cheap enough to be ubiquitous in our lives today.

Packets

A packet is a unit of data sent or received over the network. Computers communicate to one another over the network by sending and receiving packets. Packets have a maximum size, so if a computer wants to send data that does not fit in a single packet, it will have to split the data to multiple packets and send them separately. Each packet contains:

- Addresses (source and destination)
- Checksum
  - to detect data corruption during transmission
- Ports (source and destination)
  - to distinguish logical connections to the same machines
- Actual payload data

Networking system calls

A networking program uses a set of system calls to send and receive information over the network. The first and foremost system call is called `socket()`.

- `socket()`: Analogous to `pipe()`, it creates a networking socket and returns a file descriptor to the socket.

The returned file descriptor is *non-connected* -- it has just been initialized but it is neither connected to the network nor backed up by any files or pipes. You can think of `socket()` as merely reserving kernel state for a future network connection.

Recall how we connect two processes using pipes. There is a parent process which is responsible for setting everything up (calling `pipe()`, `fork()`, `dup2()`, `close()`, etc.) before the child process gets to run a new program by calling `execvp()`. This approach clearly doesn't work here, because there is no equivalent of such "parent process" when it comes to completely different computers trying to communicate with one another. Therefore a connection must be set up using a different process with different abstractions.

In network connections, we introduce another pair of abstractions: a client and a server.

- The client is the *active endpoint* of a connection: It actively creates a connection to a specified server. **Example:** When you visit `google.com`, your browser functions as a client. It knows which server to connect to!
- The server is the *passive endpoint* of a connection: It waits to accept connections from what are usually unspecified clients. **Example:** `google.com` servers serve all clients visiting Google.

Client- and server-sides use different networking system calls.

Client-side system call -- `connect`

- `connect(fd, addr, len) -> int`: Establish a connection.
  - `fd`: socket file descriptor returned by `socket()`
  - `addr` and `len`: C struct containing server address information (including port) and length of the struct
  - Returns 0 on success and a negative value on failure.

Server-side system calls

On the server side things get a bit more complicated. There are 3 system calls:

- `bind(fd, ...) -> int`: Picks a port and associate it with the socket `fd`.
- `listen(fd) -> int`: Set the state of socket `fd` to indicate that it can accept incoming connections.
- `accept(fd) -> cfd`: Wait for a client connection, and returns a new socket file descriptor `cfd` after establishing a new incoming connection from the client. `cfd` correspond to the active connection with the client.

The server is not ready to accept incoming connections until after calling `listen()`. It means that before the server calls `listen()` all incoming connection requests from the clients will fail.

Among all these system calls mentioned above, only `connect()` and `accept()` involves actual communication over the network, all other calls simply manipulate local state. So only `connect()` and `accept()` system calls can block.

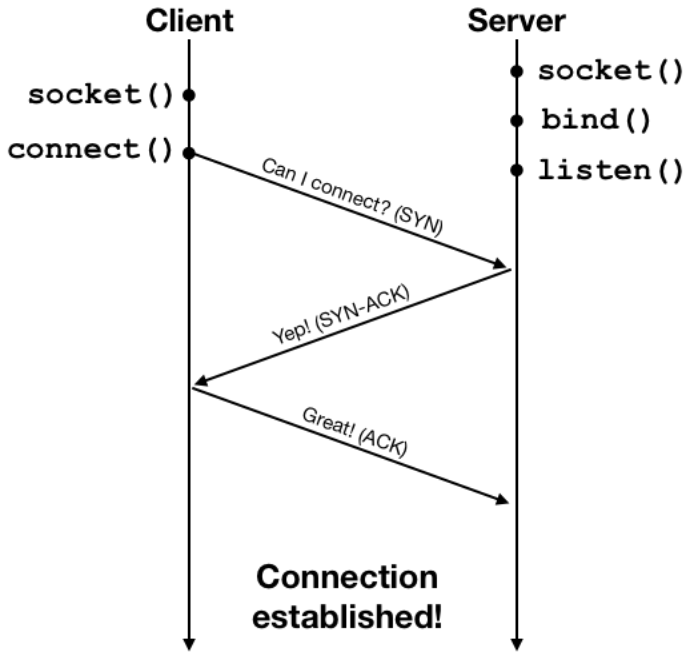
> One interesting distinction between pipes are sockets is that pipes are one way, but sockets are two-way: one can only read from the read end of the pipe and write to the write end of the pipe, but one are free to both read and write from a socket. Unlike regular file descriptors for files opened in Read/Write mode, writing to a socket sends data to the network, and reading from the same socket will receive data from the network. Sockets hence represents a two-way connection between the client and the server, they only need to establish one connect to communicate back and forth.

Connections

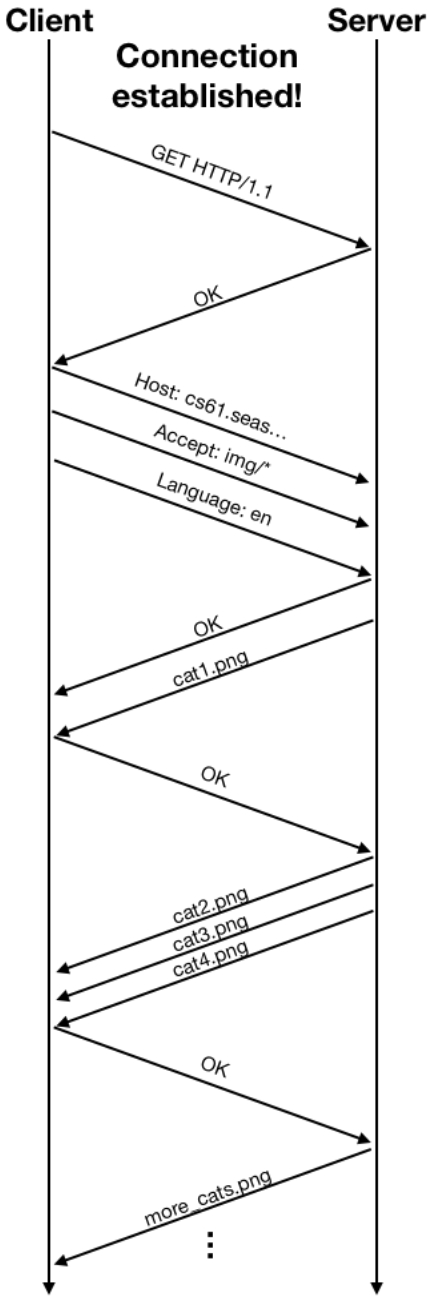
A connection is an abstraction built on top of raw network packets. It presents an illusion of a reliable data stream between two endpoints of the connection. Connections are set up in phases, again by sending packets.

> Here we are describing the Transmission Control Protocol (TCP). There are other networking protocols that do not use the notion of a connection and deals with packets directly. Google "User Datagram Protocol" or simply "UDP" for more information.

A connection is established by what is known as a *three-way handshake* process. The client initiates the connection request using a network packet, and then the server and the client exchange one round of acknowledgment packets to establish the connection. This process is illustrated below.



Once the connection is established, the client and the server can exchange data using the connection. The connection provides an abstraction of a reliable data stream, but at a lower level data are still sent in packets. The networking protocol also performs *congestion control*: the client would send some data, wait for an acknowledgment from the server, and then send more data, and wait for another acknowledgment. The acknowledgment packets are used by the protocol as indicators of the condition of the network. The the network suffers from high packet loss rate or high delay due to heavy traffic, the protocol will lower the rate at which data are sent to alleviate congestion. The following diagram shows an example of the packet exchange between the client and the server using HTTP over an established connection.



WeensyDB

We now look at a simple network database, WeensyDB, to see how networking and synchronization are integrated in the same program. A database is a program that stores data. In our case the WeensyDB application has a server side and a client side. The server side is where data is actually stored, and the client simply queries the database over the network and retrieves data from the server.

Version 1: Single-threaded server

The first version of the server-side database is in `synch4/weensydb-01.cc`. We use a simple hash table to represent our database. The hash table has 1024 buckets, and each bucket is a list of elements to handle collisions.

The `handle_connection()` function performs most of the server-side logic given a established connection from a client. It reads from the network and parses the command sent by the client. It then processes the command by accessing the database (hash table), and then writes the result of the command back to the network connection so the client can receive it.

We can run this database application by running both the server and the client. Note that merely starting the server does not generate any packets over the network -- it's a purely local operation that simply prepares the listening socket and put it into the appropriate state. After we start the client and type in a command, we do see packets being sent over the network, as a result an active connection being established and data being exchanged.

> The example we showed in the lecture had both the client and the server running on the same computer. The packets were exchanged through the local loop-back network interface and nothing gets actually sent over the wire (or WiFi). For understanding you can just imagine that the two programs are running on distinct machines. In fact the two programs do not use anything other than the network to communicate, so it is very much like they operate from different physical machines.

The database server should serve any many clients as possible, without one client being able to interfere with other clients' connections. This is like the process isolation properties provided by the OS kernel. Our first version of WeensyDB doesn't actually provide this property. As we see in its implementation -- it is a single-threaded server. It simply can't handle two client connections concurrently, the clients must wait to be served one by one.

This opens up door for possible attacks. A malicious client who never closes its connection to the server will block all other clients from making progress. `synch4/wdbclientloop.cc` contains such a bad behaving client, and when we use it with our first version of WeensyDB we observe this effect.

Version 2: Handle connection in a new thread

The next version of the WeensyDB server tries to solve this problem using multi-threading. The program is in `synch4/weensydb-02.cc`. It's very similar to the previous version, except that it handles a client connection in a new thread. Relevant code is shown below:

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

    while (true) {
        // Accept connection on listening socket
        int cfd = accept(fd, nullptr, nullptr);
        if (cfd < 0) {
            perror("accept");
            exit(1);
        }

        // Handle connection
        std::thread t(handle_connection, cfd);
        t.detach();
    }
}
```

This code no longer blocks the main thread while a client connection is being handled, so concurrent client connections can proceed in parallel. Great! Does that really work though?

Now look at how we actually handle the client connection, in the `while` loop `handle_connection()` function:

```
void handle_connection(int cfd) {
    ...
    while (fgets(buf, BUFSIZ, fin)) {
        if ... {
            ...
        } else if (sscanf(buf, "set %s %zu ", key, &sz) == 2) {
            // find item; insert if missing
            auto b = string_hash(key) % NBUCKETS;
            auto it = hfind(hash[b], key);
            if (it == hash[b].end()) {
                it = hash[b].insert(it, hash_item(key));
            }

            // set value
            it->value = std::string(sz, '\0');
            fread(it->value.data(), 1, sz, fin);
            fprintf(f, "STORED %p\r\n", &*it);
            fflush(f);
        } else if ...
    }
    ...
}
```

We see that while handling a `set` operation, we modify the underlying hash table, which is shared among all threads created by parallel client connections. And these modifications are not synchronized at all! Indeed if we turn on thread sanitizer we see a lot of complaints indicating serious race conditions in our code when clients issue `set` commands in parallel.

We have been through this before when dealing with bounded buffers, so we know we need to fix this using a mutex. **But** we can't just simply protect the *entire* `while` loop in question using a scoped lock, as we did for the bounded buffer. If we did that, the program would be properly synchronized, but it loses parallelism -- the mutex ensures that only one connection can be handled at a time, and we fall back to the single-threaded version which is subject to attack by the bad client.

The key to using mutex in a networked program is to realize the danger of holding the mutex while blocking on network system calls. Blocking can happen when sending/receiving data on the network, so during network communications the mutex must never be locked by the server thread.

One way to re-organize the program to avoid these issues is to make sure the mutex is only locked after a complete request has been received from the client, and is unlocked before sending the response. In the `weensydb-02.cc` code it is equivalent to put one scoped lock within each `if/else if/else` block in the `while` loop in `handle_connection()`.

Version 3: Fine-grained locking

By this point we are still doing coarse-grained locking in that the entire hash table is protected by one mutex. In a hash table, it's natural to assign a mutex to each hash table *bucket* to achieve fine-grained locking, because different hash table buckets are guaranteed to contain different keys. `synch4/weensydb-05.cc` implements this

version.

Version 4: Exchange

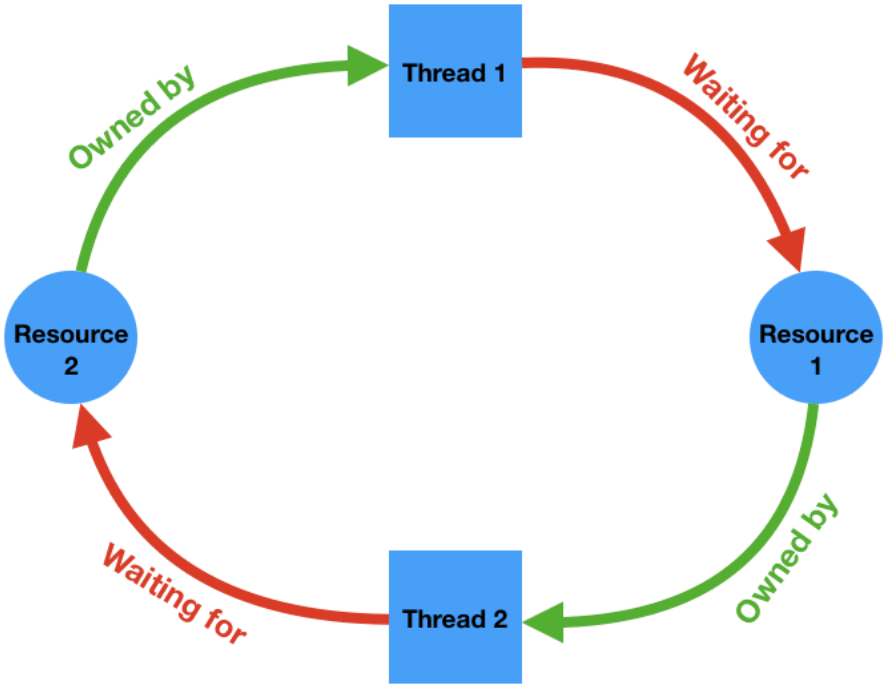
Now consider we are to add a new feature to the database: an `exchange` command that swaps the values of two keys. We still follow the previous fine- grained locking design, but this time we will need to lock up to 2 buckets for every operations. When we lock multiple mutexes in a program we should always be wary of the possibility of deadlocks. We will have more discussions about deadlocks the next time!

Synchronization 5: Deadlock and Server Programming

Deadlock

Deadlock is a situation in a multi-threaded process where each thread is waiting (forever) for a resource that is held by another thread.

It is necessary to have a cycle in a resource ownership-wait diagram to have a deadlock. The following diagram illustrates a deadlock where two threads are waiting for some resources held by each other and are therefore unable to ever make progress.



Cycles like this can potentially be detected by the runtime system and it can intervene to break the deadlock, for example by forcing one of the threads to release the resources it owns. The C++ runtime does not perform these checks for performance reasons, therefore it's the programmer's responsibility to make sure the program is free of deadlocks.

There are many ways to lead to a deadlock, even with just a single mutex and a single thread. Consider the following code:

```
std::mutex m;

void f() {
    std::scoped_lock guard(m);
    g();
}

void g() {
    std::scoped_lock guard(m);
    // Do something else
}
```

This code will not make any progress because the lock acquisition `scoped_lock` in `g()` is waiting for the `scoped_lock` in `f()` to go out of scope, which it never will.

Also, the resources that are involved in a ownership-wait cycle don't have to be mutexes. Recall the extra credit in the stdio problem set (problem set 4). The two resources are two pipe buffers. Let's say the two processes are following the procedure below:

```
Process 1:
writes 3 requests to the pipe, 33000 bytes each
reads 3 responses from the pipe, 66000 bytes each

Process 2:
while (true) {
    reads a request from the pipe
    writes a response to the pipe
}
```

Process 1 should successfully write two requests to the pipe, as Process 2 consumes one request so there is space in the pipe to hold the second request. The third request write will however be blocked because there is not enough space in the pipe buffer (assuming pipe buffers are 65536 bytes). Process 2 can not consume another request because it is also blocked while writing the large response for the first request, which does not fit in the pipe buffer. A cycle occurs and no process can make progress.

WeensyDB's `exch` command

Our simple networked database, WeensyDB, from the last lecture now supports a new "exchange" (or `exch`) command, which swaps the values associated with two keys. The relevant code enabling this command is in `synch4/weensydb-06.cc#L112:L137`, which is also shown below:

```
        } else if (sscanf(buf, "exch %s %s ", key, key2) == 2) {
            // find item
            auto b1 = string_hash(key) % NBUCKETS;
            auto b2 = string_hash(key2) % NBUCKETS;
            hash_mutex[b1].lock();
            hash_mutex[b2].lock();
            auto it1 = hfind(hash[b1], key);
            auto it2 = hfind(hash[b2], key2);

            // exchange items
            if (it1 != hash[b1].end() && it2 != hash[b2].end()) {
                std::swap(it1->value, it2->value);
                fprintf(f, "EXCHANGED %p %p\r\n", &*it1, &*it2);
            } else {
                fprintf(f, "NOT_FOUND\r\n");
            }
            fflush(f);

            hash_mutex[b1].unlock();
            hash_mutex[b2].unlock();

        } else if (remove_trailing_whitespace(buf)) {
            fprintf(f, "ERROR\r\n");
            fflush(f);
        }
    }
```

The two lines

```
        hash_mutex[b1].lock();
        hash_mutex[b2].lock();
```

smells deadlock. This code deadlocks if the two keys simply hash to the same bucket.

Changes were made in `synch4/weensydb-07.cc` to address this issue. The two lines above are now replaced by the following (albeit slightly more complex) code:

```
        hash_mutex[b1].lock();
        if (b1 != b2) {
            hash_mutex[b2].lock();
        }
```

There is still risk of deadlock as we are not locking the mutexes in a fixed global order. In many cases we can handily represent this global order by something we already keep track of in our code. In this example the bucket numbers `b1` and `b2` are natural indicators of such a global order. Another commonly used order is based on the memory address values of the mutex objects themselves. `synch4/weensydb-08.cc` shows a locking order based on hash bucket numbers:

```
        if (b1 > b2) {
            hash_mutex[b2].lock();
            hash_mutex[b1].lock();
        } else if (b1 == b2) {
            hash_mutex[b1].lock();
        } else {
            hash_mutex[b1].lock();
            hash_mutex[b2].lock();
        }
```

Servers and the "C10K" problem

"C10K" stands for handling 10,000 connections at a time. Back at the time when the Internet just came into being this is considered a "large-scale" problem. In today's scale the problem is more like "C10M" or "C10B" -- building servers or server architectures that can handle tens of millions (or even billions) of connections at the same time.

Early server software spawns a thread for each incoming connection. This means to handle 10,000 connections at once, the server must run 10,000 threads at the same time. Every thread has its own set of registers (stored in the kernel) and its own stack (takes up user-space memory). Each thread also has a control block (called a `task` struct in Linux) managed by the kernel. These resources add up to some very significant per-thread space overhead:

- Socket buffers: ~ 32KB each
- Stack: ~ 8MB
- Task struct + kernel thread stack: ~ 8KB

10,000 thread stacks alone add up to ~ 80GB of memory! It is clearly not scalable. **Is it possible to handle 10,000 connections with just one thread?**

It is possible to handle 10,000 *mostly idle* connections with just one thread. What we need to make sure is that whenever a message comes through in one of these connections, it is timely handled.

Conventional system calls like `read()` and `write()` block, so they prevent us from using a single thread to handle concurrent connections, because our single thread will just be blocked on an idle connection. The solution to this problem is to not use blocking: we need non-blocking I/O!

Non-blocking I/O: Handle more than 1 connection per thread

`read()` and `write()` system calls can in fact be configured to operate in non-blocking mode. Where a normal `read()` or `write()` call would block, the non-blocking version would simply return -1 with `errno` set to `EAGAIN`. Let's see how we can use non-blocking I/O to implement a beefy thread that can handle 10,000 connections at once:

```
int cfd[10000]; // 10K socket fds representing 10K connections

int main(...) {
    ...
    while (true) {
        for (int c : cfd) {
            int r = read(c);
            if (r < 0)
                continue;

            // handle message
            ...

            r = write(c);
            if (r < 0) {
                // queue/buffer response
                continue;
            }
        }
    }
}
```

This style of programming using non-blocking I/O is also called *event-driven* programming. Of course, in order to handle this many connections within one thread, we will have to maintain more state in the user level to do bookkeeping for each connection. For example, when we read from a connection and it didn't return a message, we need to somehow mark that we are still awaiting a message from this connection. Likewise, when a `write()` system calls fails due to `EAGAIN` while sending a response, the server will need to temporarily store the response message somewhere in memory and queue it for delivery when the network becomes available. These states are not required when we use blocking I/O because the server processes everything in order. A non-blocking server can essentially handle connections out-of-order within a single thread. Nevertheless these user-level overhead is much lower than the overhead of maintaining thousands of threads.

The server program described above has one significant problem: it does polling. Polling means high CPU utilization, which means the server runs hot and consumes a lot of power! It would be ideal if we can block when there is really no messages to be processes, but wake up as soon as something becomes available. The `select()` system call allows us to do exactly that.

`select()`: Block on multiple fds to avoid polling

Recall the signature of the `select()` system call:

```
select(nfds, rfds, wfds, xfds, timeout)
            ^ read set
                  ^ write set
                        ^ exception set
```

The `select()` system call blocks until either the following occurs:

- The timeout elapses
- At least one of the fds in the read set `rfds` becomes readable (including `EOF`)

We can solve the problem of polling by calling `select()` after the inner `for` loop in the server program above:

```
int cfd[10000]; // 10K socket fds representing 10K connections

int main(...) {
    ...
    while (true) {
        for (int c : cfd) {
            ...
        }
        select(...);
    }
}
```

We need to add all fds in the `cfd[]` array to the read set passed to `select()`. Now it should solve our problem!

But there is still a lot of overhead in this program. Imagine among the 10,000 connections that are established, most of them are idle and we only have one connection that are active at a time. In order to find that one active connection, the `for` loop has to search through the entire array. And that's just the overhead in the user space. The `select()` system call itself has O(N) performance, where N is the number of fds in its read set. For these reasons, this code still results in high CPU usage even most connections are idle. It would be nice if we don't have to do any of this extra work when we just have a single interesting connection to handle.

Our problem here is that `select()` does work for each connection in its read set on every call. We can solve this problem by maintaining a persistent read set for connections in the kernel, and a blocking system call that returns only *active* connections when waking up. This is in comparison to the interface of `select()`, where the read set is passed in as a parameter, and system call API itself is stateless. This API design is the reason why `select()` must do O(N) work.

The Linux version of this system call is called `epoll`.

`epoll`: Maintaining a persistent read set in the kernel

There are three system calls in the `epoll` system call family:

- `epoll_create()`: Returns a new `epoll` file descriptor representing the persistent read set in kernel.
- `epoll_ctl()`: Adds/Removes fds to/from the read set.
- `epoll_wait()`: The blocking system call. Returns information about active fds in the read set.

> This is an example of improved performance by having an alternative design of the interface. By moving the read set to be persistently maintained in the kernel, the interface looks drastically different from `select` albeit achieving the same functionalities. This enables much better per-call performance when the number of active fds is small but the read set itself is large. The `nginx` web server makes extensive use of the `epoll` system calls to handle multiple connections per thread.

Here are some latency numbers on `select()` vs. `epoll_wait()`, based on 100K calls to a server (in microseconds, lowers are better):

number of fds	`select()`	`epoll_wait()`
10	0.73	0.41
100	3.0	0.42
10000	930	0.66

This O(N) vs. O(1) difference is achieved by an interface change. Identifying issues in interface designs and addressing them by proposing alternative interfaces is one of the goals of systems research.

Where to go next...

Congratulations! You have reached the end of the last lecture! If you become interested in topics introduced in this class and wonder where to go next, here are some suggestions.

If you are hooked with kernels and want to build a full-blown OS with multi- processor support and its own file systems, **take CS161 next term with Eddie Kohler and James Mickens**.

If you want to hear more about server-less computing, you can take **CS260r by Eddie**.

If you find the assembly unit most interesting to you in this class, you might want to check out the Computer Architecture course **CS141 by David Brooks**, where you will learn at a low and deep level of how modern processors work.

If you want to learn about cloud computing, take **CS145 with Minlan Yu**.

If you want to have the best teacher and learn about programming languages, go check out **CS152/CS252 by Steve Chong**.

Database classes like **CS165/CS265 by Stratos Idreos** are also great continuations of CS61, where you will see how topics you learned in this class are really important in designing efficient software.

Thank you all for your effort in this class, and wish everyone a good year!