Section 1

Outline

- Review
  - Segments
  - Size and Alignment
  - (Basic) Pointer Arithmetic
  - Memory Bugs
- C++ Patterns
  - STL Containers
  - The `system_allocator<>` class in PSet 1
- Programming Exercises
  - Greetings!
  - Image Inverter

Review

Segments

Place all variables in question into the correct sections of memory and identify the scope of each variable.

```c
#include <cstdlib>
#include <cstdio>

#define MY_VALUE 10
int g = 45;                                     // Q#1

const char* a_string = "A fixed string";        // Q#2

void add_helper(int* c, int* d, int* e) {       // Q#3
    *c = *d + *e;                               // Q#4
}

int add(int a, int b) {
    int *answer = (int*) malloc(sizeof(int));  // Q#5
    add_helper(answer, &a, &b);
    int ans = *answer;                          // Q#6
    free(answer);
    return ans;
}

int main() {
    printf("%d\n", add(MY_VALUE, 6));
    return 0;
}
```

What is the construct called and where does it live in memory:

1. `g`
2. `a_string`
3. `add_helper`
4. `c`, `d`, and `e`
5. `*answer`
6. `ans`

High Addresses **Stack** **Heap** **Data (read/write)** **Data (read)** **Text**

Size and Alignment

Write down the size and the alignment of the following `struct`s. Hint: draw out how an instance of each `struct` would look in memory.

```c
struct struct1 {
    int i;
};
```

- Alignment:

- Size:

```
struct struct2 {
    short a;
    short b;
    short c;
};
```

- Alignment:
- Size:

```
struct struct3 {
    char a;
    int b;
};
```

- Alignment:
- Size:

```
struct struct4 {
    int b;
    char a;
};
```

- Alignment:
- Size:

```
struct struct5 {
    char a;
    char b;
    int c;
};
```

- Alignment:
- Size:

```
struct struct6 {
    char a;
    struct {
        char b;
        int c;
    } s;
};
```

- Alignment:
- Size:

```
union union1 {
    char a;
    int b;
};
```

- Alignment:
- Size:

```
struct struct7 {
    char a;
    union {
        char b;
        int c;
    } u;
};
```

- Alignment:
- Size:

Pointer Arithmetic

Given the following definitions what does each function do, and what do they return? (Assume the arguments are cast to the right types.)

```
int sum(int a, int b){
    return a + b;
}

sum(8, 6) = ?
```

```
char* sum(char* a, int b) {
    return &a[b];
}

char s[20];
sum(s + 8, 6) = ?
```

```
int* sum(int* a, int b) {
    return &a[b];
}

int x[1000];
sum(&x[8], 6) = ?
```

```
ptrdiff_t sub(int* a, int* b) {
    return a - b;
}

int x[1000];
sub(&x[996], &x[988]) = ?
```

Pointer Equivalence

You can think of comparisons in three ways: data/value, point to the same memory, are the same memory.

If we have:

```
int c, d;
c = 10;
d = 10;
```

clearly `(c == d)` because we are comparing values. If we then say

```
int* e = &c;
int* f = &d;
```

then `(*e == *f)`, since `(10 == 10)`, but `(e != f)`. Why?

Pointer comparison involves comparing the underlying addresses. Since the pointers point to different objects —`e` points to `c`, and `f` points to `d`—and C/C++ guarantees that different objects have distinct, non-overlapping addresses (except in a union), it follows that `(e != f)`.

Equivalence Exercises

```
int a = 5;
int b = 5;
int* x = &a;  // x==0xf4dc
int* y = &b;  // y==0xf4d8
int* z = x;
int** p = &z; // p==0xf4d0
int array[10] = { 5, 0 ... 0 };
int yarra[6] = { 1, 2, 3, 4, 5, 6 };
int* w = array + 4;
int* group[3];
group[0] = array;
group[1] = yarra;
group[2] = y;
```

True of false?

1. `(a == b)`

2. `(x == y)`
3. `(y == &a)`
4. `(*z == a)`
5. `(group[0][0] == a)`
6. `(group[1][3] == 1)`
7. `(group[1][3] == 4)`
8. `(group[2][0] == 5)`
9. `(group[0][0] == *group[0])`
10. `(group[0][4] == *(group[0] + 4))`
11. `(group[0][4] == *group[0] + 4)`
12. `(group[0][4] == (*group)[4])`
13. `((*group)[0] == (*(group+1))[4])`
14. `(*w == 4)`
15. `(*w == 0)`
16. `(w == 0)`

Symbol	Type	Value
`z`		
`p-2`		
`&z`		
`yarra[3]`		
`&b`		
`group[2][0]`		
`array[6]`		
`*p`		
`**(x-3)`		
`**(&group[1]-1)`		
`*w`		
`group+2`		
`*(group+3)`		

Memory Bugs

Go to `cs61-sections/s01`. For each of the `membug*.cc` files, describe the memory bugs present in each file. Some example bugs include “invalid free”, “invalid free: pointer not on heap”, “double free”, “use of uninitialized pointer.” Before running, predict the effect of the memory bug -- will the program crash? Or will it fail silently? Run `make`, then `./membug1`, `./membug2`, etc. to observe the effects of each bug.

- `membug1.cc`:
- `membug2.cc`:

- `membug3.cc`:
- `membug4.cc`:
- `membug5.cc`:
- `membug6.cc`:
- `membug7.cc`:
- `membug8.cc`:

C++ Patterns

`new` and `delete` vs. `new ...[]` and `delete ...[]`

C++ uses four operators for allocating and deleting memory. They are:

1. `new T` — dynamically allocates a single new object of type `T`.
2. `delete ptr` — frees the single object `ptr` previously allocated by `new`.
3. `new T[N]` — dynamically allocates an array of `N` objects of type `T`.
4. `delete[] ptr` — frees the array `ptr` previously allocated by `new ...[]`.

Note that you're supposed to call the right one. If you allocate an array, use `delete[]` to free it. (This differs from C.)

Advanced `new` syntax can *initialize* an object when it is allocated. For instance, `new int{3}` (or `new int(3)`) returns a pointer to a dynamically-allocated `int` initially set to 3.

Standard Template Library (STL) Containers

STL is just a library full of useful stuff! It comes with a collection of containers, or data structures, that you may notice in the handout code. You may also want to use these data structures yourself.

Vector (Growable Array)

```cpp
#include <vector>

// A vector of integers
std::vector<int> my_vec = { 1, 2, 3, 4 };

int vec_2 = my_vec[2]; // Reading from vector
my_vec[3] = 4;         // Writing to vector
// The vector `[]` operator is like array dereference: the caller must
// check bounds. But there's another call that always checks bounds.
vec_2 = my_vec.at(2);
my_vec.at(3) = 4;

my_vec.push_back(5);   // Adding element to the end of the vector
my_vec.back();         // Return the last element of the vector (must not be empty)
my_vec.pop_back();     // Remove element at the end
my_vec.size();         // Return length of vector
my_vec.empty();        // Return true iff `size() == 0`
```

Iterators

STL containers and algorithms rely on an abstraction called the *iterator*. An iterator is like a "smart pointer" into a data structure. It indicates a current position in the container. In a vector, iterators are like pointers into arrays.

The most important iterator methods are `container.begin()`, which returns an iterator that points to the "beginning" of the container (in a vector, this is the first element), and `container.end()`, which returns an iterator that points to the "end" of the container and also represents absent elements (in a vector, this points one past the last element).

These codes behave the same:

```cpp
int my_array[4] = { 1, 2, 3, 4 };
// first iterate using an index
for (int i = 0; i != 4; ++i) {
    printf("%d\n", my_array[i]);
}

// that's equivalent to iterate using a pointer into the array,
// thanks to pointer arithmetic!!!!!!!!!!
for (int* a = my_array; a != &my_array[4] /* one past end */; ++a) {
    printf("%d\n", *a);
}

// the C++ vector version looks like the "pointer arithmetic"
// version, and can be just as fast, but safer.
std::vector<int> my_vec = { 1, 2, 3, 4 };
for (auto it = my_vec.begin(); it != my_vec.end(); ++it) {
    printf("%d\n", *it);
}
```

C++'s "for-each" loops also use iterators behind the scenes.

```cpp
for (auto& a : my_vec) {
    printf("%d\n", a);
}
```

Vector Exercises

Fill in the ?s!

```cpp
std::vector<int> my_vec = { 1, 2, 3, 4, 5 };

my_vec.size() == ?
my_vec[3] == ?

my_vec.push_back(6);

my_vec.size() == ?
my_vec.back() == ?

my_vec.erase(my_vec.begin(), my_vec.begin() + 1);

my_vec.size() == ?
my_vec.back() == ?
my_vec[0] == ?
```

Can you do this?

```cpp
std::vector<int> my_vec = { 1, 2, 3 };

my_vec[4] = 1;
```

What about this?

```cpp
std::vector<int> my_vec = { 1, 2, 3 };

printf("%d\n", my_vec.at(4));
```

Map (Ordered Search Tree)

Comparison for key type is required.

```cpp
#include <map>
#include <string>

std::map<int, std::string> my_map;
my_map[1] = "one";                      // Insert into map (with overwrite semantics)
std::string s = my_map[1];              // Map lookup (inserts default if not found)

// test if key is in map
size_t exists = my_map.count(2);        // 0 if not in map, 1 if in map

// Insert without overwriting (leaves map unchanged if key present)
my_map.insert({1, "ONE"});

// Remove key
my_map.erase(1);

// Number of keys in map
size_t x = my_map.size();
```

Maps can find keys quickly using binary search tree algorithms. The `find` method returns an iterator.

```cpp
auto it = my_map.find(2);
if (it != my_map.end()) {
    // Then `2` is present in the map as a key.
    assert(it->first == 2);
    // And we can access the value.
    printf("%s\n", it->second.c_str());
} else {
    // it == my_map.end() -- key is not found
}
```

As usual `begin()` and `end()` can iterate over *all* the elements of the map.

Map Exercises

Fill in the ?s!

```
std::map<int, int> my_map;

my_map.insert({1, 2});
my_map[1] == ?
my_map.count(1) == ?
my_map.count(2) == ?
my_map.size() == ?

int x = my_map[2];
x == ?
my_map.size() == ?

my_map.insert({1, 3});
my_map[1] == ?

my_map.erase(1);
my_map.size() == ?
my_map.insert({1, 3});
my_map[1] == ?
```

What is printed?

```
std::vector<int> my_vec = { 9, 1, 4, 5, 8, 2, 3, 6, 7 };
std::map<int, int> my_map;
for (auto it = my_vec.begin(); it != my_vec.end(); ++it) {
    my_map.insert({*it, *it});
}
for (auto it = my_map.begin(); it != my_map.end(); ++it) {
    printf("my_map[%d] = %d\n", it->first, it->second);
}
```

Unordered Map (Hash Table)

Basically the same syntax as `std::map`, except that a hash function and equality checker for the key type are required.

If you see a horrible error like

```
error: static_assert failed "the specified hash does not meet the Hash requirements"
```

or (EUUUUUGGGGGHHHHHHHHH)

```
/usr/include/c++/7/bits/hashtable_policy.h: In instantiation of ‘struct
std::__detail::__is_noexcept_hash<std::pair<int, int>, std::hash<std::pair<int, int> >
>’:
/usr/include/c++/7/type_traits:143:12:   required from ‘struct
std::__and_<std::__is_fast_hash<std::hash<std::pair<int, int> > >,
std::__detail::__is_noexcept_hash<std::pair<int, int>, std::hash<std::pair<int, int> > >
>’
/usr/include/c++/7/type_traits:154:31:   required from ‘struct
std::__not_<std::__and_<std::__is_fast_hash<std::hash<std::pair<int, int> > >,
std::__detail::__is_noexcept_hash<std::pair<int, int>, std::hash<std::pair<int, int> > >
> >’
/usr/include/c++/7/bits/unordered_map.h:103:66:   required from ‘class
std::unordered_map<std::pair<int, int>, int>’
membug8.cc:4:54:   required from here
/usr/include/c++/7/bits/hashtable_policy.h:87:34: error: no match for call to ‘(const
std::hash<std::pair<int, int> >) (const std::pair<int, int>&)’
```

that means that the key type has no hash function yet. You can use a `std::map` (which doesn't require a hash function, but features slower lookup), or write a hash function:

```
namespace std {
     template <> struct hash<MY_TYPE> {
          size_t operator()(const MY_TYPE& x) const {
               return ... whatever ...;
          }
     };
}
```

Here is a pretty good hash function for pairs (based on the one in Boost). Do not ask your TF about it during section but feel free to ask us offline :)

```
namespace std {
    template <typename T, typename U> struct hash<pair<T, U>> {
        size_t operator()(const pair<T, U>& x) const {
            size_t h1 = std::hash<T>{}(x.first);
            size_t h2 = std::hash<U>{}(x.second);
            size_t k = 0xC6A4A7935BD1E995UL;
            h2 = ((h2 * k) >> 47) * k;
            return (h1 ^ h2) * k;
        }
    };
}
```

And here is a garbage hash function. You could use it as a placeholder if you like making your computer work hard.

```
namespace std {
    template <> struct hash<MY_TYPE> {
        size_t operator()(const MY_TYPE& x) const {
            return 0;
        }
    };
}
```

Q: How would you use a pair of a string and a number (`std::pair<std::string, int>`) as the key type in unordered map, without writing your own hash function?

Understanding `system_allocator<>` in Pset 1

There is a class called `system_allocator` in problem set 1's handout code (in `m61.hh`):

```
/// This magic class lets standard C++ containers use the system allocator,
/// instead of the debugging allocator.
template <typename T>
class system_allocator {
public:
    typedef T value_type;
    system_allocator() noexcept = default;
    template <typename U> system_allocator(system_allocator<U>&) noexcept {}

    T* allocate(size_t n) {
        return reinterpret_cast<T*>((malloc)(n * sizeof(T)));
    }
    void deallocate(T* ptr, size_t) {
        (free)(ptr);
    }
};
```

According to the comments, this code lets C++ STL containers use the system allocator instead of the debugging allocator. How does it work? Let's take a closer look.

This class defines two methods, `allocate()` and `deallocate()`. Within these method, `malloc` and `free` are invoked. The parentheses around `malloc` and `free` ensure that these are the *system* `malloc` and `free`, not the (possibly-macro-defined) `m61_malloc` and `m61_free`.

Now look at `basealloc.cc` within the pset directory, where STL containers are used. Pay attention to how the containers are declared. For example, the vector:

```
static std::vector<base_allocation, system_allocator<base_allocation>> frees;
```

We pass two parameters to `std::vector`. The first one, `base_allocation`, is the element type of the vector: this is a vector of `base_allocation`s. The second one is the `system_allocator` class we just defined! This tells STL to only use the system `malloc`, rather than the m61 allocator you're writing. Nearly all STL containers take an allocator class as a parameter (which defaults to a class is called `std::allocate<>`). The container uses the `allocate()` and `deallocate()` methods defined by the allocator class to manage dynamic memory.

Q: Which allocator would STL containers use without passing in the `system_allocator` parameter?

TL;DR: This is what you should do if you choose to use STL containers *within your debugging allocator code* in PSet 1:

```
// If you want to use std::vector
std::vector<T, system_allocator<T>> my_vec;

// If you want to use std::map or std::unordered_map
// It looks a bit awful because of C++ template specialization rules
std::map<K, V, std::less<K>, system_allocator<std::pair<const K, V>>> my_map;
std::unordered_map<K, V, std::hash<K>, std::equal_to<K>,
system_allocator<std::pair<const K, V>>> my_hash;

// Even strings need an allocator! Don't use `std::string`, use
std::basic_string<char, std::char_traits<char>, system_allocator<char>> my_string;

// Or give these things shorter names!
typedef std::vector<T, system_allocator<T>> my_vec_t;
my_vec_t my_vec;
```

Additional C++ Resources

There are a lot of online resources for common C++ patterns and STL documentation. cppreference.com is a good place to look for such information.

Programming Exercises

Greetings!

Take a look at `cs61-sections/s01/greet.cc/`. It's a very short program:

```cpp
#include <cstdio>
#include <cstdlib>

void greet() {
    char buf[16];

    printf("Hello! What is your name?\n");
    scanf("%s", buf);
    printf("Nice to meet you, %s!\n", buf);
}

int main() {
    greet();
    return 0;
}
```

Try running it and see what it does.

Q: Can you crash this program, without changing the program itself?

Q: Why did it crash, and what are the implications?

Q: How can you make this program safe? Hint: `man scanf`!

Image Inverter

Goals

- Work with C++ arrays and pointers
- Understand how malformed input can crash a program
- And the security implications of the previous bullet point

Overview

In this exercise we will be implementing a program that inverts a digital image stored in PPM format.

You can learn more about PPM format here. Basically, PPM is a uncompressed image format that's easy for programs to write and read. For our purposes we assume a PPM image file always conforms to the following structure:

1. ASCII characters `"P6"`, magic value for the PPM format
2. Whitespace
3. Width of the image, as ASCII string (decimal representation)
4. Whitespace
5. Height of the image, in ASCII decimal

6. Whitespace
7. Maximum pixel value, in ASCII decimal
8. A single whitespace character
9. Pixel data

Pixel data is just a sequence of bytes. Each pixel in the image is represented by 3 bytes of data (R, G, and B), where each byte takes value 0 to the "max pixel value" defined in the file earlier.

We call the textual portion of the file (before the pixel data) the *header* of a PPM file. The following is a valid PPM header:

```
P6
100 100
255
```

Your job is to write a program that reads a PPM file, color-inverts the image, and save the inverted image to a new file.

We've provided a skeleton in `inv.cc`, but it's not complete.

Invert!

Find image.ppm in the `cs61-sections/s01` directory. You can view your image using the display command:

```
display sample.ppm      # Linux
open sample.ppm         # Mac OS X
```

You will see the following image displayed:

The inverted version of this image can be seen in `sample-inverted.ppm`:

Finish the program in `inv.cc`. If you run your program as

```
make inv && ./inv < sample.ppm > x.ppm
```

you should find that `x.ppm` is identical to `sample-inverted.ppm`.

Your program should color-invert the image pixel by pixel (red should become green, black should become white, and so on). **You can achieve this by replacing each pixel value with its difference from the maximum pixel value.**

Security

Q. Can you produce an input that causes `./inv` to crash or abort with a memory error? (Assertion failures don't count.)

Q. Update `inv.cc` so that `./inv` *never* causes a memory error, crash, or abort. (Again, assertion failures don't count.)

Section 2

In this section we're going to have fun.

Update your section repository, `cd s02`, and `make`. This will built a number of fun programs.

Setup

Let's run one:

```
$ ./fun01
😿😿😿😿😿😿😿😿 no fun 😿😿😿😿😿😿😿😿
```

That wasn't fun!

These programs are puzzles. Look at `fundriver.cc` and you'll see the ground rules. The driver's `main` function first creates a single C string that contains all program arguments, separated by spaces. It then calls the `fun` function, passing in that string. The `fun` function returns an integer; if `fun(str)` returns 0, then the driver has fun, and if it returns anything else, no fun is had (the function `boo()` is called, which prints the `no fun` message).

We want to have fun, how can we have fun? Well might as well look to see what the function is doing! (Open `fun01.cc`)

Looks like this `fun` function will return 0 if and only if the arguments contain an exclamation point. Let's test that:

```
$ ./fun01 !
🎉🎉🎉🎉🎊🎊🎊🎊🌽🌽🌽🎊🎊🎊🎊🎉🎉🎉🎉
                    FUN
🎉🎉🎉🎉🎊🎊🎊🎊🌽🌽🌽🎊🎊🎊🎊🎉🎉🎉🎉
$ ./fun01 'yay!'
🎉🎉🎉🎉🎊🎊🎊🎊🌽🌽🌽🎊🎊🎊🎊🎉🎉🎉🎉
                    FUN
🎉🎉🎉🎉🎊🎊🎊🎊🌽🌽🌽🎊🎊🎊🎊🎉🎉🎉🎉
$ ./fun01 'amazing!!!!!!!!!!!!!!!!'
🎉🎉🎉🎉🎊🎊🎊🎊🌽🌽🌽🎊🎊🎊🎊🎉🎉🎉🎉
                    FUN
🎉🎉🎉🎉🎊🎊🎊🎊🌽🌽🌽🎊🎊🎊🎊🎉🎉🎉🎉
$ ./fun01 'amazing?'
😿😿😿😿😿😿😿😿 no fun 😿😿😿😿😿😿😿😿
```

It works.

GDB

Now let's say that the idea of not having fun is so painful to you that you will do literally anything to avoid it.

Is there any way that you could, for example, prevent the `boo()` function from running? That you could stop the program if it reached `boo()`?

This calls for a *debugger breakpoint*. A debugger is a program that manages the execution of another program. It lets you run a program, stop it, and examine variables, registers, and the contents of memory. Among the most powerful debugger features is the ability to stop a program if it ever reaches an instruction. This is called "setting a breakpoint": the breakpoint marks a location that, when reached, "breaks" the program and returns control to the debugger.

How would you stop the program from executing `boo`?

```
$ gdb fun01
(gdb) b boo
```

Now if we run the program with non-fun arguments

```
(gdb) r
```

we will stop before printing "no fun"!

If you're not careful, though, it's possible to accidentally step through and print the message. You can do this one step at a time (demo `r`, followed by several `s`es); or you can do it by continuing the program by accident (demo `r` followed by `c`).

What if you wanted to make this kind of accident wicked unlikely? Well, you could set more breakpoints!

```
(gdb) b foo
(gdb) r
Breakpoint 1, main (argc=1, argv=0x7fffffffdf88) at fundriver.cc:34
34          boo();
(gdb) x/20i $pc
=> 0x400c47 <main(int, char**)+375>:  lea    0x344(%rip),%rsi        # 0x400f92
   0x400c4e <main(int, char**)+382>:  lea    0x20156b(%rip),%rdi        # 0x6021c0
<_ZSt4cerr@@GLIBCXX_3.4>
   0x400c55 <main(int, char**)+389>:  callq  0x400a70
<_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
   0x400c5a <main(int, char**)+394>:  mov    $0x1,%edi
   0x400c5f <main(int, char**)+399>:  callq  0x400a90 <exit@plt>
```

We've stopped at the first instruction in the `boo` function (which has actually been inlined into `main`, but never mind). But can also set more breakpoints! For example, at the second instruction and the third:

```
(gdb) b *0x400c4e
(gdb) b *0x400c55
```

(demo)

But what if you forgot these breakpoints??? Well, that's a good case for `.gdbinit`.

GDB usage

- `x` for examining memory
- `x/20i $pc` for examining instructions
- autodisplay instructions: `display/20i $pc`
- `-tui`
- `info registers`
- `s` vs. `n`

More fun

Now let's work through a couple more funs. We'll try to understand the operation of the funs using GDB and assembly, though for the first 6 funs, the C++ is there if you get stuck.

ASSEMBLY IS HARD. And trying to understand assembly from first principles, without running it, is **really** hard! As with many aspects of systems, you will have more luck with an approach motivated by experimental science. Try and *guess* at an input that will work, using *cues* from the assembly. Develop a hypothesis and test it. For the bomb, you don't need to fully understand the assembly, you just need to find an input that passes each phase. (That said, you will often end up understanding the assembly—but only after completing the phase with the help of experiments.)

It is also often effective to alternate between working *top down*, starting from the entry to a function, and *bottom up*, starting at the return statement. Working from the bottom up, you can eliminate error paths and trace through how the desired result is calculated. Working from the top down, you can develop hypothesis about how the input should look. As long as you have breakpoints set, you can experiment with a free and easy heart. (And if the bomb goes off, who really cares?)

Fun cheatsheet

- `fun02`: Calls `strtol`. Hypothesis: string should be an integer! Adds 1 to the return from `strtol`. Answer: `-1`
- `fun03`: Checks first character of string, then returns second. Answer: any single-character string. Introduces control flow.
- `fun04`: Any two-character string with both characters same. More control flow.
- `fun05`: Any nonzero-character string with all characters same. A loop.
- `fun06`: Any string with length a multiple of 4. For loop.
- `fun07`: Checks for primes. Work top-down: expects an integer. Test a bunch of integers at command line. Ask students to spot the pattern.
- `fun08`: Reads three integers, returns `a + b - c`. Examples: `1 1 2`, `0 0 0`.

- `fun09`: Expects a filename of an openable file. Pretty simple.
- `fun10`: A power of 2 greater than 128. Bitwise arithmetic.
- `fun11`: A six-or-more character string consisting of uppercase letters.
- `fun12`: A five-or-more character palindrome.

Other stuff

The section cheatsheet has a bunch of material on assembly that can be used for reference or presentation if the above doesn't work for you.

Section 3

System calls

System calls are special "function calls" that are handled not by the calling program, but rather by the operating system. Programs interact with other programs, and with the outside world, by making system calls. Whenever a program prints something or accepts input, a system call was involved. (A program's registers and memory are private to that program, so normal function calls and instruction executions do not interact with the outside world.)

System calls are invoked by special hardware instructions, such as the x86-64 `syscall` instruction. When the processor encounters a `syscall`, it saves the machine state and switches context to the **kernel**, which is the special operating system program that executes with all privilege, allowing it to control hardware. The kernel interprets the system call according to its rules and resumes the calling process when complete.

How to read an strace

`strace` is a Linux program that's great for debugging the system calls made by another program.

You run `strace` like this:

```
$ strace -o strace.out PROGRAMNAME ARGUMENTS...
```

This runs `PROGRAMNAME` with the given `ARGUMENTS`, but it simultaneously snoops on all the system calls `PROGRAMNAME` makes and writes a readable summary of those system calls to `strace.out`. Look at that output by running `less strace.out`.

The first thirty or more lines of an strace are boilerplate caused by program startup. It's usually pretty easy to tell where the boilerplate ends and program execution begins; for example, you might look for the first `open` calls that refer to data files, rather than program libraries.

How to read a blktrace

`blktrace` is a Linux program that helps debug the disk requests made by an operating system in the course of executing other programs. This is below the level of system calls: a single disk request might be made in response to multiple system calls, and many system calls (such as those that access the buffer cache) do not cause disk requests at all.

Install `blktrace` with `sudo yum install blktrace` (on Ubuntu), and run it like this:

```
$ df .                                  # figure out what disk you're on
Filesystem     1K-blocks     Used Available Use% Mounted on
/dev/sda1       61897344 18080492  41107316  31% /
$ sudo sh -c "blktrace /dev/sda1 &"    # puts machine-readable output in
`sda1.blktrace.*`
$ PROGRAMNAME ARGUMENTS...
$ sudo killall blktrace
=== sda1 ===
  CPU  0:                  282 events,       14 KiB data
  CPU  1:                  658 events,       31 KiB data
  Total:                   940 events (dropped 0),       45 KiB data
$ blkparse sda1 | less
```

The default output of `blkparse` has everything anyone might want to know, and is a result is quite hard to read. Try this command line to restrict the output to disk requests ("`-a issue`", which show up as lines with "`D`") and request completions ("`-a complete`" and "`C`").

```
$ blkparse -a issue -a complete -f "%5T.%9t: %2a %3d: %8NB @%9S [%C]\n" sda1 | less
```

With those arguments, `blkparse` output looks like this:

```
    0.000055499:  D   R:     16384B @ 33902656 [bash]
    0.073827694:  C   R:     16384B @ 33902656 [swapper/0]
    0.074596596:  D  RM:      4096B @ 33556536 [bash]
    0.090967020:  C  RM:      4096B @ 33556536 [swapper/0]
    0.091078346:  D  RM:      4096B @ 33556528 [bash]
    0.091270951:  C  RM:      4096B @ 33556528 [swapper/0]
    0.091432386:  D  WS:      4096B @ 24284592 [bash]
    0.091619438:  C  WS:      4096B @ 24284592 [swapper/0]
    0.100106375:  D  WS:     61440B @ 17181720 [kworker/0:1H]
    0.100526435:  C  WS:     61440B @ 17181720 [swapper/0]
    0.100583894:  D  WS:      4096B @ 17181840 [jbd2/sda1-8]
         |        |   |          |         |       |
         |        |   |          |         |       |
         |        |   |          |         |       |
         |        |   |          |         |       +--- responsible command
         |        |   |          |         |
         |        |   |          |         +----------- disk offset of read or write
         |        |   |          +--------------------- number of bytes read or written
         |        |   |
         |        |   +------------------------------- Read or Write (plus other info)
         |        +----------------------------------- D = request issued to device,
         |                                             C = request completed
         |
         +--------------------------------------------- time of request
```

Access patterns

An **access pattern** is a high-level description of a class of reference strings. That's a bit technical: at a higher level, think of an access pattern as a general flavor of file or data access, described by the kinds of addresses that are accessed.

The different `...61` programs in your problem set perform different kinds of access, and have different access patterns. Some especially imporant access patterns are as foll

Sequential access

A sequential access pattern accesses a contiguous increasing sequence of addresses with nothing skipped, like 1, 2, 3, 4, 5, 6... or 1009, 1010, 1011, 1012, 1013....

Random access

A random access pattern skips around in address space at random, like 1, 10403, 96, 2, 51934, ...

Reverse-sequential access

A reverse-sequential access pattern access a contiguous *decreasing* sequence of addresses with nothing skipped, like 60274, 60273, 60272, 60271, 60270, ...

Strided access

A strided access pattern accesses an increasing sequence of addresses, but with large uniform skips, like 1, 1001, 2001, 3001, 4001, 5001, ... The "stride" of an access pattern is the distance between adjacent accesses. Sequential access is a kind of strided access but with "stride 1."

Variants and combinations

These terms describe whole access patterns, but they are often used more loosely, to describe *parts* of a reference string. For instance, we might see a reference string that starts with a sequential region, then skips ahead by tens of thousands of bytes and has another sequential region, then does some stuff that you don't understand so you call it "random," etc.

Access size

Many caches transform access sizes, or units of access, in the process of handling an access request. Let the *requested size* be the amount of data requested from above by the cache's user, and the *implementation size* be the amount of data that the cache itself requests from its underlying storage. Then these sizes need not be the same!

For example, consider the processor cache. The processor cache mostly has requested sizes of 1, 2, 4, or 8 bytes (corresponding to `movb`, `movw`, `movl`, and `movq`); but when a processor cache experiences a miss, its implementation size is 64 or 128 bytes: it always loads 64 or 128 aligned bytes of memory from the primary memory, then answers the request using that region.

As an important side effect, access size transformations often transform a reference string effectively to have more repeated accesses. For instance, consider a sequential access pattern of bytes on the processor, starting at address 0x401001. Then the processor accesses bytes in this order:

Byte address	Cache line address (aligned by 64)
0x401001	0x401000
0x401002	0x401000
0x401003	0x401000
…	…
0x40103e	0x401000
0x40103f	0x401000
0x401040	**0x401040**
0x401041	0x401040
0x401042	0x401040
…	…

This is why single-slot caches can be effective in practice even on reference strings that don't just access the same byte over and over and over again.

Strace exercises

Test your knowledge by characterizing access patterns and cache types given some straces!

In the `s03` directory, you'll see a bunch of truncated strace output in files `straceNN.out`. For each output, characterize its **access pattern**, and describe **what kind of I/O cache the program might have implemented, if any**.

Show solution

1. Sequential, byte-at-a-time input, 4096-byte block output
2. Sequential, 4096-byte block input, 1024-byte block output
3. Sequential, 3-byte block input, 1024-byte block output
4. 4096-byte stride, direct characterwise system calls
5. Reverse-sequential, stdio (4096-byte single-block cache)
6. Reverse-sequential, bytewise (no cache)
7. Sequential, 4096-byte block input, byte-at-a-time output
8. 100-byte stride, stdio
9. Sequential, stdio (4096-byte block input and output)
10. Sequential, 4096-byte block input, 4096-byte block output (could be stdio)
11. 4096-byte stride, stdio (so it refreshes 4096 bytes on every call)

Hide solution

Hide solution

Single-slot cache

In the rest of section, we're going to work on a specific representation of a single-slot I/O cache. The purpose is to explain such caches and to get you used to thinking in terms of file offsets. Here's the representation.

```
struct io61_file {
    int fd;
    static constexpr off_t bufsize = 4096; // or whatever
    unsigned char cbuf[bufsize];
    off_t tag;      // file offset of first byte in cache (0 when file is opened)
    off_t end_tag;  // file offset one past last valid byte in cache
    off_t pos_tag;  // file offset of next char to read in cache
};
```

`pos_tag` refers to the current read position. We include it to ensure that the single-slot cache can alter the access size and respond to any requested number of bytes.

Here are some facts about this cache representation.

- `tag <= end_tag`. (This is an **invariant**: an expression that is always true for every cache. You could encode the invariant as an assertion: `assert(tag <= end_tag)`.)
- `end_tag - tag <= bufsize`.
- If `tag == end_tag`, then the cache's slot is empty.
- `tag <= pos_tag && pos_tag <= end_tag` (the `pos_tag` lies between the `tag` and `end_tag`).
- The file descriptor's current file position equals `end_tag` for read caches (and seekable files). It equals `tag` for write caches.
- The next `io61_read` or `io61_write` call should logically start at file offset `pos_tag`. That means the first byte read by `io61_read` should come from file offset `pos_tag`, and similarly for `io61_write`.

- If `i >= tag` and `i < end_tag`, then byte `cbuf[i - tag]` contains the byte at file offset `i`.

Fill the cache

Implement this function, which should fill the cache with data read from the file. You will make a `read` system call.

```
void io61_fill(io61_file* f) {
    // Fill the read cache with new data, starting from file offset `end_tag`.
    // Only called for read caches.

    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    // Reset the cache to empty.
    f->tag = f->pos_tag = f->end_tag;









    // Recheck invariants (good practice!).
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);
}
```

Show solution

```
void io61_fill(io61_file* f) {
    // Fill the read cache with new data, starting from file offset `end_tag`.
    // Only called for read caches.

    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    /* ANSWER */
    // Reset the cache to empty.
    f->tag = f->pos_tag = f->end_tag;
    // Read data.
    ssize_t n = read(f->fd, f->cbuf, f->bufsize);
    if (n >= 0) {
        f->end_tag = f->tag + n;
    }

    // Recheck invariants (good practice!).
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);
}
```

Hide solution

Hide solution

Easy read

Implement `io61_read`, **assuming that the desired data is entirely contained within the current cache slot**.

```
ssize_t io61_read(io61_file* f, char* buf, size_t sz) {
    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    // The desired data is guaranteed to lie within this cache slot.
    assert(sz <= f->bufsize && f->pos_tag + sz <= f->end_tag);




}
```

Show solution

```
ssize_t io61_read(io61_file* f, char* buf, size_t sz) {
    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    // The desired data is guaranteed to lie within this cache slot.
    assert(sz <= f->bufsize && f->pos_tag + sz <= f->end_tag);

    /* ANSWER */
    memcpy(buf, &f->cbuf[f->pos_tag - f->tag], sz);
    f->pos_tag += sz;
}
```

Hide solution

Hide solution

Full read

Implement `io61_read` **without** that assumption. You will need to call `io61_fill`, and will use a loop. You may assume that no call to `read` ever returns an error.

> Our tests do not check whether your IO61 library handles errors correctly. This means that you may assume that no `read` and `write` system call returns a permanent error. But the *best* implementations will handle errors gracefully: if a `read` system call returns a permanent error, then `io61_read` should return -1. (What to do with restartable errors is up to you, but most I/O libraries retry on encountering `EINTR`.)

```
ssize_t io61_read(io61_file* f, char* buf, size_t sz) {
    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);









}
```

Show solution

```
ssize_t io61_read(io61_file* f, char* buf, size_t sz) {
    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    /* ANSWER */
    size_t pos = 0;
    while (pos < sz) {
        if (f->pos_tag == f->end_tag) {
            io61_fill(f);
            if (f->pos_tag == f->end_tag) {
                break;
            }
        }

        // This would be faster if you used `memcpy`!
        buf[pos] = f->cbuf[f->pos_tag - f->tag];
        ++f->pos_tag;
        ++pos;
    }
    return pos;
}
```

Hide solution

Hide solution

Easy write

When reading from a cached file, the library fills the cache using system calls and empties it to the user. Writing to a cached file is the converse: the library fills the cache with user data and empties it using system calls.

For a read cache, the cache buffer region between `pos_tag` and `end_tag` contains data that has not yet been read, and the region after `end_tag` (if any) is invalid. There are several reasonable choices for write caches; in these exercises we force `pos_tag == end_tag` as an additional invariant.

Implement `io61_write`, **assuming that space for the desired data is available within the current cache slot**.

```
ssize_t io61_write(io61_file* f, const char* buf, size_t sz) {
    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    // Write cache invariant.
    assert(f->pos_tag == f->end_tag);

    // The desired data is guaranteed to fit within this cache slot.
    assert(sz <= f->bufsize && f->pos_tag + sz <= f->tag + f->bufsize);




}
```

Show solution

```
ssize_t io61_write(io61_file* f, const char* buf, size_t sz) {
    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    // Write cache invariant.
    assert(f->pos_tag == f->end_tag);

    // The desired data is guaranteed to lie within this cache slot.
    assert(sz <= f->bufsize && f->pos_tag + sz <= f->tag + f->bufsize);

    /* ANSWER */
    memcpy(&f->cbuf[f->pos_tag - f->tag], buf, sz);
    f->pos_tag += sz;
    f->end_tag += sz;
    return sz;
}
```

Hide solution

Hide solution

Flush

Implement a function that empties a write cache by flushing its data using a system call. As before, you may assume that the system call succeeds (all data is written).

```
void io61_flush(io61_file* f) {
    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    // Write cache invariant.
    assert(f->pos_tag == f->end_tag);




}
```

Show solution

```
void io61_flush(io61_file* f) {
    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    // Write cache invariant.
    assert(f->pos_tag == f->end_tag);

    /* ANSWER */
    ssize_t n = write(f->fd, f->cbuf, f->pos_tag - f->tag);
    assert((size_t) n == f->pos_tag - f->tag);
    f->tag = f->pos_tag;
}
```

Hide solution

Hide solution

Full write

Implement a function that implements a full write. You will call `io61_flush` and use a loop.

```
ssize_t io61_write(io61_file* f, const char* buf, size_t sz) {
    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    // Write cache invariant.
    assert(f->pos_tag == f->end_tag);




}
```

Show solution

```
ssize_t io61_write(io61_file* f, const char* buf, size_t sz) {
    // Check invariants.
    assert(f->tag <= f->pos_tag && f->pos_tag <= f->end_tag);
    assert(f->end_tag - f->pos_tag <= f->bufsize);

    // Write cache invariant.
    assert(f->pos_tag == f->end_tag);

    /* ANSWER */
    size_t pos = 0;
    while (pos < sz) {
        if (f->end_tag == f->tag + f->bufsize) {
            io61_flush(f);
        }

        // This would be faster if you used `memcpy`!
        f->cbuf[f->pos_tag - f->tag] = buf[pos];
        ++f->pos_tag;
        ++f->end_tag;
        ++pos;
    }
    return pos;
}
```

Hide solution

Hide solution

Seeks

Now, add seek to the mix. How should a seek be implemented for this single-slot cache?

Section 4

Process isolation and virtual memory

We don't want processes to be able to trample over the memory of other processes, as discussed in class. So, we need process isolation. Virtual memory is a mechanism for the OS to provide process isolation. Today's section will be focused on diving deeper into virtual memory.

When you print a pointer in C program, you see the pointer's virtual address in that process's address space. The actual place where the underlying bytes live is in **physical memory**, at an address only known to the OS. This address can be determined using the process's **page table** (which maps virtual addresses to physical addresses). This page table is set up by the operating system **kernel**, and interpreted by the processor hardware.

Multi-level page tables

Many processors, including x86-64, use **multi-level page tables** to translate virtual addresses to physical addresses. A multi-level page table is structured in a tree format. The root of the tree is configured via a register (`%cr3` on x86-64); the root's entries point to lower-level page tables. The virtual address being looked up provides the index of the entry to follow at each level of the pagetable. We are going to walk through how the pagetable structure can be determined from the page size and address size.

Example: x86-32

On 32-bit x86 architecture (virtual addresses are 32 bits long), each page is 4096 = 2^{12} bytes. Each physical page contains 2^{12} separate addresses, and the architecture supports up to $2^{20} = 2^{32} / 2^{12}$ virtual pages.

QUESTION: What is the size of the offset for an x86-32 address?

Show solution

> 12 bits.
>
> Hide solution

Hide solution

QUESTION: How many 32-bit addresses can fit in an x86-32 page?

Show solution

$2^{12} / 4 = 2^{10} = 1024$.

Hide solution

Hide solution

QUESTION: Assume that x86-32 used a **single-level** page table (like the page table from the Kernel 2 lecture with 6-bit addresses). Such a page table is indexed by the top portion of the address—everything but the offset. That means it must occupy enough contiguous physical memory to account for every possible index. How big would the single-level page table be?

Show solution

2^{32} addresses / 2^{12} addresses/page * 2^{2} bytes/page table entry = 2^{22} bytes of page table.

Hide solution

Hide solution

x86-32 does *not* use a single-level page table. It uses a two-level page table. The x86-32 address divides into three sections: level-2 index, level-1 index, and offset.

bits 31-22	bits 21-12	bits 11-0
Level 2 index	Level 1 index	Offset

The address of the level-2 page table is stored in the x86-32 `%cr3` register. This page table is indexed by level 2 index to find the *physical* address of a level-1 page table; there can be many of these, one per level-2 index. The level-1 page table is indexed by level 1 index to find the physical address of the final physical page.

QUESTION: Why is 10 bits per level the correct number?

Show solution

10 bits = 2^{10} entries. 2^{10} entries * 4 bytes/entry = 2^{12} bytes = the number of bytes on a page. This division means that page tables naturally break into units of single pages, which makes page management convenient.

Hide solution

Hide solution

QUESTION: What is the minimum number of page table pages necessary to provide access to 2^{20} different bytes of physical memory?

Show solution

2^{20} bytes = 2^{8} physical pages. A single level-1 page table can point to 2^{10} different pages, so we only need one level-1 page table page. We always need a root: the level-2 page table page. So the answer would seem to be 2.

But if we want to be very clever, we can actually *reuse* the level-2 page table page as a level-1 page table page! So the true answer is 1.

For instance, consider a level-2 page table page, located at physical address 0x1000, whose index-0 entry contained physical address 0x1000!

(This kind of reuse is occasionally useful for special purposes, but honestly it's more of a party trick.)

Hide solution

Hide solution

QUESTION: What is the minimum number of page tables necessary to provide access to 2^{20} different *pages* of physical memory?

Show solution

2^{20} pages requires 2^{10} different level-1 page tables. So a total of 2^{10} + 1 pages are required.

Hide solution

Hide solution

Moving to x86-64

x86-64 uses a similar structure, but each address consists of 48 meaningful bits (stored in a 64-bit, or 8-byte, number). This is so many bits that more levels of page table are required. x86-64 has a **four**-level page table, with addresses divided as follows:

bits 63-48	bits 47-39	bits 38-30	bits 29-21	bits 20-12	bits 11-0
(ignored)	Level 4 index	Level 3 index	Level 2 index	Level 1 index	Offset

Try working through the above problems for x86-64 (probably on your own time). What numbers do you get?

QUESTION: What is the maximum number of page table pages in each level? What is the maximum number of physical pages that are accessible with 4 levels of the pagetable? Include both physical pages corresponding to virtual memory and pagetable pages.

Show solution

There is 1 level-4 pagetable. There are at most 512 = 2^9 level-3 pagetable pages. There are most 512 * 512 = 2^18 level-2 page table pages. There are at most 512 * 512 * 512 = 2^27 level-1 pagetable pages. Each level-1 pagetable entry points to a physical page, so there are at most 512 * 512 * 512 * 512 = 2^36 physical pages = 2^48 bytes pointed to by virtual memory. There are also 1 + 2^9 + 2^18 + 2^27 pagetable pages.

Hide solution

Hide solution

QUESTION: If the upper 16 bits of x86-64 addresses were meaningful, how many levels would the page table require?

Show solution

Six: there 16 more bits, and a level can hold at most 9 bits' worth of index.

Hide solution

Hide solution

QUESTION: What is the *minimum* number of physical pages required on x86-64 to allocate the following allocations? Draw an example pagetable mapping for each scenario (start from scratch each time).

1. 1 byte of memory
2. 1 allocation of size 2^12 bytes of memory
3. 2^9 allocations of size of 2^12 bytes of memory each
4. 2^9 + 1 allocations of size of 2^12 bytes of memory each
5. 2^18 + 1 allocations of size 2^12 bytes of memory each

Show solution

1. 5 pages (1 per level plus the destination physical page)
2. Same as part (a)
3. 2^9 + 4 pages: 1 L4, 1 L3, 1 L2, and 1 L1 pages, plus 2^9 physical pages
4. 2^9 + 6 pages: 1 L4, 1 L3, 1 L2, 2 L1 pages, plus 2^9 + 1 physical pages
5. 2^18 + 518 pages: 1 L4, 1 L3, 2 L2, 513 L1 pages plus 2^18 + 1 physical pages

Hide solution

Hide solution

WeensyOS

The pset for this unit asks you to add to WeensyOS, an operating system that we'll use in this course. WeensyOS has internal interfaces for dealing with virtual memory that we'll now discuss.

`vmiter` and `ptiter` are iterators in WeensyOS for processing page tables. Vmiter helps iterate through pages in virtual memory, and lets you check the underlying physical pages for properties like permissions and physical addresses. Ptiter helps you iterate through page table pages. Let's walk through the code:

Vmiter

The `vmiter` class represents an iterator over virtual memory mappings. Here is its interface:

```
// `vmiter` walks over virtual address mappings.
// `pa()` and `perm()` read current addresses and permissions;
// `map()` installs new mappings.

class vmiter {
  public:
    // initialize a `vmiter` for `pt` pointing at `va`
    inline vmiter(x86_64_pagetable* pt, uintptr_t va = 0);
    inline vmiter(const proc* p, uintptr_t va = 0);

    inline uintptr_t va() const;      // current virtual address
    inline uint64_t pa() const;       // current physical address
    inline uint64_t perm() const;     // current permissions
    inline bool present() const;      // is va present?
    inline bool writable() const;     // is va writable?
    inline bool user() const;         // is va user-accessible (unprivileged)?

    inline vmiter& find(uintptr_t va);   // change virtual address to `va`
    inline vmiter& operator+=(intptr_t delta);  // advance `va` by `delta`

    // map current va to `pa` with permissions `perm`
    // Current va must be page-aligned. Calls kallocpage() to allocate
    // page table pages if necessary. Returns 0 on success,
    // negative on failure.
    int map(uintptr_t pa, int perm = PTE_P | PTE_W | PTE_U)
        __attribute__((warn_unused_result));
};
```

This type parses x86-64 page tables and manages virtual-to-physical address mappings. `vmiter(pt, va)` creates a `vmiter` object that's examining virtual address `va` in page table `pt`. Methods on this object can return the corresponding physical address:

```
x86_64_pagetable* pt = ...;
uintptr_t pa = vmiter(pt, va).pa();    // returns uintptr_t(-1) if unmapped
```

Or the permissions:

```
if (vmiter(pt, va).writable()) {
    // then `va` is present and writable (PTE_P | PTE_W) in `pt`
}
```

It's also possible to use `vmiter` as a loop variable, calling methods that query its state and methods that change its current virtual address. For example, this loop prints all present mappings in the lower 64KiB of memory:

```
for (vmiter it(pt, 0); it.va() < 0x10000; it += PAGESIZE) {
    if (it.present()) {
        log_printf("%p maps to %p\n", it.va(), it.pa());
    }
}
```

This loop goes one page at a time (the `it += PAGESIZE` expression increases `it.va()` by `PAGESIZE`).

The `vmiter.map()` function is used to *add* mappings to a page table. This maps physical page 0x3000 at virtual address 0x2000:

```
int r = vmiter(pt, 0x2000).map(0x3000, PTE_P | PTE_W | PTE_U);
// r == 0 on success, r < 0 on failure
```

Ptiter

```
class ptiter {
  public:
    // initialize a `ptiter` for `pt` pointing at `va`
    inline ptiter(x86_64_pagetable* pt, uintptr_t va = 0);
    inline ptiter(const proc* p, uintptr_t va = 0);

    inline uintptr_t va() const;             // current virtual address
    inline uintptr_t last_va() const;        // one past last va covered by ptp
    inline bool active() const;              // does va exist?
    inline int level() const;                // current level (0–2)
    inline x86_64_pagetable* ptp() const;    // current page table page
    inline uintptr_t ptp_pa() const;         // physical address of ptp

    // move to next page table page in depth-first order
    inline void next();
};
```

`ptiter` visits the *individual page table pages* in a multi-level page table, in depth-first order (so all level-1 page tables under a level-2 page table are visited before the level-2 is visited). A `ptiter` loop makes it easy to find all the page table pages owned by a process, which is usually at least 4 page tables in x86-64 (one per level).

```
for (ptiter it(pt, 0); it.va() < MEMSIZE_VIRTUAL; it.next()) {
    log_printf("[%p, %p): ptp at va %p, pa %p\n",
               it.va(), it.end_va(), it.ptp(), it.ptp_pa());
}
```

A WeensyOS process might print the following:

```
[0x0, 0x200000): ptp at va 0xb000, pa 0xb000
[0x200000, 0x400000): ptp at va 0xe000, pa 0xe000
[0x0, 0x40000000): ptp at va 0xa000, pa 0xa000
[0x0, 0x8000000000): ptp at va 0x9000, pa 0x9000
```

Note the depth-first order: the level-1 page table pages are visited first, then level-2, then level-3. Because of this order (and other implementation choices), a `ptiter` loop may be used to free a page table—for instance, when you're implementing process exit later :)

`ptiter` never visits the top level-4 page table page, so that will need to be freed independently of the `ptiter` loop.

Using iterators

On this problem set, you will implement some of the system calls in WeensyOS. Two of the main system calls that you will implement are `fork()` and `exit()`.

The fork system call allows a process to spawn a child process that is essentially a copy of the parent process. The child process has a copy of the parent processes' memory, and after the new child process created, both processes execute the next instruction after fork(). The registers and file descriptors are also copied.

The exit system call allows a process to stop its execution. All of the memory belonging to that process, including its page table pages, are returned to the kernel for reuse.

The Lifecycle of a Process

A process: the singular unit of life in an operating system. When a new process is born, it needs its own page table pages, which it gets from its parent. For processes not born from a `fork()` call, this parent is the kernel, and thus, the process's page table pages will be a copy of the kernel's.

During the normal execution of a process, it refers to memory addresses by their virtual memory addresses -- which means that, to actually access the underlying memory, those virtual memory addresses need to be translated into physical addresses. Fortunately, you have just learned about the basic mechanism behind this process. ;)

Once a process has completed its run, there is some cleanup to be done -- namely, the relinquishment of its virtual memory mappings. First, the process needs to disassociate itself from the kernel's page table pages -- an `exit`ed process shouldn't need them anymore, anyway. Next, the process needs to tell the kernel that it is no longer using its page table pages -- this is not just the level-1 page table pages, but also the higher-level ones as well! Finally, the process can rest in peace. RIP.

QUESTION: Use `vmiter` to implement the following function.

```
void copy_mappings(x86_64_pagetable* dst, x86_64_pagetable* src) {
    // Copy all virtual memory mappings from `src` into `dst`
    // for addresses in the range [0, MEMSIZE_VIRTUAL).
    // You may assume that `dst` starts out empty (has no mappings).

    // After this function completes, for any va with
    // 0 <= va < MEMSIZE_VIRTUAL, dst and src should map that va
    // to the same pa with the same permissions.



}
```

QUESTION: Use `vmiter` and `ptiter` to implement the following function.

```
void free_everything(x86_64_pagetable* pt) {
    // Free the following pages by passing their physical addresses
    // to `kfree_pa()`:
    // 1. All memory accessible by unprivileged mappings in `pt`.
    // 2. All page table pages that are part of `pt`.



}
```

Section 5

The UNIX shell 🐚

> Please get the latest section code from the `cs61-sections` repo.

Open up a new terminal window on your computer. Congratulations, you've opened up a UNIX Shell (unless you opened Command Prompt)! So far, you've used this wonderful utility to run around your filesystem, possibly open some code, build your pset, and/or play around with Git. Isn't it magical? Whoever built such a tool must've been true geniuses.

Fortunately, you too are a genius, because you will be implementing a (slightly simpler) shell for pset 5.

Digging deeper into The Inner Life of a ~~Cell~~ Shell

If you looked at the above video (it's not too relevant to CS 61), you might now be in awe of the fluid mosaic, phospholipid bilayer, etc. etc.—we certainly are. We also think the UNIX shell is an interesting work of art. For example, the shell can:

1. Interpret commands, e.g. `ls`,
2. Run some programs, e.g. `sudo apt-get install sl -y && sl`, and
3. It can even fork bomb your computer! 😈

Deep down, though, the shell is just doing the same thing in all of these cases: it's parsing the input you type into the shell, forking a subprocess to run the command, waiting for that process to finish running, and subsequently execute the next part of the command. Let's start off by playing around with the shell to familiarize yourself with it.

In pset 5, you will be given the rudimentary tools to parse the command line input and some shell code (🥁🥁) to get you started. You will:

1. Use `fork()` to create subprocesses, use `exec()` to load the program's code, and examine the output of `exit()` to determine the "success" of the execution.
2. Interpret command line syntax to emulate more complex behavior, e.g. `a && b` will only run `b` if `a` is successful.
3. Use pipes to direct input to and output from a program.

Shell exercises

Refer to the shell descriptions at the bottom of these section notes to complete these exercises. We'll start off on one to warm up, and move on to more throughout the section.

Exercise: Use only shell commands to print the contents of `eve_returns.cc` and `evil.sh` into the console, in that order.

Show solution

```
cat eve_returns.cc evil.sh
```

Hide solution

Hide solution

Process I/O and redirections

As you saw in pset 3, I/O is very important! It's so important, in fact, that reading from and writing to memory is a kernel-only power, and it's invoked by processes through a `syscall`. You may also have noticed that new files had file descriptors starting at 3—what happened to 0, 1, and 2? By convention, file descriptor 0 is `STDIN`, 1 is `STDOUT`, and 2 is `STDERR`.

How, then, are `STDIN`, `STDOUT`, and `STDERR` created, handled, and used? Remember that they're simply file descriptors, so all the `strace` calls you looked at in pset 3, including the `read()` and `write()` syscalls, can be used for this purpose. With that out of the way... how *can* we play around with these three file descriptors?

Redirections allow us to change the behavior of a program's `STDIN` and `STDOUT`—in particular, their sources. Try running these three commands:

```
$ echo hello kitty
$ echo hello kitty > cs61.txt
$ cat < cs61.txt
```

The first echo will print

```
hello kitty
```

to the terminal.

Exercise: What effects does the second `echo` have?

Show solution

Nothing is printed to the terminal, but a file called `cs61.txt` now contains the string, `hello kitty`.

Hide solution

Hide solution

Exercise: What effects does the `cat` have?

Show solution

The string, `hello kitty` is printed to the terminal, despite nothing in the shell command containing the string, `hello kitty`! Strangely enough, this is also the exact string that we had printed to `cs61.txt` earlier. Coincidence? I think not.

Hide solution

Hide solution

As it turns out, `program > sometxt` will cause `program` to print its standard output into a file called `sometxt` (creating the file if it does not exist, and clobbering any of its previous contents if it does exist), and `program < sometxt` will cause `program` to read from `sometxt` as its standard input. How convenient!

More kinds of redirection

`>>` is like `>`, but will append to the file if it already exists.

`<<` is not a thing.

`2> sometxt` means "redirect file descriptor 2 to sometxt for writing." Note that there is *no space* between `2` and `>`.

`2>&1` means "redirect file descriptor 2 to file descriptor 1." This causes all of `STDERR` to be printed directly to `STDOUT` instead. Again, note that there is *no space* between `>` and `&`.

Exercise: Repeat Shell Exercise #1, but store the result in a file called `cs61.txt`.

Show solution

```
cat eve_returns.cc evil.sh > cs61.txt
```

Hide solution

Hide solution

Exercise: Do it again, but with a different sequence of commands.

Show solution

```
cat eve_returns.cc > cs61.txt
cat evil.sh >> cs61.txt
```

Or `cat eve_returns.cc > cs61.txt ; cat evil.sh >> cs61.txt`

There's an infinite number of possibilities.

Hide solution

Hide solution

Pipes

Among the redirections briefly mentioned above is a funny-looking one: `2>&1`, which redirects file descriptor 2 into file descriptor 1. File descriptors are actually rather arbitrary... what if we could redirect one process's file descriptor into a different process's file descriptor? More importantly, *what if we could redirect `command1`'s `STDOUT` into `command2`'s `STDIN`?*

Of course, as with all loaded, leading questions, this question is a bit disingenuous—there's a reason why this section is titled "Pipes." Let's do a quick investigation of these magical *pipes*.

Consider this line of shell commands:

```
ls | head -n 2
```

Exercise: What does this do?

Show solution

The same thing we were doing earlier with two redirections: finding the first two items listed in the output of `ls`! Isn't this so much nicer than redirecting into and out from a file?

Hide solution

Hide solution

Pipes take information from one process and spit it into another. Aside from being more robust against typos, pipes also have another distinct advantage: in `command1 | command2`, `command2` can start working on the output of `command1` before `command1` has fully completed!

Pipes can also be chained together more than one at a time:

```
ls | head -n 2 | sed 's/[a-z]/?/g'
```

will print out the first two items listed in `ls`, but with all their lowercase letters replaced by question marks.

Question: How many pipes can be chained together in one line?

Show solution

> Technically, infinite. Practically, this is constrained mostly by memory and CPU... but assuming infinite RAM and infinite CPU, you could in theory have as many pipes chained together as you'd like.
>
> Hide solution
>
> Hide solution

We call a sequence of processes piped to each other a **pipeline**.

Use only shell commands to complete the following tasks. Refer to the the shell descrptions to help you find and use the proper shell utilities.

Exercise: Count the number of lines in the file `words` (and print the count).

Show solution

> ```
> wc -l words
> ```
>
> Hide solution
>
> Hide solution

Exercise: Print every unique line in the file `words` exactly once.

Show solution

> ```
> sort -u words
> ```
>
> Hide solution
>
> Hide solution

Exercise: Count the number of unique lines in the file `words` (and print the count).

Show solution

> ```
> sort -u words | wc -l
> ```
>
> Hide solution
>
> Hide solution

Exercise: Write a command that could help you discover whether a shell really executes the two sides of a pipeline in parallel. Describe the result if (1) the shell executed the left side to completion first (and buffered the output for the right side to read), (2) the shell executed the sides in parallel.

Show solution

Any number of solutions, but `sleep 10 | echo foo` is a classic one. If the pipeline is executed left-to-right with intermediate buffering, we'd see nothing printed for 10 seconds while the `sleep` counted down, then we might see `foo` printed afterward. If the pipeline is indeed executed in parallel, then we'd see `foo` printed immediately instead.

Hide solution

Hide solution

Putting commands together

Pipes aren't the only way to put commands together! Shells also allow us to run commands in sequences called **command lists**. And some of those ways let us detect and respond to failures.

First, the `;` operator says "run this, then run that." The semicolon waits for the left-hand command to complete. Then it executes the right-hand command.

Exercise: Write a *single* shell command list that repeats Shell Exercise #1, but runs two separate commands in a single command list.

Show solution

```
cat eve_returns.cc > cs61.txt ; cat evil.sh >> cs61.txt
```

Hide solution

Hide solution

Now, shell commands, and processes in general, can either succeed or fail. A process indicates its status when it exits by passing an integer (between 0 and 255) to the `exit()` library function. By convention, the zero status indicates success, and all non-zero statuses indicate failure. That is, successful processes are all alike; every failing process fails in its own way.

To test different statuses, we often use the `true` and `false` programs. `true` just exits with successful status: `exit(0)`. `false` just exits with the unsuccessful status 1: `exit(1)`.

Exercise: Write C++ code that compiles to a program equivalent to `true`.

Show solution

```c
#include <stdlib.h>
int main() {
    exit(0);
}
```

Hide solution

Hide solution

Exercise: Write C++ code that compiles to a program equivalent to `false`, but don't call `exit` explicitly.

Show solution

```c
int main() {
    return 1;
}
```

Hide solution

Hide solution

In well-written programs, errors that cause the program to exit should cause a failing exit status. For instance, `cat` will fail if it is passed a nonexistent file.

Exit status is commonly checked by the shell `&&` and `||` operators. These are like `;`—they execute the left-hand command first, then turn to the right-hand command—but unlike `;`, they check the status of the left-hand command, and only run the right-hand command if the left-hand command has an expected status.

Exercise: Use `true`, `false`, and `echo` to figure out what statuses `&&` and `||` expect.

Show solution

The && operator executes the right-hand command if the left-hand command succeeds. The || operator executes the right-hand command if the left-hand command fails. We might figure that out with commands like this:

```
true && echo left side succeeded 1
false && echo left side failed 2
true || echo left side succeeded 3
false || echo left side failed 4
```

That will print:

```
left side succeeded 1
left side failed 4
```

Hide solution

Hide solution

In command lists with multiple conditionals, the conditionals run left to right.

Exercise: What status does `false && echo foo` return? Use more conditionals to figure it out.

Show solution

It returns the status of `false`, which is 1. `false && echo foo || echo bar` and `false && echo foo && echo quux` would help you figure out that `false && echo foo` has a failing exit status; you'd have to use other features, such as `$?`, to figure out the exact status.

Hide solution

Hide solution

Exercise: Run and print the numerical exit status of `true` to the shell. (If you're working through on your own, read manual pages here, or look below at our documentation.)

Show solution

```
true; echo $?
```

Hide solution

Hide solution

More exploration

In lecture, we discussed some of the shell utilities—if you'd like to discover more and have more practice with them, here are some exercises to help you along your way.

Use only shell commands to complete the following tasks.

Exercise: Run and print the exit status of `false` to the shell.

Show solution

```
false; echo $?
```

Hide solution

Hide solution

Exercise: Print the exit status of `curl http://ipinfo.io/ip` to the shell.

Show solution

```
curl http://ipinfo.io/ip; echo $?
```

Hide solution

Hide solution

Exercise: Print the exit status of `curl cs61://ipinfo.io/ip` to the shell.

Show solution

```
curl cs61://ipinfo.io/ip; echo $?
```

Hide solution

Hide solution

Exercise: `curl` a URL and print `Success` if the download succeeds, or `Failure` if the download fails.

Show solution

```
curl lcdf.org && echo "Success" || echo "Failure"
```

Hide solution

Hide solution

Exercise: Do it again, but store the downloaded result in a file called `ip`.

Show solution

```
curl lcdf.org > ip && echo "Success" || echo "Failure"
```

Hide solution

Hide solution

Representing a parsed command line

In `sh61`, commands are parsed into tokens. The `sh61` grammar discusses more about how `sh61` parses commands and how to understand that process some more. For now, let's take a look at what we can do with a parsed command line.

There are two common ways to represent our parsed command line in a data structure appropriate for our shell, a tree representation and a list representation. Students often are biased toward the tree representation, which precisely represents the structure of the grammar, but the list representation can be easier to handle! The tradeoff is simplicity vs. execution time: the list representation requires more work to answer some questions about commands, but command lines are small enough in practice that the extra work doesn't matter.

The overall structure of a line, as described by the grammar, is:

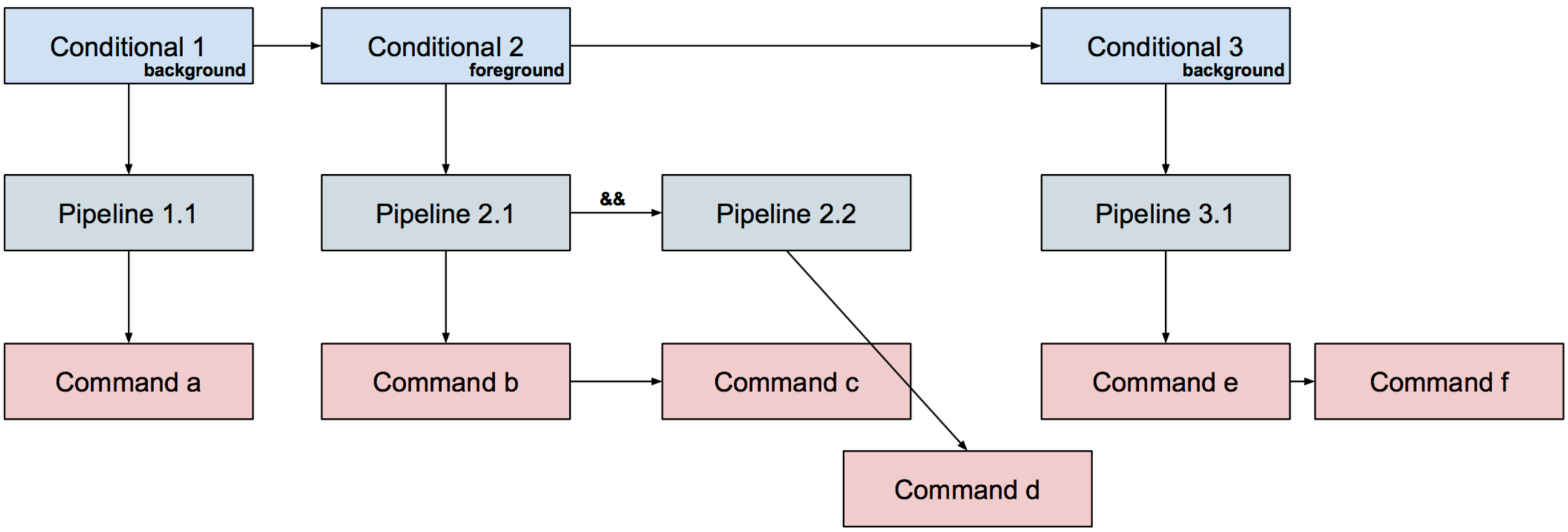
- A `command` is composed of `word`s and `redirection`s.
- A `pipeline` is composed of `command`s joined by `|`.
- A `conditional` is composed of `pipeline`s joined by `&&` or `||`.
- A `list` is composed of `conditional`s joined by `;` or `&` (and the last `command` in the `list` might or might not end with `&`).

Tree representation

The following tree-formatted data structure precisely represents this grammar structure.

- A `command` contains its executable, arguments, and any redirections.
- A `pipeline` is a linked list of `command`s, all of which are joined by `|`.
- A `conditional` is a linked list of `pipelines`. But since adjacent `conditional`s can be connected by either `&&` or `||`, we need to store whether the link is `&&` or `||`.
- A `list` is a linked list of `conditional`s, each flagged as foreground (i.e., joined by `;`) or background (i.e., joined by `&`). Note that while the `conditional` linktype doesn't matter for the last pipeline in a `conditional`, the background flag does matter for the last `conditional` in a `list`, since the last `conditional` in a `list` might be run in the foreground or background.

For instance, consider this tree structure for a command line (the `word`s and `redirection`s that are there, but not being shown for the sake of simplifying the image somewhat):



Exercise: Assuming that `a`, `b`, `c`, `d`, `e`, and `f` are distinct commands, what could be a command line that would produce this tree structure?

Show solution

```
a & b | c && d ; e | f &
```

Hide solution

Hide solution

Exercise: Sketch a set of C++ structures corresponding to this design.

Show solution

Here's one:

```
struct command {
    std::vector<std::string> args;
    command* next_in_pipeline;

    // initialize members to nullptr by default (good practice)
    command() {
        next_in_pipeline = nullptr;
    }
};
struct pipeline {
    command* child;
    pipeline* next_in_conditional;
    bool is_or;
    pipeline() {
        child = nullptr;
        next_in_conditional = nullptr;
    }
};
struct conditional {
    pipeline* child;
    conditional* next_in_list;
    bool is_background;
    conditional() {
        child = nullptr;
        next_in_list = nullptr;
    }
};
```

There are many other solutions. Logically, we need a struct for conditional chains that indicates whether the chain should be run in the foreground or background; a struct for pipelines; and a struct for commands. Each struct needs to hold the necessary pointers to form a linked list, as well as a pointer to the first item of its sublist (i.e., the overall command line struct needs a pointer to the head of the lists linked list, each list struct needs to hold a pointer to the head of its conditionals linked list, etc).

Hide solution

Hide solution

Exercise: Given a C++ structure corresponding to a conditional chain, explain how to tell whether that chain should be executed in the foreground or the background.

Show solution

For our solution, that's as simple as `c->is_background`.

Hide solution

Hide solution

Exercise: Given a C++ structure corresponding to a command, how can one determine whether the command is on the left-hand side of a pipe?

Show solution

For our solution, a command `c` is on the left-hand side of a pipe iff `c->next_in_pipeline != nullptr`.

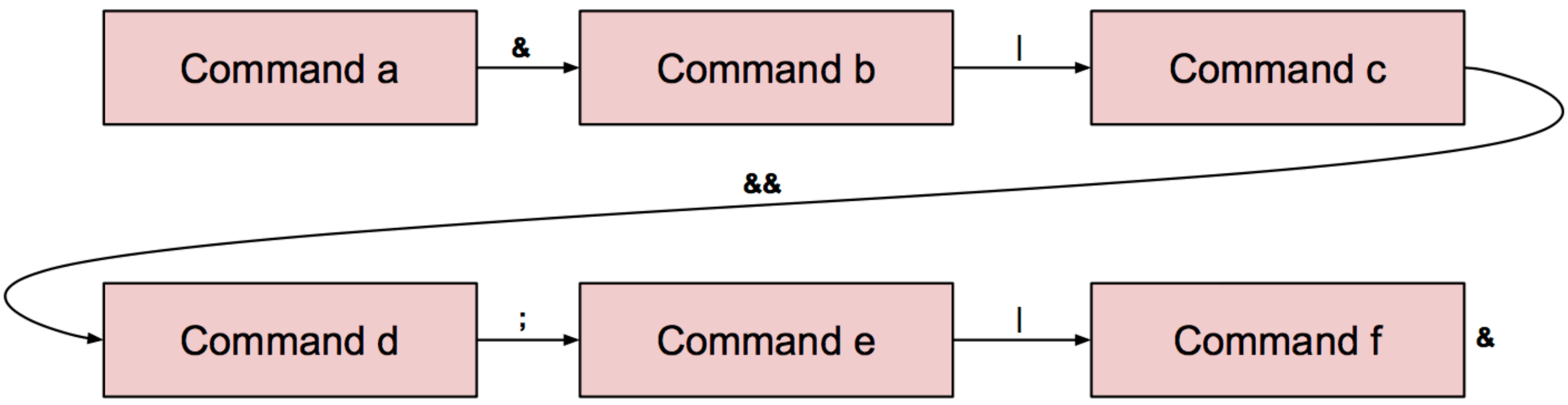
Hide solution

Hide solution

Exercise (left to the reader): Given a C++ structure corresponding to a command, how can one determine whether the command is on the *right*-hand side of a pipe?

Flat linked list representation

Alternatively, we can create a single linked list of all of the commands. In this case, we also store the connecting operator (one of `&` `;` `|` `&&` or `||`). For our example command line above, the flat linked list structure might look something like:



Exercise: Write a set of C++ structures corresponding to this design.

Show solution

Here's an example:

```
struct command {
    std::vector<std::string> args;
    command* next;
    int op;          // Operator following this command. Equals one of the TOKEN_
constants;
                     // always TOKEN_SEQUENCE or TOKEN_BACKGROUND for last in
list.

    // Initialize `next` and `op` when a `command` is allocated.
    command() {
        next = nullptr;
        op = TOKEN_SEQUENCE;
    }
};
```

Much simpler!

Another good one, with fewer pointers and thus fewer opportunities for memory mistakes, but harder to traverse (you need C++ iterators):

```
struct command {
    std::vector<std::string> args;
    int op;          // as above
};
using command_list = std::list<command>;
```

Hide solution

Hide solution

Exercise: Given a C++ structure corresponding to a conditional chain, explain how to tell whether that chain should be executed in the foreground or the background.

Show solution

```
bool chain_in_background(command* c) {
    while (c->op != TOKEN_SEQUENCE && c->op != TOKEN_BACKGROUND) {
        c = c->next;
    }
    return c->op != TOKEN_BACKGROUND;
}
```

Things are slightly more complicated with a flat linked list. We are still executing commands sequentially, but some subsequences of commands may need to be executed in the foreground or the background. During parsing, we can skip ahead in the command line to find the end of our current list (i.e., our current chain of conditionals).

Hide solution

Hide solution

Exercise (left to the reader): Given a C++ structure corresponding to a command, how can one determine whether the command is on the left-hand side of a pipe?

Exercise (left to the reader): Given a C++ structure corresponding to a command, how can one determine whether the command is on the *right*-hand side of a pipe?

Exercise (left to the reader): Sketch out code for parsing a command line into these C structures.

Documentation and References

Feel free to refer to these during section and as you work on pset 5.

sh61 shell subset

For pset 5, you will be expected to implement a subset of common UNIX shell features. Try a few of them now, and start thinking about how you might consider implementing them!

Syntax	Description
`command1 ; command2`	Sequencing. Run `command1`, and when it finishes, run `command2`.
`command1 & command2`	Backgrounding. Start `command1`, but don't wait for it to finish. Run `command2` right away.
`command1 && command2`	And conditional. Run `command1`; if it finishes by exiting with status `0`, run `command2`.
`command1 || command2`	Or conditional. Run `command1`; if it finishes by exiting with a status ≠ `0`, then run `command2`.
	Pipe. Run `command1` and `command2` in parallel. `command1`'s standard output is hooked up to `command2`'s standard input. Thus, `command2` reads what

`command1 | command2`	`command1` wrote. **The exit status of the pipeline is the exit status of `command2`.**
`command > file`	STDOUT redirection. Run `command` with its standard output writing to file. The file is truncated before the command is run.
`command < file`	STDIN redirection. Run `command` with its standard input reading from file.
`command 2> file`	STDERR redirection. Run `command` with its standard error writing to file. The file is truncated before the command is run.

Useful shell utilities

Here are some commonly-installed, commonly-used programs that you can call directly from the shell. Documentation for these programs can be accessed via the `man` page, e.g. `man cat`, or often also through a help switch, e.g. `cat --help`.

Shell Program	Description
`cat`	Write standard input to standard output.
`wc`	Count lines, words, and characters in standard input, write result to standard output.
`head -n N`	Print first `N` lines of standard input.
`tail -n N`	Print last `N` lines of standard input.
`echo ARG1 ARG2...`	Print arguments.
`printf FORMAT ARG...`	Print arguments with printf-style formatting.
`true`	Always succeed (exit with status `0`).
`false`	Always fail (exit with status `1`).
`sort`	Sort lines in input.
`uniq`	Drop duplicate lines in input (or print only duplicate lines).
`tr`	Change characters; e.g., `tr a-z A-Z` makes all letters uppercase.
`ps`	List processes.
`curl URL`	Download `URL` and write result to standard output.
`sleep N`	Pause for `N` seconds, then exit with status `0`.
`cut`	Cut selected portions of each line of a file.

Non-sh61 shell features

Here are some commonly-used features of a more fully-implemented shell (like the one you generally use!), but won't be part of the `sh61` assignment. Take a look, some of them are incredibly powerful!

Syntax	Description

`var=value`	Sets a variable to a value.
`$var`	Variable reference. Replaced with the variable's value. There are several special variables; for instance:
`$?`	The numeric exit status of the most recently executed foreground pipeline.
`$$`	The shell's own process ID.
`command >> file`	Run `command` with its standard output appending to `file`. The file is not truncated before the command is run.
`command 2>&1`	Run `command` with its standard error redirected to go to the same place as standard output.
`command 1>&2`	Run `command` with its standard output redirected to go to the same place as standard error. Thus, `echo Error 1>&2` prints `Error` to standard error.
`(command1; command2)`	Parentheses group commands into a "subshell." The entire subshell can have redirections, and can have its output put into a pipe.
`command1 $(command2)`	Command substitution. The shell runs `command2`, then passes its output as the first argument to `command1`.

sh61 grammar

This is the BNF grammar for `sh61`'s parser, as found in pset 5:

```
commandline ::= list
          |  list ";"
          |  list "&"

list      ::=  conditional
          |   list ";" conditional
          |   list "&" conditional

conditional ::=  pipeline
          |   conditional "&&" pipeline
          |   conditional "||" pipeline

pipeline ::=  command
          |   pipeline "|" command

command  ::=  [word or redirection]...

redirection  ::=  redirectionop filename
redirectionop  ::=  "<"  |  ">"  |  "2>"
```

A BNF grammar gives recursive definitions for a few terms (the "words" of the grammar). The `::=` indicates definition (i.e., `commandline` is defined to be `list | list ";" | list "&"`. On the definition side, the `|` is a logical or. For example, in `sh61`'s grammar, a `commandline` is composed of a `list`, or a `list` followed by a semicolon, or a `list` followed by an ampersand.

You may notice that in some of the later definitions, the term being defined is used in the definition. This recursive definition allows for lists or trees of terms to be chained together. Let's take the definition of `list` for example:

```
list     ::=  conditional
          |   list ";" conditional
          |   list "&" conditional
```

This reads "a `list` is a `conditional`, or a `list` followed by a semicolon and then a `conditional`, or a `list` followed by an `ampersand` and then a `conditional`." But this means that the `list` is just a bunch of `conditional`s, linked by semicolons or ampersands! Notice that the other recursive definitions in `sh61`'s grammar also follow this pattern. In other words:

- A `list` is a series of `conditionals`, concatenated by `;` or `&`.
- A `conditional` is a series of `pipelines`, concatenated by `&&` or `||`.
- A `pipeline` is a series of `command`s, concatenated by `|`.
- A `redirection` is one of `<`, `>`, or `2>`, followed by a `filename`.

What about the definition of `command`? `[word or redirection]`... seems a bit vague; in this case, you should use your intuition. When you type a command into the terminal, it's just a series of words representing the program name and its arguments, possibly followed by some number of redirection commands.

Section 6

Threads reference

We've talked about the concept of threads plenty in lecture, so we're going to start off with just a brief review of thread syntax.

Threads in C++

`std::thread`'s constructor takes the name of the function to run on the new thread and any arguments, like `std::thread t(func, param1, ...)`.

Given a thread `t`, `t.join()` blocks the current thread until the thread being joined exits. (This is roughly analogous to waitpid.) `t.detach()` sets the `t` thread free. Each thread object **must** be detached or joined before it goes out of scope (is destroyed).

An example of how we might combine these is:

```
#include <thread>

void foo(...) {
    ...
}

void bar(...) {
    ...
}

int main() {
    {
        std::thread t1(foo, ...);
        t1.detach();
    }

    {
        std::thread t2(bar, ...);
        t2.join();
    }
}
```

Exercise: What can this code print?

```
int main() {
    for (int i = 0; i != 3; ++i) {
        std::thread t(printf, "%d\n", i + 1);
        t.join();
    }
}
```

Show solution

`1\n2\n3\n`

Hide solution

Hide solution

Exercise: What can this code print?

```
int main() {
    for (int i = 0; i != 3; ++i) {
        std::thread t(printf, "%d\n", i + 1);
        t.detach();
    }
}
```

Show solution

1 2 3, in any order, and with any number of those lines left out—because when the main thread exits (by returning from `main`) all other threads are killed too!

Hide solution

Hide solution

Exercise: What can this code print?

```
int main() {
    for (int i = 0; i != 3; ++i) {
        std::thread t(printf, "%d\n", i + 1);
    }
}
```

Show solution

Literally anything. This is not C++ code: when the `t` thread goes out of scope without having been joined or detached, it invokes undefined behavior.

Hide solution

Hide solution

Threads in C

C doesn't have built-in threads or synchronization, but a common library is `pthread` (POSIX threads). Since we're using C++ threads in this class we won't talk about details here, but be aware that it exists in case you're ever working in C. (On Linux, standard C++ threads are implemented using the `pthread` library.)

Synchronization reference

Recall the high-level definition of a mutex: we want some way to *synchronize* access to data that is accessed simultaneously by multiple threads to avoid *race conditions*. Mutexes (also known as locks) achieve this by providing *mutual exclusion*, meaning that only *one* thread can be running in a particular *critical section* at any time.

The abstract interface for a mutex or lock is:

- `lock`: takes hold of the lock in a **blocking** fashion
- `try lock`: takes hold of the lock in a **polling** fashion; returns a boolean indicating whether the lock attempt succeeded
- `unlock`: instantly lets go of the lock

Note that it is disallowed to lock a lock that you already hold or unlock a lock that you do not hold. Depending on the implementation or language you're using, these actions might explicitly fail or—even worse—silently fail, and cause deadlock or incorrect synchronization.

If you've ever had the pleasure of participating in an icebreaker, you can think of a single lock as the object that is passed around to allow people to speak. You have to acquire the object before you can speak, and you release it so that another person can use it once you're done. Because there's one object and only one person can hold it at once, only one person can speak at a time, achieving mutual exclusion.

For multiple mutexes, a better analogy is the locks on a set of bathroom stalls. When you enter a stall (the critical section) you lock the lock associated with that stall. Then, you perform whatever operation needs to be synchronized inside the stall. Finally, once you're done, you unlock the stall and go on your merry way. This system ensures that each stall can only have one person in it at a time. Here the difference between `lock` and `try lock` is better motivated: if one particular stall is full, we might want to check to see if other stalls are available instead of waiting for someone to leave that one.

Synchronization in C++

At some point in the next few weeks you might want to synchronize something in C++. Here are some of the tools you can use to do that (and here is some even more comprehensive documentation).

Mutexes

C++ has a few different kinds of locks built in.

`std::mutex` is the basic mutex we all know and love. Once we declare and construct a `std::mutex m`, we can call:

- `m.lock()`
- `m.try_lock()`
- `m.unlock()`

which correspond exactly to the abstract lock operations defined above. A typical use might look like this, though later we're going to talk about an easier way to get the same behavior:

```
#include <mutex>

int i;
std::mutex m; // protects write access to i

void increase() {
    m.lock();
    i++;
    m.unlock();
}

...
```

`std::recursive_mutex` is like a normal mutex, but a thread that holds a lock may recursively acquire it again (as long as they recursively release it the same number of times). This might be useful if you need to acquire several locks, and sometimes some of those locks might actually be the same lock: with recursive locks, you don't need to keep track of which ones you already hold, to avoid acquiring the same lock twice.

`std::shared_lock` is an instantiation of a different kind of lock altogether: a readers-writer lock, or a shared-exclusive lock. The idea behind this kind of lock is that rather than just have one level of access--mutual exclusion--there are two levels of access: shared access and exclusive access. When a thread holds the lock with shared access, that guarantees that all other threads that hold the lock have shared access. When a thread holds the lock with exclusive access, that guarantees that it is the only thread to have any kind of access. We call this a readers-writer lock because the two levels of access often correspond to multiple reader threads, which require the shared data not to change while they're accessing it, but don't modify it themselves; and a single writer thread, which needs to be the only thread with access to the data it's modifying. You **do not need** the functionality of `std::shared_lock` for the problem set, and in fact it is not particularly helpful for this particular application.

The **only** mutex you need for the problem set is `std::mutex`, though `std::recursive_mutex` may simplify some logic for `simpong61`.

Mutex Management

C++ also defines some handy tools to acquire locks while avoiding deadlock. Instead of calling `.lock()` on the mutex itself, you can pass several mutexes as arguments to the function `std::lock`, which uses C++ magic to acquire the locks in an order that automatically avoids deadlock.

Here's an example of how this works:

```
#include <mutex>

std::mutex a;
std::mutex b;

void worker1() {
    std::lock(a, b);
    ...
    a.unlock(); // the order of these doesn't matter,
    b.unlock(); // since unlocking can't cause deadlock
}

void worker2() {
    std::lock(b, a); // look ma, no deadlock!
    ...
    b.unlock();
    a.unlock();
}


...
```

If it's too much of a burden to manually unlock all your locks, or if you don't trust yourself to remember to do so (we humans are quite careless), then `std::scoped_lock` is your best bet. This works the same as `std::lock`, except it automatically unlocks the mutexes when it leaves scope (leave the level of curly braces in which the `scoped_lock` occurs). This is an instance of an idiom called *Resource acquisition is initialization (RAII)*, which is the idea that an object's resources are automatically acquired and released by its constructor and destructor, which are triggered by scope. This means that `std::scoped_lock` constructs an *object* that acquires all the locks passed to it, and releases them when destructed, whereas `std::lock` is just a *function* that acquires the locks. Here's how it would simplify the code above:

```cpp
#include <mutex>

std::mutex a;
std::mutex b;

void worker1() {
    std::scoped_lock guard(a,b);
    ...
}

void worker2() {
    std::scoped_lock guard(b,a);
    ...
}

...
```

Condition Variables

Suppose you have several threads waiting for some event to happen, which doesn't neatly correspond to a single unlock event. (For example, to go back to the bathroom example, we might want people lining up to wait for *any* stall to open, not some particular stall door to be unlocked.) We could achieve this behavior naively by polling, but we've seen that this is wasteful. *Condition variables* are a synchronization primitive built on top of mutexes that allows us to form a queue of threads, which wait in line for a wake up call. (Note that condition variables usually do not actually guarantee that they function as FIFO queues; an arbitrary thread may be the head of the queue.)

The abstract interface for a condition variable is:

- `wait`: go to sleep on the condition variable **and a held lock** until signaled
- `notify one` or `signal`: wake up one thread waiting on the CV
- `notify all` or `broadcast`: wake up all threads waiting on the CV

Once we create a condition variable `std::condition_variable cv` in C++, we can call:

- `cv.wait(std::unique_lock<std::mutex>& lock)`
- `cv.notify_one()`
- `cv.notify_all()`

You probably noticed that `cv.wait` takes something we haven't seen before as a parameter: a `std::unique_lock<std::mutex>&`. Rather than use `std::lock` or `std::scoped_lock`, you **must** use `std::unique_lock` for condition variables. For other purposes, there is no meaningful difference between `std::unique_lock` and `std::scoped_lock`, but for condition variables the types must match up. The lock **must** be locked when calling `wait()`. **`wait()` unlocks the associated mutex while waiting, and re-locks the mutex before it returns.**

Condition variables are almost always used in loops that look like this:

```
std::unique_lock<std::mutex> lock;
...
while (!CONDITION) {
    cv.wait(lock);
}
```

In this code:

- `CONDITION` is some expression, such as `phase != 1`.
- The `CONDITION` can safely be evaluated when `lock` is locked.
- `cv` is associated with `CONDITION` in the following way: Any code that might change the truth value of `CONDITION` will call `cv.notify_all()`.

The loop is important because in most cases, a condition variable `wait` can wake up *spuriously*—in other words, by the time the `wait` operation returns, the condition has gone back to false.

> If you want to use a condition variable with something other than a `std::mutex`, you'll have to use `std::condition_variable_any` instead.

Synchronization: Counting to three

Here is some code.

```cpp
#include <thread>
#include <mutex>
#include <condition_variable>
#include <cstdio>

/* G1 */

void a() {
    /* A1 */
    printf("1\n");
    /* A2 */
}

void b() {
    /* B1 */
    printf("2\n");
    /* B2 */
}

void c() {
    /* C1 */
    printf("3\n");
    /* C2 */
}

int main() {
    /* M1 */
    std::thread at(a);
    std::thread bt(b);
    std::thread ct(c);
    /* M2 */
    ct.join();
    bt.join();
    at.join();
}
```

Exercise: What are the possible outputs of this program as is?

Show solution

> `1`, `2`, and `3`, on separate lines, in any order. (`printf` is thread-safe, so no two calls to `printf` will interleave characters.)
>
> Hide solution

Hide solution

Exercise: The program will *never* exit with the output `1\n2\n`. Why not?

Show solution

> The `ct.join()` line means that thread `ct`, which is running `c()`, must exit before the main program exits.
>
> Hide solution
>
> Hide solution

Exercise: Add code to this program so that the program only prints `1\n2\n3\n`. Your code must only use atomics, and you can only change the locations marked with comments.

Show solution

> ```
> G1: std::atomic<int> phase;
> A2: phase.store(1);
> B1: while (phase.load() != 1) {}
> B2: phase.store(2);
> C1: while (phase.load() != 2) {}
> ```
>
> Hide solution
>
> Hide solution

Exercise: Does your atomics-based code wait using blocking or polling?

Show solution

> Polling
>
> Hide solution
>
> Hide solution

Exercise: Add code to this program so that the program only prints `1\n2\n3\n`. Your code must only use `std::mutex` and/or `std::condition_variable`, and you must not ever unlock a mutex in a thread different from the thread that locked the mutex. Again, only change the locations marked with comments.

Show solution

```
G1: int phase = 0; std::mutex mutex; std::condition_variable cv;

A2: std::unique_lock<std::mutex> lock(mutex);
    phase = 1;
    cv.notify_all();

B1: std::unique_lock<std::mutex> lock(mutex);
    while (phase != 1) {
        cv.wait(lock);
    }

B2: phase = 2;
    cv.notify_all();

C1: std::unique_lock<std::mutex> lock(mutex);
    while (phase != 2) {
        cv.wait(lock);
    }
```

Hide solution

Hide solution

Deadlock

A **deadlock** is a situation in which two or more threads block forever, because each thread is waiting for a resource that's held by another thread.

Deadlock in multithreaded programming commonly involves mutual exclusion locks. For instance, thread 1 might have mutex `m1` locked while it tries to acquire mutex `m2`, while thread 2 has `m2` locked while it tries to acquire `m1`. Neither thread will ever make progress because each is waiting for the other.

There are simple ways to avoid deadlock in practice. One is to lock at most one mutex at a time: no deadlock! Another is to have a fixed order in which mutexes are locked, called a **lock order**. For instance, given lock order `m1`, `m2`, then thread 2 above cannot happen (it's acquiring locks against the lock order).

Deadlocks can also occur with other resources, such as space in pipe buffers (remember the extra credit in pset 3?) and even pavement.

Bathroom synchronization

Ah, the bathroom. Does anything deserve more careful consideration? Is anything more suitable for poop jokes?

Bathroom 1: Stall privacy

In this exercise, the setting is a bathroom with *K* stalls numbered 0 through *K–1*. A number of users enter the bathroom; each has a preferred stall. Each user can be modeled as an independent thread running the following code:

```
void user(int preferred_stall) {
    poop_into(preferred_stall);
}
```

Two users should never `poop_into` the same stall at the same time.

You probably can intuitively *smell* that this bathroom model has a synchronization problem. Answer the following questions without referring to the solutions first.

Question: What synchronization property is missing?

Show solution

> Mutual exclusion (two users can poop into the same stall at the same time).
>
> Hide solution
>
> Hide solution

Question: Provide synchronization using a single mutex, i.e., using **coarse-grained** locking. Your solution should be correct (it should avoid *gross conditions* [which are race conditions in the context of a bathroom-themed synchronization problem]), but it need **not** be efficient. Your solution will consist of (1) a definition for the mutex, which must be a global variable, and (2) new code for `user()`. You don't need to sweat the syntax.

Show solution

```
std::mutex bmutex;
void user(int preferred_stall) {
    std::scoped_lock guard(bmutex);
    poop_into(preferred_stall);
}
```

Hide solution

Hide solution

Question: Provide synchronization using **fine-grained** locking. Your solution should be more efficient than the previous solution. Specifically, it should allow the bathroom to achieve *full utilization*: it should be possible for all K stalls to be in use at the same time. You will need to change both the global mutex—there will be more than one global mutex—and the code for `user()`.

Show solution

```
std::mutex stall[K];
void user(int preferred_stall) {
    std::scoped_lock guard(stall[preferred_stall]);
    poop_into(preferred_stall);
}
```

Hide solution

Hide solution

Bathroom 2: Stall indifference

The setting is the same, except that now users have no predetermined preferred stall:

```
void user() {
    int preferred_stall = random() % K;
    poop_into(preferred_stall);
}
```

It should be pretty easy to adapt your synchronization solution to this setting, by locking the preferred stall once it's determined by random choice. But that can leave stalls unused: a user will wait indefinitely if they pick an occupied stall randomly, no matter how many other stalls are available.

Question: Design a solution that solves this problem: each user chooses any available stall, or waits if none are available. Use a single mutex and a single condition variable. You'll also need some subsidiary data.

Show solution

```
std::mutex bmutex;
std::condition_variable cv;
bool stall_full[K];
int noccupied;

void user() {
    std::unique_lock<std::mutex> lock(bmutex);
    while (noccupied == K) {
        cv.wait(lock);
    }
    ++noccupied;
    int stall = 0;
    while (stall_full[stall]) {
        ++stall;
    }
    stall_full[stall] = true;
    lock.unlock();

    poop_into(stall);

    lock.lock();
    --noccupied;
    stall_full[stall] = false;
    cv.notify_all();
    lock.unlock();
}
```

Hide solution

Hide solution

Question: Another solution is possible that uses K mutexes and no condition variables, and the `std::mutex::try_lock()` function. Design that solution.

Show solution

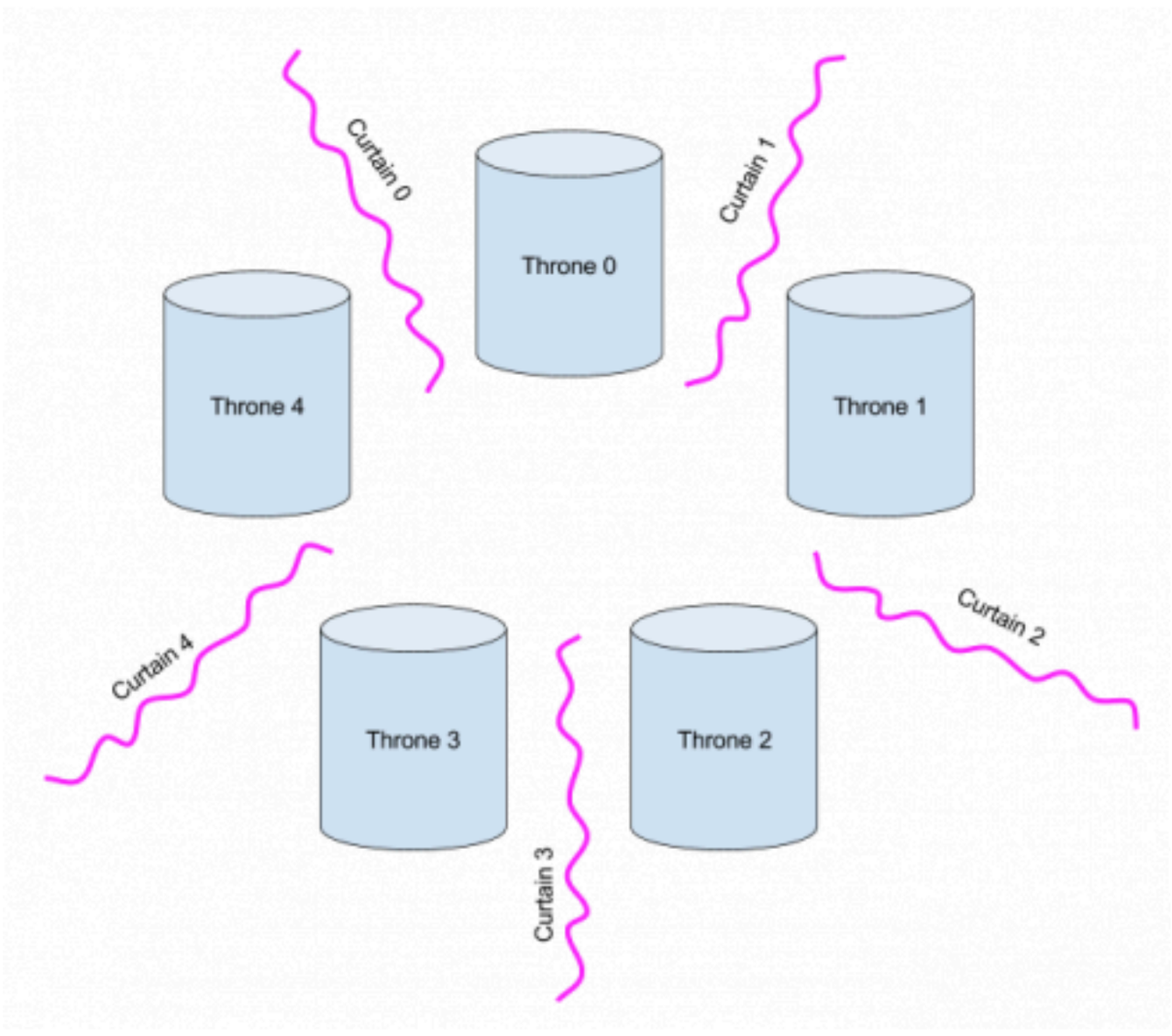
```
std::mutex stall[K];
void user() {
    for (int i = 0; true; i = (i + 1) % K) {
        if (stall[i].try_lock()) {
            poop_into(i);
            stall[i].unlock();
            return;
        }
    }
}
```

Hide solution

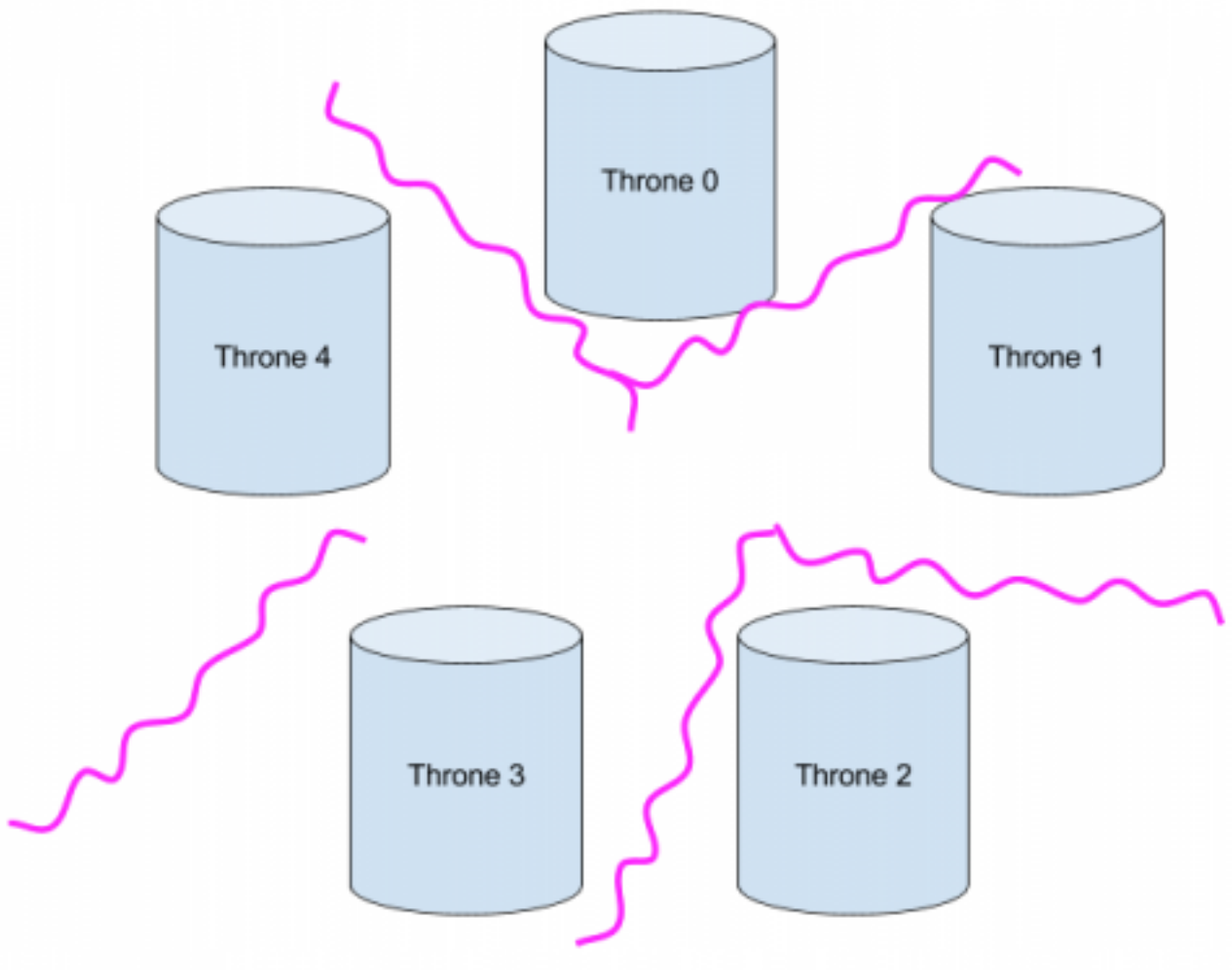
Hide solution

Bathroom 3: Pooping philosophers

To facilitate the communal philosophical life, Plato decreed that philosophy should be performed in a bathroom with K thrones and K curtains, arranged like so:



Each philosopher sits on a throne with two curtains, one on each side. The philosophers normally argue with one another. But sometimes a philosopher's got to poop, and to poop one needs privacy. A pooping philosopher must first draw the curtains on either side; philosopher X will need curtains X and (X+1) mod K. For instance, here, philosophers 0 and 2 can poop simultaneously, but philosopher 4 cannot poop until philosopher 0 is done (because philosopher 4 needs curtain 0), and philosopher 1 cannot poop until philosophers 0 and 2 are both done:



Question: In high-level terms, what is the mutual exclusion problem here—that is, what resources are subject to mutual exclusion?

Show solution

> Curtains. The mutual exclusion problem here is that each philosopher must have 2 curtains in order to poop, and a curtain can only be held by one philosopher at a time.
>
> Hide solution

Hide solution

A philosopher could be described like this:

```
void philosopher(int x) {
    while (true) {
        argue();
        draw curtains x and (x + 1) % K;
        poop();
        release curtains x and (x + 1) % K;
    }
}
```

Question: Write a correctly-synchronized version of this philosopher. Your solution should not suffer from deadlock, and philosophers should block (not poll) while waiting to poop. First, use `std::scoped_lock`—which can (magically) lock more than one mutex at a time while avoiding deadlock.

Show solution

```
std::mutex curtains[K];
void philosopher(int x) {
    while (true) {
        argue();
        std::scoped_lock guard(curtains[x], curtains[(x + 1) % K]);
        poop();
    }
}
```

Hide solution

Hide solution

Question: Can you do it without `std::scoped_lock`?

Show solution

Sure; we just enforce our own lock order.

```
std::mutex curtains[K];
void philosopher(int x) {
    while (true) {
        argue();
        int l1 = std::min(x, (x + 1) % K);
        int l2 = std::max(x, (x + 1) % K);
        std::unique_lock<std::mutex> guard1(curtains[l1]);
        std::unique_lock<std::mutex> guard2(curtains[l2]);
        poop();
    }
}
```

Hide solution

Hide solution

Designing synchronization

Recall the Fundamental Law of Synchronization: If two threads access the same object concurrently, and at least one of those accesses is a write, that invokes undefined behavior.

In the `simpong61` part of the problem set, you're given code that works correctly as long as there's only one thread, and you must synchronize it. This is actually a very common situation: it's much easier to start with correct single-threaded code than to write correct multi-threaded code all at once.

You can make progress on your synchronization strategy for the problem set by reasoning about which threads modify which objects. This will help you determine which accesses might run afoul of the Fundamental Law.

As is common, many objects in `simpong61` are *initialized* to some value in the main thread, before any other threads start running. Don't worry about those writes as you answer these questions: since no other threads have started, there's no problem.

Answer these questions by referring to the handout code.

Question: Which threads modify `pong_board::width_` and `height_`?

Show solution

None of them do. You can access those variables without synchronization.

Hide solution

Hide solution

Question: Which threads modify `pong_cell::type_`?

Show solution

Again, none.

Hide solution

Hide solution

Question: Which threads modify `pong_cell::ball_`?

Show solution

Any thread can modify these objects. However, the thread for a ball will only move *that* ball: it will set `pong_cell::ball_ = nullptr` for its own old position (which must have equaled `this` before), and it will set `pong_cell::ball_ = this` for its own new position (which must have equaled `nullptr` before).

Hide solution

Hide solution

Question: Which threads modify `pong_ball::x_` and `y_` for ball `b`? The answer is not "any thread!"

Show solution

Only the thread for ball `b` modifies `b->x_` and `b->y_`. That means that the thread for ball `b` can examine its `x_` and `y_` without holding any locks! However, any **other** ball's thread **cannot** access `b->x_` and `b->y_`, unless you design some synchronization. Luckily, in our handout code, you'll notice that each thread only accesses its own ball's `x_` and `y_`.

Hide solution

Hide solution

Question: Which threads modify `pong_ball::dx_` and `dy_` for ball `b`?

Show solution

Any ball's thread can modify any other ball's `dx_` and `dy_`, including its own.

Hide solution

Hide solution

For the remaining questions, we are not providing solutions; but if you think about the questions, you may be able to come up with a synchronization strategy. Assume a pong board of size `W`x`H` with `N` balls.

Exercise: Design a synchronization strategy with `W*H` mutexes. How many mutexes must `pong_ball::move` lock?

Exercise: Design a synchronization strategy with `W` mutexes.

Exercise: Design a synchronization strategy with `W+2` mutexes. Why is this easier to program than the strategy with exactly `W`?

Networking reference

So far we've learned about how to communicate between threads with synchronized variables and processes with inter-process communication like pipes, but how do we communicate between logical computers? (This might mean between your VM and host machine, host machine and local network, or even host machine and the internet at large.) The physical resources that computers use to connect over networks are *network sockets*, which, like all system resources, are managed by the operating system. Our use of sockets is therefore mediated by the `socket` family of *system calls*.

Sockets have two sets of behavior: client and server. The server socket is set up ahead of time and waits for incoming connections, then the client later sets up its own socket and connects to the waiting server socket. Once the client and server are connected, each of their file descriptors works as a bidirectional stream.

In both the client and server cases, you use the `socket()` system call to create a new, not-yet-connected network file descriptor.

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

In the server case, we next have to perform the following system calls:

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)
```

- `bind()` assigns a particular address to the socket file descriptor created with `socket()`. This is non-blocking.

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

- `listen()` starts listening for incoming connection requests, and allows a queue of length `backlog` of unattended requests to form. This is non-blocking.

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
```

- `accept()` **blocks** until there is a connection request in the queue, removes the connection from the backlog, and returns a new file descriptor for the new connection. This new file descriptor works as a bidirectional stream of communication with the client, wherever they may be. `accept()` **does not change** the underlying socket.

In the client case, we only have to perform one additional system call:

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

- `connect()` connects to the server socket specified by `addr`, and **blocks** until its request is fulfilled by the server calling `accept()`. Once this happens, the socket file descriptor passed to `connect()` **becomes** a bidirectional stream for communication with the server.

When you're done with a socket or connection as either a client or a server, make sure to `close()` it: it's ultimately just another kind of file.

Question: How are sockets different from pipes?

Show solution

Sockets have two different modes of use, client and server, which require different system calls to set up, while pipes have only one. Sockets can be used across machines with protocols like UDP or TCP, while pipes connect processes on the same logical machine. Each file descriptor associated with a socket is also bidirectional, while a pipe has two unidirectional file descriptors: one for reading and the other for writing.

Hide solution

Hide solution