CS 253: Introduction to Systems Programming
Slides

Introduction to Systems Programming (1 of 2)

- Roster and passwords
- Our `pub` directory:

  `onyx:~jbuffenb/classes/253/pub`

  `pub/etc/gcd.c`
- Our lecture slides, table of contents, and code:

  `pub/slides/slides.pdf`

  `pub/slides/code.tar`
- Review syllabus:

  `http://cs.boisestate.edu/~buff`

  `pub/syllabus/syllabus.pdf`

Introduction to Systems Programming (2 of 2)

- Our "textbooks" are:
  - *The C Programming Language*, by Brian Kernighan and Dennis Ritchie. Prentice Hall, second edition, 1988.
  - *Managing Projects with GNU Make*, by Robert Meclenburg, 2005:

    `pub/etc/mpwgm.pdf`
  - *Advanced Bash-Scripting Guide*, by Mendel Cooper, 2014:

    `pub/etc/abs-guide.pdf`
  - *The Art of Unix Programming*, by Eric Steven Raymond, 2003:

    `pub/etc/taoup.pdf`
- What is Systems Programming?
- What is a "systems" program?
- C for Java Programmers

C for Java Programmers: Similarities and Differences

- Since C begat C++, which begat Java, Java is quite similar to C.
- The biggest difference is that Java is an object-oriented (OO) programming language (PL), and C is not.
- Other differences are due to Java making what could be called improvements to C.
- First, we'll enumerate the similarities. Then, we'll examine the differences.

C for Java Programmers: Similarities (1 of 2)

- Both PLs have simple scalar data types: `char`, `int`, `float`, and `double`.
- Both PLs support various type modifiers: `short`, `long`, `unsigned`, `signed`, etc.
- A C `char` is an 8-bit byte, like a Java `byte`. A Java `char` is Unicode.
- The size of the other C types is not portable.

C for Java Programmers: Similarities (2 of 2)

- Both PLs have the same kinds of statements: assignments, `if`, `switch`, `for`, `while`, `do`, and function call. Both have `return`, `break`, and `continue`.
- Both PLs have the same kinds of expressions, although C has a few more operators. Recall that an expression is what can appear on the RHS of an assignment statement, or what can be passed as a parameter to a function.
- Both PLs have static/lexical scope. However, C allows a name in a nested scope to "shadow" the same name in an outer scope.

A Simple C Example (1 of 3)

- Since a “Hello world!” program is too simple, let’s start with an implementation of Euclid’s solution to the greatest common divisor (GCD) problem.

  `pub/etc/gcd.c`

  `pub/etc/GNUmakefile`
- The program gets its input in a silly way, so I can introduce the GNU Compiler Collection (GCC) preprocessor, and GNU Make.
- Two preprocessor macros, `X` and `Y`, provide input. They can be redefined, but only when the program is recompiled. This might make sense for configuration values, but not input values. Command-line arguments are more realistic.

A Simple C Example (2 of 3)

- The makefile is awful, but simple. It can redefine macro values. Eventually, we'll discuss better makefiles.
- The object file, `gcd.o`, depends on the makefile, to cause recompilation upon macro redefinition.
- The makefile separates compilation and linking, as is done with larger programs. Compilation is much slower than linking.
- The comment in `gcd.c` shows how to combine compilation and linking.

A Simple C Example (3 of 3)

- Comments in `gcd.c` also show how to execute the GNU Debugger (GDB) and the GNU Data Display Debugger (DDD).
- GDB has a jillion commands, but here are some good ones:
  - b: set a breakpoint
  - r: run the program
  - c: continue running the program
  - n: execute the next statement
  - s: execute one statement
  - p: print a value

  See the info documentation and/or the quick reference card for details.

  `pub/etc/gdb-refcard.pdf`
- DDD is an X11 GUI for GDB.

Java/C `boolean` versus `int`

- We can begin to examine Java/C differences, by returning to the GCD program.
- Consider the `while` loop's termination condition. In Java, it must be an expression of the builtin type `boolean`. C does not have a boolean type.
- Instead of an expression that evaluates to true or false, you can use an expression of other types (e.g., `int`). C treats an all-zero-bits value as false, and any other value as true.
- Thus, we *could* use `while (a-b)`, rather than `while (a!=b)`. Don't get *too* tricky!

Java/C `println` versus `printf`

- Neither Java nor C provides input/output (I/O) in the PL. Libraries provide I/O.
- In Java, `System.out.println` (et al.) is overloaded for different types.
- C provides `printf` (et al.), which takes a variable number of arguments. The first argument is a string, with embedded format specifiers, which describes the remaining arguments. A specifier contains a percent sign. See the `man` page for gory details.
- For C, we will use `getline` for input and `printf` for output, for reasons given later. We might also use `sscanf` and `asprintf`, but only in special ways.

Java/C `String` versus `char[]` or `char*` (1 of 5)

- Neither Java nor C provides character strings in the PL. Java libraries provide string classes (e.g., `String`). C libraries provide string functions, which assume the data representation described next.
- In C, a character string is a sequence of zero or more (contiguous, with increasing addresses) non-zero bytes in memory, terminated by a zero byte.
- A string is denoted by the address of the beginning of the sequence. For a zero-length string, the address is that of the terminating zero byte.
- Some people use the preprocessor macro `NULL`, rather than an eight-bit zero, when referring to the terminator, but `NULL` is defined to be some kind of zero.

Java/C `String` versus `char[]` or `char*` (2 of 5)

- A string's characters, and terminator, can be accessed in two ways:
  - as an element of an array of `char`, by indexing the array (e.g., `s[i]`)
  - as a standalone `char`, through a pointer to `char`, by dereferencing the pointer (e.g., `*s`)
- Adjacent characters can be accessed by index or pointer arithmetic, respectively.
- Choosing which way to access a string is often difficult, especially for C newbies.
- We will talk, much more, about arrays and pointers, soon.

Java/C `String` versus `char[]` or `char*` (3 of 5)

- Here's an example, which, repeatedly, reads a line from `stdin`, uppercases it, and writes it to `stdout`:

  `pub/Shout/Shout.c`
- A good way to read a line is `getline`. It allocates, reuses, or reallocates memory, the size of which is in `n`. Eventually, you should deallocate it with `free`, or you'll have a *memory leak*.
- The number of characters read is in `len`, which may be less then `n`. Both `size_t` and `ssize_t` are integers.
- When `getline` is called, the *addresses* of `line` and `n` are passed, so `getline` can change their values.
- The variable `line` can be considered a character pointer, a character array, or a string (pick one).
- For practice, you could change the `for` loop to use pointers, rather than indices.

Java/C `String` versus `char[]` or `char*` (4 of 5)

- Regardless of whether arrays or pointers (or both) are used, managing memory for strings, and accessing them, is a *huge* source of bugs in C programs. From CERN Computer Security:

  Most vulnerabilities in C are related to buffer overflows and string manipulation.

- You can read all about it here:

  `http://en.wikipedia.org/wiki/Buffer_overflow`

Java/C `String` versus `char[]` or `char*` (5 of 5)

- The basic problem is that C allows a programmer to access memory at an address below or above the range of addresses allocated for a particular variable. This can:
  - cause an operating-system intervention, due to a memory-access privilege violation (e.g., a segmentation fault), thereby terminating the program
  - change or corrupt the value of an adjacent variable
  - change the sequence of execution of the program, by changing a function call's return address

Java/C Arrays (1 of 8)

- In Java, an array variable contains a reference to an array object, allocated by the operators **`new`** and `[]`. An array variable does *not* contain the array elements.
- Also in Java, an array is also indexed by the operator `[]`, which is, essentially, just the name of an method, called on the array object. The index is passed as a parameter to the method.

Java/C Arrays (2 of 8)

- In C, and other PLs:
  - An *aggregate* variable is one that can hold multiple values simultaneously. If the values are of the same type, the variable is *homogeneous*. Otherwise, it is *heterogeneous*.
  - An *array* is a homogeneous aggregate variable, which can be indexed to obtain individual values.
  - A *struct* is a heterogeneous aggregate variable, with named individual values. We'll see these, later.

Java/C Arrays (3 of 8)

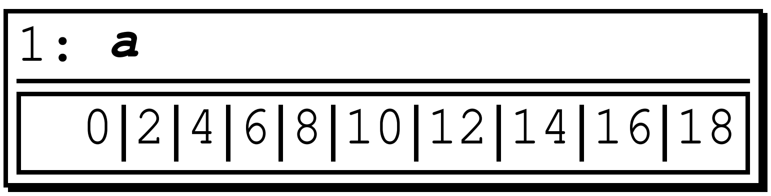
- In Java and C, indices run from $0 \cdots n-1$, where n is the size.
- But in C, if you access outside the bounds of the array, neither the compiler nor the run-time system will tell you.
- Here's a simple example:

  `pub/ArraySimple/ArraySimple.c`
- Surprisingly, `SIZE` is a macro, rather than a `const` variable, for simpler allocation. We'll see VLAs, soon.
- When possible, let the compiler count, and use the (compile-time) `sizeof` operator:

  `pub/ArrayInit/ArrayInit.c`

Java/C Arrays (4 of 8)

- This display was created with `ddd`:



- In plain `gdb`, just:

```
(gdb) p a
$1 = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18}
```

Java/C Arrays (5 of 8)

- An array can be passed as a parameter to a function, but in an weird way.
- Even though C passes all parameters by value (i.e., by passing a copy of the parameter), the "value" of an array is its starting address. The address is copied, not the array elements. The effect is that of pass by reference.
- This allows a function to access array parameters of differing sizes, but there is no builtin way for the function to determine the array's size. It's the programmer's job.

Java/C Arrays (6 of 8)

- Here's an example of passing an array as a parameter:

  pub/ArrayArg/ArrayArg.c

- These displays were created with **ddd**. I set a breakpoint in **inc**, where the array is treated as a pointer:

```
1: a
(int *) 0x7fffffffe130
```

```
X
0x7fffffffe130: 0        2        4        6
0x7fffffffe140: 8        11       12       14
0x7fffffffe150: 16       18
```

```
2: *a@size
  0|2|4|6|8|11|12|14|16|18
```

Java/C Arrays (7 of 8)

- In original C, the size of an array had to be static: computable by the compiler, at compile time. This was true even for local variables. That's why, so far, our examples used a macro for the size, rather than a `const` variable.
- Later, C adopted so-called variable-length arrays (aka, VLAs):

  `pub/ArrayVLA/ArrayVLA.c`

  Note: Sometimes, `sizeof` must be computed at run time.
- Different instances of local-variable arrays can have different sizes. But, once allocated, an array's size cannot change.
- Don't use this feature! There are better ways to do this.

Java/C Arrays (8 of 8)

- C, like many PLs, only supports arrays of one dimension. However, since each element of an array can be an array, arrays of two or more dimensions can be simulated:

  `pub/ArrayTwoD/ArrayTwoD.c`

  `pub/ArrayTwoD/array.pdf`

- If you initialize such an array, you can let the compiler count for the "outer" dimension:

  `pub/Array2dInit/Array2dInit.c`

- Note that you cannot initialize a variable-length array. This produces a puzzling error message:

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

- Computer scientists rarely use multi-dimensional arrays.

Java/C Pointers (1 of 9)

- We've mentioned pointers, but we need to know more about them. Java does not have pointers.
- A *pointer* is a variable whose value is an address, which can be explicitly dereferenced to obtain the value at that address.
- In a declaration, an asterisk denotes a pointer. In an expression, an asterisk denotes a dereference.

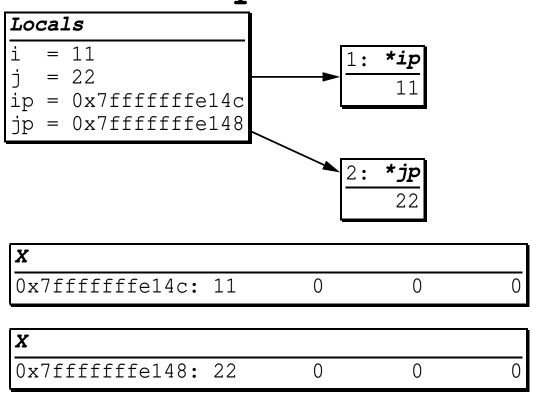
Java/C Pointers (2 of 9)

- The address assigned to a pointer can be computed in several ways:
  - The "address-of" operator (i.e., `&`) returns the *l-value* of an *r-value*.
  - The "pointer arithmetic" operators (e.g., `++`, `--`, `+`, and `-`) manipulate addresses.
  - A literal address, usually `0`, looks like an integer literal. Some people use the macro `NULL`, rather than `0`. A `0` is like Java's `null`.
- Here's an example:

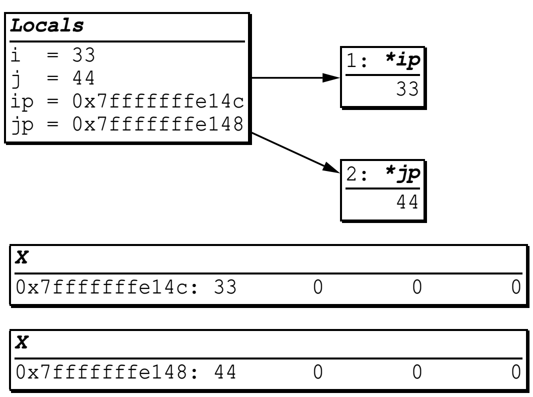
  `pub/PtrSimple/PtrSimple.c`

Java/C Pointers (3 of 9)

- Here's a `ddd` graph after a breakpoint at the first `printf`:



- Here's at the second `printf`:



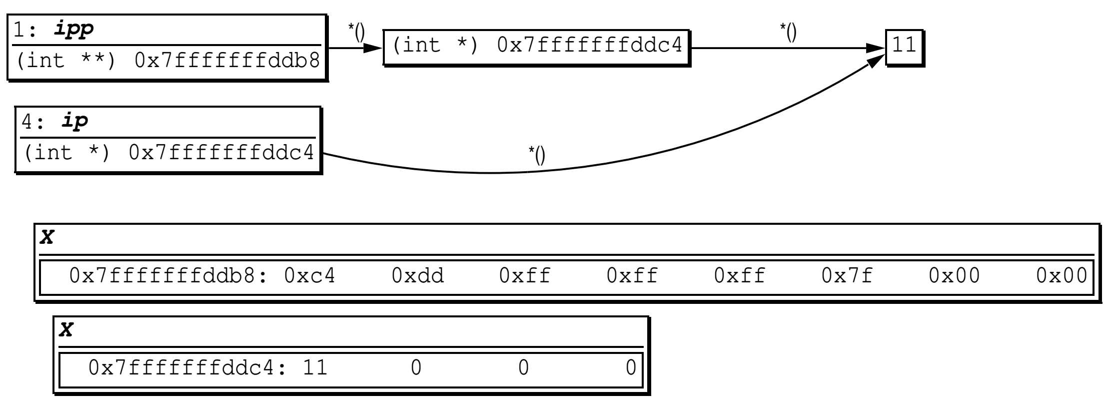
Java/C Pointers (4 of 9)

- These displays were produced by:
  - Select `Data`, then `Display Local Variables`.
  - Select `ip`, then click `Display *()`.
  - Select `jp`, then click `Display *()`.
  - Select `Data`, then `Memory...`, then `4 decimal bytes from ip`, then `4 decimal bytes from jp`.
- This also demonstrates that our Intel processor is little endian.

Java/C Pointers (5 of 9)

- A pointer can point to another pointer.
- Here's an example:

  `pub/PtrMultiple/PtrMultiple.c`
- In `ddd`, double click on a pointer name in a definition, then double click on the display:



Java/C Pointers (6 of 9)

- A pointer can point to an array. Indeed, as we saw with character strings, such a pointer is pretty much interchangable with the array. Sometimes, it's hard to decide which to use.
- Here's a pointer-palooza example:

  `pub/ArrayPtr/ArrayPtr.c`
- Notice how pointer arithmetic conforms to the size of the type that is pointed to (e.g., `p++` does *not* add one to the address).
- Here's even more examples:

  `pub/ArrayPtr/arrptr1.c`

  `pub/ArrayPtr/arrptr2.c`

Java/C Pointers (7 of 9)

- So far, our pointer examples are silly.
- A *real* pointer accesses dynamically-allocated memory, managed by `malloc` and `free`. These functions are part of the Standard C Library, like `printf`, but they are declared in `stdlib.h`, rather than `stdio.h`.
- Although `malloc` is not part of C, it's the closest cousin to Java's operator `new`.
- We've been allocating memory for parameters and local (`auto`) variables, which are deallocated when the enclosing scope ends or function returns.
- When you allocate memory with `malloc`, it is not deallocated until you call `free`. If you forget, you have a memory leak.

Java/C Pointers (8 of 9)

- This example demonstrates:
  - `malloc` and `free`
  - printing an error message
  - accessing the program's name
  - size-in-bytes versus size-in-elements
  - even more array-versus-pointer stuff

  `pub/PtrAlloc/PtrAlloc.c`
- From `ddd`:

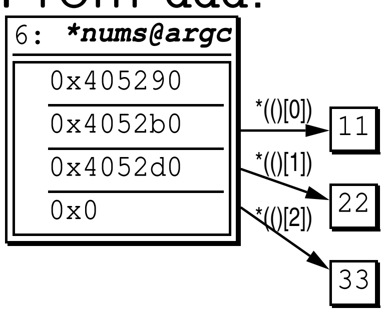
```
2: &prog
(char **) 0x404070 <prog>

4: prog
0x7fffffffe326 "/home/buff/ws/253/pub/PtrAlloc/PtrAlloc"

5: &a
(int (*)[3]) 0x7fffffffdd74

6: a
11|4|33

X
0x7fffffffdd74: 11    0    0    0    4    0    0    0
0x7fffffffdd7c: 33    0    0    0

7: &b
(int **) 0x7fffffffdd88

8: b
(int *) 0x4052a0

10: *b@3
11|4|33

X
0x4052a0:       11    0    0    0    4    0    0    0
0x4052a8:       33    0    0    0
```

Java/C Pointers (9 of 9)

- In Java, a common data structure is a sequence of object references (e.g., `ArrayList<Student>`).
- A typical C implementation of such a data structure is a dynamically-allocated array of pointers to dynamically-allocated structures.
- We'll see `struct` examples, but here we'll use `int` as our "object" type.

  `pub/PtrArrPtrs/PtrArrPtrs.c`

- From `ddd`:



Valgrind (1 of 3)

- Let's revisit our last program: a `malloc`-ed array of pointers to `malloc`-ed memory:
  `pub/PtrArrPtrs/PtrArrPtrs.c`
- How can we check whether:
  - it erroneously accesses outside any of those `malloc`-ed blocks?
  - we remembered to call `free` on all of those `malloc`-ed blocks?
- Valgrind to the rescue!

Valgrind (2 of 3)

- Valgrind is an system for debugging and profiling programs. It can automatically detect many memory-management and threading bugs, avoiding hours of frustrating bug hunting, making your programs more stable. You can also perform detailed profiling, to speed up and reduce memory use of your programs.
- Valgrind takes control of your program before it starts. Debugging information is read from the executable and associated libraries, so that error messages and other outputs can be phrased in terms of source code locations.
- Your program is then run on a synthetic CPU provided by the Valgrind core.

Valgrind (3 of 3)

- Our makefile knows how to analyze our program with Valgrind. Just run:

  ```
  make valgrind
  ```
- No problems were found.
- What if our we forgot the first terminator, and our program is run with no arguments?
- What if we miscounted the command-line arguments?
- What if we forgot to free the data?

Java/C Command-Line Arguments

- As we saw in the last example, `main` has (optional) formal parameters. They contain the program's name and command-line arguments. They are traditionally named `argc` ("count") and `argv` ("vector").
- `argc` is an integer whose value is the number of "words" on the command line, including the program's name.
- `argv` is an array of character strings (i.e., an array of pointers to characters).
- `envp` is a third (optional) argument, containing environment variables and their values, as an array of strings.
- Here's an example:

  `pub/Args/Args.c`
- Functions `atoi` and `atof` convert a string to an `int` and a `double`, respectively.

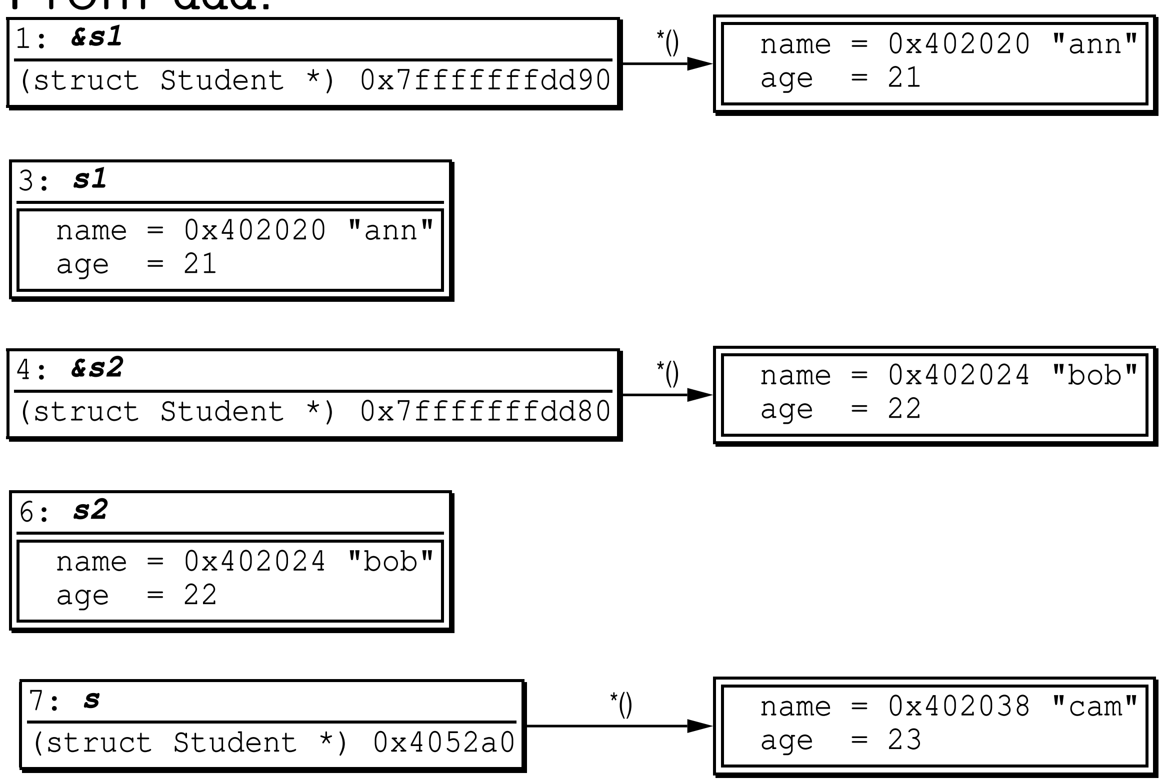
Java/C Structures (1 of 4)

- A Java `class` can be mostly simulated by a C `struct`. Indeed, a C++ `class` is essentially a `struct`. (Since C++ never throws anything away, it still has both.)
- A `struct` sort-of defines a new type, which can be used to define a variable. People use "struct" to describe both the type and variable.
- As with other types, memory for a `struct` can be allocated statically, on the stack, or in the heap.

Java/C Structures (2 of 4)

- A `struct` is a heterogeneous aggregate variable, with named individual values (aka, *members*). A member's name is analogous to an array index.
- Here's an example:

  `pub/StructSimple/StructSimple.c`
- From `ddd`:



Java/C Structures (3 of 4)

- The element type of an array can be a `struct`, or a pointer to a `struct`.
- A `struct` can contain an array.
- We saw that passing a `struct` to a function makes a copy. This can be a clever way to pass a *copy* of an array to a function.
- Returning a `struct` from a function also makes a copy. This can be a clever way to return “multiple” values from a function.
- A `struct` declaration *can* be nested within another `struct` declaration, but this usually isn't a good idea. Usually, you want `struct` names to be global, or file global.

Java/C Structures (4 of 4)

- Here's an example:

  `pub/StructType/StructType.c`
- You can omit the *tag*.
- You can combine the declaration and variable definition/initialization.

Java/C Alignment, Padding, and Unions (1 of 2)

- Most modern CPUs use byte-addressed memory: each byte has its own address.
- Some 64-bit CPUs can read an eight-byte word, with one machine instruction, in one memory cycle, if it begins at an address with zero as the low three bits ($8 = 2^3$). Otherwise, more cycles are required.
- Other 64-bit CPUs can *only* read an eight-byte word, with one machine instruction, if it has such an address. Otherwise more instructions are required.
- Thus, there are benefits to storing a 2^n-byte word at an address with the low n bits zero. This is called *alignment*. Any wasted memory, due to alignment, is called *padding*.

Java/C Alignment, Padding, and Unions (2 of 2)

- In C, array elements are contiguous, with no special alignment, and no padding.
- In C, `struct` members *may* be aligned, with padding. They will not be reordered.
- Thus, a `struct` member's offset, from the beginning of the `struct`, might be larger than the sum of the sizes of previous members:

  `pub/StructAlign/StructAlign.c`
- A `union` is just a `struct` with all members having an offset of zero (i.e., they overlap). This allows a value of one type to be accessed as a value of another type:

  `pub/UnionSimple/UnionSimple.c`

Java/C Enumerations (1 of 3)

- An *enumeration* is a type with a set of named values. An example is the set of three-letter days of the week, containing: Sun, Mon, Tue, and so on.
- Without enumerations, programmers tend to use integer literals, or variables with integer values, to encode the enumeration values. If the PL supports it, these variables should be constants.
- The problem with using integer literals is that no can remember whether Sunday is encoded as 0, 1, 6, or 7.
- Also, changing the encoding for December, from 12 to 11, might change the number of eggs in a dozen.

Java/C Enumerations (2 of 3)

- Java has enumerations, now, but they are a recent addition. A Java `enum` is a class. Each value is a `static final` variable.
- A C `enum` sort-of defines a type (i.e., a tag, like a `struct` tag), and a set of `int` constants. The constants' values start at zero, and increase, but you can also explicitly define values.
- Surprisingly, the scope of the constant name is that of the tag. They can be referenced without qualification.
- Here's an example:

  `pub/Enum/Enum.c`

Java/C Enumerations (3 of 3)

- Since an `enum` constant is "constant enough" to be a `switch` case value, it can also be used to define the size of a non-variable-length array, which avoids using a macro for this purpose, and makes the constant visible to a debugger.
- In this example, the capitalization, and parameter order, is quite subtle:

  `pub/ArrayEnum/ArrayEnum.c`

  This allows the following, which won't work if `ROWS` is a macro:

  ```
  (gdb) b pr
  (gdb) r
  (gdb) p a[0]@ROWS
  ```

C for Java Programmers: Differences

- Comparative Skeletal Anatomy
- Comparative Visibility Etiquette
- Modules
- Functions
- Variables
- Types

Java/C Comparative Skeletal Anatomy

- In both PLs, of course, a source program is a set of named files.
- File names matter to the Java compiler.
- File names do *not* matter to the C compiler. But, they matter to us. We will follow historical convention.
- Here's a comparison:

    `pub/etc/comp-anat.pdf`

Java/C Comparative Visibility Etiquette

- Java
  - Avoid using class (aka, `static`) variables.
  - Instance variables, class or object, should be private.
- C
  - Avoid using variables declared outside a function definition (aka, static variables), whether declared `static` or not.
  - Variables declared outside a function definition should be declared `static`, giving them module-global visibility.

Java/C Modules (1 of 2)

- A Java module is a `class`: a set of function and variable definitions. The class name defines a type.
- A C *module* is a pair of files: an interface (`.h`) file and an implementation (`.c`) file.
- A C *interface* is a set of function declarations and type definitions.
- A C *implementation* is a set of function, variable, and type definitions.
- In C, a *function definition* has a body; a *declaration* does not. A declaration is also called a signature, prototype, or header.
- In C, a *variable definition* allocates memory for the variable; a *declaration* does not.
- In C, a *type definition* (i.e., `typedef`) simply names a type.

Java/C Modules (2 of 2)

- This example demonstrates a simple module, which exposes its data representation:

  `pub/ModulePublic/ModulePublic.c`

  `pub/ModulePublic/Student.h`

  `pub/ModulePublic/Student.c`

- This module hides its data representation, behind a `void` pointer:

  `pub/ModulePrivate/Student.h`

  `pub/ModulePrivate/Student.c`

  Notice that a well-behaved client is unchanged.

Java/C Functions

- Java is OO; C is not. A non-static Java function makes no sense in C.

  `pub/etc/javafunc.c`
- Java's `public` is like C's `extern`.
- Java's `private` is like C's `static`.
- In C, all declarations of a function, and the one definition must match.
- In both Java and C, function parameters are passed by value. Neither supports pass-by-reference, as some people claim. Let's argue about this!

Java/C Variables (1 of 2)

- Java is OO; C is not. Passing an object reference, as a function parameter, makes no sense in C. However, C has pointers. A pointer can be passed as a function parameter. It can then be dereferenced.

  `pub/etc/javavar.c`

- In Java and C, a variable that is local (`auto`) to a function definition is like a function parameter that was not assigned a value by the caller.

Java/C Variables (2 of 2)

- In C, a module-global variable is much like a `private static` class variable in Java. Its visibility is limited to the module. Its lifetime is that of the program. Try not to use!
- In C, a global variable is much like a `public static` class variable in Java. Its visibility is unlimited, and its name need not be qualified by the module name. Its lifetime is that of the program. Do not use!

Java/C Types (1 of 2)

- In Java, a type is defined by creating and naming a new class. A class is created by inheriting from a superclass. The subclass, at least at first, is just a copy of the superclass. Of course, functions and variables are typically added to the subclass.
- Java is OO; C is not. Inheriting from a superclass makes no sense in C. However, C has structures, which can be created with a `struct`. A structure can simulate a class, but not inheritance. Indeed, C++ implements classes with structures.
- In C, a type is defined by naming an already existing type, with a `typedef`. If the existing type is a structure, the name is much like a Java class name.

Java/C Types (2 of 2)

- In C, the possible combinations of `struct`, `union`, `typedef`, pointers, and arrays are very confusing. This is partly because C has "evolved" and partly because there is more than one way to do it. We'll ignore `union` and arrays, for now, but it's still confusing. Here are some bad ways, and the good ways:

  `pub/etc/javaclass.c`
- The sort-of type `struct Int1` is an example of a so-called *structure tag*. We'll need them, but not now. Try to avoid them.
- The C library function `malloc` is like Java's `new` operator, but way dumber.

Linux `x86_64` Process Memory

- A 64-bit (8-byte) address has 16 hex digits (e.g., `7fff ffff ffff ffff`). It can address any byte in a 64-exabyte region:

  `pub/etc/mem-proc.pdf`
- The 256-terabyte user-space region needs the low 48 bits:

  `pub/etc/mem-user.pdf`
- Now, we can see where `auto`, `static`, and `malloc()`-ed data is allocated in memory. And, we can understand the addresses that `gdb` shows us:

  `pub/etc/javaclass.c`

Java/C Containers (1 of 2)

- We saw a C module that implements a container, as an array of pointers to `int`:
  `pub/PtrArrPtrs/PtrArrPtrs.c`
- We saw that `argv` and `envp` use this idea, as an array of pointers to `char`.
- This module implements the container, as an array of `Student` pointers:
  `pub/PtrArrStudents/Picture.pdf`
  `pub/PtrArrStudents/PtrArrStudents.c`
- Would `ModulePrivate` work?
- How many times do we have to do this???

Java/C Containers (2 of 2)

- A popular alternative to an array of pointers to structures is a linked list of pointers to structures.
- Such a structure cannot be indexed as an array, but is more flexible for insertions and deletions:

  `pub/PtrPtrStudents/PtrPtrStudents.c`
- There are many different ways of doing this sort of thing.
- How many times do we have to do this???

Java/C Generic/Polymorphic Containers (1 of 3)

- We saw a C module that approximates Java's `ArrayList<Student>`:

  `pub/PtrPtrStudents/PtrPtrStudents.c`
- How can we avoid having to implement this sort of module for each type of value we want to contain?
- We saw how to hide a module's data representation, behind a `void` pointer:

  `pub/ModulePrivate/Student.h`

  `pub/ModulePrivate/Student.c`
- We can combine these ideas, to create a reusable `List` module, for containing lists of values of any type.
- However, we have some design decisions to make.

Java/C Generic/Polymorphic Containers (2 of 3)

- Should our lists be only one-level deep?

```
typedef void *Elm;
typedef struct Elms {
  Elm elm;
  struct Elms *elms;
} *Elms;
```

- Or, should we support nested lists, to unbounded depth?

```
typedef void *List;
typedef struct {
  List car;
  List cdr;
} *Pair;
```

Java/C Generic/Polymorphic Containers (3 of 3)

- As sometimes happens, the more general solution is also simpler and more elegant.

  `pub/ListStudents/ListStudents.c`
- The names of the list functions, are borrowed from Lisp:
  - `cons(List car, List cdr)` constructs a pair from its arguments. Intuitively, `car` is added to the front of the list `cdr`. It's like `add`.
  - `car(List list)` returns the first part of the pair at the front of `list`. It's like `head` or `first`.
  - `cdr(List list)` returns the second part of the pair at the front of `list`. It's like `tail` or `rest`.

Java/C Function Pointers (1 of 4)

- A Java class typically defines methods. An object of that class holds references to those methods, because the methods are shared between objects of that class.
- A subclass can override a method's definition. A sublass object then holds a reference to the overriding (i.e., new) method. The method can be called, through the reference, which is the address of the first machine instruction of the method. In Java:

```
r=o.m(a);
```

Java/C Function Pointers (2 of 4)

- In C lingo, the reference/address is a *function pointer*. Such a value can be stored in a scalar variable, an array, or a structure. It can also be passed as a parameter to, or returned from, a function.
- A typical use of function pointers is to have some task g that needs to perform some subtask f, but there's more than one way to do f: $f_a, f_b, \cdots$. So, simply define the f_i you want, and pass it to g, like this: $g(\cdots, f_i, \cdots)$.
- For example, reconsider:

  `pub/ListStudents/ListStudents.c`

  Notice the redundancy. We are already iterating (aka, cdr-ing) down the student list twice: once to print and once to free. How many times to we have to do this???

Java/C Function Pointers (3 of 4)

- Our g is the cdr-ing task. Our name for g is `map`, because it "maps" a function across the elements of a list:

  pub/ListStudentsMap/listext.h

  pub/ListStudentsMap/listext.c

- Our f_i, for now, are:
  - f_a: return a string representation of a student. This is `studentToString`.
  - f_b: print a string. This is a new function, `prtstr`, a `printf` wrapper.
  - f_c: free a student's memory. This is `freeStudent`.

    pub/ListStudentsMap/ListStudents.c

Java/C Function Pointers (4 of 4)

- The trickiest part about function pointers is getting the signatures, or casts, right. GCC's error messages can help.
- Our `map` allocates a list. Deallocating it can be tricky. Valgrind can help.
- We could have merged the toStringing, printing, and/or freeing functions. Modularity is better.
- In general, function pointers are handy for: a sort function's need to compare elements, framework callbacks (e.g., signals), and functional programming. For example, see the `man` pages for `qsort` and `signal`.

Java/C Variadic Functions

- Java allows a function to have a variable number of parameters, with a definition like this:

```
public static void foo(int ... a) {
  for (int i: a)
    System.out.println(i);
}
```

- C does it quite a bit differently, using the `...` syntax, and library functions/macros:
  `pub/VarArgs/VarArgs.c`
- Notice how:
  - You have to indicate the last regular parameter, before the varying parameters.
  - You have to recognize the last varying parameter.
  - There's no type checking.

I/O from the C Library (1 of 7)

- As we've discussed, C has no I/O constructs. It relies on library functions to provide simple and fancy I/O, at multiple levels of abstraction.
- We've been, and will continue to use, `getline` for input and `printf` for output. Line-oriented I/O is often convenient and sufficient. More critically, `getline` is, by far, the best way to avoid buffer-overflow bugs.

I/O from the C Library (2 of 7)

- Nevertheless, you will most certainly see, and may need, functions from lower abstraction levels (from low to higher):
  - Low-level buffer-oriented I/O: `read` and `write`.
  - Buffered stream-oriented character-oriented I/O: `getchar`, `putchar`, `getc`, `putc`, `fgetc`, and `fputc`.
  - Buffered stream-oriented string-oriented I/O: `fgets` and `fputs`.
  - Buffered stream-oriented formatted I/O: `printf`, `fprintf`, `scanf`, and `fscanf`.
  - Formatted strings: `snprintf` and `sscanf`.

I/O from the C Library (3 of 7)

- Example programs for each level follow.
- Note well: The programs ignore I/O errors (e.g., full disk). Such errors can be detected, but I've ignored them to simplify the examples. See the `man` pages for more information.

I/O from the C Library (4 of 7)

- This demonstrates low-level I/O:
  `pub/CopyRW/CopyRW.c`
- The type of `stdin` and `stdout` is `FILE *`, which is also called a stream. The function `fileno` returns the integer file descriptor associated with a stream. We could have just used `0` and `1`.
- Function `strcmp` compares character strings. A return value of `0` means the strings are equal.
- A file name of `-` denotes `stdinp` or `stdout`.
- The call to `creat` also sets the permission bits on the output file.
- We could have used any value for the buffer size.

I/O from the C Library (5 of 7)

- This demonstrates medium-level I/O:
  `pub/CopyFGP/CopyFGP.c`
- There's a macro version:
  `pub/CopyGP/CopyGP.c`
- These are significantly simpler than the last example.
- We could have used `getchar` and `putchar`, which are macros that use `stdin` and `stdout`.
- Don't use a mixture.

I/O from the C Library (6 of 7)

- This demonstrates line-oriented I/O:
  `pub/CopyLn/CopyLn.c`
- Don't use `gets`! Your program will be vulnerable to buffer overflows.
- Multiple invocations of `fgets` may be needed to read an entire line.

I/O from the C Library (7 of 7)

- This demonstrates formatted I/O:
  `pub/CopySP/CopySP.c`
- These functions perform type conversions.
- The address of a variable is passed to `fscanf`, so it can change the variable's value.
- The `%ms` (nee, `%as`) `malloc`-allocation format is extremely convenient for avoiding buffer overflows. Here, we have a memory leak.
- The `sscanf` and `asprintf` functions allow reading from, and writing to, a string, rather than a file. Don't use `sprintf`! Your program will be vulnerable to buffer overflows.

Shell (Bash) Scripts

- We'll be talking about Bash scripts, but other shells are similar (e.g., `ash` – `zsh`).
- Bash is an acronym for Bourne Again SHell, a pun on the name of the second Unix shell, `sh`, developed by Stephen Bourne. The first was developed by Ken Thompson. Both were at Bell Labs.
- Bash was developed by Brian Fox, and is maintained by Chet Ramey.
- In its simplest form, a (shell) script is an "executable" text file, containing a sequence of program invocations (e.g., `cp`, `mv`, and `rm`). A shell just reads and executes each line.
- A modern shell, like Bash, is an interpreter for a full-featured general-purpose PL.

Bash Introduction (1 of 9)

- To ensure that your script is executed by the *right* shell, the first line should be:

  `#!/bin/bash`

  The ASCII values of the first two characters form a 16-bit "magic number" defining the file type. (see the `man` page for `file`). This causes the `exec` library functions to execute Bash on the script.
- Typically, you'll make it executable:

  `chmod a+x` *script*

Bash Introduction (2 of 9)

- A newline can be significant. For two commands on one line, separate them with a:

  `;` (sequencing)

  `&` (concurrency)

  `||` (short-circuit "or")

  `&&` (short-circuit "and")

- A compound command is surrounded by curly braces:

  `pub/bash/braces`

- Comments extend from a pound sign to the end of the line.
- A function is a named compound command that can be called with parameters.

  `pub/bash/function`

Bash Introduction (3 of 9)

- *Filename globbing* is Bash's notation for expressing a list of files. It's similar to regular expressions, but not the same.

  `pub/bash/globbing`
- Input and output (I/O) can be redirected from and to other files. When Bash starts a process, it has three open file descriptors:

  0: `stdin`, typically the keyboard

  1: `stdout`, typically the display

  2: `stderr`, typically the display
- Here are some examples:

  `pub/bash/io`
  - The descriptor for < is 0.
  - The descriptor for > is 1.
  - The ampersand allows you to refer to a particular descriptor.
  - Send your error messages to `stderr`!

Bash Introduction (4 of 9)

- I/O can also be piped from and to the output and input of other processes.

  `pub/bash/pipes`

  - This avoids temporary files.
  - This enables concurrency.
  - A pipeline's commands are executed in separate processes, with separate environments (e.g., variables and working directory).

Bash Introduction (5 of 9)

- A process's environment includes a set of variables, each of which has a value. A variable does not really have a type, but if its value looks like an integer, you can apply arithmetic operations.
- A command can refer to a variable: the reference is replaced by the variable's value. This is called *variable substitution*:

  `pub/bash/vars`

  - Curly braces are usually unnecessary.
  - There are simple, and complex, manipulation mechanisms.
  - You can remember left versus right by keyboard placement.

Bash Introduction (6 of 9)

- A reference to an undefined variable is substituted with the empty string.
- There are many predefined variables. For example:

  `$$`: the process identification number (PID)

  `$?`: the exit code of the last command

  `$#`: the number of arguments

  `$@`: all arguments

  `$`i: (i is an integer) an argument

  `$PWD`: the working directory

  `$HOME`: the user's home directory

  `$PATH`: directories to search for commands

Bash Introduction (7 of 9)

- A command can use the output of another command, produced on `stdout`, as one or more command-line arguments. This is called *command substitution*:

  `pub/bash/cmd`
- Backticks are the old way (e.g., `` `ls` ``). Use the new way.
- Alternatively, the output of another command can be accessed through a command-line argument naming a file (actually, a named pipe). This is called *process substitution*:

  `pub/bash/proc`

Bash Introduction (8 of 9)

- Bash has several quoting mechanisms. The vast majority of Bash programmers do not understand them:

  `pub/bash/quotes`

  My guidelines are:
  - If a command-line argument is the empty string or contains glob characters, surround it with double quotes. This will allow variable substitution.
  - If a command-line argument contains a dollar sign, double quote, or backslash, surround it with single quotes. This will prevent variable substitution.
  - If a command-line argument contains a single quote, surround it with double quotes.

Bash Introduction (9 of 9)

- When a process exits, it produces an integer exit code, indicating, in some way, whether it was successful or not. Zero means success. Bash assigns this integer to the variable `$?`. Many Bash built-in commands test this value.
- Bash's `test` built-in command can evaluate a variety of expressions, producing an appropriate exit code:

  `pub/bash/cmdtest`
- `test` can be abbreviated as [. It has a fancier [[. Linux also has a real program named `/usr/bin/[`.

Control Structures: `if`

- Of course, `if` commands are handy:
  `pub/bash/cmdif`
- For the `if`, `while`, and `until` control structures, the "test" is an arbitrary command. Actually, it can be a sequence of commands.
- Often, the "test" is just `test`, in the form of `[`. Notice how `]` is the last argument of `[`. Cute!
- For `if`, the end of the "test" is marked by a newline, or semicolon, followed by a `then`.

Control Structures: `while` and `until`

- These loops allow the same "test" as `if`.
- However, a `while` loop is especially good for processing the content of a file:

  `pub/bash/cmdwhile`

  - The `read` builtin command reads a line from `stdin`, splits it into fields, and sets variables to the field values.
  - The last variable (e.g., `others`) is set to all remaining fields.
  - The `IFS` variable controls field splitting. The default is whitespace.
- You can process complete lines:

  `pub/bash/cmdwhileline`
- An `until` loop just negates the "test".
- Bash loops support `break` and `continue` builtin commands.

Control Structures: `for`

- A `for` loop is much different than a `while` or `until` loop. Rather than a "test," it has a loop variable and a sequence of values.
- A `for` loop is especially good for iterating through file names, command-line arguments, or function parameters.

  `pub/bash/cmdfor`
- Nevertheless, you'll see silly code like this:

  `pub/bash/cmdforbad`
- When you omit the `in` part, the default is `in "$@"`. We'll discuss this, more, soon.
- We'll see arithmetic loops, too, soon.

Control Structures: `case`

- A script often needs to test whether a file name, or string, matches a pattern. Bash has several ways to do this:
  - Globbing expands a glob to one or more matching file names.
  - `test` or [can compare strings for equality, inequality, or order.
  - [[can try to match a pattern or regular expression against a string.
  - `case` tries to match a pattern against a string.
- They can be difficult to differentiate.
- Bash's `case` command is like a `switch` in other PLs. It is often better than an `if` command, because you can use patterns:

  `pub/bash/cmdcase`

  These patterns are neither globs nor regular expressions.

More on Quotation

- Two constructs deserve special mention.
- The variable references `$@` and `$*` are substituted by all of a script's command-line arguments, or all of a function's parameters, but differently:

  `pub/bash/allargs`
- I always double quote them.

Arithmetic

- A variable's value is a string, even if it looks like a number, but arithmetic can be done in the context of a `let` builtin command.
- `let` is typically abbreviated, with double parentheses:

  `pub/bash/arithwhile`
- Or, with the "alternate" form of `for` loop:

  `pub/bash/arithfor`

Temporary Files: `trap`

- A script sometimes needs temporary files. Names must be chosen carefully. They should be removed, afterwards.
- This work well:

  `pub/bash/trap`
- `$0` is the path to the script.
- `$prg` is the name of the script.
- `$$` is the process id.
- The `trap` builtin command allows a script to react to a signal. Here, `EXIT` isn't really a signal, but it causes the command to be executed just before the script exits, for whatever reason.

Arrays (1 of 2)

- When I'm thinking about how to solve a problem, without using C/C++, my usual "escalation path" is:
  1. Can I just do it at the Bash prompt?
  2. Will a (pure) Bash script be enough?
  3. Do I need `grep`, `sed`, `cmp`, etc? These tools are good for simple subtasks.
  4. If I need coordinated multi-line manipulations, or single-process execution, I use Awk. It's syntax is simple enough to skip its manual.
  5. If I need a particular library, or object orientation, I use Perl or Python. No one can remember Perl syntax. Both have awful syntax.

  Although Bash's array syntax is overly punctuational, its arrays are useful enough to avoid using another PL.

Arrays (2 of 2)

- Bash has indexed *and* associative arrays.
- Array declaration can be combined with assignment.
- Array declaration is optional for indexed arrays.
- Indexed arrays use integer indices, which start at zero:

  `pub/bash/arrindex`
- Associative arrays use string indices:

  `pub/bash/arrassoc`
- As I said, array syntax is nasty, but better than with Perl.
- If you are tempted to use `while` and `read` to slurp an entire file into an array, consider the `readarray` (aka, `mapfile`) builtin command:

  `pub/bash/arrread`

Make and Makefiles

- We'll be talking about GNU Make.
- To a programmer, Make keeps an executable file up-to-date, with respect to its source files.
- In general, Make records and implements a *system model*, which specifies how *target* files are built from their *dependency* files.
- A system model may, or may not, involve compilation and linking.
- Make stores a system model in a *makefile*, written in a domain-specific language. This language is declarative, imperative, and functional: a combination of all three PL paradigms. A typical programmer only knows imperative PLs, and therefore, struggles with makefiles.

A Simple Makefile (1 of 2)

- We first saw this (overly) simple makefile:
  `pub/etc/GNUmakefile`
- It could be even simpler:
  `pub/etc/gmf1.mf`
- It specifies, directly:
  - The executable file depends on the one object file, and is built by the linker.
  - The object file depends on the one source file, and is built by the compiler.
- Make determines, transitively, that the executable file depends on the source file.

A Simple Makefile (2 of 2)

- That makefile is, still, clearly hardcoded for a particular set of files.
- It also contains redundancy: "gcd" appears seven times.
- Make has ways to address these problems. For example:

  `pub/etc/gmf2.mf`

  or:

  `pub/etc/gmf3.mf`

  or:

  `pub/etc/gmf4.mf`
- These versions employ:
  - user-defined variables
  - automatic variables
  - pattern rules
- We'll even see how to avoid mentioning "gcd" at all, in a reusable makefile.

Makefile Syntax (1 of 2)

- Essentially, a makefile contains a set of rules. Their order is unimportant. We'll discuss rules, in detail, soon.
- It can also contain variable definitions. For example:

  ```
  prog := gcd
  flags = -o $(prog) $<
  ```

  Whitespace around the assignment token is optional, but the definition extends to the end of the line. The first form is eager; the second lazy. Prefer the first.
- It can also contain directives, to do something, while the makefile is read (e.g., `include` another file).
- Finally, comments start with `#` and extend to the end of the line

Makefile Syntax (2 of 2)

- A (simple) rule has the following syntax:

  target: *dependency*$_1$ $\cdots$ *dependency*$_m$
  command$_1$
  $\vdots$
  command$_n$
- A *target* is (typically) a file to be built (e.g., an object or executable file).
- A *dependency* is a file upon which *target* depends (e.g., an object file depends upon its corresponding source file).
- Each *command* is a shell (e.g., Bash) command, which must begin with a tab.

Makefile Semantics

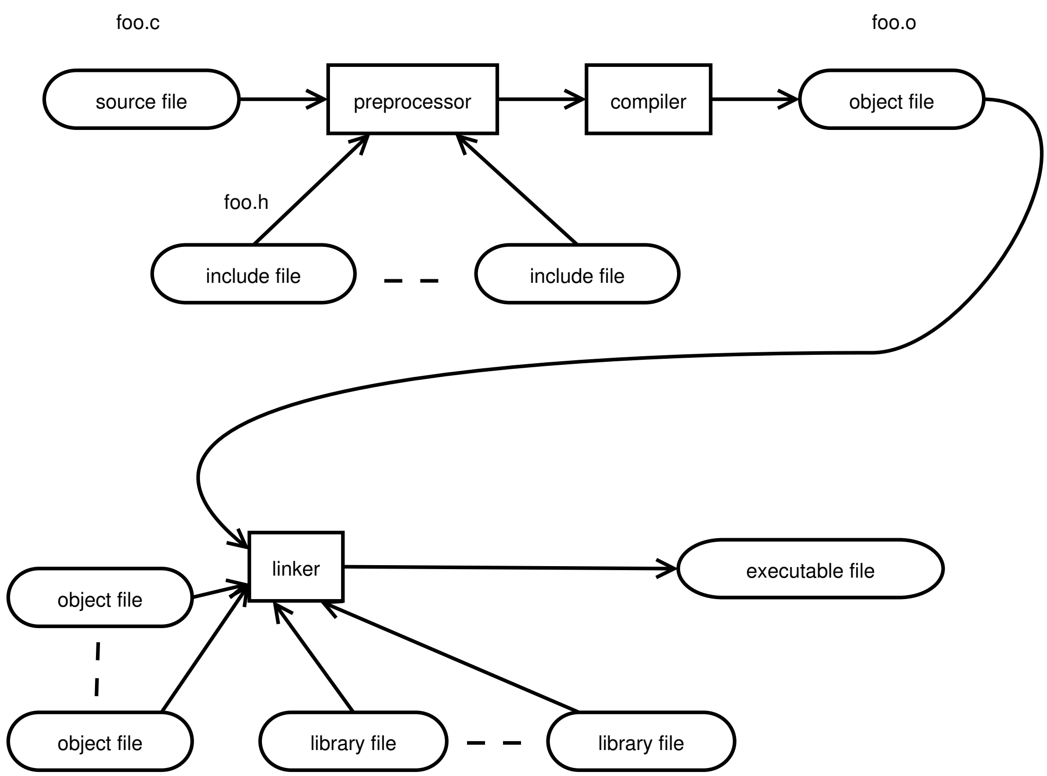
- When Make is executed, it tries to update a *goal* target. By default, this is the target of the first rule in the makefile.
- To do so, it (recursively) treats the goal's dependencies as targets, and tries to update them.
- When a target file is older that any of its dependency files, each command is executed, in order, in a separate subshell. Typically, a target file is rebuilt from all of its dependency files.
- The *older* relation between files is based on the time-of-last-modification of each file, maintained by the operating system.
- If a command fails, Make stops.

Unix/GCC Tool Chain (1 of 2)

- Our simple makefile can be improved.
- To do so, we first need to understand what's involved in translating source files into the end product.
- The "end product" might just be a single executable file, but it might be much more than that (e.g., for an embedded system).
- Additional complexity can also be due to programs that generate "source" files from other files (e.g., Flex, Bison, or home-grown tools).
- We call any generated file a *derived* file, whether it's intermediate or final.

Unix/GCC Toolchain (2 of 2)

- Here's a picture of a simple *toolchain* for a Unix/C/C++ development environment:



Building a Better Makefile (1 of 7)

- With our toolchain in mind, we can pursue the following goals.
- Recognize when a source file has been changed, and perform the minimum amount of work needed to generate up-to-date derived files.
- Eliminate or minimize redundancy within each rule, and across all rules.
- Automate dependency maintenance, for:
  - `#include` files
  - object files
- Provide convenience targets (e.g., `clean`).

Building a Better Makefile (2 of 7)

- Convenience targets are easy.
- For some, like `clean`, there is no file associated with the target. They are never up to date. Make always rebuilds them, but the commands never create a file with the same name as the target:

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

  The `.PHONY` "special target" tells Make to ignore a file named `clean`, if one is accidentally created.
- For others, there is a file associated with the target (e.g., an assembly file). We'll see some soon.

Building a Better Makefile (3 of 7)

- Pattern rules avoid redundancy between rules. There are two styles. I'll show you the new, more flexible, style.
- Rather than having one rule for each source file, specifying how to translate it into a derived file, we can use a single pattern rule for all of them.

Building a Better Makefile (4 of 7)

- For example:

```
%.o: %.c
        gcc -c $< $(flags)
```

- The `%` sign on the LHS of the colon is the "wildcard" character. It can match any substring of a target's name.
- The zero or more `%` signs on the RHS of the colon expands to what was matched by the `%` on the LHS of the colon.
- Make considers this rule when it needs to build an object file, and there is a corresponding C source file.
- In the commands, the "automatic variable" `$<` expands to the first dependency (i.e., the source file).

Building a Better Makefile (5 of 7)

- Of course, there are other automatic variables:
  - `$<` the name of the first dependency
  - `$@` the name of the target
  - `$^` the names of all the dependencies
  - `$*` basically, the wildcard string
- We can now eliminate more redundancy within a rule:

```
prog: prog.o util.o
        gcc -o $@ $^
```

Building a Better Makefile (6 of 7)

- Functions and variables help eliminate redundancy, and automate dependency maintenance.
- Let's compute the name of the executable file to be the name of its directory:
  `prog:=$(notdir $(PWD))`
- It likely depends on all of the object files, the names of which can be computed from the names of the source files:
  `objs:=$(subst .c,.o,$(wildcard *.c))`
  Why can't you expect `*.o` to work?
- There are many functions, as well as user-defined functions:
  - `dir`, `notdir`
  - `suffix`, `basename`
  - `addsuffix`, `addprefix`
  - `wildcard`
  - `foreach`
  - `call`, `eval`

Building a Better Makefile (7 of 7)

- Conveniently, GCC can help us automatically maintain `#include` dependencies:

```
%.o: %.c
        gcc -c $< -MMD
sinclude *.d
```

- The `-MMD` option causes GCC (actually, the preprocessor) to produce a `.d` file for each `.o` file, which contains lines like:

```
foo.o: foo.h
foo.o: bar.h
foo.o: zap.h
```

- They are “silently” included.

A Reusable Makefile (1 of 2)

- We can combine these improvements, arriving at the makefile in our `pub` directory:

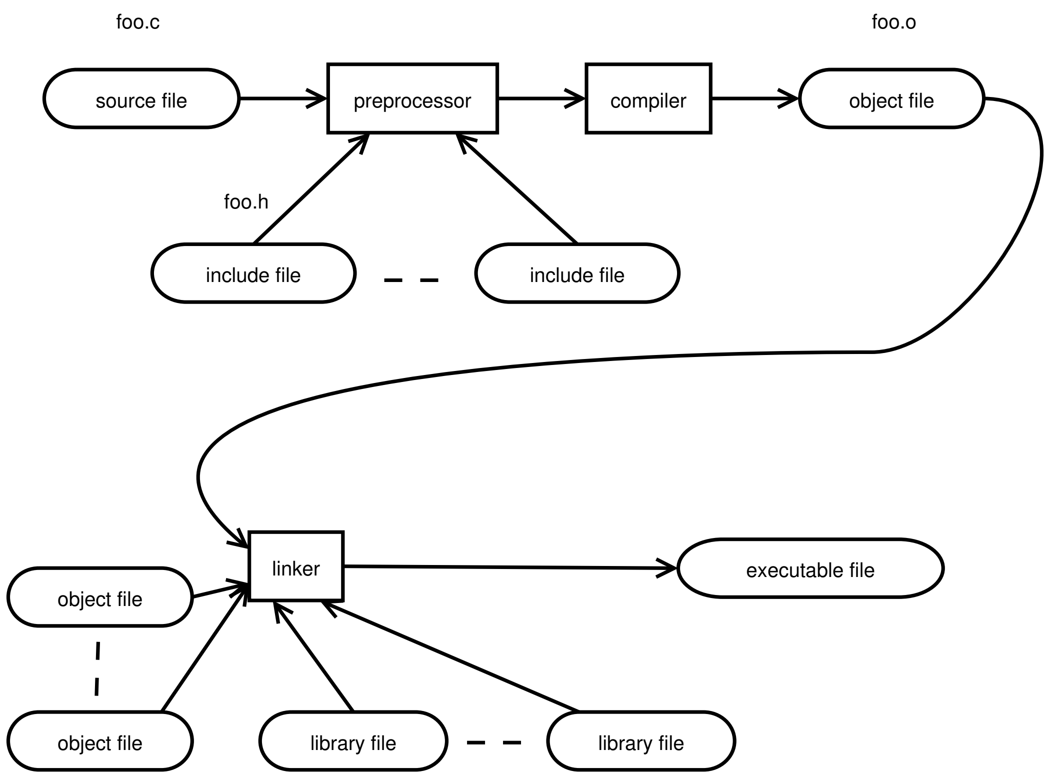
  `pub/GNUmakefile`
- You can create a symbolic link to it from a "project" directory.
- You can `include` it from a makefile in a "project" directory, adding to it and overriding parts of it. This is a bit like inheritance.

A Reusable Makefile (2 of 2)

- It computes the executable file's name from the basename of the current working directory.
- It computes the names of the object files from the names of the source files.
- It uses variables to record compiler and linker options.
- It records the compilation and linking process.
- It can produce preprocessor-output and assembly files.
- It records how to remove derived files.
- It records how to test the program.
- It uses GCC's `-MMD` option to compute include-file dependencies.

The C Preprocessor (1 of 2)

- We saw that the preprocessor, `cpp`, is at the front of the Unix/C/C++ toolchain:



- It operates on directives: source-file lines beginning with a pound sign. A directive must be a single line, but line continuation is okay.
- We've already seen some of these directives, but there are others.

The C Preprocessor (2 of 2)

- File inclusion: `#include`. Double quotes are for user files. Angle brackets are for system files.
- Macros: `#define` and `#undef`.
- Conditional inclusion: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif`. These also provide a syntax for determining whether a macro is defined or not (i.e., `defined()` and `!defined()`). Sometimes, `#error` is handy.
- Line control: `#line`.
- Compiler features: `#pragma`.

Macros

- A macro is a symbol, defined to have a value (e.g., `DOZEN`). A macro can take arguments, sort-of like a function, but there is no type checking.
- A macro can then be referenced, causing its definition to be expanded.
- Macros manipulate C tokens, although some string operations are supported.
- Macros can be tricky because argument passing is token substitution, rather than C's pass-by-value semantics:

  `pub/CppMacro/CppMacro.c`
- Macros allow you to do extra-linguistic things, like:

  `pub/CppAlloc/CppAlloc.c`

Conditional Inclusion (1 of 3)

- Conditional inclusion (e.g., `#if`) can prevent source code from being translated by the compiler proper. The compiler never even "sees" it.
- Perhaps, the source code *cannot* be compiled (e.g., it's for a different environment).
- Perhaps, a smaller executable file is important (e.g., it's only needed for debugging).

Conditional Inclusion (2 of 3)

- Conditional inclusion can be tricky because the semantics is different than C's `if` statement. For example:

  `pub/CppIfBug/GNUmakefile`

  `pub/CppIfBug/CppIfBug.c`
- GCC's `-D` option allows you to `#define` a macro at compile time.
- `#error` stops preprocessing.
- The problem is that `==` in a preprocessor expression compares integers, not strings, and an undefined macro evaluates to zero. The following technique does work:

  `pub/CppIf/GNUmakefile`

  `pub/CppIf/CppIf.c`

Conditional Inclusion (3 of 3)

- Conditional inclusion is often used to enable debugging code during development.

  `pub/CppDebug/GNUmakefile`

  `pub/CppDebug/CppDebug.c`

`#line` and `#pragma` Directives

- Most lines beginning with a pound sign are treated as comments by the compiler proper, but not these.
- The preprocessor adds `#line` directives to its output, so the compiler proper can keep track of source-file locations, for error messages and debugging.
- `#pragma` directives provide information to the compiler, controlling its behavior (e.g., for alignment).

Object Files and `binutils` (1 of 5)

- A compiler for a PL translates a source program into either: some sort of intermediate language (e.g., Java byte code), or assembly language.
- An assembly language can be fairly generic (ala, GCC), or CPU specific. Either way, the assembler generates CPU-specific object code, and stores it in an object file.
- An object file contains machine instructions, and a bunch of other stuff, as specified by a *format*. For execution, the machine instructions must be loaded into memory, according to the format.

Object Files and `binutils` (2 of 5)

- There are multiple object-file formats: a.out, COFF, ELF, etc. They are binary, non-text, formats. We use ELF:

  ```
  file /bin/cp
  ```

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

- For GCC development environments, object files are manipulated by `binutils` programs:

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

Object Files and `binutils` (3 of 5)

- What is the aforementioned “other stuff,” supported by typical object file formats? From Wikipedia:
  - header (descriptive and control information)
  - code segment (“text segment”, executable code)
  - data segment (initialized static variables)
  - read-only data segment (rodata, initialized static constants)
  - bss segment (uninitialized static data, both variables and constants)
  - external definitions and references for linking
  - relocation information
  - dynamic linking information
  - debugging information

Object Files and `binutils` (4 of 5)

- What does "uninitialized static data" mean? What does "bss" abbreviate?
  - Block Started by Symbol
  - Better Save Space
  - BS Section
- The ELF specification describes the content of a program file (i.e., *sections*), which describes the content of process memory (i.e., *segments*), for execution.

  `pub/etc/elf_link_vs_exec_view.jpg`

Object Files and `binutils` (5 of 5)

- Consider this simple program:

  `pub/Binutils/foo.c`

- We can compile it to assembly code, to see its sections:

  ```
  gcc -o foo.s -S foo.c
  grep ··· foo.s
  ```

- We can assemble it to object code, to see how sizes change:

  ```
  gcc -c foo.c
  size foo.o
  ```

- We can link it into an executable file, to see sizes:

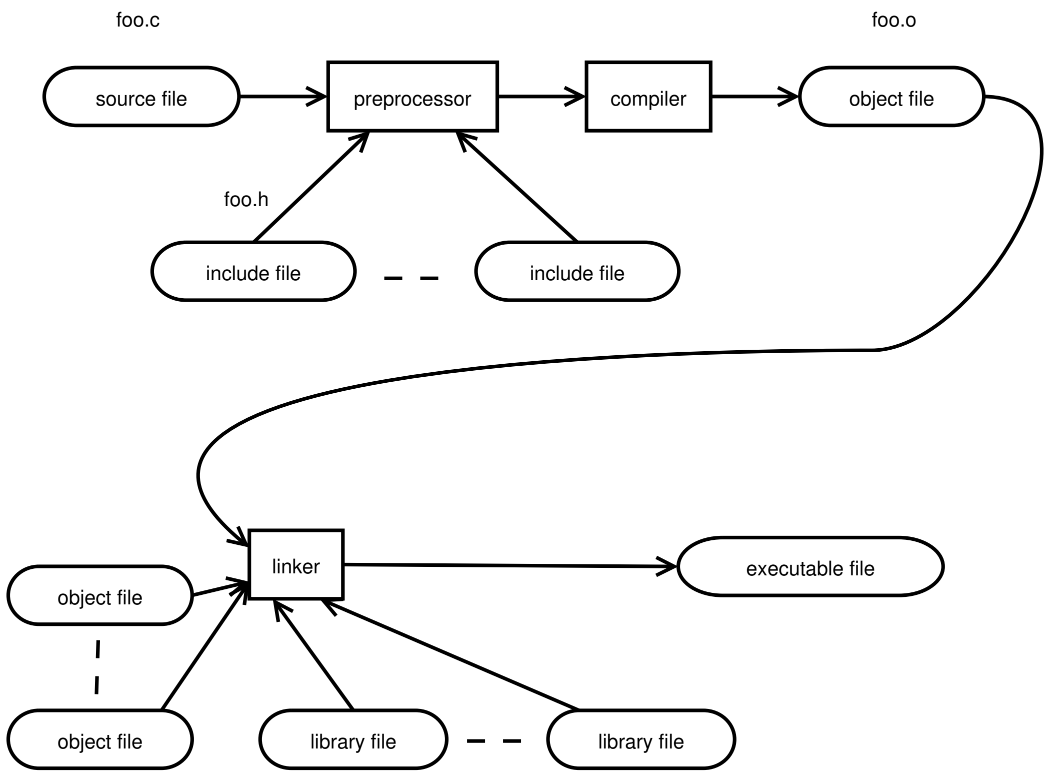
  ```
  gcc -o foo foo.c
  size -A foo
  ```

- Other handy tools:

  ```
  readelf -a foo.o
  objdump -h foo.o
  nm foo.o
  ```

Libraries (1 of 3)

- Object code can be put in a *library*, for easy access. An executable file can be *linked against* a library, to get the code.
- We saw that the linker, `ld`, is near the end of the Unix/C/C++ toolchain:



Libraries (2 of 3)

- The linker's job is to combine multiple object files into a single object file.
- The linker's input can be `.o` files, but also library files.
- The linker's output can be an executable file, but it can instead be a library file.
- A library is analogous to a Java `.jar` file. It's also much like a `.tar` or `.zip` file. However, a library is specialized for object code and linking.

Libraries (3 of 3)

- There are two kinds of library.
- When an executable file is linked against a *static* (aka, *archive*) library, object code is copied from the library into the file.
- When an executable file is linked against a *shared* (aka, *dynamic*) library, only a reference to the library is stored in the file. The library's object code is accessed later, when the executable file is executed.

Static Libraries (1 of 2)

- A static library is created by the program `ar`. It is essentially a set of object files and an index.
- During linking, a reference to a previously undefined symbol causes the object file defining that symbol to be extracted from the library.
- Here's an example:

  `pub/Libraries/TryString.c`

  `pub/Libraries/Make.static`

Static Libraries (2 of 2)

- `ar` options:

`r`	Replace library members.
`c`	Create the library.
`s`	Write an index.

- `gcc` options:

`-static`	Use static libraries, even if shared libraries exist.
`-L`	Add a library directory to the search path.
`-l`	Link against this library. Notice the abbreviated name.

- Only required object files are copied from the library to the executable.
- Symbol `gtNew` is in `GenTab.o` due to the definition. It is not in `TryString.o` or `TryString1` because it is not referenced.
- Symbol `newString` is in `TryString.o` and `TryString1` because it is referenced.

Shared Libraries (1 of 3)

- A shared library is created by `collect2`, a teammate of `ld`, whose agent is `gcc`.
- Regardless of how many currently executing programs (i.e., processes) are linked against a shared library, at most one instance of its code is in memory.
- Since each such process accesses the library's code at different addresses, the code must be compiled in a position-independent way (i.e., PIC), using relative addressing.
- A shared library's code is shared among processes, but if the library has static variables, each process has its own copy.

Shared Libraries (2 of 3)

- Here's an example:

  `pub/Libraries/Make.shared`

- `gcc` options:

`-fPIC`	Produce position-independent code, which is required for a shared library.
`-shared`	Produce a shared library, rather than an executable file.

- At run time, you can tell the dynamic loader where to search for libraries, via `LD_LIBRARY_PATH`.

Shared Libraries (3 of 3)

- Compare the sizes of the executable files:

```
ls -l TryString1 TryString2
```

- You can determine which shared libraries are required by an executable file:

```
readelf -d TryString1 | grep NEEDED
readelf -d TryString2 | grep NEEDED
```

- You can determine which shared libraries will actually be used by an executable (`ld` dependencies):

```
LD_LIBRARY_PATH=$PWD ldd TryString2
```

Plugins (1 of 3)

- In what we've seen, code from a shared library is loaded into memory, and linked with an executable file's code, automatically.
- This convenience is provided by the dynamic linker/loader, which itself is a shared library, named `ld-linux.so`.
- For more control, at run time, we can perform the loading and linking ourselves, with the functions: `dlopen`, `dlsym`, `dlerror`, and `dlclose`.

Plugins (2 of 3)

- A *plugin* is a shared library that can be loaded and unloaded, repeatedly, while a program is executing.
- Here is the source code for two plugins, which define a function named `plugin`:

  `pub/Plugins/plugin0.c`

  `pub/Plugins/plugin1.c`
- We can build the plugins with this script:

  `pub/Plugins/Make`

Plugins (3 of 3)

- When the executable file is created:
  - Don't mention the plugins.
  - Do mention `libdl.so`, the dynamic linker/loader.
- Calling the plugin's function is somewhat involved:

  `pub/Plugins/main.c`
- Many errors can occur.
- We toggle between plugins.
- Function `dlsym` returns a function pointer to the named function.

Systems Programming (1 of 2)

- Remind us again: What is systems programming? What is a systems program?
- These nebulous terms encompass: firmware, boot loaders, BIOSes, hypervisors, operating-system kernels, kernel modules, drivers, some libraries, development toolchains, debuggers, servers (aka, daemons), network utilities, etc.
- It does *not* include: text, document, spreadsheet, diagram, image, audio, or video viewers/editors; web browsers; or mail, news, or chat agents. It does not include what people call "applications."

Systems Programming (2 of 2)

- In addition to what we've already seen, we'll need to learn about: processes, threads, memory spaces, file descriptors, pipes, sockets, signals, interprocess synchronization (e.g., mutexes), etc.
- These topics cannot, effectively, be learned in isolation. They are interrelated.
- Some PLs provide these as intrinsic parts of the language. C does not. Like I/O, they are available via libraries.

Threads and Processes (1 of 7)

- Imagine a *very* primitive computer system:

  `pub/etc/proc.dia`

  - It has a CPU, with a small block of ROM, fast enough for code execution, starting at address 0x0000 (the reset address). The CPU also has a tiny amount of RAM.
  - It has an off-CPU block of RAM, fast enough for code execution.
  - It has an big off-CPU block of ROM, too slow for code execution.
- Initially, a program is stored in the fast ROM. Upon reset, it starts executing. It's variables are stored in the fast RAM.
- There is only one thread of control. There are no "processes," because the computer only does one thing. Whatever it does do, call it *A*.

Threads and Processes (2 of 7)

- Now, imagine that we want the computer to do *A* or *B*. However, the code to do both, won't fit in the fast ROM, only the slow ROM.
- We could write new code, name it *BL*, and store it in the fast ROM. Upon reset, *BL* would choose, copy either *A* or *B* to RAM, and jump to it.
- There is still only one thread of control, and no processes, because the computer only does one thing: either *A* or *B*.

Threads and Processes (3 of 7)

- Now, imagine that we want the computer to do A and B. What are our choices?
  1. We could *very carefully* combine the code to do A and B, name it AB, and store it in the slow ROM. Then, we could write new code, name it BL, and store it in the fast ROM. Upon reset, BL would copy AB to RAM, and jump to it. There is still one thread and zero processes.
  2. We could write new code, name it OS, and store it in the fast ROM. Upon reset, OS would copy A to RAM, and jump to it. At some point, A would jump back to OS, OS would save the state of A, copy B to RAM, and jump to it. At some point, B would jump back to OS, OS would save the state of B, and repeat, forever.

Threads and Processes (4 of 7)

- Our second choice is more scalable: for *C*, *D*, etc. However, we have opened a can of worms, a barrel of monkeys, or whatever.
- The “state” (aka, *context*) of *A* includes a snapshot of what *A* was about to do before it jumped back to *OS*: the CPU’s program-counter (PC) value and some of the CPU’s RAM contents. Same for *B*.
- Thus, there are two *threads*.
- Since *A* and *B* are assumed independent, they should not access each other’s variables, in non-CPU RAM. This should be enforced, somehow.
- Thus, there are also two processes. Let’s call them A and B. The stored *A*, in slow ROM is a *program*. The executing A, in RAM is a *process*.

Threads and Processes (5 of 7)

- Of course, *OS* is the operating system (OS). Since a process "jumps back" to the OS, we can say that the OS has a thread of execution, too. But, the OS is *not* a process.
- When should process A or B jump back to the OS?
  - When it wants to? This threatens *fairness* and *starvation*.
  - When it needs to? This suggest OS control of resources (e.g., I/O).
  - When the OS tells it to, somehow? This introduces *interrupts* and *preemption*.

Threads and Processes (6 of 7)

- Can the OS execute two or more instances of one program (e.g., two A processes)? Yes, if the OS keeps their contexts separate. In fact, the program's code can be shared by the processes, if we can assume its immutability.
- Can a process start a new process, increasing the number of processes? Which program should the new process be executing? This leads us to the `fork` and `exec` families of library functions and system calls.

Threads and Processes (7 of 7)

- We saw that the OS should prevent processes from interfering with each other (e.g., by isolating variable sets). What if two processes *want* to communicate?
- This opens the barn doors of *parallel processing*, *distributed processing*, *interprocess synchronization*, *interprocess communication*, and *networking*.
- We'll stick with pipes, signals, sockets, and mutexes.

Unix Processes (1 of 4)

- When a process is created, it is assigned a unique identifier, called a *PID*. A PID is an integer, assigned consecutively, starting with zero, up to some constant. When the constant is reached, assignments restart at zero. A PID is reused only if its process has exited. A process can get its PID.
- A *parent* process can create a *child* process. The parent can get its child's PID, and a child can get its parent's PID.
- One approach would be for the parent to choose a program for the child to execute. Not with Unix.

Unix Processes (2 of 4)

- A new child continues to execute *the same* program as its parent. However, the child has a *copy* of the parent's context and memory. The child is a "clone" of the parent: immediately after cloning, both execute the *same next (machine) instruction*.
- Since the parent knows the child's PID, and the child has no child (with a PID) of its own, their two threads can diverge, causing the processes to behave differently.
- The library function is named `fork`. The system call is named `clone`.

Unix Processes (3 of 4)

- A process can, at any time, start executing a different program. It can even reexecute the same program. Either way, its PID does not change.
- Of course, the new program has its own code, and variables, which become part of the process. However, open file descriptors (fds) are retained, and can be accessed by the new program's code.
- The family of library functions is generically named `exec`. The system call is named `execve`.

Unix Processes (4 of 4)

- A common pattern is:
  1. The parent manipulates fds.
  2. The parent forks a child.
  3. The parent manipulates fds.
  4. The parent returns to its task.
  5. The child manipulates fds.
  6. The child begins its task, perhaps executing a new program to do so.

Process Example (1 of 6)

- Suppose we want to develop a program, named `mpexec`, that takes at least one command-line argument.
- The first argument is the name of a program that reads its input from `stdin`, and writes its output to `stdout`.
- The remaining n arguments are file names.
- For example:

  ```
  mpexec wc foo bar
  ```

  Here, the first argument is `wc`, a word-counting program, and $n = 2$. We want `mpexec` to execute `wc` twice, in two child processes: one on the content of a file named `foo`; the other on `bar`.

Process Example (2 of 6)

- The child processes execute concurrently, likely on separate cores or processors.
- This is the default, and the most natural way to do it. Temporally serializing the processes actually requires more code.
- However, we want to avoid interleaving child output. Once the parent starts to write, to its `stdout`, the output of a particular child, the parent must finish with that child, before starting another.

Process Example (3 of 6)

- Here's the code:

  `pub/mpexec/mpexec.c`
- `main`, calls `start` for each file. The parent opens a pair of file descriptors (fds) for each child. A child writes its output to its fd. The parent reads that output from its fd, for that child.
- A pair of fds, connected in that way, is called a *pipe*. Data bytes flow through a pipe from the "write end" to the "read end." The ends of a pipe can be in separate processes.
- A pipe has capacity, via kernel buffering. Trying to read from an empty pipe, or trying to write to a full pipe, causes a process to block.

Process Example (4 of 6)

- For a given file, `start` creates a pipe with an open fd for each end, by calling `pipe`. It then calls `fork`, creating that file's child. Both parent and child have the pipe's fds.
- The parent saves the pipe's read end in its set of fds, and cleans up, avoiding an fd "leak."
- The child hooks the pipe's write end to its `stdout`, cleans up, and calls `exec` to execute the desired program (e.g., `wc`). When that program writes to its `stdout`, it unwittingly writes to the pipe.

Process Example (5 of 6)

- After all children are executing, `main` calls `finish`, to wait for a child, any child, to produce output. To wait for input, from any of a set of fds, call `select`.
- When `select` returns, data might be available from multiple fds. The `for` loop processes each fd. Function `cat` just copies the child's output to the parent's `stdout`.

Process Example (6 of 6)

- When a child exits, the kernel must save the child's exit status, for the parent. The kernel "signals" the parent that its child has exited, by calling a parent function, via a function pointer, provided by calling `signal`.
- In that "signal handler," any exited children are "reaped," by calling `waitpid`, which can also provide the child's exit status.
- Don't be disturbed by something like:

    After spawning children, a parent grimly waits, reaping them as they die, thereby preventing zombies.

Summary of Process (`fork`) Example

- Parent and children each have separate *code* memory spaces. Each can be executing different programs.
- Even if they are all executing the same program, they each have separate *data* memory spaces. They cannot access each other's variables.
- Interprocess synchronization and communication must employ other resources (e.g., pipes and signals).
- A parent resource leak might be very bad, especially if the parent is a long-lived server.
- A child leak is not so bad, because a child is likely to exit soon. The OS then deallocates all of the child's resources.

Segue from Processes to Threads (1 of 2)

- From the `fork(2)` man page:
  - The child process and the parent process run in separate memory spaces.
  - The child inherits copies of the parent's set of open file descriptors.
- From the `clone(2)` man page:
  - One use of `clone` is to implement threads: multiple flows of control in a program that run concurrently in a shared address space.
- You heard that right! In a single process, with a single PID, you can create multiple threads, each with their own TID (thread identifier).

Segue from Processes to Threads (2 of 2)

- We won't call `clone`, directly. We'll use an abstraction library, named Pthreads: POSIX Threads.
- All of the threads in a process share its memory. Sort of:
  - They share the code. They are all executing the same program.
  - They share static variables and `malloc`-ed memory.
  - Each thread has its own stack, for function parameters and local (`auto`) variables.

Thread Example (1 of 8)

- Suppose we want to develop our own word-counting program, named `mtwc`. It that takes one command-line argument: the number of threads to use for counting.
- For example:

  ```
  mtwc 16 < foo
  ```

- As before, for the most part, children threads execute concurrently, likely on separate cores, or maybe even separate processors. Shared memory favors cores.
- Again, this is the default. Serializing requires more code, as we shall see.

Thread Example (2 of 8)

- Input comes from `stdin`; output goes to `stdout`. As before, we must avoid some forms of interleaving, but now, both input and output must be synchronized.
- Children share not only the file itself, but the kernel's "open file description" and the process's "open file descriptor."
- In particular, this means they share a file offset (i.e., next-read pointer). So, they better be polite, and take turns reading.

Thread Example (3 of 8)

- With this in mind, we can divide the work, which is an important design decision.
- Each child thread will repeatedly: read a chunk of input, then count the chunk's words. When a thread is done, it contributes its count to the total.
- Here's the code:

  `pub/mtwc/mtwc.c`
- `main` calls `start`, for each child thread, passing an array of threads, rather than a set of fds. We could have called `malloc` to allocate the array.

Thread Example (4 of 8)

- All threads have also have access to two global variables: `eof` and `mutex`. Both could be local to `main`, but passing them to the children would be tedious.
- `eof` is the state of `stdin`.
- `mutex` is for *mutual exclusion*, a form of interprocess synchronization. A *mutex* is a *binary semaphore*, a variable with one of two possible values: unlocked or locked. By respecting a mutex, threads can politely take turns, sharing a resource.

Thread Example (5 of 8)

- Of course, a mutex *is* a shared resource. However, the kernel ensures serialized access to a mutex, with special machine instructions, and memory/cache trickery.
- An attempt to lock a locked mutex blocks, until it is unlocked.
- Then what??
  [This should raise many questions!]

Thread Example (6 of 8)

- Each time `start` is called, it creates a new thread, which (eventually) calls `count`, via a function pointer.
- When `count` is done, it will return its share of the word count to its parent.
- As designed, `count` repeatedly:
  - reads a chunk of input (i.e., a line) from `stdin`
  - counts the words in the line, according to a simple algorithm
  - updates its share of the count, which is stored in an (unshared) local variable
- Important: `count` locks the mutex, but *only* while it is modifying `stdin` and `eof`.

Thread Example (7 of 8)

- When `count` sees `eof` nonzero, it returns its share of the count to `thread_func`, which passes it to `pthread_exit`.
- By then, or soon, `main` has called `finish`, which loops through the children threads, calling `pthread_join` on each one.
- The parameter passed to `pthread_exit`, by a thread, is the return value of `pthread_join` for that thread.
- The per-thread word counts are totaled, then written to `stdout`, and we're done.

Thread Example (8 of 8)

- Questions:
  - What if the mutex is left locked?
  - How serious is a memory leak?
- From Wikipedia:

  > *The requirement of mutual exclusion was first identified and solved by Edsger W. Dijkstra in his seminal 1965 paper* Solution of a Problem in Concurrent Programming Control, *which is credited as the first topic in the study of concurrent algorithms.*

  Dijkstra was Dutch. `P()` and `V()` are part of his terminology.