1

The C Programming Language

The C Programming Language

"[S]nake oil has the most impressive names, otherwise you would be selling nothing, like 'Object Orientation'..."

— Edsger Dijkstra, *The Strengths of the Academic Enterprise*

Example

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]) {
4     printf("Hello, world!\n");
5
6     return 0;
7 }
```

The C Programming Language

- In C, the entry point is the function "`main`".
  - `main`'s arguments are the command line arguments; if they are unused, they may be declared as "`void`" instead.
  - `main`'s return value is an exit status; by convention, returning `0` indicates "no error".
- In C, comments begin with "`/*`" and end with "`*/`".

Example

Consider the following minimal, complete C program:

```
1 int main(void) {
2     /* Execution starts here. */
3
4     return 0;
5 }
```

Flag	Purpose
`-std=`*STD*	Specify the language standard
`-pedantic`	Enforce strict compliance with standards
`-Wall`, `-Wextra`	Display all warnings
`-Werror`	Treat all warnings as errors
`-o` *FILE*	Specify the executable filename
`-E`	Output the preprocessed source file
`-c`	Output the assembled object file

Example *(cont.)*

```
>$ gcc -Wall -Wextra -Werror -ansi -pedantic hello.c
>$ ./a.out
```

Makefiles

- A **Makefile** contains rules of the basic form:

TARGET : *DEPENDENCY* ...
COMMANDS...

  - If the target is nonexistent or older than a dependency, then the commands will be executed.
  - Note that the commands must be indented using tabs.

Example

```
1 a.out: hello.c
2     gcc -Wall -Wextra -Werror -ansi -pedantic hello.c
```

Example *(cont.)*

```
>$ make a.out
make: 'a.out' is up to date.
```

Types

- In C, there are 5 basic types:
  - `int`
  - `char`
  - `float`
  - `double`
  - `void`
- In C, basic types may have several modifiers, including:
  - `short`
  - `long`
  - `unsigned`
  - `const`
  - `static`
  - `extern`

Example

An `int` guarantees at least 16 bits, and is typically 32 bits; a `short int` is typically 16 bits; a `long int` guarantees 32.

Example

An `int` guarantees the range $\{-32767, \ldots, 32767\}$; an `unsigned long int` guarantees the range $\{0, \ldots, 4294967295\}$.

Variables

- In C, variables are statically typed and generally mutable.
  - A value may be cast to another type at runtime.
  - A variable may not be converted to another type at runtime.
- In C, uninitialized local variables are not zeroed out.

Example

Suppose `int x`. Then `x` evaluates to an unknown value.

Example

Suppose `int x = 0`. Then `x = "0"` fails to compile.

Example

Suppose `int x = 0.5`. Then `(double)x` evaluates to `0.0`.

Operators

Arithmetic Operator	C Syntax
Addition, Subtraction	+, -
Multiplication	*
Division	/
Modulus	%
Increment	++
Decrement	--

Example

Suppose `int x = 1` and `int y = 2`. Then `x / y` evaluates to 0.

Example

Suppose `int x = 1`. Then `x++` assigns `x = 2` and evaluates to 1.

Operators

Logical Operator	C Syntax
Logical "Not"	`!`
Logical "And"	`&&`
Logical "Or"	`||`
Ternary Expression	`p ? x : y`

(where zero is falsey; anything else is truthy)

Example

Suppose `int p = 0` and `int q = 3`. Then `p || q` evaluates to `1`.

Example

Suppose `int p = 0`. Then `p ? 1 : 2` evaluates to 2.

Operators

Relational Operator	C Syntax
"Equal"	==
"Not Equal"	!=
"Less Than (or Equal)"	<, <=
"Greater Than (or Equal)"	>, >=

Example

Suppose `int x = 1` and `int y = 2`. Then `x >= y` evaluates to 0.

Example

Suppose `int x = 1`. Then `0 < x && x < 2` evaluates to 1.

Operators

Bitwise Operator	C Syntax
Bitwise "Not"	`~`
Bitwise "And"	`&`
Bitwise "Or"	`|`
Bitwise "Exclusive Or"	`^`
Left Shift	`<<`
Right Shift	`>>`

Example

Suppose `int s = 1` and `int t = 2`. Then `s & t` evaluates to 0.

Example

Suppose `int s = 1`. Then `s << 2` evaluates to 4.

Control Flow

- In C, statements are terminated by semicolons.
  - Multiple statements may be collected into a block, delimited by curly braces, that can be treated as a single statement.
  - Local variables have block scope.
- In ANSI C, local variables must be declared after a left brace.

Example

```
1 int x = 1;                    /* x has value 1. */
2 {
3     x = 2;                    /* x has value 2. */
4     {
5         int x = 3;            /* x has value 3. */
6     }
7 }                             /* x has value 2. */
```

Control Flow

- In C, an "`if`" statement has the form:

```
if (CONDITION)
    STATEMENT
```

  - The body's indentation is not meaningful.
  - Any curly braces are not part of the conditional syntax.

Example

```
1 if (1) x = 2;
```

Example

```
1 if (1) {
2     x = 2;
3     y = 3;
4 }
```

Control Flow

- In C, an "`if…else`" statement has the form:

```
if (CONDITION)
    STATEMENT
else
    STATEMENT
```

  - The body of the "`else`" may be another "`if`" statement.

Example

```
1 if (x > y) {
2     z = x;
3 }
4 else if (x < y) {
5     z = y;
6 }
```

Control Flow

- In C, a "`while`" loop has the form:

  `while (`*CONDITION*`)`
  STATEMENT

  - A loop can be exited early with a "`break`" statement.
  - An iteration can be skipped with a "`continue`" statement.
  - Note that both apply only to the nearest enclosing loop.

Example

```
   int sum = 0, i = 0;

1  while (i < 10) {
2      sum += i;
3      i++;
4  }
```

Control Flow

- In C, a "`do…while`" loop has the form:

```
do
    STATEMENT
while (CONDITION);
```

  - The body will be executed before the condition is evaluated.
  - A terminating semicolon is required.

Example

```
   int sum = 0, i = 10;

1  do {
2      sum += i;
3      i++;
4  } while (i < 10);
```

Control Flow

- In C, a "`for`" loop has the form:

  `for (`*`INITIALIZATION`*`; `*`CONDITION`*`; `*`INCREMENT`*`)`
  `STATEMENT`

  - Any of the parenthesized expressions may be omitted.
  - An omitted condition is always considered truthy.
  - In ANSI C, variables may not be declared in the initialization.

Example

```
   int sum = 0, i;

1  for (i = 0; i < 10; i++) {
2      sum += i;
3  }
```

Input and Output

- In C, output is printed with the library function:

  `printf(`*STRING*`,` *EXPRESSION*`, …);`

  - The string must contain a format specifier for each expression.
  - "%d" for integers, "%c" for characters, "%f" for doubles...
  - Note that `printf` will not append a newline.
- `printf` is defined by the `stdio` library.

Example

```
#include <stdio.h>

int x = 5;
printf("x: %d\n", x);
```

Input and Output

- In C, input is scanned with the library function:

  `scanf(`*`STRING`*`, &`*`VARIABLE`*`, ...);`

  - The string must contain a format specifier for each variable.
  - "%d" for integers, "%c" for characters, "%f" for doubles...
  - Note that variables must typically be preceded by "&".
- `scanf` is defined by the `stdio` library.

Example

```
   #include <stdio.h>

1  int x;
2  printf("x? ");
3  scanf("%d", &x);
```

Functions

- In C, a function definition has the form:

```
TYPE NAME(PARAMETER, ...) {
    STATEMENTS...
}
```

  - The curly braces are part of the function definition syntax.
  - Unless the return type is declared as "`void`", control flow must always end with a "`return`" statement.

Example

```
1 int add(int x, int y) {
2     int z = x + y;
3
4     return z;
5 }
```

Functions

- In C, a function **prototype** has the form:

$$\textit{TYPE}\ \ \textit{NAME}(\textit{TYPE},\ \ldots);$$

- In C, a function call has the form:

$$\textit{NAME}(\textit{ARGUMENT},\ \ldots);$$

  - A function's prototype or its definition must precede any calls.
  - The compiler must know a function's name and type signature in order to call it, but will not look ahead for its definition.

Example *(cont.)*

```
   int add(int, int);

1  printf("add(1, 2): %d\n", add(1, 2));
2  printf("add(3, 4): %d\n", add(3, 4));
```

Header Files

- In C, copying a **header file**'s contents has one of the forms:

  `#include <PATH>`

  `#include "PATH"`

  - Angle brackets indicate a header in the standard library.
  - Double quotes indicate a header in the same directory.
  - Header files contain definitions to be shared among source files.

Example

```
1 #include <stdio.h>
```

Example

```
1 #include "add.h"
```

Macros

- In C, a **macro** has one of the forms:

  `#define` *TOKEN REPLACEMENT*

  `#define` *TOKEN*`(`*PARAMETER*`, …)` *REPLACEMENT*

  - Any occurrences of the token will be substituted with its replacement before compilation.
  - Note that substitution applies non-recursively, to full tokens occurring after macro definition and outside of strings.

Example

```
1 #define PI 3.14
```

Example

```
1 #define ADD(X, Y) ((X) + (Y))
```

The C Standard Library

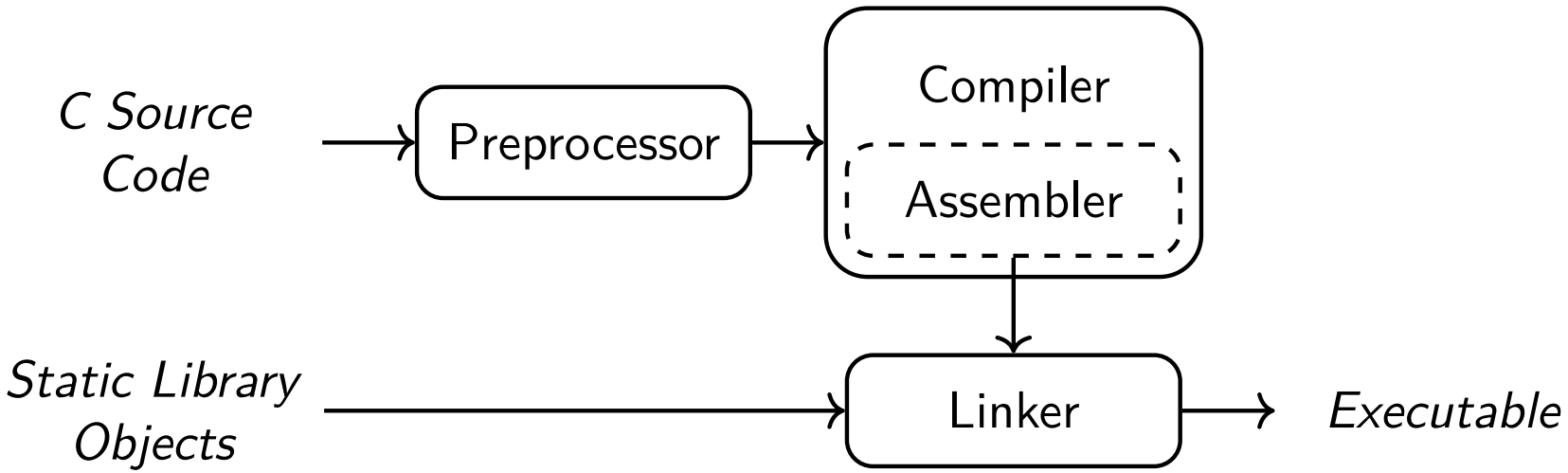
- The ANSI C standard library is declared by standard headers:
  - `assert.h`
  - `ctype.h`
  - `errno.h`
  - `float.h`
  - `limits.h`
  - `locale.h`
  - `math.h`
  - `setjmp.h`
  - `signal.h`
  - `stdarg.h`
  - `stddef.h`
  - `stdio.h`
  - `stdlib.h`
  - `string.h`
  - `time.h`
- Standard library headers may be included in any file.

Example

```
   #include <assert.h>
1  assert(add(1, 2) == 3);
2  assert(add(3, 4) == ADD(3, 4));
```

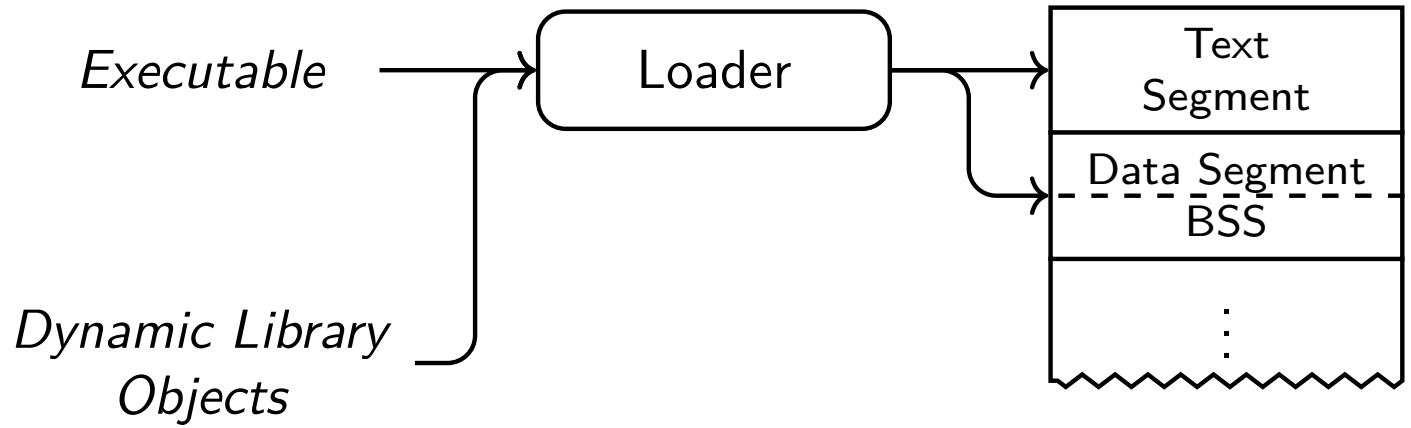
Pointers and Arrays

The C Compiler



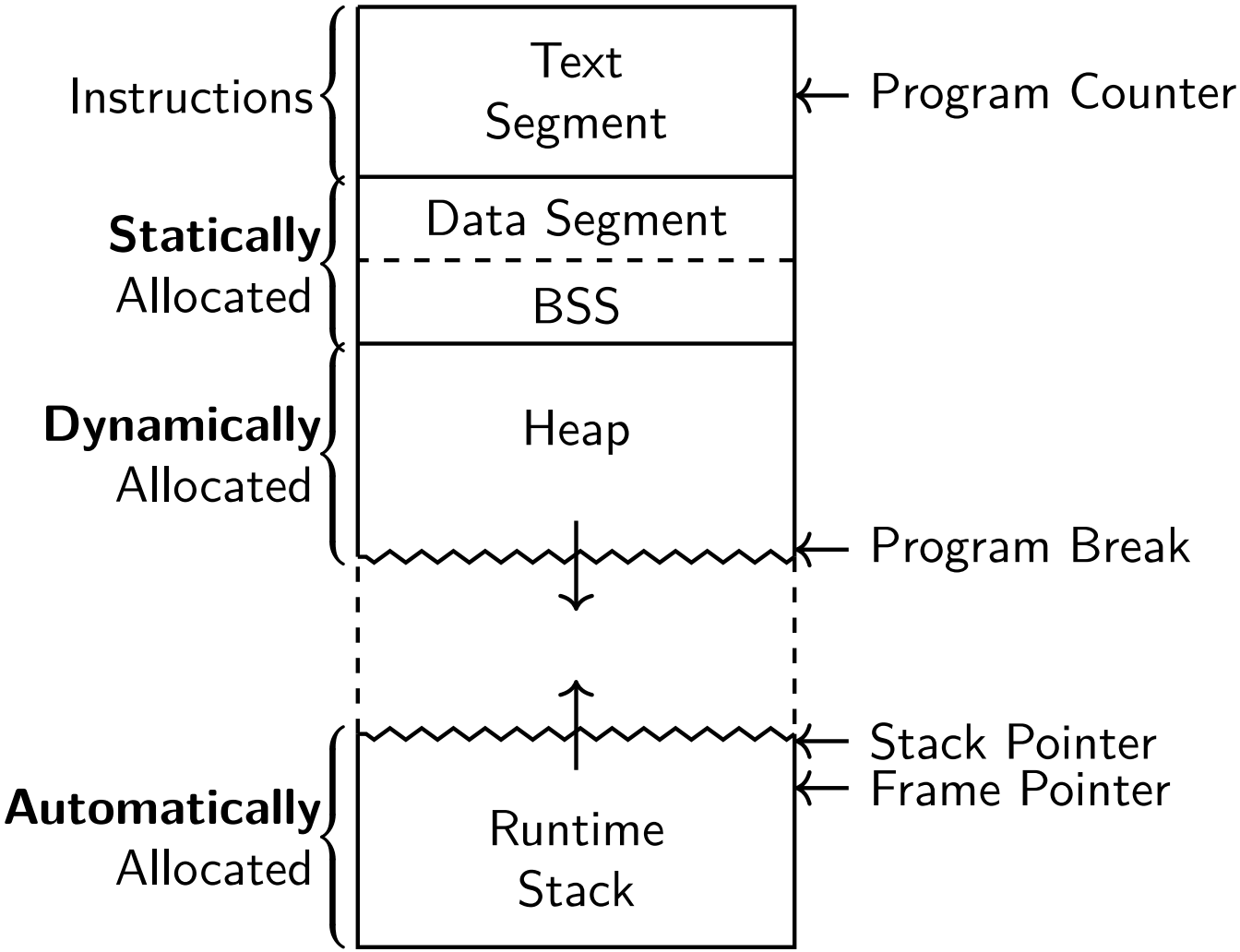
1. The **preprocessor** handles any directives for the compiler.
2. The **compiler** optimizes and translates C code into assembly or machine instructions, producing object files.
3. The **linker** combines object files into an executable.

The C Compiler



4. The **loader** reads executables from disk into memory:
   - Machine instructions are loaded into the **text segment**.
   - Initialized globals are loaded into the **data segment**.
   - Uninitialized globals are allocated in the **BSS**.
5. The hardware executes the instructions in the text segment.
   - Locals are allocated on the **runtime stack**.
   - Dynamic memory is allocated on the **heap**.

The C Compiler



The Runtime Stack

Definition

A **stack frame** contains data associated with a single function call.

- A stack frame stores data for an application, not a definition.
  - When a function is called, a stack frame is created and pushed.
  - Once a function call returns, its stack frame is popped.
- The **stack pointer** indicates the top-of-stack.

Example

A typical stack frame includes:

- Arguments
- Return value
- Local variables
- Return address
- Saved registers

Pointers

- Every variable contains a value stored in memory.
- Every location in memory has a unique address.
- Every address is itself a numerical value.

Definition

A **pointer** is a variable that contains the address of another variable.

- A pointer must be declared as pointing to a particular type.

Example

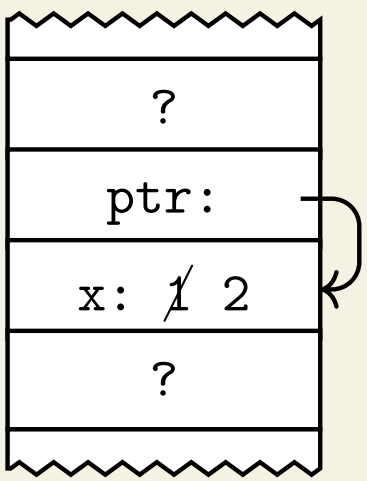
The declaration `int *ptr` declares a variable named "`ptr`" that may contain the address of an integer.

Pointers

- The unary "*" operator **dereferences** a pointer.
  - Dereferencing accesses the variable to which a pointer points.
  - Dereferencing may be used on the left-hand side of assignment.
- The unary "&" operator evaluates to the address of a variable.

Example

```
1 int x = 1, *ptr;
2
3 ptr = &x;
4 *ptr = 2;
```



Passing by Value

Definition

A programming language is **pass-by-value** if and only if functions are passed the values of their arguments.

- In pass-by-value, a function has a *copy* of its arguments.
- In C, arguments are always pass-by-value.

Example

```
1 void swap(int x, int y) {
2     int temp = x;
3     x = y;
4     y = temp;
5 }
```

Passing by Reference

Definition

A programming language is **pass-by-reference** if functions are passed references to their arguments.

- In pass-by-reference, a function *shares* its arguments.
- In C, a pointer is a variable whose value is a reference.

Example

```
1 void swap(int *x, int *y) {
2     int temp = *x;
3     *x = *y;
4     *y = temp;
5 }
```

Null Pointers

Definition

A **null pointer**, having the value `NULL`, is a pointer that is guaranteed to contain an invalid address for data.

- `NULL` is defined by `stddef.h` as *equivalent* to the integer 0.
  - Using a constant "`0`" as a pointer evaluates to a null pointer.
  - Comparing a null pointer to a constant "`0`" evaluates to true.
- By definition, a null pointer is equal to another null pointer, and is not equal to any valid pointer to data.

Example

`NULL` should be used to initialize a pointer that is not immediately pointed to data.

Segmentation Faults

Definition

A **segmentation fault**, or "segfault", occurs when a program attempts to access memory outside its allotted "segment".

- Often, segfaults are caused by dereferencing invalid pointers.

Example

```
1 int *ptr;
2 *ptr = 5;
```

Example

```
1 int *ptr = NULL;
2 *ptr = 5;
```

Segmentation Faults

- The **GNU Debugger** can examine a program's state as it runs.
  - GDB recommends first compiling with "-g".
  - GDB can step through code and print values.
  - GDB can provide a **stack trace** after a failure.

Example

```
>$ gcc -Wall -Wextra -Werror -ansi -pedantic -g swap.c
>$ gdb ./a.out
(gdb) run
Starting program: ./a.out
Program received signal SIGSEGV, Segmentation fault.
...
(gdb) bt
...
```

Arrays

- In C, an **array** is declared using one of the forms:

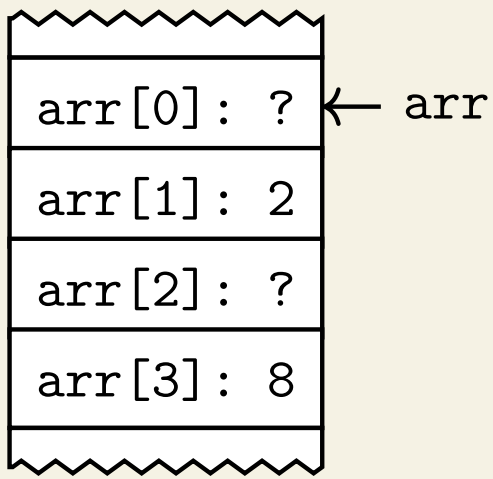
  TYPE *NAME*[*LENGTH*]

  TYPE *NAME*[] = {*ELEMENT*, …}

  - Every element in an array must have the same type.
  - Every array's length must be a compile-time constant.

Example

```
   #define LENGTH 4

1  int arr[LENGTH];
2
3  arr[1] = 2;
4  *(arr + 3) = 8;
```



Arrays as Pointers

- A reference to an array is the address of its first element.
- Indexing an array is offsetting and dereferencing its address.

Example

Suppose `int arr[4]` and `int *ptr = &arr[1]`. Then `ptr++` advances `ptr` to point to `arr[2]`.

Example

Suppose `int arr[4]` and `int *ptr = &arr[1]`. Then `arr < ptr` evaluates to `1`.

Example

Suppose `int arr[4]` and `int *ptr = &arr[1]`. Then `arr--` fails to compile.

Arrays as Arguments

- Since an array is an address, it **decays** into a pointer when passed as an argument.
- An array's length must be passed separately if needed.

Example

```
   void swap(int *, int *);

1  void reverse(int arr[], int len) {
2      int i;
3
4      for (i = 0; i < len / 2; i++) {
5          swap(arr + i, arr + (len - 1 - i));
6      }
7  }
```

Arrays as Return Values

- Since an array is an address, it is returned as a pointer; note that local arrays are deallocated upon returning.
- An array's elements are never copied on the stack.

Example

```
1 int *pair(int first, int second) {
2     int arr[2];
3
4     arr[0] = first;
5     arr[1] = second;
6
7     return arr;
8 }
```

Double Pointers

- Every variable has an address in memory.
- Every pointer is itself a variable with an address of its own.

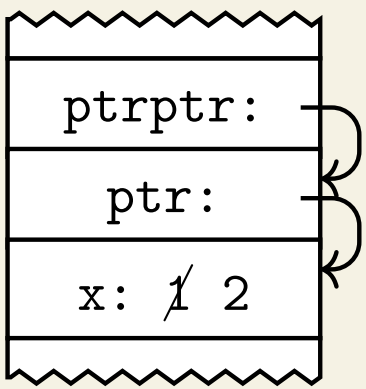
Definition

A **double pointer** is a pointer to a pointer to a (normal) variable.

- C supports arbitrarily many levels of pointers to pointers.

Example

```
1 int x = 1, *ptr, **ptrptr;
2
3 ptrptr = &ptr;
4 *ptrptr = &x;
5 **ptrptr = 2;
```

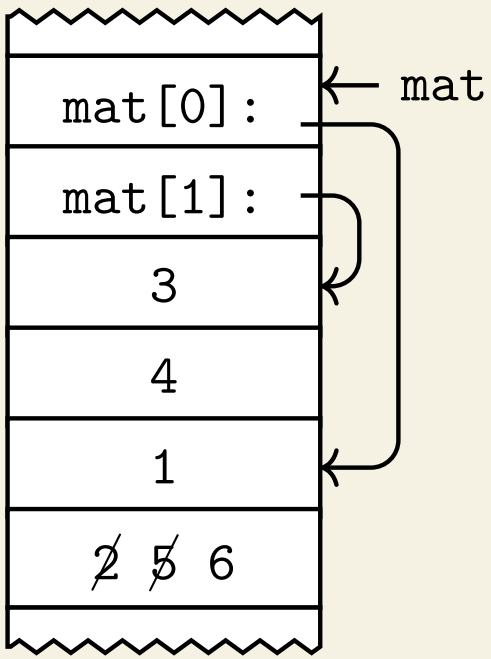


Arrays of Pointers

- Arrays of pointers are arrays whose elements are pointers.
  - The "outer" array is the address of the first "inner" pointer.
  - The "inner" pointers may themselves point to arrays.

Example

```
1 int r0[] = {1, 2}, r1[] = {3, 4};
2 int *mat[2];
3
4 mat[0] = r0;
5 mat[1] = r1;
6 mat[0][1] = 5;
7 *(*(mat + 0) + 1) = 6;
```

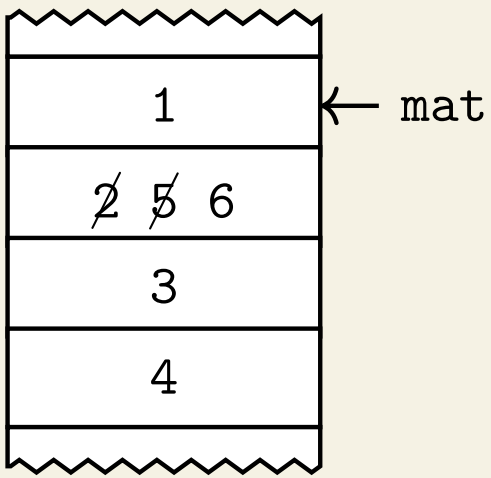


Multidimensional Arrays

- Multidimensional arrays are stored in **row-major order**.
  - The "outer" array is a contiguous block of "inner" arrays.
  - The "inner" arrays must all have the same length.

Example

```
1 int mat[][2] = {{1, 2}, {3, 4}};
2
3 mat[0][1] = 5;
4 *((int *)mat + (0 * 2) + 1) = 6;
```



Strings

- In C, a string is a null-terminated array of characters.
  - Strings may generally be declared as character arrays, which are stored on the runtime stack.
  - Literal strings may be declared as character pointers, in which case the string is typically stored in the read-only data segment.
- Literal strings are denoted with double quotes.

Example

Suppose `char str[] = "bar"`. Then `str[2] = 'y'` mutates the string to `"bay"`.

Example

Suppose `char *str = "bar"`. Then `str[2] = 'y'` is undefined and typically fails at runtime.

Function Pointers

- Every function is compiled into machine instructions.
- Every instruction is stored at an address in the text segment.

Definition

A **function pointer** is a pointer to a function.

- A function pointer must be declared as pointing to a function with a particular type signature.

Example

The declaration `int (*ptr)(int, int)` declares a variable named "`ptr`" that may contain the address of a function that takes two integer arguments and returns an integer.

Function Pointers

Example

```
1 int reduce(int arr[], int n, int (*fn)(int, int)) {
2     int val = arr[0], i;
3
4     for (i = 1; i < n; i++) {
5         val = (*fn)(val, arr[i]);
6     }
7
8     return val;
9 }
```

Dynamic Allocation

The Heap

- Statically allocated global data in the data segment must be known at compile-time.
- Automatically allocated local data on the runtime stack must be deallocated upon returning.
- Dynamically allocated data is stored on the **heap**; note that the "heap" is *not* a heap data structure.

Example

In Java, instantiated objects are automatically managed on the heap.

Example

In C, the heap is manually managed with functions `malloc`, `free`.

Dynamic Allocation

- In C, memory is allocated with the library function:

  `malloc(`*SIZE*`);`

  - The size must be given as a number of bytes.
  - The `sizeof` operator evaluates to the bytes required by a type.
  - `malloc` returns a `void` **pointer** to a block on the heap.
- `malloc` is defined by the `stdlib` library.

Example

```
#include <stdlib.h>

int *ptr = (int *)malloc(sizeof(int) * 2);
```

Dynamic Allocation

- Since an array is an address, a pointer can be indexed as though it were an array.
- A pointer to an address on the heap can be safely returned.

Example

```
1 int *pair(int first, int second) {
2     int *arr = (int *)malloc(sizeof(int) * 2);
3
4     arr[0] = first;
5     arr[1] = second;
6
7     return arr;
8 }
```

Dynamic Deallocation

- In C, memory is deallocated with the library function:

  `free(`*`POINTER`*`);`

  - The pointer must point to dynamically allocated memory.
  - If the pointer is `NULL`, `free` does nothing.
  - `free` takes a `void` pointer to a block on the heap.
- `free` is defined by the `stdlib` library.

Example *(cont.)*

```
#include <stdlib.h>

free(ptr);
```

Memory Leaks

Definition

A **memory leak** occurs when a program fails to deallocate memory that it is no longer using.

- Often, memory leaks do not immediately cause a fatal error.

Example *(cont.)*

```
1 int main(void) {
2     int *arr;
3
4     arr = pair(1, 2);
5
6     return 0;
7 }
```

Memory Leaks

- **Valgrind** can analyze memory usage and detect leaks.
  - Valgrind recommends first compiling with "-g".
  - Valgrind identifies where leaked memory originated.
  - Note that modern OSs will free such memory on termination.

Example

```
>$ gcc -Wall -Werror -Wextra -ansi -pedantic -g pair.c
>$ valgrind --leak-check=full ./a.out
...
8 bytes in 1 blocks are definitely lost
   at 0x4C29F73: malloc (vg_replace_malloc.c:309)
   by 0x400614: pair (pair.c:17)
   by 0x400598: main (pair.c:9)
...
```

Dynamic Reallocation

- In C, memory is reallocated with the library function:

  `realloc(`*`POINTER`*`, `*`SIZE`*`);`

  - The pointer must point to dynamically allocated memory.
  - If the pointer is `NULL`, `realloc` is identical to `malloc`.
  - `realloc` may return a new `void` pointer to a new block.
- A `realloc` is essentially a `malloc`, a `memcpy`, and a `free`.

Example

```
   #include <stdlib.h>

1  int *ptr = (int *)malloc(sizeof(int) * 2);
2  ptr = (int *)realloc(ptr, sizeof(int) * 4);
3  free(ptr);
```

Structures

Structures

Definition

A **structure** defines a new type by composing variables.

- Structures include only variables, not their associated functions.
  - All structures are classes.
  - Not all classes are structures.
- Note that a variable in a structure may be a function pointer.

Example

```
1 /* The elements of a pair: */
2 int first;
  ...
3 int second;
```

Structures

- In C, a **structure definition** has the form:

```
struct NAME {
    MEMBER
    ...
};
```

  - The **members** of a structure may have differing types.
  - Member declarations may not include initial values.

Example *(cont.)*

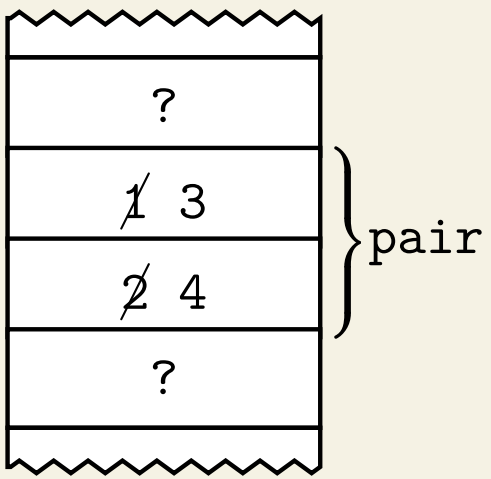
```
1 struct Pair {
2     int first;
3     int second;
4 };
```

Structures

- The "." operator accesses a member variable.
  - A structure is stored as a contiguous blocks of memory.
  - Accessing a member is offsetting within its structure's memory.
- ".", "=", and "&" are the only valid operations on structures.

Example *(cont.)*

```
1 struct Pair pair = {1, 2};
2
3 pair.first = 3;
4 pair.second = 4;
```



Type Definitions

- In C, a **type definition** has the form:

  `typedef` *TYPE NAME*`;`

  - Like a macro, a type definition effectively substitutes one fragment of code for another.
  - Unlike a macro, a type definition only applies to types and is handled by the compiler instead of the preprocessor.

Example

```
1 typedef struct Pair Pair;
```

Example

```
1 typedef struct Pair {
2     int first, second;
3 } Pair;
```

Structure Pointers

Example

```
1 Pair initp(Pair pair) {
2     pair.first = 0;
3     pair.second = 0;
4
5     return pair;
6 }
```

Example

```
1 void initp(Pair *pair) {
2     (*pair).first = 0;
3     (*pair).second = 0;
4 }
```

Structure Pointers

- Note that "." has higher precedence than "*".
  - "`*structure.member`" equates to "`*(structure.member)`".
  - "`structure->member`" equates to "`(*structure).member`".
- Note that member variables may themselves be pointers.
  - "`*(structure.member)`" dereferences a pointer *in* a structure.
  - "`(*structure).member`" dereferences a pointer *to* a structure.

Example *(cont.)*

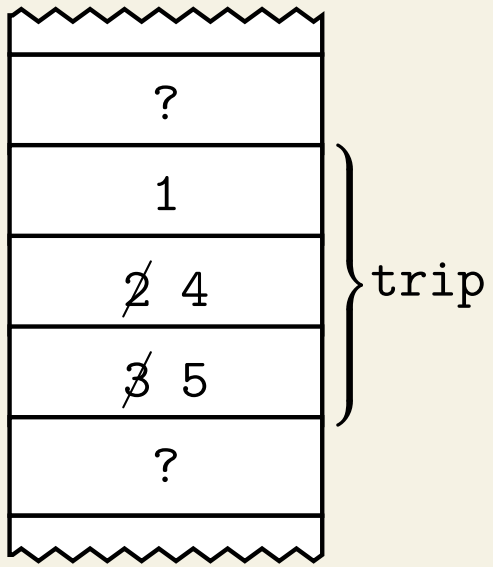
```
1 void initp(Pair *pair) {
2     pair->first = 0;
3     pair->second = 0;
4 }
```

Nested Structures

- A structure may contain structures of other types.
- A structure may not be defined before its members' types.

Example

```
   typedef struct Trip {
       Pair pair;
       int third;
   } Trip;

1  Trip trip = {{1, 2}, 3};
2
3  trip.pair.second = 4;
4  trip.third = 5;
```



Recursive Structures

- A structure may not contain a structure of its own type.
- A structure may contain a *pointer* to one of its own type.

Example

```
typedef struct Node {
    void *val;
    struct Node *next;
} Node;

Node *head = (Node *)malloc(sizeof(Node));

head->val = &x;
head->next = NULL;
```

64

The UNIX Environment

Operating Systems

"[T]he interrupt mechanism turned the computer into a nondeterministic machine...and could we control such a beast?"

— Edsger Dijkstra, *Recollections of Operating System Design*

Definition

An **operating system** is a program that manages the hardware resources allocated to application software.

- Operating systems must manage resources such as:
  - CPU time
  - Privileged instructions
  - Memory segments
  - Files and directories
  - Interprocess communications
  - I/O devices

System Calls

Definition

A **system call** allows a program to request a service from the operating system.

- System calls protect programmers from having to know the details of system resources.
- System calls protect system resources from malicious or incompetent programmers.

Example

`write` requests that the operating system write bytes to a file.

Example

`printf` calls `write` in order to write text to the terminal.

System Calls

- Often, system calls are exposed by assembly instructions or C functions, which temporarily transfer control to the OS.
- In UNIX, these are documented by Section 2 of the **manpages**.

Example

```
>$ man 2 write
```

- Often, system calls are abstracted by standard library functions, which encapsulate the most commonly desired functionality.
- In UNIX, these are documented by Section 3 of the manpages.

Example

```
>$ man 3 printf
```

Unbuffered I/O

- In UNIX, files are opened with the system call:

  `open(`*`PATH`*`, `*`OPTIONS`*`);`

  `open(`*`PATH`*`, `*`OPTIONS`*`, `*`PERMISSIONS`*`);`

  - The options must be a **bitmask** of one or more flags.
  - The permissions must be specified iff the file is created.
  - `open` returns an integer **file descriptor**.

Example

```
   #include <fcntl.h>
   
1  int fd = open(
2   "hello.txt", O_WRONLY | O_CREAT | O_TRUNC,
3   S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
```

Unbuffered I/O

- In UNIX, open files are accessed with the system calls:

  `read(`*`DESCRIPTOR`*`, `*`BUFFER`*`, `*`SIZE`*`);`

  `write(`*`DESCRIPTOR`*`, `*`BUFFER`*`, `*`SIZE`*`);`

  - The buffer must be a pointer to pre-allocated memory.
  - `read` and `write` advance the offset within a file.
  - `read` and `write` return the number of bytes read or written.

Example *(cont.)*

```
   #include <unistd.h>

1  char *buf = "Hello, world!\n";
2  write(fd, buf, sizeof(char) * 14);
```

Unbuffered I/O

- In UNIX, open files are managed with the system calls:

  `lseek(`*`DESCRIPTOR`*`, `*`OFFSET`*`, `*`WHENCE`*`);`

  `close(`*`DESCRIPTOR`*`);`

  - `lseek` repositions the offset within a file.
  - `close` releases a file descriptor for later reuse.
  - Note that there is a finite number of file descriptors available.

Example *(cont.)*

```
#include <unistd.h>

close(fd);
```

Buffered I/O

- In C, files are opened with the library functions:

  `fopen(`*`PATH`*`, `*`MODE`*`);`

  `freopen(`*`PATH`*`, `*`MODE`*`, `*`STREAM`*`);`

  - The mode must be a string containing one or more flags.
  - All created files will have permissions `rw-rw-rw-`.
  - `fopen` returns a `FILE` pointer associated with a **stream**.

Example

```
   #include <stdio.h>
1
1  FILE *file = fopen("hello.txt", "w");
```

Buffered I/O

- In C, open streams are accessed with the library functions:

  `fread(`*BUFFER*`, `*SIZE*`, `*QUANTITY*`, `*STREAM*`);`

  `fwrite(`*BUFFER*`, `*SIZE*`, `*QUANTITY*`, `*STREAM*`);`

  - The buffer must be a pointer to pre-allocated memory.
  - `fread` and `fwrite` advance the offset within a file.
  - `fread` and `fwrite` return the number of bytes read or written.

Example *(cont.)*

```
   #include <stdio.h>

1  char *buf = "Hello, world!\n";
2  fwrite(buf, sizeof(char), 14, file);
```

Buffered I/O

- In C, open streams are managed with the library functions:

  `fseek(`*STREAM*`, `*OFFSET*`, `*WHENCE*`);`

  `fclose(`*STREAM*`);`

  - `fseek` repositions the offset within a file.
  - `fclose` deallocates a `FILE` and dissociates its stream.
  - Failing to close a file will cause a memory leak.

Example *(cont.)*

```
#include <stdio.h>

fclose(file);
```

File Streams

- Every process begins with 3 open file streams:
  - `stdin` (file descriptor `STDIN_FILENO`, typically `0`)
  - `stdout` (file descriptor `STDOUT_FILENO`, typically `1`)
  - `stderr` (file descriptor `STDERR_FILENO`, typically `2`)
- Each stream is typically **buffered** in order to limit system calls.

Example

Reading 10000 bytes from a file, 1 byte at a time, requires 10000 calls to `read`.

Example

Reading 10000 bytes from a file into a buffer, 4096 bytes at a time, and iterating over them requires 3 calls to `read`.

Bitpacking

- Bits can be "packed" into bytes using bitwise operations.
  - A bit can be set to '1' using a bitwise "or" with '1'.
  - A bit can be set to '0' using a bitwise "and" with '0'.
  - A bit can be flipped using a bitwise "exclusive or" with '1'.
  - A bit can be extracted using a bitwise "and" with '1'.
- Some values require numbers of bits that are not multiples of 8.

Example

`O_WRONLY` and `O_CREAT` are bitmasks, and `O_WRONLY | O_CREAT` evaluates to an integer with their corresponding bits set to '1'.

Example

`mode_t mode` is a bit field, and `mode & S_IRUSR` evaluates to a non-zero integer if and only if the corresponding bit is set to '1'.

Bitpacking

Example

```
 1 unsigned char stob(char *bits) {
 2     unsigned char byte = 0, mask;
 3
 4     for (mask = 1 << 7; mask > 0; mask >>= 1) {
 5         if (*(bits++) == '1') {
 6             byte |= mask;
 7         }
 8     }
 9
10     return byte;
11 }
```

Error Numbers

- System calls will set an **error number** to indicate failure.
  - The global variable `errno` contains the number of the last error.
  - The function `perror` prints to `stderr` based on `errno`.
- Programs ought to return `EXIT_FAILURE` to indicate failure.

Example

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

1  FILE *file;
2  if ((file = fopen("dne.txt", "r")) == NULL) {
3      perror("fopen");
4      exit(EXIT_FAILURE);
5  }
```

Files and Directories

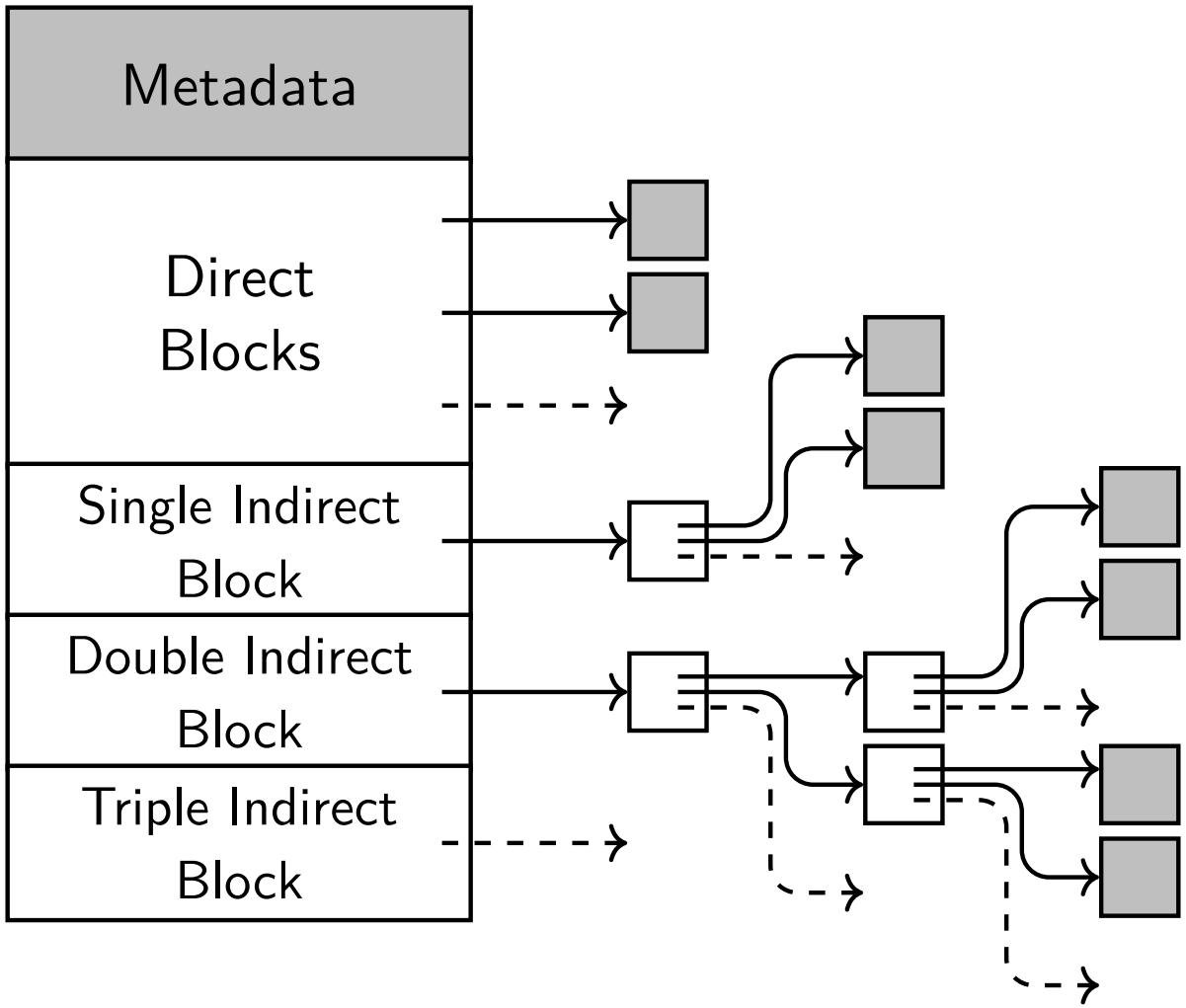
Files

Definition

A **file** is a sequence of bytes that can be read from and written to a **file system** on nonvolatile storage devices.

- In UNIX, data is stored on disk in one or more **blocks**.
  - A direct block contains a fixed amount of a file's data.
  - An indirect block contains locations of direct blocks.
  - Note that there will be a finite number of blocks available.
- In UNIX, each file is associated with exactly one **inode**.
  - Each storage device is identified by a unique integer.
  - Each inode is identified by an integer, unique within its device.
  - Note that there will be a finite number of inodes available.

Files


Metadata
Direct Blocks
Single Indirect Block
Double Indirect Block
Triple Indirect Block

Files

- In UNIX, files are inspected with the system calls:

```
stat(PATH, BUFFER);
lstat(PATH, BUFFER);
fstat(DESCRIPTOR, BUFFER);
```

  - The buffer must be a `struct stat` pointer.
  - Note that `stat` will follow symbolic links; `lstat` will not.

Example

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

1 struct stat buf;
2 stat("test.txt", &buf);
```

Directories

Definition

A **directory** is a file containing a mapping of filenames to inodes.

- Note that inodes do not contain filenames; they are associated with filenames within their parent directories.
- Every process has a **current working directory**; in UNIX, it can be changed using the system call `chdir`.

Example

```
>$ ls -ai testdir
 67205332 .          67205326 ..
119230730 bar.txt   119230731 foo.txt
119230732 subdir
```

Directories

- In UNIX, directories are traversed with the library functions:

`opendir(`*PATH*`);`

`readdir(`*DIRECTORY*`);`

`closedir(`*DIRECTORY*`);`

  - `readdir` returns the next `struct dirent` pointer.
  - Entries are only guaranteed to have inode numbers, filenames.

Example

```
   #include <dirent.h>

1  DIR *dir;
2  struct dirent *entry;
3  dir = opendir("testdir");
4  entry = readdir(dir);
```

Links

Definition

A **hard link** is a filename-inode mapping in a directory.

Definition

A **soft link** is a file containing a path to another file.

- Multiple filenames may be mapped to the same inode.
- If no filenames are mapped to an inode, and no process has its file open, then it will be removed.

Example

```
>$ ln foo.txt tmp.txt
>$ ls -i foo.txt tmp.txt
119230731 foo.txt  119230731 tmp.txt
```

Links

- In UNIX, links are managed with the system calls:

  `link(`*PATH*`, `*NAME*`);`

  `symlink(`*PATH*`, `*NAME*`);`

  `unlink(`*PATH*`);`

  - `link` creates a hard link; `symlink` creates a soft link.
  - Note that users may not create hard links to directories.

Example

```
   #include <unistd.h>
1
1  link("foo.txt", "tmp.txt");
2  unlink("tmp.txt");
```

Permissions

- In UNIX, every file has an **owner** and a **group**.
  - The owner is typically the file's creator.
  - The group is that of either the creator or the parent directory.
- In UNIX, every file has **read**, **write**, and **execute** permissions.
  - Each file has permission fields for owner, group, and other.
  - The **superuser**, "`root`", has all permissions to all files.
- Each permission field is represented as an octal digit.

Example

```
>$ chmod 644 foo.txt
>$ ls -l foo.txt
-rw-r--r-- 1 cesiu domain users 5 Nov  4 15:39 foo.txt
```

Permissions

- A directory's permissions are used when accessing its entries.
  - To read from a directory is to list its entries' filenames.
  - To write to a directory is to modify its entries.
  - To execute a directory is to search for an entry by filename.
- A soft link's permissions are typically meaningless.

Example

```
>$ chmod 500 testdir
>$ ls testdir
bar.txt  foo.txt  subdir
>$ touch testdir/bay.txt
touch: cannot touch 'testdir/bay.txt': Permission denied
>$ rm testdir/foo.txt
rm: cannot remove 'testdir/foo.txt': Permission denied
```

Permissions

- In UNIX, permissions are managed with the system calls:

  `chmod(`*PATH*`, `*PERMISSIONS*`);`

  `chown(`*PATH*`, `*OWNER*`, `*GROUP*`);`

  - Only the owner or `root` may change a file's permissions.
  - Only the owner may change a file's group to one of their own.
  - Only `root` may change a file's owner, or its group arbitrarily.

Example

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

chmod("test.txt", S_IRWXU);
```

Processes

Processes

Definition

A **process** is a running instance of a program.

- Every process has a unique **process identifier**.
  - While a process runs, no other process has the same PID.
  - Once a process terminates, its PID may be reused.
- Almost every process has an invoking **parent** process.

Example

In UNIX, the kernel's system process has PID `0` and no parent.

Example

In UNIX, the `init` process has PID `1` and PPID `0`.

- In UNIX, process identifiers are retrieved with the system calls:

```
getpid();
getppid();
```

  - Both `getpid` and `getppid` always return an integer PID.
  - Neither `getpid` nor `getppid` set any error numbers.

Example

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

1 char fname[13];
2 FILE *tmpfile;
3 snprintf(fname, 13, "tmp%05d.txt", getpid());
4 tmpfile = fopen(fname, "w");
```

Process Creation

- In UNIX, child processes are created with the system call:

```
fork();
```

  - `fork` duplicates the (now-parent) process.
  - In the parent process, `fork` returns the child's PID.
  - In the child process, `fork` returns 0.

Example

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

1 if (!fork()) {
2     printf("Process %d is the child.\n", getpid());
3     exit(94);
4 }
```

Process Termination

- In UNIX, parents wait for children with the system calls:

  `wait(`*`STATUS`*`);`

  `waitpid(`*`PID`*`, `*`STATUS`*`, `*`OPTIONS`*`);`

  - The status must be an integer pointer, or `NULL`.
  - If the PID is `-1`, `waitpid` waits for any child.
  - If the PID is positive, `waitpid` waits for that child.

Example *(cont.)*

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

else {
    int status;
    printf("Process %d is the parent.\n", getpid());
    wait(&status);
```

Process Termination

- In UNIX, a child's exit status is tested with the basic macros:

  `WIFEXITED(`*STATUS*`);`

  `WEXITSTATUS(`*STATUS*`);`

  - `WIFEXITED` evaluates to true iff the child terminated normally.
  - `WEXITSTATUS` then evaluates to the child's exit status.

Example *(cont.)*

```
 9 ||     if (WIFEXITED(status)) {
10 ||         printf("Child exited with status %d.\n",
11 ||                WEXITSTATUS(status));
12 ||     }
13 ||
14 ||     exit(0);
15 || }
```

Process Termination

- When a child terminates, the **zombie** process's resources are not completely deallocated until its parent waits for it.
- If a parent terminates without waiting for a child, the **orphan** process will be adopted by `init`, which will wait for it.

Example

```
>$ ps u
  PID %CPU %MEM STAT START   TIME COMMAND
10130  0.0  0.0 Ss   15:52   0:00 -bash
10456  0.0  0.0 T    15:56   0:00 ./a.out
10457  0.0  0.0 Z    15:56   0:00 [a.out] <defunct>
10458  0.0  0.0 Z    15:56   0:00 [a.out] <defunct>
10494  0.0  0.0 R+   15:57   0:00 ps u
```

Process Replacement

- In UNIX, a process is replaced with the library functions:

  `execlp(`*PATH*`, `*ARGUMENT*`, …, NULL);`

  `execvp(`*PATH*`, `*ARGUMENTS*`);`

  - The path must identify an executable file.
  - `execlp` takes one or more command line arguments.
  - `execvp` takes an array of command line arguments.

Example

```
   #include <unistd.h>

1  if (!fork()) {
2      execlp("ls", "ls", "-a", ".", NULL);
3      exit(EXIT_FAILURE);
4  }
```

Process Groups

- In UNIX, **process groups** are managed with the system calls:

  `getpgid(`*PID*`);`

  `setpgid(`*PID*`, `*PGID*`);`

  - Every process group has a **leader** whose PID is its PGID.
  - Child processes inherit their parents' PGIDs.
  - A process may only change the PGID of itself or its children.

Example

```
   #include <unistd.h>

1  if (!fork()) {
2      setpgid(0, 0);
3      execlp("ls", "ls", "-a", ".", NULL);
4      exit(EXIT_FAILURE);
5  }
```

Resource Limits

- In UNIX, resource limits are managed with the system calls:

```
getrlimit(RESOURCE, LIMIT);
setrlimit(RESOURCE, LIMIT);
```

  - A process may temporarily change a resource's "soft limit" to any value below its "hard limit".
  - A process may permanently lower a resource's "hard limit" to any value above its "soft limit".

Example

```
>$ ulimit -a -S
max user processes                  (-u) 256
>$ ulimit -a -H
max user processes                  (-u) 256
```

Resource Limits

- In UNIX, standard resources include:
  - `RLIMIT_CPU`
  - `RLIMIT_DATA`
  - `RLIMIT_FSIZE`
  - `RLIMIT_NPROC`
  - `RLIMIT_STACK`
  - `RLIMIT_NOFILE`
- Child processes inherit their parents' resource limits.

Example

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

1  struct rlimit limit;
2
3  getrlimit(RLIMIT_NPROC, &limit);
4  limit.rlim_cur /= 2;
5  limit.rlim_max /= 2;
6  setrlimit(RLIMIT_NPROC, &limit);
```

Fork Bombs

Definition

A **fork bomb** is a denial-of-service attack based on one or more indefinitely replicating processes.

- Often, fork bombs can only be fixed by rebooting.

Example

```
   #include <unistd.h>

1  /* NOTE: If you must try this at home, lower your
2   *       resource limits first so that it can be
3   *       killed from another terminal. */
4  while (1) {
5      fork();
6  }
```

Signals

Signals

Definition

A **signal** notifies a process that an external or extraordinary event has occurred.

- Some signals indicate asynchronous conditions.
- Some signals indicate errors.

Example

A `SIGALRM` is sent when a timer has elapsed.

Example

A `SIGSEGV` is sent when a segfault has occurred.

Signals

- In UNIX, standard signals include:
  - `SIGINT`
  - `SIGQUIT`
  - `SIGTERM`
  - `SIGSEGV`
  - `SIGFPE`
  - `SIGBUS`
  - `SIGKILL`
  - `SIGALRM`
  - `SIGPIPE`
  - `SIGSTOP`
  - `SIGCONT`
  - `SIGCHLD`
- Each signal is associated with a default action.

Example

The default action for `SIGCONT` is to resume the process if stopped.

Example

The default action for `SIGCHLD` is to ignore the signal.

Example

The default action for `SIGSEGV` is to terminate the process.

Signal Handlers

- In UNIX, a signal's action is set with the system call:

  `sigaction(`*`SIGNAL`*`, `*`ACTION`*`, `*`ACTION`*`);`

  - The actions must be `struct sigaction` pointers.
  - The old action will be saved in the second structure.
  - The new action will be set from the first structure.

Definition

A **signal handler** is a function that responds to signals.

Example

```
1 void handler(int signum) {
2     printf("Handling signal %d.\n", signum);
3 }
```

Signal Handlers

- By default, a signal is **blocked** while it is being handled, so that a signal does not interrupt its own handler.
- Note that `SIGKILL`, `SIGSTOP` cannot be blocked or handled.

Example *(cont.)*

```
   #include <signal.h>

1  struct sigaction action;
2
3  action.sa_handler = handler;
4  action.sa_flags = SA_RESTART;
5  sigemptyset(&action.sa_mask);
6  sigaction(SIGALRM, &action, NULL);
```

Signal Handlers

- In UNIX, certain functions are **async-signal-safe**, including:
  - `sigaction`
  - `kill`
  - `sleep`
  - `pause`
  - `open`
  - `close`
  - `read`
  - `write`
  - `fork`
  - `execve`
  - `wait`
  - `waitpid`
- Note that a signal interrupts the normal flow of control.

Example

It is safe to call `write` within a signal handler.

Example

It is not safe to call `printf` within a signal handler, which could alter the state of an interrupted call to `printf`.

Timers

- In UNIX, a timer's interval is set with the system call:

  `setitimer(`*TIMER*`,` *INTERVAL*`,` *INTERVAL*`);`

  - There are three available timers: `ITIMER_REAL`, `ITIMER_VIRTUAL`, and `ITIMER_PROF`.
  - Each timer will send a signal every time it elapses.

Example *(cont.)*

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

1  struct itimerval timer;
2
3  timer.it_interval.tv_sec = 1;
4  timer.it_interval.tv_usec = 0;
5  timer.it_value = timer.it_interval;
6  setitimer(ITIMER_REAL, &timer, NULL);
```

Waiting for Signals

- In UNIX, signals are awaited with library function:

```
pause();
```

  - Until a signal is received, execution will be suspended, so as to avoid the need for **busy waiting**.
  - Once that signal has been handled, `pause` will return.

Example *(cont.)*

```
#include <unistd.h>

while (1) {
    pause();
}
```

Sending Signals

- In UNIX, signals are sent with the system calls:

  `kill(`*PID*`, `*SIGNAL*`);`

  `killpg(`*PGID*`, `*SIGNAL*`);`

  - If PID > 0, the signal is sent to that process.
  - If PID $= 0$, the signal is sent to the current process group.
  - If PID $= -1$, the signal is sent to all of the user's processes.
  - Only `root` may send signals to other users' processes.

Example

```
#include <signal.h>

kill(getpid(), SIGALRM);
```

Pipes

Pipes

Definition

A **pipe** allows data to be sent from one process to another.

- A pipe is essentially a temporary file managed by the OS.
  - Like files, pipes can be accessed within different processes.
  - Unlike files, pipes need not be stored on disk.
- A pipe is essentially a queue accessible by multiple processes.

Example

```
>$ ls > tmp.txt; sort < tmp.txt
```

Example

```
>$ ls | sort
```

Pipes

- In UNIX, pipes are created with the system call:

  `pipe(`*`DESCRIPTORS`*`);`

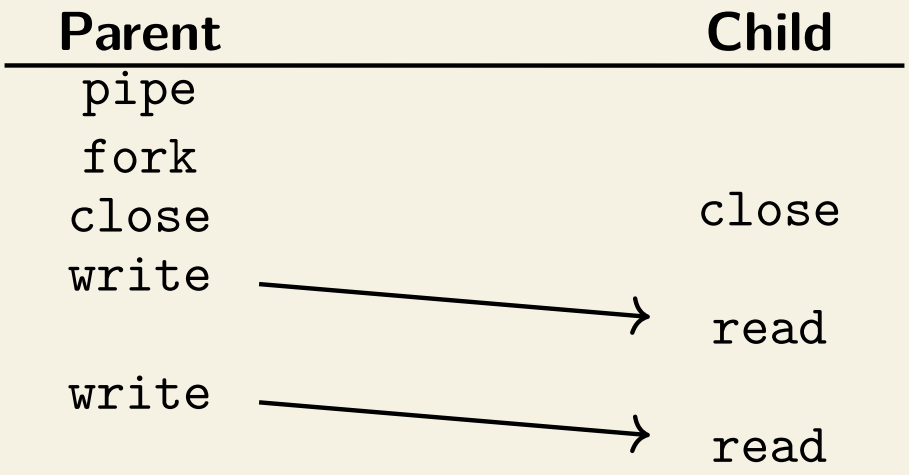
  - The descriptors must be an array of two integers.
  - The first element will be set to the **read end**'s file descriptor.
  - The second element will be set to the **write end**'s file descriptor.

Example

```
   #include <unistd.h>

1  int fds[2];
2  pipe(fds);

   close(fds[0]);
   close(fds[1]);
```

Pipes

Example



1. Parent creates a `pipe` and `forks` a child.
2. Parent `closes` the pipe's read end.
3. Child `closes` the pipe's write end.
4. Parent `writes` to the pipe's write end.
5. Child `reads` from the pipe's read end.

Writing to Pipes

- Writing to and reading from pipes are **blocking** calls:
  - Writing to a full pipe will block, unless no process has its read end open (in which case a `SIGPIPE` is sent).
  - Reading from an empty pipe will block, unless no process has its write end open (in which case an EOF is indicated).
- Note that child processes inherit their parents' file descriptors.

Example *(cont.)*

```
3 if (fork()) {
4     close(fds[0]);
5     write(fds[1], "Hello, child!", 13);
6     close(fds[1]);
7     wait(NULL);
8 }
```

Reading from Pipes

- A pipe must be created before any processes that need access.
- A pipe should be closed by all processes that have access.

Example *(cont.)*

```
 9 else {
10     char buf[4];
11     int n;
12
13     close(fds[1]);
14     while ((n = read(fds[0], buf, 4)) > 0) {
15         write(STDOUT_FILENO, buf, n);
16     }
17     close(fds[0]);
18 }
```

Replacing Streams with Pipes

- In UNIX, file descriptors are replaced with the system call:

  `dup2(`*`DESCRIPTOR`*`, `*`DESCRIPTOR`*`);`

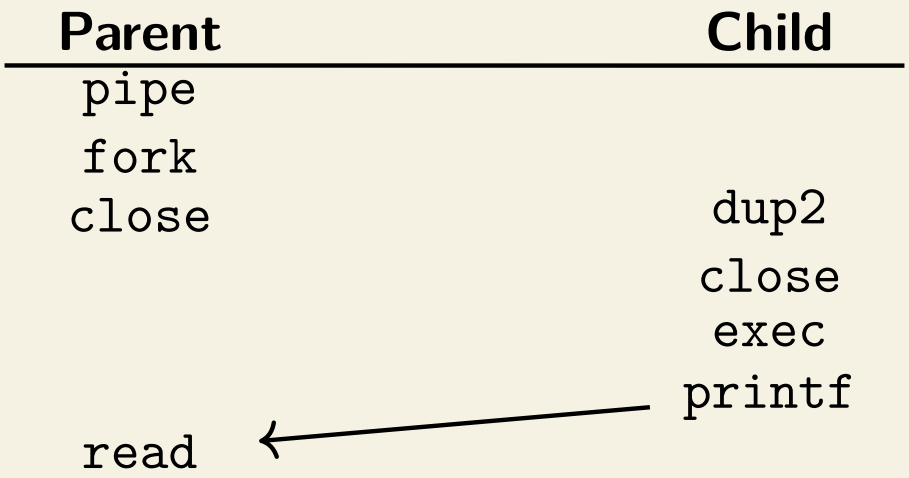
  - The second descriptor will first be closed.
  - The second descriptor will then be made a duplicate of the first.

Example

```
   #include <unistd.h>

   int fds[2];
   pipe(fds);

1  dup2(fds[1], STDOUT_FILENO);
2  close(fds[0]);
3  close(fds[1]);
```

Replacing Streams with Pipes

Example



1. Parent creates a `pipe` and `forks` a child.
2. Parent `closes` the pipe's write end.
3. Child `dup2`s the pipe's write end onto `stdout`.
4. Child `closes` the pipe's read and write ends.
5. Child `execs` an executable that prints to `stdout`.
6. Parent `reads` from the pipe's read end.

Deadlocks

Definition

A **deadlock** occurs when two or more processes are waiting for an event that can only be caused by one of them.

- Often, deadlocks are caused by circularly blocked processes.

Example

Suppose 2 processes are each reading from a pipe to which the other will later write. Then neither process will ever unblock.

Example

Suppose 2 processes are each paused awaiting a signal that the other will later send. Then neither process will ever unblock.

The POSIX Standard

Application Programming Interfaces

"Whenever computer science has made significant progress…ignoring those *de facto* standards has each been a necessary condition."

— Edsger Dijkstra, *Can Computer Science Save the Industry?*

Definition

An **application programming interface** specifies how two pieces of software may interact.

- The POSIX standard attempts to unify the APIs and utilities implemented by "UNIX-like" operating systems.
- Largely POSIX-compliant operating systems include macOS, most Linux distributions, and most BSD derivatives.

Application Programming Interfaces

Example

A POSIX-compliant OS must refer to the current directory as "`.`", the parent directory as "`..`", and the root directory as "`/`".

Example

A POSIX-compliant OS must have the `ed`, `ex`, `sed`, and `vi` text editors preinstalled.

Example

A POSIX-compliant OS must implement extended C system calls and standard libraries, including sockets and threads.

Networks and Sockets

Networks

Definition

A **network** is a collection of connected devices, each identified by at least one **address**.

- Devices on the same network may communicate according to one or more **protocols**.
- Note that not all networks are the Internet, and thus not all addresses are IP addresses.

Example

```
>$ ip a | grep inet
    inet 127.0.0.1/8 scope host lo
    inet6 ::1/128 scope host
    inet 129.65.128.61/24 brd 129.65.128.255 scope g...
    inet6 fe80::250:56ff:fe89:de58/64 scope link nop...
```

Ports

Definition

A **port** is a number that uniquely identifies a single connection associated with an address.

- Two networked processes on the same device will typically use the same IP address but different ports.
- Ports ≤ 1023 are reserved for existing services and typically cannot be used without `root` privileges.

Example

Port 80 is reserved for HTTP; port 443 is reserved for HTTPS.

Example

Port 666 is reserved for the video game *Doom*.

Addresses

- In POSIX, addresses are queried with the library functions:

  `getaddrinfo(`*HOSTNAME*`, `*PORT*`, `*HINTS*`, `*ADDRESS*`);`

  `freeaddrinfo(`*ADDRESS*`);`

Example

```
   #include <sys/socket.h>
   #include <netdb.h>

1  struct addrinfo hints = {0}, *addr;
2  hints.ai_family = AF_INET;
3  hints.ai_socktype = SOCK_STREAM;
4  getaddrinfo("localhost", "2187", &hints, &addr);

   freeaddrinfo(addr);
```

Endianness

Definition

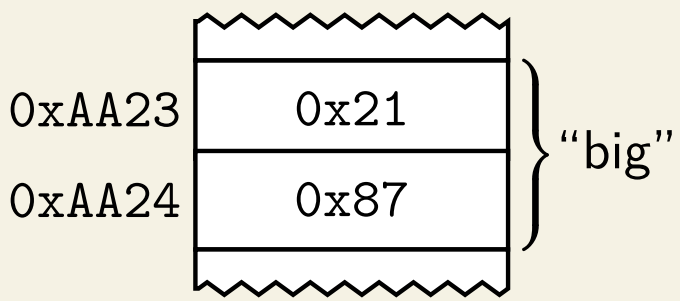
Every value consists of bytes, which must be stored in memory:

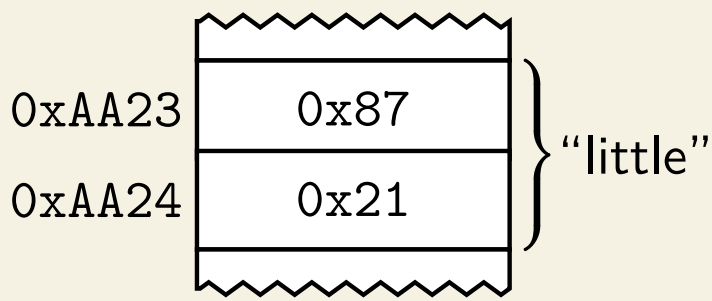
- A **big-endian** system stores the MSB at the lowest address.
- A **little-endian** system stores the LSB at the lowest address.

- By convention, most network protocols are big-endian.
- For historical reasons, most CPUs are little-endian.

Example

Given that the 16-bit value `0x2187` is stored at address `0xAA23`:





Endianness

- In POSIX, endianness is converted with the library functions:

  `htons(`*SHORT*`);` `ntohs(`*SHORT*`);`

  `htonl(`*LONG*`);` `ntohl(`*LONG*`);`

  - If the **host byte order** is little-endian, these functions convert to and from **network byte order**, big-endian integers.
  - If the host byte order is big-endian, these functions do nothing.

Example

Suppose the host is little-endian. Then `ntohs(0x2187)` returns `0x8721`.

Example

Suppose the host is big-endian. Then `htonl(0xDEADBEEF)` returns `0xDEADBEEF`.

Sockets

- In POSIX, **sockets** are created with the system calls:

  `socket(`*FAMILY*`, `*TYPE*`, `*PROTOCOL*`);`

  `bind(`*SOCKET*`, `*ADDRESS*`, `*LENGTH*`);`

  - `socket` returns a file descriptor for a new socket.
  - `bind` associates an existing socket with a specific port.
  - Connecting to an existing socket requires knowing its port.

Example *(cont.)*

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

int fd = socket(addr->ai_family,
 addr->ai_socktype, addr->ai_protocol);

close(fd);
```

Sockets

- In POSIX, two sockets are connected with the system calls:

```
listen(SOCKET, LIMIT);
connect(SOCKET, ADDRESS, LENGTH);
accept(SOCKET, ADDRESS, LENGTH);
```

  - `listen` creates a queue for pending connections.
  - `connect` initiates a pending connection.
  - `accept` creates a new socket for the next pending connection.

Example *(cont.)*

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

connect(fd, addr->ai_addr, addr->ai_addrlen);
```

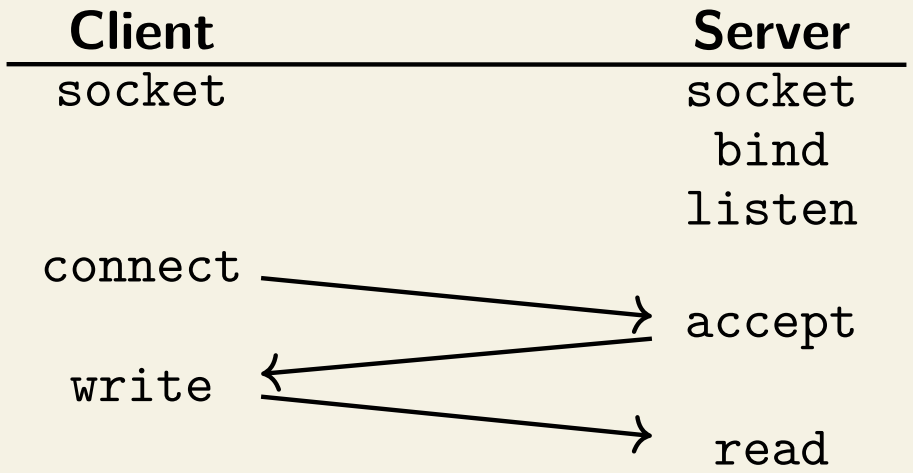
Sockets

Example

```
   struct addrinfo hints = {0}, *addr;
   hints.af_family = AF_INET;
   hints.ai_socktype = SOCK_STREAM;
   hints.ai_flags = AI_PASSIVE;
   getaddrinfo(NULL, "2187", &hints, &addr);

   int fd = socket(addr->ai_family,
    addr->ai_socktype, addr->ai_protocol);
   bind(fd, addr->ai_addr, addr->ai_addrlen);

1  listen(fd, 16);
2  int client = accept(fd, NULL, NULL);

   close(client);
   close(fd);
```

Sockets

Example



1. Client and server each create a `socket`.
2. Server `binds` its socket to a port and `listens`.
3. Client `connects` its socket to the server's.
4. Server `accepts` the pending connection.
5. Client `writes` to its already existing socket.
6. Server `reads` from its newly created socket.

Writing to Sockets

- Note that `write` returns the number of bytes written, which may not be the length of entire buffer.
- If the entire buffer cannot be sent at once, it is the caller's responsibility to send the remainder later.

Example *(cont.)*

```
    int i = 0;
    char buf[15] = "Hello, server!\n";

 8  while (i < 15) {
 9      i += write(fd, buf + i, 15 - i);
10  }
```

Reading from Sockets

- Note that `read` returns the number of bytes read, where 0 indicates that the **peer** has closed the connection.
- Note that sockets are **duplex**: the same socket can be used for both reading and writing, in both directions.

Example *(cont.)*

```
   int n;
   char buf[4];

3  while ((n = read(client, buf, 4)) > 0) {
4      write(STDOUT_FILENO, buf, n);
5  }
```

Non-Blocking I/O

- In POSIX, files are set to non-blocking with the system call:

  `fcntl(`*`DESCRIPTOR`*`, F_SETFL, O_NONBLOCK);`

  - If no data is available to read, `read` will instead return -1 and set the error number `EAGAIN`.
  - If no space is available to write, `write` will instead return -1 and set the error number `EAGAIN`.

Example

```
   #include <fcntl.h>

1  fcntl(client, F_SETFL, O_NONBLOCK);
2  while ((n = read(client, buf, 4)) < 0
3         && errno == EAGAIN) {
4      sleep(1);
5  }
```

Non-Blocking I/O

- In POSIX, open files are polled with the system call:

  `poll(`*DESCRIPTORS*`,` *LENGTH*`,` *TIMEOUT*`);`

  - If the timeout is -1, `poll` blocks indefinitely; if the timeout is 0, `poll` does not block.
  - Note that `poll` is used to "poll" a set of file descriptors; it does not do so by "polling" while busy-waiting.

Example

```
   #include <poll.h>

1  struct pollfd fds[1];
2  fds[0].fd = client;
3  fds[0].events = POLLIN;
4  poll(fds, 1, 1000);
```

Threads

Threads

Definition

A **thread** is an independent sequence of instructions in a process.

- Like processes, threads can be executed concurrently with their own registers and runtime stacks.
- Unlike processes, threads share a data segment, a heap, and resources such as open files.

Example

```
>$ ps -L u
  PID   LWP %CPU NLWP %MEM STAT START   TIME COMMAND
10130 10130  0.0    1  0.0 Ss   15:52   0:00 -bash
12257 12257  0.0    1  0.0 R+   16:05   0:00 ps -L u
```

Thread Creation

- In POSIX, threads are created with the library function:

  `pthread_create(`*`THREAD`*`, NULL,`
  `FUNCTION``, `*`ARGUMENT`*`);`

  - When a thread is created, a given function is called.
  - The function must take as argument a `void` pointer.

Example

```
#include <pthread.h>

void *f(void *x) {
    return x;
}

pthread_t tid;
pthread_create(&tid, NULL, f, NULL);
```

Thread Termination

- In POSIX, threads are joined with the library function:

  `pthread_join(`*THREAD*`,` *RETURN*`);`

  - When a thread's original function returns, the thread exits.
  - The function must return a `void` pointer.
- The function given to `pthread_create` is its thread's "`main`".
  - A thread may also terminate by calling `pthread_exit`.
  - Note that calling `exit` in any thread terminates the *process*.

Example *(cont.)*

```
   #include <pthread.h>

1  void *retval;
2  pthread_join(tid, &retval);
```

Task Parallelism

Definition

Task parallelism parallelizes a problem by distributing smaller problems of the same instance.

- In other words, each subproblem concerns a subtask of the overall task, ideally taking equal amounts of time.
- Note that some subproblems may depend on another's data.

Example

Consider the following problem:

- Given a stream of characters, copy them to another stream.

...a single buffer must be read and written in serial, but the current buffer can be written in parallel with the next buffer's being read.

Data Parallelism

Definition

Data parallelism parallelizes a problem by distributing smaller instances of the same problem.

- In other words, each subproblem concerns a subset of the overall dataset, ideally of equal sizes.
- Note that some subsolutions may not be overall solutions.

Example

Consider the following problem:

- Given an array of integers, sort them in ascending order.

...the halves of the array can be sorted in parallel, but they must then be merged together in serial.

Race Conditions

Definition

A **race condition** occurs when multiple threads attempt to access data concurrently, such that behavior depends on order of execution.

- Often, race conditions cause programs to be *sometimes* correct.

Example

Suppose 2 threads attempt to increment the same global integer. Then the integer may only be incremented by 1.

Example

Suppose 2 processes attempt to write to the same file. Then their writes may be interleaved, depending on how they were buffered.

Mutex Locks

Definition

A **mutex lock** is a resource which at most one thread may possess at any given time.

- A mutex must be acquired **atomically**, so that no thread can check it while another is in the process of setting it.

Example

```
   int lock = 0, count = 0;

1  if (lock == 0) {    /* Other threads may check the */
2      lock = 1;       /* lock between these lines.   */
3      count++;
4      lock = 0;
5  }
```

Thread Synchronization

- In POSIX, mutexes are created with the library functions:

  `pthread_mutex_init(`*`MUTEX`*`, NULL);`

  `pthread_mutex_destroy(`*`MUTEX`*`);`

  - Only a mutex itself is guaranteed to be mutually exclusive.
  - Threads must cooperatively agree to acquire a mutex.

Example

```
   #include <pthread.h>

1  pthread_mutex_t lock;
2  pthread_mutex_init(&lock, NULL);

   pthread_mutex_destroy(&lock)
```

Thread Synchronization

- In POSIX, mutexes are acquired with the library functions:

```
pthread_mutex_lock(MUTEX);
pthread_mutex_trylock(MUTEX);
pthread_mutex_unlock(MUTEX);
```

  - `pthread_mutex_lock` will block until the mutex is acquired.
  - `pthread_mutex_trylock` will error if the mutex is locked.

Example *(cont.)*

```
#include <pthread.h>

pthread_mutex_lock(&lock);
count++;
pthread_mutex_unlock(&lock)
```

Shell Scripts

Shells

Definition

A **shell** is an interpreter for commands that expose system services.

- A **terminal** is a device (or program emulating such a device) that provides a textual user interface.
- A **shell script** is a sequence of shell commands saved in a file, typically with execute permission.

Example

The default shell on most Linux distributions is `bash`.

Example

The default shell on macOS is `zsh`.

Shell Scripts

- In POSIX, an executable shell script begins with a **shebang**:

  `#!`*PATH*

  - The path indicates to the operating system which shell should interpret the script.
  - Note that the shebang itself will be treated as a comment by the shell, and it will not be expanded.
- In Bash, comments begin with "#" and end with a newline.

Example

```
1 #!/bin/bash
2
3 echo "Hello, world!"
```

Variables

- In Bash, a variable is assigned and referenced with the forms:

$$NAME=VALUE$$

$$\$NAME$$

  - By convention, variable names are typically in all-caps.
  - Note that the assignment operator is not surrounded by spaces.
- Effectively, all variables contain strings; integer arithmetic is allowed on strings that consist only of digits.

Example

```
1 #!/bin/bash
2
3 STR="Hello, world!"
4 echo $STR
```

Variables

- Variables referenced within double quotes will be substituted into the string; those within single quotes will be ignored.
- Command-line arguments are numbered from 0. Numbers greater than 9 must be wrapped in curly braces.

Example

```
1 #!/bin/bash
2
3 echo "Hello, $1!"
```

Example *(cont.)*

```
>$ ./hello.sh Christopher
Hello, Christopher!
```

Redirection

- In Bash, output is redirected with the basic forms:

COMMAND `>` *PATH*

COMMAND `>>` *PATH*

COMMAND `&>` *PATH*

  - A single arrow writes a command's `stdout` to a file.
  - A double arrow appends a command's `stdout` to a file.
  - An ampersand arrow writes `stdout` and `stderr` to a file.

Example

```
1 #!/bin/bash
2
3 date >> logfile.txt
4 w >> logfile.txt
```

Redirection

- In Bash, input is redirected with the basic forms:

$$\textit{COMMAND} < \textit{PATH}$$

$$\textit{COMMAND} \mid \textit{COMMAND}$$

  - An arrow redirects a file to a command's `stdin`.
  - A pipe redirects one command's `stdout` to another's `stdin`.

Example

```
1 #!/bin/bash
2
3 ls *.txt | sort -R
```

Substitution

- In Bash, output is substituted with the form:

  $(*COMMAND*)

  - The command will be executed in a **subshell**.
  - The substitution will expand to the command's output.
- Substitutions may be used within commands or assignments.

Example

```
1 #!/bin/bash
2
3 FILES="$(ls *.txt | sort -R)"
4 cat $FILES
```