CS241 System Programming Wikibook

Contents

Chapter 1

Introduction

./introduction

Chapter 2

C Programming Language

If you want teach systems, don't drum up the programmers, sort the issues, and make PRs. Instead, teach them to yearn for the vast and endless C.

Antoine de Saint-Exupéry (Kinda)

C is the de-facto programming language to do serious system serious programming. Why? Most kernels are written in largely in C. The Linux kernel Love (2010) and the XNU kernel Inc. (2017) of which Mac OS X is based off. The Windows Kernel uses C++, but doing system programming on that is much harder on windows that UNIX for beginner system programmers. Most of you have some experience with C++, but C is a different beast entirely. You don't have nice abstractions like classes and RAII to clean up memory. You are going to have to do that yourself. C gives you much more of an opportunity to shoot yourself in the foot but lets you do thinks at a much finer grain level.

History of C

C was developed by Dennis Ritchie and Ken Thompson at Bell Labs back in 1973 Ritchie (1993). Back then, we had gems of programming languages like Fortran, ALGOL, and LISP. The goal of C was two fold. One, to target the most popular computers at the time liek the PDP-7. Two, try and remove some of the lower level constructs like managing registers, programming assembly for jumps and instead create a language that had the power to express programs procedurally (as opposed to mathematically like lisp) with more readable code all while still having the ability to interface with the operating system. It sounded like a tough feat. At first, it was only used internally at Bell Labs along with the UNIX operating system.

The first "real" standardization is with Brian Kerninghan and Dennis Ritchies book Kernighan and Ritchie (1988). It is still widely regarded today as the only portable set of C instructions. There were different standards of C from ANSI to ISO after the Unix guides. The one that we will be mainly focusing on is the POSIX C library. Now to get the elephant out of the room, the Linux kernel is not entirely POSIX compliant. It instead basis itself on the

Differences Between Other Languages

Features of C

Crash course intro to C

The only way to start learning C is in true Kerninghan and Ritchie fashion, with dissecting a hello world program. The K&R book is known as the de-facto standard for learning C. There have been additions to it, but they have been few and far between over the years.

How do you write a complete hello world program in C?

The only way to start learning C is by starting with hello world. As per the original example that Kernighan and Ritchie proposed way back when, the hello world hasn't changed that much.

```
#include <stdio.h>
int main(void) {
    printf("Hello World\n");
    return 0;
}
```

1. The `#include` directive takes the file `stdio.h` (which stands for **st**an**d**ard **i**nput and **o**utput) located somewhere in your operating system, copies the text, and substitutes it where the `#include` was.

2. The `int main(void)` is a function declaration. The first word `int` tells the compiler what the return type of the function is. The part before the parens (`main`) is the function name. In C, no two functions can have the same name in a single compiled program, shared libraries are a different touchy subject. Then, what comes after is the paramater list. When we give the parameter list for regular functions `(void)` that means that the compiler should error if the function is called with any arguments. For regular functions having a declaration like `void func()` means that you are allowed to call the function like `func(1, 2, 3)` because there is no delimiter **?**. In the case of `main`, it is a special function. There are many ways of declaring `main` but the ones that you will be familiar with are `int main(void)`, `int main()`, and `int main(int argc, char *argv[])`.

3. `printf("Hello World");` is what we call a function call. `printf` is defined as a part of `stdio.h`. The function has been compiled and lives somewhere else on our machine. All we need to do is include the header and call the function with the appropriate parameters (a string literal `"Hello World"`). If you don't have the newline, the buffer will not be flushed. It is by convention that buffered IO is not flushed until a newline. **?**

4. `return 0;`. `main` has to return an integer. By convention, `return 0` means success and anything else means failure **?**.

```
$ gcc main.c -o main
$ ./main
Hello World
$
```

1. `gcc` is short for the GNU-Compiler-Collection which has a host of compilers ready for use. The compiler infers from the extension that you are trying to compile a .c file
2. `./main` tells your shell to execute the program in the current directory called main. The program then prints out hello world

Preprocessor

What is the preprocessor? It is an operation that the compiler performs **before** actually compiling the program. It is a copy and paste command. Meaning that if I do the following.

```
#define MAX_LENGTH 10
char buffer[MAX_LENGTH]
```

After preprocessing, it'll look like this.

```
char buffer[10]
```

But there are side effects to the preprocessor. Does the following code print out.

```
#define min(a,b) ((a)<(b) ? (a) : (b))
int x = 4;
if(min(x++, 100)) printf("%d is six", x);
```

Macros are simple text substitution so the above example expands to `x++ < 100 ? x++ : 100` (parenthesis omitted for clarity). Now for this case, it will probably still print 6 but consider the edge case when `x = 99`. Also consider the edge case when operator precedence comes into play.

```
#define min(a,b) a<b ? a : b
int x = 99;
int r = 10 + min(99, 100); // r is 100!
```

Macros are simple text substitution so the above example expands to `10 + 99 < 100 ? 99 : 100`

C Preprocessor logical gotcha

```
#define ARRAY_LENGTH(A) (sizeof((A)) / sizeof((A)[0]))
int static_array[10]; // ARRAY_LENGTH(static_array) = 10
int* dynamic_array = malloc(10); //
   ARRAY_LENGTH(dynamic_array) = 2 or 1
```

What is wrong with the macro? Well, it works if we have a static array like the first array because `sizeof` a static array returns the bytes that array takes up, and dividing it by the `sizeof(an_element)` would give you the number of entries. But if we use a pointer to a piece of memory, taking the sizeof the pointer and dividing it by the size of the first entry won't always give us the size of the array.

Parsing

Syntactic Parsing

Language Facilities

Keywords

C has an assortment of control-flow keywords. Here are some constructs that you should know briefly as of C99.

1. `break`
2. `char`
3. `const`
4. `continue`
5. `do  while(0);`
6. `double`
7. `else`
8. `enum`
9. `extern`
10. `float`
11. `for`
12. `goto`
13. `if else else-if`
14. `inline`
15. `register`

16. `restrict`
17. `return`
18. `signed`
19. `sizeof`
20. `static`
21. `struct`
22. `switch case default` Switches are essentially glorified jump statements. Meaning that you take either a byte or an integer and the control flow of the program jumps to that location.

    ```
    switch(/* char or int */) {
      case INT1: puts("1");
      case INT2: puts("2");
      case INT3: puts("3");
    }
    ```

    If we give a value of 2 then

    ```
    switch(2) {
      case INT1: puts("1"); /* Doesn't run this */
      case INT2: puts("2"); /* Runs this */
      case INT3: puts("3"); /* Also runs this */
    }
    ```

    The break statement
23. `typedef`
24. `union`
25. `unsigned`
26. `void`
27. `volatile` is a compiler keyword. This means that the compiler should not optimize its value out. Consider the following simple function.

    ```
    int flag = 1;
    pass_flag(&flag);
    while(flag) {
        // Do things unrelated to flag
    ```

```
}
```

The compiler may, since the internals of the while loop have nothing to do with the flag, optimize it to the following even though a function may alter the data.

```
while(1) {
    // Do things unrelated to flag
}
```

If you put the volatile keyword

28. `while`

C data types

char short int long (long int) long long float double

Operators

Common C Functions

To find more information about any functions, use the man pages. Note the man pages are organized into sections. Section 2 are System calls. Section 3 are C libraries. On the web, Google `man 7 open`. In the shell, `man -S2 open` or `man -S3 printf`

Input/Output

Parsing

string.h

stdlib.h

To allocate space on the heap, use malloc. There's also realloc and calloc. Typically used with sizeof. e.g. enough space to hold 10 integers

```
int *space = malloc(sizeof(int) * 10);
```

Conventions/Errno

System Calls

What is a system call?

The interplay with library functions

Does `printf` call write or does write call `printf`?

`printf` calls `write`. `printf` includes an internal buffer so, to increase performance `printf` may not call `write` everytime you call `printf`. `printf` is a C library function. `write` is a system call and as we know system calls are expensive. On the other hand, `printf` uses a buffer which suits our needs better at that point

C Memory Model

C Null-Terminated Strings

Strings in C are represented as characters in memory. The end of the string includes a NULL (0) byte **?**. So "ABC" requires four(4) bytes `[A,B,C,\0]`. The only way to find out the length of a C string is to keep reading memory until you find the NULL byte. C characters are always exactly one byte each.

When you write a string literal `"ABC"` in an expression the string literal evaluates to a char pointer (`char *`), which points to the first byte/char of the string. This means `ptr` in the example below will hold the memory address of the first character in the string.

String constants are constant

```
char array[] = "Hi!"; // array contains a mutable copy
strcpy(array, "OK");

char *ptr = "Can't change me"; // ptr points to some
   immutable memory
strcpy(ptr, "Will not work");
```

String literals are character arrays stored in the code segment of the program, which is immutable. Two string literals may share the same space in memory. An example follows:

```
char *str1 = "Brandon Chong is the best TA";
char *str2 = "Brandon Chong is the best TA";
```

The strings pointed to by `str1` and `str2` may actually reside in the same location in memory.

Char arrays, however, contain the literal value which has been copied from the code segment into either the stack or static memory. These following char arrays do not reside in the same place in memory.

```
char arr1[] = "Brandon Chong didn't write this";
char arr2[] = "Brandon Chong didn't write this";
```

```
char *ptr = "ABC"
```

Some common ways to initialize a string include:

```c
char *str = "ABC";
char str[] = "ABC";
char str[]={'A','B','C','\0'};
```

Pointers

Pointer Basics

Declaring a Pointer

A pointer refers to a memory address. The type of the pointer is useful - it tells the compiler how many bytes need to be read/written. You can declare a pointer as follows.

```c
int *ptr1;
char *ptr2;
```

Due to C's grammar, an `int*` or any pointer is not actually its own type. You have to precede each pointer variable with an asterisk. As a common gotcha, the following

```c
int* ptr3, ptr4;
```

Will only declare `*ptr3` as a pointer. `ptr4` will actually be a regular int variable. To fix this declaration, keep the `*` preceding to the pointer

```c
int *ptr3, *ptr4;
```

Keep this in mind for structs as well. If one does not typedef them, then the pointer goes after the type.

```c
struct person *ptr3;
```

Reading/Writing with pointers

Let's say that we declare a pointer `int *ptr`. For the sake of discussion, let's say that `ptr` points to memory address `0x1000`. If we want to write to a pointer, we can dereference and assign `*ptr`.

```c
*ptr = 0; // Writes some memory.
```

What C will do is take the type of the pointer which is an `int` and writes `sizeof(int)` bytes from the start of the pointer, meaning that bytes 0x1000, 0x1001, 0x1002, 0x1003 will all be zero. The number of bytes written depends on the pointer type. It is the same for all primitive types but structs are a little different.

Pointer Arithmetic

You can add an integer to a pointer. However, the pointer type is used to determine how much to increment the pointer. For char pointers this is trivial because characters are always one byte:

```
char *ptr = "Hello"; // ptr holds the memory location of
    'H'
ptr += 2; //ptr now points to the first'l'
```

If an int is 4 bytes then ptr+1 points to 4 bytes after whatever ptr is pointing at.

```
char *ptr = "ABCDEFGH";
int *bna = (int *) ptr;
bna +=1; // Would cause iterate by one integer space (i.e
    4 bytes on some systems)
ptr = (char *) bna;
printf("%s", ptr);
/* Notice how only 'EFGH' is printed. Why is that? Well
    as mentioned above, when performing 'bna+=1' we are
    increasing the **integer** pointer by 1, (translates
    to 4 bytes on most systems) which is equivalent to 4
    characters (each character is only 1 byte)*/
return 0;
```

Because pointer arithmetic in C is always automatically scaled by the size of the type that is pointed to, you can't perform pointer arithmetic on void pointers.

You can think of pointer arithmetic in C as essentially doing the following

If I want to do

```
int *ptr1 = ...;
int *offset = ptr1 + 4;
```

Think

```
int *ptr1 = ...;
char *temp_ptr1 = (char*) ptr1;
int *offset = (int*)(temp_ptr1 + sizeof(int)*4);
```

To get the value. **Every time you do pointer arithmetic, take a deep breath and make sure that you are shifting over the number of bytes you think you are shifting over.**

What is a void pointer?

A pointer without a type (very similar to a void variable). Void pointers are used when either a datatype you're dealing with is unknown or when you're interfacing C code with other programming languages. You can think of this as a raw pointer, or just a memory address. You cannot directly read or write to it because the void type does not have a size. For Example

```
void *give_me_space = malloc(10);
char *string = give_me_space;
```

This does not require a cast because C automatically promotes `void*` to its appropriate type. **Note:**
gcc and clang are not total ISO-C compliant, meaning that they will let you do arithmetic on a void pointer. They will treat it as a char pointer but do not do this because it may not work with all compilers!

Topics

- C Strings representation
- C Strings as pointers
- char p[]vs char* p
- Simple C string functions (strcmp, strcat, strcpy)
- sizeof char
- sizeof x vs x*
- Heap memory lifetime
- Calls to heap allocation
- Deferencing pointers
- Address-of operator
- Pointer arithmetic
- String duplication
- String truncation
- double-free error
- String literals
- Print formatting.

- memory out of bounds errors
- static memory
- fileio POSIX vs. C library
- C io fprintf and printf
- POSIX file IO (read, write, open)
- Buffering of stdout

Questions/Exercises

- What does the following print out?

```
int main(){
fprintf(stderr, "Hello ");
fprintf(stdout, "It's a small ");
fprintf(stderr, "World\n");
fprintf(stdout, "place\n");
return 0;
}
```

- What are the differences between the following two declarations? What does `sizeof` return for one of them?

```
char str1[] = "bhuvan";
char *str2 = "another one";
```

- What is a string in c?
- Code up a simple `my_strcmp`. How about `my_strcat`, `my_strcpy`, or `my_strdup`? Bonus: Code the functions while only going through the strings *once*.
- What should the following usually return?

```
int *ptr;
sizeof(ptr);
sizeof(*ptr);
```

- What is `malloc`? How is it different than `calloc`. Once memory is `malloc`ed how can I use `realloc`?

- What is the & operator? How about *?
- Pointer Arithmetic. Assume the following addresses. What are the following shifts?

```c
char** ptr = malloc(10); //0x100
ptr[0] = malloc(20); //0x200
ptr[1] = malloc(20); //0x300
```

  - `ptr + 2`
  - `ptr + 4`
  - `ptr[0] + 4`
  - `ptr[1] + 2000`
  - `*((int)(ptr + 1)) + 3`

- How do we prevent double free errors?
- What is the printf specifier to print a string, `int`, or `char`?
- Is the following code valid? If so, why? Where is `output` located?

```c
char *foo(int var){
static char output[20];
snprintf(output, 20, "%d", var);
return output;
}
```

- Write a function that accepts a string and opens that file prints out the file 40 bytes at a time but every other print reverses the string (try using POSIX API for this).
- What are some differences between the POSIX filedescriptor model and C's `FILE*` (ie what function calls are used and which is buffered)? Does POSIX use C's `FILE*` internally or vice versa?

Bibliography

Apple Inc. Xnu kernel. `https://github.com/apple/darwin-xnu`, 2017.

B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988. ISBN 9780131103627. URL `https://books.google.com/books?id=161QAAAAMAAJ`.

Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. ISBN 0672329468, 9780672329463.

Dennis M. Ritchie. The development of the c language. *SIGPLAN Not.*, 28(3):201–208, March 1993. ISSN 0362-1340. doi: 10.1145/155360.155580. URL `http://doi.acm.org/10.1145/155360.155580`.

Chapter 3

Appendix

The Hitchhiker's Guide to Debugging C Programs

This is going to be a massive guide to helping you debug your C programs. There are different levels that you can check errors and we will be going through most of them. Feel free to add anything that you found helpful in debugging C programs including but not limited to, debugger usage, recognizing common error types, gotchas, and effective googling tips.

In-Code Debugging

Clean code

Make your code modular using helper functions. If there is a repeated task (getting the pointers to contiguous blocks in the malloc MP, for example), make them helper functions. And make sure each function does one thing very well, so that you don't have to debug twice.

Let's say that we are doing selection sort by finding the minimum element each iteration like so,

```c
void selection_sort(int *a, long len){
     for(long i = len-1; i > 0; --i){
         long max_index = i;
         for(long j = len-1; j >= 0; --j){
             if(a[max_index] < a[j]){
                  max_index = j;
             }
         }
         int temp = a[i];
         a[i] = a[max_index];
         a[max_index] = temp;
     }

}
```

Many can see the bug in the code, but it can help to refactor the above method into

```
long max_index(int *a, long start, long end);
void swap(int *a, long idx1, long idx2);
void selection_sort(int *a, long len);
```

And the error is specifically in one function. In the end, we are not a class about refactoring/debugging your code. In fact, most kernel code is so atrocious that you don't want to read it. But for the sake of debugging, it may benefit you in the long run to adopt some of these practices.

Asserts!

Use assertions to make sure your code works up to a certain point – and importantly, to make sure you don't break it later. For example, if your data structure is a doubly linked list, you can do something like `assert(node->size == node->next->prev->size)` to assert that the next node has a pointer to the current node. You can also check the pointer is pointing to an expected range of memory address, not null, ->size is reasonable etc. The `NDEBUG` macro will disable all assertions, so don't forget to set that once you finish debugging. assert link

Here's a quick example with assert. Let's say that I'm writing code using memcpy

```
assert(!(src < dest+n && dest < src+n)); //Checks overlap
memcpy(dest, src, n);
```

This check can be turned off at compile time, but will save you **tons** of trouble debugging!

printfs

When all else fails, print like crazy! Each of your functions should have an idea of what it is going to do (ie find_min better find the minimum element). You want to test that each of your functions is doing what it set out to do and see exactly where your code breaks. In the case with race conditions, tsan may be able to help, but having each thread print out data at certain times could help you identify the race condition.

Valgrind

Valgrind is a suite of tools designed to provide debugging and profiling tools to make your programs more correct and detect some runtime issues. The most used of these tools is Memcheck, which can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behaviour (for example, unfreed memory buffers). To run Valgrind on your program:

```
valgrind --leak-check=full --show-leak-kinds=all
   myprogram arg1 arg2
```

Arguments are optional and the default tool that will run is Memcheck. The output will be presented in form of number of allocations, number of freed allocations, and the number of errors.

Suppose we have a simple program like this:

```
#include <stdlib.h>

void dummy_function() {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;       // error 1:as you can see here we
        write to an out of bound memory address
}                    // error 2: memory leak the allocated
    x not freed

int main(void) {
    dummy_function();
    return 0;
}
```

This program compiles and run with no errors. Let's see what Valgrind will output.

```
==29515== Memcheck, a memory error detector
==29515== Copyright (C) 2002-2015, and GNU GPL'd, by
    Julian Seward et al.
==29515== Using Valgrind-3.11.0 and LibVEX; rerun with -h
    for copyright info
==29515== Command: ./a
==29515==
==29515== Invalid write of size 4
==29515==    at 0x400544: dummy_function (in
    /home/rafi/projects/exocpp/a)
==29515==    by 0x40055A: main (in
    /home/rafi/projects/exocpp/a)
==29515== Address 0x5203068 is 0 bytes after a block of
    size 40 alloc'd
==29515==    at 0x4C2DB8F: malloc (in
    /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==29515==    by 0x400537: dummy_function (in
    /home/rafi/projects/exocpp/a)
==29515==    by 0x40055A: main (in
    /home/rafi/projects/exocpp/a)
==29515==
==29515==
==29515== HEAP SUMMARY:
==29515==     in use at exit: 40 bytes in 1 blocks
==29515==   total heap usage: 1 allocs, 0 frees, 40 bytes
    allocated
==29515==
==29515== LEAK SUMMARY:
==29515==    definitely lost: 40 bytes in 1 blocks
```

```
==29515==   indirectly lost: 0 bytes in 0 blocks
==29515==     possibly lost: 0 bytes in 0 blocks
==29515==   still reachable: 0 bytes in 0 blocks
==29515==        suppressed: 0 bytes in 0 blocks
==29515== Rerun with --leak-check=full to see details of
    leaked memory
==29515==
==29515== For counts of detected and suppressed errors,
    rerun with: -v
==29515== ERROR SUMMARY: 1 errors from 1 contexts
    (suppressed: 0 from 0)
```

Invalid write: It detected our heap block overrun, writing outside of allocated block.

Definitely lost: Memory leak — you probably forgot to free a memory block.

Valgrind is a very effective tool to check for errors at runtime. C is very special when it comes to such behavior, so after compiling your program you can use Valgrind to fix errors that your compiler may not catch and that usually happen when your program is running.

For more information, you can refer to the valgrind website.

TSAN

ThreadSanitizer is a tool from Google, built into clang and gcc, to help you detect race conditions in your code. For more information about the tool, see the Github wiki. Note, that running with tsan will slow your code down a bit. Consider the following code.

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

int global;

void *Thread1(void *x) {
    global++;
    return NULL;
}

int main() {
    pthread_t t[2];
    pthread_create(&t[0], NULL, Thread1, NULL);
    global = 100;
    pthread_join(t[0], NULL);
}
// compile with gcc -fsanitize=thread -pie -fPIC -ltsan
    -g simple_race.c
```

We can see that there is a race condition on the variable `global`. Both the main thread and the thread created with pthread_create will try to change the value at the same time. But, does ThreadSantizer catch it?

```
$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=28888)
  Read of size 4 at 0x7f73ed91c078 by thread T1:
    #0 Thread1 /home/zmick2/simple_race.c:7
        (exe+0x000000000a50)
    #1  :0 (libtsan.so.0+0x00000001b459)

  Previous write of size 4 at 0x7f73ed91c078 by main
      thread:
    #0 main /home/zmick2/simple_race.c:14
        (exe+0x000000000ac8)

  Thread T1 (tid=28889, running) created by main thread
      at:
    #0  :0 (libtsan.so.0+0x00000001f6ab)
    #1 main /home/zmick2/simple_race.c:13
        (exe+0x000000000ab8)

SUMMARY: ThreadSanitizer: data race
    /home/zmick2/simple_race.c:7 Thread1
==================
ThreadSanitizer: reported 1 warnings
```

If we compiled with the debug flag, then it would give us the variable name as well.

GDB

Introduction to gdb

Setting breakpoints programmatically A very useful trick when debugging complex C programs with GDB is setting breakpoints in the source code.

```
int main() {
    int val = 1;
    val = 42;
    asm("int $3"); // set a breakpoint here
    val = 7;
}
```

```
$ gcc main.c -g -o main && ./main
```

```
(gdb) r
[...]
Program received signal SIGTRAP, Trace/breakpoint trap.
main () at main.c:6
6     val = 7;
(gdb) p val
$1 = 42
```

Checking memory content   Memory Content

For example,

```
int main() {
    char bad_string[3] = {'C', 'a', 't'};
    printf("%s", bad_string);
}
```

```
$ gcc main.c -g -o main && ./main
$ Cat ZVQï£¡- $
```

```
(gdb) l
1 #include <stdio.h>
2 int main() {
3     char bad_string[3] = {'C', 'a', 't'};
4     printf("%s", bad_string);
5 }
(gdb) b 4
Breakpoint 1 at 0x100000f57: file main.c, line 4.
(gdb) r
[...]
Breakpoint 1, main () at main.c:4
4     printf("%s", bad_string);
(gdb) x/16xb bad_string
0x7fff5fbff9cd: 0x63 0x61 0x74 0xe0 0xf9  0xbf  0x5f  0xff
0x7fff5fbff9d5: 0x7f 0x00 0x00 0xfd 0xb5  0x23  0x89  0xff

(gdb)
```

Here, by using the x command with parameters 16xb, we can see that starting at memory address

`0x7fff5fbff9c` (value of `bad_string`), `printf` would actually see the following sequence of bytes as a string because we provided a malformed string without a null terminator.

Life in the terminal

```
0x43 0x61 0x74 0xe0 0xf9 0xbf 0x5f 0xff 0x7f 0x00
```

System Programming Jokes

Warning: Authors are not responsible for any neuro-apoptosis caused by these "jokes." - Groaners are allowed.

Light bulb jokes

Q. How many system programmers does it take to change a lightbulb?

A. Just one but they keep changing it until it returns zero.

A. None they prefer an empty socket.

A. Well you start with one but actually it waits for a child to do all of the work.

Groaners

Why did the baby system programmer like their new colorful blankie? It was multithreaded.

Why are your programs so fine and soft? I only use 400-thread-count or higher programs.

Where do bad student shell processes go when they die? Forking Hell.

Why are C programmers so messy? They store everything in one big heap.

System Programmer (Definition)

A system programmer is…

Someone who knows `sleepsort` is a bad idea but still dreams of an excuse to use it.

Someone who never lets their code deadlock… but when it does, causes more problems than everyone else combined.

Someone who believes zombies are real.

Someone who doesn't trust their process to run correctly without testing with the same data, kernel, compiler, RAM, filesystem size,file system format, disk brand, core count, CPU load, weather, magnetic flux, orientation, pixie dust, horoscope sign, wall color, wall gloss and reflectance, motherboard, vibration, illumination, backup battery, time of day, temperature, humidity, lunar position, sun-moon, co-position…

System Program (Definition)

A system program …

Evolves until it can send email.

Evolves until it has the potential to create, connect and kill other programs and consume all possible CPU,memory,network,… resources on all possible devices but chooses not to. Today.

Glossary

portable Works on multiple operating systems or machines. 7

POSIX TODO. 7