CS241 System Programming Wikibook

Contents

Chapter 1

Introduction

./introduction

Chapter 2

C Programming Language

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.

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. After

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.

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.

```c
#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"`)

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

Preprocessor

Parsing

Syntactic Parsing

Language Facilities

Keywords

C has an assortment of keywords

Operators

Common Functions

Input/Output

Parsing

string.h

Conventions/Errno

C Memory Model

Pointer Arithmetic

How are C strings represented?

They 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.

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

Some common ways to initialize a string include:

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

How do you declare 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.

```
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

```
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

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

How do you use a pointer to read/write some memory?

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`.

```
*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.

What is 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!

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

How do you print out pointer values? integers? strings?

Use format specifiers "%p" for pointers, "%d" for integers and "%s" for Strings. A full list of all of the format specifiers is found here

Example of integer:

```
int num1 =10;
printf("%d", num1); //prints num1
```

Example of integer pointer:

```
int *ptr =(int *) malloc(sizeof(int));
*ptr =10;
printf("%p\n", ptr); //prints the address pointed to by the pointer
printf("%p\n", &ptr); /*prints the address of pointer -- extremely useful
when dealing with double pointers*/
printf("%d", *ptr); //prints the integer content of ptr
return 0;
```

Example of string:

```
char *str =(char *) malloc(256 * sizeof(char));
strcpy(str, "Hello there!");
printf("%p\n", str); // print the address in the heap
printf("%s", str);
return 0;
```

Strings as Pointers & Arrays @ BU

How would you make standard out be saved to a file?

Simplest way: run your program and use shell redirection e.g.

```
./program > output.txt

#To read the contents of the file,
cat output.txt
```

More complicated way: close(1) and then use open to re-open the file descriptor. See [[http://cs-education.github.io/sys/#chapte
What's the difference between a pointer and an array? Give an example of something you can do with one but not the other.

```
char ary[] ="Hello";
char *ptr ="Hello";
```

Example

The array name points to the first byte of the array. Both `ary` and `ptr` can be printed out:

```
char ary[] ="Hello";
char *ptr ="Hello";
// Print out address and contents
printf("%p : %s\n", ary, ary);
printf("%p : %s\n", ptr, ptr);
```

The array is mutable, so we can change its contents (be careful not to write bytes beyond the end of the array though). Fortunately, 'World' is no longer than 'Hello"

In this case, the char pointer `ptr` points to some read-only memory (where the statically allocated string literal is stored), so we cannot change those contents.

```
strcpy(ary, "World"); // OK
strcpy(ptr, "World"); // NOT OK - Segmentation fault (crashes)
```

We can, however, unlike the array, we change `ptr` to point to another piece of memory,

```
ptr ="World"; // OK!
ptr =ary; // OK!
ary =(..anything..) ; // WONT COMPILE
// ary is doomed to always refer to the original array.
printf("%p : %s\n", ptr, ptr);
strcpy(ptr, "World"); // OK because now ptr is pointing to mutable memory (the
    array)
```

What to take away from this is that pointers * can point to any type of memory while C arrays [] can only point to memory on the stack. In a more common case, pointers will point to heap memory in which case the memory referred to by the pointer CAN be modified.

`sizeof()` returns the number of bytes. So using above code, what is sizeof(ary) and sizeof(ptr)?

`sizeof(ary)`: `ary` is an array. Returns the number of bytes required for the entire array (5 chars + zero byte = 6 bytes) `sizeof(ptr)`: Same as sizeof(char *). Returns the number bytes required for a pointer (e.g. 4 or 8 for a 32 bit or 64-bit machine) `sizeof` is a special operator. Really it's something the compiler substitutes in before compiling the program because the size of all types is known at compile time. When you have `sizeof(char*)` that takes the size of a pointer on your machine (8 bytes for a 64-bit machine and 4 for a 32 bit and so on). When you try `sizeof(char[])`, the compiler looks at that and substitutes the number of bytes that the **entire** array contains because the total size of the array is known at compile time.

```
char str1[] ="will be 11";
char* str2 ="will be 8";
sizeof(str1) //11 because it is an array
sizeof(str2) //8 because it is a pointer
```

Be careful, using sizeof for the length of a string!

Which of the following code is incorrect or correct and why?

```
int* f1(int *p) {
    *p = 42;
    return p;
} // This code is correct;

char* f2() {
    char p[] ="Hello";
    return p;
} // Incorrect!
```

Explanation: An array p is created on the stack for the correct size to hold H,e,l,l,o, and a null byte i.e. (6) bytes. This array is stored on the stack and is invalid after we return from f2.

```
char* f3() {
    char *p ="Hello";
    return p;
} // OK
```

Explanation: p is a pointer. It holds the address of the string constant. The string constant is unchanged and valid even after f3 returns.

```
char* f4() {
    static char p[] ="Hello";
    return p;
} // OK
```

Explanation: The array is static meaning it exists for the lifetime of the process (static variables are not on the heap or the stack).

How do you look up information C library calls and system calls?

Use the man pages. Note the man pages are organized into sections. Section 2 = System calls. Section 3 = C libraries. Web: Google “man7 open” shell: man -S2 open or man -S3 printf

How do you allocate memory 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);
```

What’s wrong with this string copy code?

```
void mystrcpy(char*dest, char* src) {
  // void means no return value
  while( *src ) {dest =src; src ++; dest++; }
}
```

In the above code it simply changes the dest pointer to point to source string. Also the nuls bytes are not copied. Here’s a better version -

```
  while( *src ) {*dest =*src; src ++; dest++; }
  *dest =*src;
```

Note it’s also usual to see the following kind of implementation, which does everything inside the expression test, including copying the nul byte.

```
  while( (*dest++ =*src++ )) {};
```

How do you write a strdup replacement?

```
// Use strlen+1 to find the zero byte...
char* mystrdup(char*source) {
   char *p =(char *) malloc ( strlen(source)+1 );
   strcpy(p,source);
   return p;
}
```

How do you unallocate memory on the heap?

Use free!

```
int *n =(int *) malloc(sizeof(int));
*n = 10;
//Do some work
free(n);
```

What is double free error? How can you avoid? What is a dangling pointer? How do you avoid?

A double free error is when you accidentally attempt to free the same allocation twice.

```
int *p =malloc(sizeof(int));
free(p);

*p = 123; // Oops! - Dangling pointer! Writing to memory we don't own anymore

free(p); // Oops! - Double free!
```

The fix is first to write correct programs! Secondly, it's good programming hygiene to reset pointers once the memory has been freed. This ensures the pointer can't be used incorrectly without the program crashing.

Fix:

```
p = NULL; // Now you can't use this pointer by mistake
```

What is an example of buffer overflow?

Famous example: Heart Bleed (performed a memcpy into a buffer that was of insufficient size). Simple example: implement a strcpy and forget to add one to strlen, when determining the size of the memory required.

What is 'typedef' and how do you use it?

Declares an alias for a type. Often used with structs to reduce the visual clutter of having to write 'struct' as part of the type.

```
typedef float real;
real gravity =10;
// Also typedef gives us an abstraction over the underlying type used.
// In the future, we only need to change this typedef if we
// wanted our physics library to use doubles instead of floats.

typedef struct link link_t;
//With structs, include the keyword 'struct' as part of the original types
```

In this class, we regularly typedef functions. A typedef for a function can be this for example

```
typedef int (*comparator)(void*,void*);

int greater_than(void* a, void* b){
    return a > b;
```

```
}
comparator gt =greater_than;
```

This declares a function type comparator that accepts two `void*` params and returns an integer.

Printing to Streams

How do I print strings, ints, chars to the standard output stream?

Use `printf`. The first parameter is a format string that includes placeholders for the data to be printed. Common format specifiers are `%s` treat the argument as a c string pointer, keep printing all characters until the NULL-character is reached; `%d` print the argument as an integer; `%p` print the argument as a memory address.

A simple example is shown below:

```
char *name =... ; int score =...;
printf("Hello %s, your result is %d\n", name, score);
printf("Debug: The string and int are stored at: %p and %p\n", name, &score );
// name already is a char pointer and points to the start of the array.
// We need "&" to get the address of the int variable
```

By default, for performance, `printf` does not actually write anything out (by calling write) until its buffer is full or a newline is printed.

How else can I print strings and single characters?

Use `puts( name )` and `putchar( c )` where name is a pointer to a C string and c is just a `char`

How do I print to other file streams?

Use `fprintf( _file_ , "Hello %s, score: %d", name, score);` Where _file_ is either predefined 'stdout' 'stderr' or a FILE pointer that was returned by `fopen` or `fdopen`

Can I use file descriptors?

Yes! Just use `dprintf(int fd, char* format_string, ...);` Just remember the stream may be buffered, so you will need to assure that the data is written to the file descriptor.

How do I print data into a C string?

Use `sprintf` or better `snprintf`.

```
char result[200];
int len =snprintf(result, sizeof(result), "%s:%d", name, score);
```

snprintf returns the number of characters written excluding the terminating byte. In the above example, this would be a maximum of 199.

What if I really really want `printf` to call `write` without a newline?

Use `fflush( FILE* inp )`. The contents of the file will be written. If I wanted to write "Hello World" with no newline, I could write it like this.

```
int main(){
    fprintf(stdout, "Hello World");
    fflush(stdout);
    return 0;
}
```

How is `perror` helpful?

Let's say that you have a function call that just failed (because you checked the man page and it is a failing return code). `perror(const char* message)` will print the English version of the error to stderr

```c
int main(){
    int ret =open("IDoNotExist.txt", O_RDONLY);
    if(ret < 0){
        perror("Opening IDoNotExist:");
    }
    //...
    return 0;
}
```

Parsing Input

How do I parse numbers from strings?

Use `long int strtol(const char *nptr, char **endptr, int base);` or `long long int strtoll(const char *nptr, char **endptr, int base);`.

What these functions do is take the pointer to your string `*nptr` and a `base` (ie binary, octal, decimal, hexadecimal etc) and an optional pointer `endptr` and returns a parsed value.

```c
int main(){
    const char *nptr ="1A2436";
    char* endptr;
    long int result =strtol(nptr, &endptr, 16);
    return 0;
}
```

Be careful though! Error handling is tricky because the function won't return an error code. If you give it a string that is not a number it will return 0. This means you cant differentiate between a valid "0" and an invalid string. See the man page for more details on strol behavior with invalid and out of bounds values. A safer alternative is use to `sscanf` (and check the return value).

```c
int main(){
    const char *input ="0"; // or "!##@" or ""
    char* endptr;
    long int parsed =strtol(input, &endptr, 10);
    if(parsed ==0){
        // Either the input string was not a valid base-10 number or it really was
            zero!

    }
    return 0;
}
```

How do I parse input using `scanf` into parameters?

Use `scanf` (or `fscanf` or `sscanf`) to get input from the default input stream, an arbitrary file stream or a C string respectively. It's a good idea to check the return value to see how many items were parsed. `scanf` functions require valid pointers. It's a common source of error to pass in an incorrect pointer value. For example,

```c
int *data =(int *) malloc(sizeof(int));
char *line ="v 10";
char type;
// Good practice: Check scanf parsed the line and read two values:
int ok =2 == sscanf(line, "%c %d", &type, &data); // pointer error
```

We wanted to write the character value into c and the integer value into the malloc'd memory. However, we passed the address of the data pointer, not what the pointer is pointing to! So `sscanf` will change the pointer itself. i.e. the pointer will now point to address 10 so this code will later fail e.g. when free(data) is called.

How do I stop scanf from causing a buffer overflow?

The following code assumes the scanf won't read more than 10 characters (including the terminating byte) into the buffer.

```
char buffer[10];
scanf("%s",buffer);
```

You can include an optional integer to specify how many characters EXCLUDING the terminating byte:

```
char buffer[10];
scanf("%9s", buffer); // reads up to 9 charactes from input (leave room for the
    10th byte to be the terminating byte)
```

Why is `gets` dangerous? What should I use instead?

The following code is vulnerable to buffer overflow. It assumes or trusts that the input line will be no more than 10 characters, including the terminating byte.

```
char buf[10];
gets(buf); // Remember the array name means the first byte of the array
```

`gets` is deprecated in C99 standard and has been removed from the latest C standard (C11). Programs should use `fgets` or `getline` instead.

Where each has the following structure respectively:

```
char *fgets (char *str, int num, FILE *stream);

ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Here's a simple, safe way to read a single line. Lines longer than 9 characters will be truncated:

```
char buffer[10];
char *result =fgets(buffer, sizeof(buffer), stdin);
```

The result is NULL if there was an error or the end of the file is reached. Note, unlike `gets`, `fgets` copies the newline into the buffer, which you may want to discard-

```
if (!result) {return; /* no data - don't read the buffer contents */}

int i =strlen(buffer) - 1;
if (buffer[i] =='\n')
    buffer[i] ='\0';
```

How do I use `getline`?

One of the advantages of `getline` is that will automatically (re-) allocate a buffer on the heap of sufficient size.

```
// ssize_t getline(char **lineptr, size_t *n, FILE *stream);

 /* set buffer and size to 0; they will be changed by getline */
char *buffer =NULL;
size_t size =0;

ssize_t chars =getline(&buffer, &size, stdin);

// Discard newline character if it is present,
if (chars > 0 && buffer[chars-1] =='\n')
    buffer[chars-1] ='\0';

// Read another line.
// The existing buffer will be re-used, or, if necessary,
```

```
// It will be ‘free‘’d and a new larger buffer will ‘malloc‘’d
chars =getline(&buffer, &size, stdin);

// Later... don’t forget to free the buffer!
free(buffer);
```

What common mistakes do C programmers make?

Memory mistakes

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";
```

Buffer overflow/ underflow

```
#define N (10)
int i =N, array[N];
for( ; i >= 0; i--) array[i] =i;
```

C does not check that pointers are valid. The above example writes into `array[10]` which is outside the array bounds. This can cause memory corruption because that memory location is probably being used for something else. In practice, this can be harder to spot because the overflow/underflow may occur in a library call e.g.

```
gets(array); // Let’s hope the input is shorter than my array!
```

Returning pointers to automatic variables

```
int *f() {
    int result =42;
    static int imok;
    return &imok; // OK - static variables are not on the stack
    return &result; // Not OK
}
```

Automatic variables are bound to stack memory only for the lifetime of the function. After the function returns it is an error to continue to use the memory. ## Insufficient memory allocation

```
struct User {
   char name[100];
};
typedef struct User user_t;

user_t *user =(user_t *) malloc(sizeof(user));
```

In the above example, we needed to allocate enough bytes for the struct. Instead, we allocated enough bytes to hold a pointer. Once we start using the user pointer we will corrupt memory. The correct code is shown below.

```
struct User {
   char name[100];
};
typedef struct User user_t;

user_t * user =(user_t *) malloc(sizeof(user_t));
```

Strings require `strlen(s)+1` bytes Every string must have a null byte after the last characters. To store the string "Hi" it takes 3 bytes: [H] [i] [].

```
  char *strdup(const char *input) {/* return a copy of 'input' */
    char *copy;
    copy =malloc(sizeof(char*));   /* nope! this allocates space for a pointer, not
        a string */
    copy =malloc(strlen(input));   /* Almost...but what about the null terminator?
        */
    copy =malloc(strlen(input) + 1); /* That's right. */
    strcpy(copy, input); /* strcpy will provide the null terminator */
    return copy;
}
```

Using uninitialized variables

```
int myfunction() {
  int x;
  int y =x + 2;
...
```

Automatic variables hold garbage (whatever bit pattern happened to be in memory). It is an error to assume that it will always be initialized to zero.

Assuming Uninitialized memory will be zeroed

```
void myfunct() {
   char array[10];
   char *p =malloc(10);
```

Automatic (temporary variables) are not automatically initialized to zero. Heap allocations using malloc are not automatically initialized to zero.

Double-free

```
  char *p =malloc(10);
  free(p);
//  .. later ...
  free(p);
```

It is an error to free the same block of memory twice. ## Dangling pointers

```
  char *p =malloc(10);
  strcpy(p, "Hello");
  free(p);
//  .. later ...
  strcpy(p,"World");
```

Pointers to freed memory should not be used. A defensive programming practice is to set pointers to null as soon as the memory is freed.

It is a good idea to turn free into the following snippet that automatically sets the freed variable to null right after:(vim - ultisnips)

```
snippet free "free(something)" b
free(${1});
$1 = NULL;
${2}
endsnippet
```

Logic and Program flow mistakes

Forgetting break

```
int flag =1; // Will print all three lines.
switch(flag) {
  case 1: printf("I'm printed\n");
  case 2: printf("Me too\n");
  case 3: printf("Me three\n");
}
```

Case statements without a break will just continue onto the code of the next case statement. The correct code is shown below. The break for the last statements is unnecessary because there are no more cases to be executed after the last one. If more are added, it can cause some bugs.

```
int flag =1; // Will print only "I'm printed\n"
switch(flag) {
  case 1:
    printf("I'm printed\n");
    break;
  case 2:
    printf("Me too\n");
    break;
  case 3:
    printf("Me three\n");
    break; //unnecessary
}
```

Equal vs. equality

```
int answer =3; // Will print out the answer.
if (answer =42) {printf("I've solved the answer! It's %d", answer);}
```

Undeclared or incorrectly prototyped functions

```
time_t start =time();
```

The system function 'time' actually takes a parameter (a pointer to some memory that can receive the time_t structure). The compiler did not catch this error because the programmer did not provide a valid function prototype by including `time.h`

Extra Semicolons

```
for(int i =0; i < 5; i++) ; printf("I'm printed once");
while(x < 10); x++ ; // X is never incremented
```

However, the following code is perfectly OK.

```
for(int i =0; i < 5; i++){
    printf("%d\n", i);;;;;;;;;;;;;
}
```

It is OK to have this kind of code, because the C language uses semicolons (;) to separate statements. If there is no statement in between semicolons, then there is nothing to do and the compiler moves on to the next statement

Other Gotchas

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]
```

C Preprocessor macros and side-effects

```
#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)

C Preprocessor macros and precedence

```
#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.

Does `sizeof` do anything?

```
int a =0;
size_t size =sizeof(a++);
printf("size: %lu, a: %d", size, a);
```

What does the code print out?

```
size: 4, a: 0
```

Because sizeof is not actually evaluated at runtime. The compiler assigns the type of all expressions and discards the extra results of the expression.

Strings, Structs, and Gotcha's

So what's a string?

In C we have Null Terminated strings rather than Length Prefixed for historical reasons. What that means for your average everyday programming is that you need to remember the null character! A string in C is defined as a

https://i.imgur.com/CgsxyZb.png

Figure 2.1: String

bunch of bytes until you reach ‘ or the Null Byte.

Two places for strings

Whenever you define a constant string (ie one in the form `char* str = "constant"`) That string is stored in the *data* or *code* segment that is **read-only** meaning that any attempt to modify the string will cause a segfault.

If one, however, `malloc`'s space, one can change that string to be whatever they want.

Memory Mismanagement

One common gotcha is when you write the following

```
char* hello_string =malloc(14);
                     --- --- --- --- --- --- --- --- --- --- --- --- --- ---
// hello_string ----> | g | a | r | b | a | g | e | g | a | r | b | a | g | e |
                     âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
                        âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
                        âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
hello_string ="Hello Bhuvan!";
// (constant string in the text segment)
// hello_string ----> [ "H" , "e" , "l" , "l" , "o" , " " , "B" , "h" , "u" , "v"
    , "a" , "n" , "!" , "\0" ]
                     --- --- --- --- --- --- --- --- --- --- --- --- --- ---
// memory_leak -----> | g | a | r | b | a | g | e | g | a | r | b | a | g | e |
                     âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
                        âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
                        âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
hello_string[9] ='t'; //segfault!!
```

What did we do? We allocated space for 14 bytes, reassigned the pointer and successfully segfaulted! Remember to keep track of what your pointers are doing. What you probably wanted to do was use a `string.h` function `strcpy`.

```
strcpy(hello_string, "Hello Bhuvan!");
```

Remember the NULL byte!

Forgetting to NULL terminate a string is a big affect on the strings! Bounds checking is important. The heart bleed bug mentioned earlier in the wiki book is partially because of this.

Where can I find an In-Depth and Assignment-Comprehensive explanation of all of these functions?

Right Here!

String Information/Comparison: `strlen` `strcmp`

`int strlen(const char *s)` returns the length of the string not including the null byte

`int strcmp(const char *s1, const char *s2)` returns an integer determining the lexicographic order of the strings. If s1 where to come before s2 in a dictionary, then a -1 is returned. If the two strings are equal, then 0. Else, 1.

With most of these functions, they expect the strings to be readable and not NULL but there is undefined behavior when you pass them NULL.

String Alteration: `strcpy strcat strdup`

`char *strcpy(char *dest, const char *src)` Copies the string at `src` to `dest`. **assumes dest has enough space for src**

`char *strcat(char *dest, const char *src)` Concatenates the string at `src` to the end of destination. **This function assumes that there is enough space for `src` at the end of destination including the NULL byte**

`char *strdup(const char *dest)` Returns a `malloc`'ed copy of the string.

String Search: `strchr strstr`

`char *strchr(const char *haystack, int needle)` Returns a pointer to the first occurrence of `needle` in the `haystack`. If none found, `NULL` is returned.

`char *strstr(const char *haystack, const char *needle)` Same as above but this time a string!

String Tokenize: `strtok`

A dangerous but useful function strtok takes a string and tokenizes it. Meaning that it will transform the strings into separate strings. This function has a lot of specs so please read the man pages a contrived example is below.

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

int main(){
    char* upped =strdup("strtok,is,tricky,!!");
    char* start =strtok(upped, ",");
    do{
        printf("%s\n", start);
    }while((start =strtok(NULL, ",")));
    return 0;
}
```

Output

```
strtok
is
tricky
!!
```

What happens when I change upped like this?

```
char* upped =strdup("strtok,is,tricky,,,!!");
```

Memory Movement: `memcpy` and `memmove`

Why are `memcpy` and `memmove` both in `<string.h>`? Because strings are essentially raw memory with a null byte at the end of them!

`void *memcpy(void *dest, const void *src, size_t n)` moves `n` bytes starting at `src` to `dest`. **Be careful**, there is undefined behavior when the memory regions overlap. This is one of the classic works on my machine examples because many times valgrind won't be able to pick it up because it will look like it works on your machine. When the autograder hits, fail. Consider the safer version which is.

`void *memmove(void *dest, const void *src, size_t n)` does the same thing as above, but if the memory regions overlap then it is guaranteed that all the bytes will get copied over correctly.

So what's a `struct`?

In low-level terms, a struct is just a piece of contiguous memory, nothing more. Just like an array, a struct has enough space to keep all of its members. But unlike an array, it can store different types. Consider the contact struct declared above

```
struct contact {
    char firstname[20];
    char lastname[20];
    unsigned int phone;
};

struct contact bhuvan;
```

Brief aside

```
/* a lot of times we will do the following typdef
 so we can just write contact contact1 */

typedef struct contact contact;
contact bhuvan;

/* You can also declare the struct like this to get
 it done in one statement */
typedef struct optional_name {
    ...
} contact;
```

If you compile the code without any optimizations and reordering, you can expect the addresses of each of the variables to look like this.

```
&bhuvan          // 0x100
&bhuvan.firstname // 0x100 =0x100+0x00
&bhuvan.lastname // 0x114 =0x100+0x14
&bhuvan.phone    // 0x128 =0x100+0x28
```

Because all your compiler does is say 'hey reserve this much space, and I will go and calculate the offsets of whatever variables you want to write to'.

What do these offsets mean?

The offsets are where the variable starts at. The phone variables starts at the 0x128th bytes and continues for sizeof(int) bytes, but not always. **Offsets don't determine where the variable ends though**. Consider the following hack that you see in a lot of kernel code.

```
typedef struct {
    int length;
    char c_str[0];
} string;

const char* to_convert ="bhuvan";
int length =strlen(to_convert);

// Let's convert to a c string
string* bhuvan_name;
bhuvan_name =malloc(sizeof(string) + length+1);
/*
Currently, our memory looks like this with junk in those black spaces
                ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___
 bhuvan_name =|   |   |   |   |   |   |   |   |   |   |   |
              âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
                 âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
*/
```

```
bhuvan_name->length =length;
/*
This writes the following values to the first four bytes
The rest is still garbage
                 --- --- --- --- --- --- --- --- --- --- ---
 bhuvan_name =| 0 | 0 | 0 | 6 | |   |   |   |   |   |   |
                âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
                   âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
*/


strcpy(bhuvan_name->c_str, to_convert);
/*
Now our string is filled in correctly at the end of the struct

                 --- --- --- --- --- --- --- --- --- --- ----
 bhuvan_name =| 0 | 0 | 0 | 6 | b | h | u | v | a | n | \0 |
                âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
                   âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿âĂ¿
*/

strcmp(bhuvan_name->c_str, "bhuvan") ==0 //The strings are equal!
```

But not all structs are perfect

Structs may require something called padding (tutorial). **We do not expect you to pack structs in this course, just know that it is there This is because in the early days (and even now) when you have to an address from memory you have to do it in 32bit or 64bit blocks. This also meant that you could only request addresses that were multiples of that. Meaning that

```
struct picture{
    int height;
    pixel** data;
    int width;
    char* enconding;
}
// You think picture looks like this
          height     data        width    encoding
          --- --- --- --- --- --- --- --- --- --- --- ---
picture =|      |             |      |             |
          âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
             âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
```

Would conceptually look like this

```
struct picture{
    int height;
    char slop1[4];
    pixel** data;
    int width;
    char slop2[4];
    char* enconding;
}
          height  slop1      data       width  slop2 encoding
```

```
          --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
picture =|      |      |             |      |      |             |
          âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
             âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
             âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
```

This is on a 64-bit system. This is not always the case because sometimes your processor supports unaligned accesses. What does this mean? Well there are two options you can set an attribute

```
struct __attribute__((packed, aligned(4))) picture{
    int height;
    pixel** data;
    int width;
    char* enconding;
}
// Will look like this
          height      data       width    encoding
          --- --- --- --- --- --- --- --- --- --- --- ---
picture =|      |             |      |             |
          âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
             âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
```

But now, every time I want to access `data` or `encoding`, I have to do two memory accesses. The other thing you can do is reorder the struct, although this is not always possible

```
struct picture{
    int height;
    int width;
    pixel** data;
    char* enconding;
}
// You think picture looks like this
          height  width       data        encoding
          --- --- --- --- --- --- --- --- --- --- --- ---
picture =|      |      |             |             |
          âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
             âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿ âĂ¿âĂ¿âĂ¿
```

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,

```
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 systems 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 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. http://www.cplusplus.com/reference/cassert/assert/

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=yes myprogram arg1 arg2
```

or

```
valgrind ./myprogram
```

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.

Example

https://i.imgur.com/ZdBWDvh.png

Figure 2.2: Valgrind Example

Here's an example to help you interpret the above results. 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;
}
```

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

```
==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 official 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.

```
#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: http://www.cs.cmu.edu/~gilpin/tutorial/

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 http://www.delorie.com/gnu/docs/gdb/gdb_56.html

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.

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

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?

```
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?

```
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

Threading

Intro to Threads

What is a thread?

A thread is short for 'thread-of-execution'. It represents the sequence of instructions that the CPU has (and will) execute. To remember how to return from function calls, and to store the values of automatic variables and parameters a thread uses a stack.

What is a Lightweight Process (LWP)? How does it relate to threads?

Well for all intents and purposes a thread is a process (meaning that creating a thread is similar to `fork`) except there is **no copying** meaning no copy on write. What this allows is for a process to share the same address space, variables, heap, file descriptors and etc.

The actual system call to create a thread is similar to `fork`; it's `clone`. We won't go into the specifics but you can read the man pages keeping in mind that it is outside the direct scope of this course.

LWP or threads are preferred to forking for a lot of scenarios because there is a lot less overhead creating them. But in some cases (notably python uses this) multiprocessing is the way to make your code faster.

How does the thread's stack work?

Your main function (and other functions you might call) has automatic variables. We will store them in memory using a stack and keep track of how large the stack is by using a simple pointer (the "stack pointer"). If the thread calls another function, we move our stack pointer down, so that we have more space for parameters and automatic variables. Once it returns from a function, we can move the stack pointer back up to its previous value. We keep a copy of the old stack pointer value - on the stack! This is why returning from a function is very quick - it's easy to 'free' the memory used by automatic variables - we just need to change the stack pointer.

In a multi threaded program, there are multiple stack but only one address space. The pthread library allocates some stack space (either in the heap or using a part of the main program's stack) and uses the `clone` function call to start the thread at that stack address. The total address space may look something like this.

https://i.imgur.com/ac2QDwu.png

Figure 3.1:

How many threads can my process have?

You can have more than one thread running inside a process. You get the first thread for free! It runs the code you write inside 'main'. If you need more threads you can call `pthread_create` to create a new thread using the pthread library. You'll need to pass a pointer to a function so that the thread knows where to start.

The threads you create all live inside the same virtual memory because they are part of the same process. Thus they can all see the heap, the global variables and the program code etc. Thus you can have two (or more) CPUs working on your program at the same time and inside the same process. It's up to the operating system to assign

the threads to CPUs. If you have more active threads than CPUs then the kernel will assign the thread to a CPU for a short duration (or until it runs out of things to do) and then will automatically switch the CPU to work on another thread. For example, one CPU might be processing the game AI while another thread is computing the graphics output.

Simple Usage

Hello world pthread example

To use pthreads you will need to include `pthread.h` AND you need to compile with `-pthread` (or `-lpthread`) compiler option. This option tells the compiler that your program requires threading support

To create a thread use the function `pthread_create`. This function takes four arguments:

[] int pthread$_create(pthread_t*thread, constpthread_attr_t * attr,void*(*start_routine)(void*),void*arg)$;

- The first is a pointer to a variable that will hold the id of the newly created thread.
- The second is a pointer to attributes that we can use to tweak and tune some of the advanced features of pthreads.
- The third is a pointer to a function that we want to run
- Fourth is a pointer that will be given to our function

The argument `void *(*start_routine) (void *)` is difficult to read! It means a pointer that takes a `void *` pointer and returns a `void *` pointer. It looks like a function declaration except that the name of the function is wrapped with `(* .... )`

Here's the simplest example:

[] include <stdio.h> include <pthread.h> // remember to set compilation option -pthread

void *busy(void *ptr) { // ptr will point to "Hi" puts("Hello World"); return NULL; } int main() { pthread$_t id; pthread_create(id, NU$

If we want to wait for our thread to finish use `pthread_join`

[] void *result; pthread$_join(id, result)$;

In the above example, `result` will be `null` because the busy function returned `null`. We need to pass the address-of result because `pthread_join` will be writing into the contents of our pointer.

More pthread functions

How do I create a pthread?

See Pthreads Part 1 which introduces `pthread_create` and `pthread_join`

If I call `pthread_create` twice, how many stacks does my process have?

Your process will contain three stacks - one for each thread. The first thread is created when the process starts, and you created two more. Actually there can be more stacks than this, but let's ignore that complication for now. The important idea is that each thread requires a stack because the stack contains automatic variables and the old CPU PC register, so that it can back to executing the calling function after the function is finished.

What is the difference between a full process and a thread?

In addition, unlike processes, threads within the same process can share the same global memory (data and heap segments).

What does `pthread_cancel` do?

Stops a thread. Note the thread may not actually be stopped immediately. For example it can be terminated when the thread makes an operating system call (e.g. `write`).

In practice, `pthread_cancel` is rarely used because it does not give a thread an opportunity to clean up after itself (for example, it may have opened some files). An alternative implementation is to use a boolean (int) variable whose value is used to inform other threads that they should finish and clean up.

What is the difference between `exit` and `pthread_exit`?

`exit(42)` exits the entire process and sets the processes exit value. This is equivalent to `return 42` in the main method. All threads inside the process are stopped.

`pthread_exit(void *)` only stops the calling thread i.e. the thread never returns after calling `pthread_exit`. The pthread library will automatically finish the process if there are no other threads running. `pthread_exit(...)` is equivalent to returning from the thread's function; both finish the thread and also set the return value (void *pointer) for the thread.

Calling `pthread_exit` in the the `main` thread is a common way for simple programs to ensure that all threads finish. For example, in the following program, the `myfunc` threads will probably not have time to get started.

[] int main() { pthread$_t$ $tid1, tid2; pthread_create(tid1, NULL, myfunc, "Jabberwocky"); pthread_create(tid2, NULL, myfu$
// No code is run after exit }

The next two programs will wait for the new threads to finish-

[] int main() { pthread$_t$ $tid1, tid2; pthread_create(tid1, NULL, myfunc, "Jabberwocky"); pthread_create(tid2, NULL, myfu$
// No code is run after pthread$_e$ $xit//Howeverprocesswillcontinuetoexistuntilboththreadshavefinished$}

Alternatively, we join on each thread (i.e. wait for it to finish) before we return from main (or call exit).

[] int main() { pthread$_t$ $tid1, tid2; pthread_create(tid1, NULL, myfunc, "Jabberwocky"); pthread_create(tid2, NULL, myfu$

Note the pthread_exit version creates thread zombies, however this is not a long-running processes, so we don't care. ## How can a thread be terminated? * Returning from the thread function * Calling `pthread_exit` * Cancelling the thread with `pthread_cancel` * Terminating the process (e.g. SIGTERM); exit(); returning from `main`

What is the purpose of pthread_join?

- Wait for a thread to finish
- Clean up thread resources
- Grabs the return value of the thread

What happens if you don't call `pthread_join`?

Finished threads will continue to consume resources. Eventually, if enough threads are created, `pthread_create` will fail. In practice, this is only an issue for long-running processes but is not an issue for simple, short-lived processes as all thread resources are automatically freed when the process exits.

Should I use `pthread_exit` or `pthread_join`?

Both `pthread_exit` and `pthread_join` will let the other threads finish on their own (even if called in the main thread). However, only `pthread_join` will return to you when the specified thread finishes. `pthread_exit` does not wait and will immediately end your thread and give you no chance to continue executing.

Can you pass pointers to stack variables from one thread to another?

Yes. However you need to be very careful about the lifetime of stack variables.

```
pthread_t start_threads() {
  int start = 42;
  pthread_t tid;
  pthread_create(&tid, 0, myfunc, &start); // ERROR!
  return tid;
}
```

The above code is invalid because the function `start_threads` will likely return before `myfunc` even starts. The function passes the address-of `start`, however by the time `myfunc` is executes, `start` is no longer in scope and its address will re-used for another variable.

The following code is valid because the lifetime of the stack variable is longer than the background thread.

```
void start_threads() {
  int start = 42;
  void *result;
  pthread_t tid;
  pthread_create(&tid, 0, myfunc, &start); // OK - start will be valid!
  pthread_join(tid, &result);
}
```

Intro to Race Conditions

How can I create ten threads with different starting values.

The following code is supposed to start ten threads with values 0,1,2,3,...9 However, when run prints out `1 7 8 8 8 8 8 8 8 10`! Can you see why?

[] include <pthread.h> void* myfunc(void* ptr) { int i = *((int *) ptr); printf("return NULL; }

int main() { // Each thread gets a different value of i to process int i; pthread$_t tid; for(i=0; i<10; i++){pthread_create(tid, NU$

The above code suffers from a `race condition` - the value of i is changing. The new threads start later (in the example output the last thread starts after the loop has finished).

To overcome this race-condition, we will give each thread a pointer to it's own data area. For example, for each thread we may want to store the id, a starting value and an output value:

[] struct T { pthread$_t id; int start; char result[100];};

These can be stored in an array -

```
struct T *info = calloc(10 , sizeof(struct T)); // reserve enough bytes for ten T structures
```

And each array element passed to each thread -

```
pthread_create(&info[i].id, NULL, func, &info[i]);
```

Why are some functions e.g. asctime,getenv, strtok, strerror not thread-safe?

To answer this, let's look at a simple function that is also not 'thread-safe'

[] char *to$_m$essage(int num){ char staticresult[256]; if(num <10)sprintf(result,"elsestrcpy(result,"Unknown");returnres

In the above code the result buffer is stored in global memory. This is good - we wouldn't want to return a pointer to an invalid address on the stack, but there's only one result buffer in the entire memory. If two threads were to use it at the same time then one would corrupt the other:

Time	Thread 1	Thread 2	Comments
1	to_m(5)		
2		to_m(99)	Now both threads will see "Unknown" stored in the result buffer

What are condition variables, semaphores, mutexes?

These are synchronization locks that are used to prevent race conditions and ensure proper synchronization between threads running in the same program. In addition, these locks are conceptually identical to the primitives used inside the kernel.

Are there any advantages of using threads over forking processes?

Yes! Sharing information between threads is easy because threads (of the same process) live inside the same virtual memory space. Also, creating a thread is significantly faster than creating(forking) a process.

Are there any dis-advantages of using threads over forking processes?

Yes! No- isolation! As threads live inside the same process, one thread has access to the same virtual memory as the other threads. A single thread can terminate the entire process (e.g. by trying to read address zero).

Can you fork a process with multiple threads?

Yes! However the child process only has a single thread (which is a clone of the thread that called `fork`. We can see this as a simple example, where the background threads never print out a second message in the child process.

[] include <pthread.h> include <stdio.h> include <unistd.h>
static pid$_t child = -2$;
void *sleepnprint(void *arg) { printf("
while (child == -2) {sleep(1);} /* Later we will use condition variables */
printf("
return NULL; } int main() { pthread$_t$ $tid1, tid2; pthread_create(tid1, NULL, sleepnprint,$"$NewThreadOne$"$);pthread_create($t
child = fork(); printf("sleep(3);
printf("
pthread$_e xit(NULL); return 0; /* Neverexecutes */$}

```
8970:New Thread One starting up...
8970:fork()ing complete
8973:fork()ing complete
8970:New Thread Two starting up...
8970:New Thread Two finishing...
8970:New Thread One finishing...
8970:Main thread finished
8973:Main thread finished
```

In practice, creating threads before forking can lead to unexpected errors because (as demonstrated above) the other threads are immediately terminated when forking. Another thread might have just lock a mutex (e.g. by calling malloc) and never unlock it again. Advanced users may find `pthread_atfork` useful however we suggest you usually try to avoid creating threads before forking unless you fully understand the limitations and difficulties of this approach.

Are there other reasons where `fork` might be preferable to creating a thread.

Creating separate processes is useful * When more security is desired (for example, Chrome browser uses different processes for different tabs) * When running an existing and complete program then a new process is required (e.g. starting 'gcc') * When you are running into synchronization primitives and each process is operating on something in the system

How can I find out more?

See the complete example in the man page And the pthread reference guide ALSO: Concise third party sample code explaining create, join and exit

Overview

The next section deals with what happens when pthreads collide, but what if we have each thread do something entirely different, no overlap?

We have found the maximum speedup parallel problems?

Embarrassingly Parallel Problems

The study of parallel algorithms has exploded over the past few years. An embarrassingly parallel problem is any problem that needs little effort to turn parallel. A lot of them have some synchronization concepts with them but not always. You already know a parallelizable algorithm, Merge Sort!

[] void merge$_s ort(int*arr, size_t len){if(len>1){//Mergesortthelefthalf//Mergesorttherighthalf//Mergethetwohalv$e

With your new understanding of threads, all you need to do is create a thread for the left half, and one for the right half. Given that your CPU has multiple real cores, you will see a speedup in accordance with Amdahl's Law. The time complexity analysis gets interesting here as well. The parallel algorithm runs in O(log^3(n)) running time (because we fancy analysis assuming that we have a lot of cores.

In practice though, we typically do two changes. One, once the array gets small enough, we ditch the parallel mergesort algorithm and do a quicksort or other algorithm that works fast on small arrays (something something cache coherency). The other thing that we know is that CPUs don't have infinite cores. To get around that, we typically keep a worker pool.

Worker Pool

We know that CPUs have a finite amount of cores. A lot of times we start up a number of threads and give them tasks as they idle.

Another problem, Parallel Map

Say we want to apply a function to an entire array, one element at a time.

[]

int *map(int (*func)(int), int *arr, size$_t$ len){$int * ret = malloc(len * sizeof(*arr));for(size_t i =0; i < len; ++i) ret[i] = func($

Since none of the elements depend on any other element, how would you go about parallelizing this? What do you think would be the best way to split up the work between threads.

Scheduling

There are a few ways to split up the work. * static scheduling: break up the problems into fixed size chunks (predetermined) and have each thread work on each of the chunks. This works well when each of the subproblems take roughly the same time because there is no additional overhead. All you need to do is write a loop and give the map function to each subarray. * dynamic scheduling: as a new problem becomes available have a thread serve it. This is useful when you don't know how long the scheduling will take * guided scheduling: This is a mix of the above with a mix of the benefits and the tradeoffs. You start with a static scheduling and move slowly to dynamic if needed * runtime scheduling: You have absolutely no idea how long the problems are going to take. Instead of deciding it yourself, let the program decide what to do!

source, but no need to memorize.

Few Drawbacks

You won't see the speedup right away because of things like cache coherency and scheduling extra threads.

Other Problems

From Wikipedia * Serving static files on a webserver to multiple users at once. * The Mandelbrot set, Perlin noise and similar images, where each point is calculated independently. * Rendering of computer graphics. In computer animation, each frame may be rendered independently (see parallel rendering). * Brute-force searches in cryptography.[8] Notable real-world examples include distributed.net and proof-of-work systems used in cryptocurrency. * BLAST searches in bioinformatics for multiple queries (but not for individual large queries) [9] * Large scale facial recognition systems that compare thousands of arbitrary acquired faces (e.g., a security or surveillance video via closed-circuit television) with similarly large number of previously stored faces (e.g., a rogues gallery or similar watch list).[10] * Computer simulations comparing many independent scenarios, such as climate models. * Evolutionary computation metaheuristics such as genetic algorithms. * Ensemble calculations of numerical weather prediction. * Event simulation and reconstruction in particle physics. * The marching squares algorithm * Sieving step of the quadratic sieve and the number field sieve. * Tree growth step of the random forest machine learning technique. * Discrete Fourier Transform where each harmonic is independently calculated.

Topics

- pthread lifecycle
- Each thread has a stack
- Capturing return values from a thread
- Using `pthread_join`
- Using `pthread_create`

- Using `pthread_exit`
- Under what conditions will a process exit

Questions

- What happens when a pthread gets created? (you don't need to go into super specifics)
- Where is each thread's stack?
- How do you get a return value given a `pthread_t`? What are the ways a thread can set that return value? What happens if you discard the return value?
- Why is `pthread_join` important (think stack space, registers, return values)?
- What does `pthread_exit` do under normal circumstances (ie you are not the last thread)? What other functions are called when you call pthread_exit?
- Give me three conditions under which a multithreaded process will exit. Can you think of any more?
- What is an embarrassingly parallel problem?

Chapter 4

Synchronization

./synchronization

Chapter 5

Deadlock

./deadlock

Chapter 6

Scheduling

./scheduling

Chapter 7

Interprocess Communication

./ipc

Chapter 8

Networking

./networking

Chapter 9

Filesystems

./filesystems

Chapter 10

Signals

./signals

Chapter 11

Review

./review

Chapter 12

Appendix

./appendix

Glossary

portable Works on multiple operating systems or machines. 7