Programming with C

3. Introduction to Scientific Computing with Linux
Part III. Basic Computing

Nidish Narayanaa B[1]

[1]Department of Aerospace Engineering
Indian Institute of Space Science & Technology, Trivandrum

FOSS Club, IIST, 2016

Outline

Outline

What C Sees When You C

The bare minimal intro

- Long story short (of what is to come in the next few slides), the computer understands in binary; people do in English
- We need software to convert english to machine language (compilers, interpretors, assemblers, ...) and from machine code back to human language (Execution)[1]

[1] Please note that C is not a person - he does not see when you C

Programming Languages

What are they anyways?

Words of Wisdom #1

A programming language is a language in which you program[2]

[2] Sorry for the awful joke - I feel the actual definition is better left to some private moments between you and google

Low-level and High-Level Languages

How **C**lose are you?

Low-Level Languages

- First there were Machine languages - you had to write instructions in binary

Low-level and High-Level Languages

How Close are you?

Low-Level Languages

- First there were Machine languages - you had to write instructions in binary
- Next, in the early 50s there came symbolic, or assembly languages (aka second-generation languages) using macros and mnemonics - these were translated into machine code by an *assembler*
- Mnemonics were actual letters representing operations instead of binary, while macros were a set of letters which were replaced by the binary instruction by the assembler

Low-level and High-Level Languages

How Close are you?

Low-Level Languages

- First there were Machine languages - you had to write instructions in binary
- Next, in the early 50s there came symbolic, or assembly languages (aka second-generation languages) using macros and mnemonics - these were translated into machine code by an *assembler*
- Mnemonics were actual letters representing operations instead of binary, while macros were a set of letters which were replaced by the binary instruction by the assembler
- These languages had a major drawback of lack of portability

Low-level and High-Level Languages

How Close are you?

Low-Level Languages

- First there were Machine languages - you had to write instructions in binary
- Next, in the early 50s there came symbolic, or assembly languages (aka second-generation languages) using macros and mnemonics - these were translated into machine code by an *assembler*
- Mnemonics were actual letters representing operations instead of binary, while macros were a set of letters which were replaced by the binary instruction by the assembler
- These languages had a major drawback of lack of portability
- To patch this portability came *high-level languages*

Low-level and High-Level Languages

How **C**lose are you?

High-Level Languages

- The programmer no longer had to care about hardware features and could be more comfortable with syntaxes that were closer to "natural language"

Low-level and High-Level Languages

How **C**lose are you?

High-Level Languages

- The programmer no longer had to care about hardware features and could be more comfortable with syntaxes that were closer to "natural language"
- These are translated into machine code by other programmes called compilers or interpretors

Low-level and High-Level Languages

How **C**lose are you?

High-Level Languages

- The programmer no longer had to care about hardware features and could be more comfortable with syntaxes that were closer to "natural language"
- These are translated into machine code by other programmes called compilers or interpretors
- Fortran is the name to remember (*For*mula *tran*slation, for those interested) - the first high-level programming language

Low-level and High-Level Languages

How **C**lose are you?

High-Level Languages

- The programmer no longer had to care about hardware features and could be more comfortable with syntaxes that were closer to "natural language"
- These are translated into machine code by other programmes called compilers or interpretors
- Fortran is the name to remember (*For*mula *tran*slation, for those interested) - the first high-level programming language
- LISP, COBOL, ALGOL, etc followed, catering to specific user subsets

Low-level and High-Level Languages

How **C**lose are you?

High-Level Languages

- The programmer no longer had to care about hardware features and could be more comfortable with syntaxes that were closer to "natural language"
- These are translated into machine code by other programmes called compilers or interpretors
- Fortran is the name to remember (*For*mula *tran*slation, for those interested) - the first high-level programming language
- LISP, COBOL, ALGOL, etc followed, catering to specific user subsets
- Then came C, created by Dennis Ritchie (Bell Laboratories)

Low-level and High-Level Languages

How Close are you?

High-Level Languages

- The programmer no longer had to care about hardware features and could be more comfortable with syntaxes that were closer to "natural language"
- These are translated into machine code by other programmes called compilers or interpretors
- Fortran is the name to remember (*For*mula *tran*slation, for those interested) - the first high-level programming language
- LISP, COBOL, ALGOL, etc followed, catering to specific user subsets
- Then came C, created by Dennis Ritchie (Bell Laboratories)
- It quickly became popular partly because Dennis Ritchie wrote a whole operating system, the UNIX, in C (a practice previously unknown in high-level languages)

Low-level and High-Level Languages

Beyond the Third Gen

- The Fourth generation of programming languages saw a host of non-procedural programming languages
- They specify what is to be accomplished without specifying how - the first of this was FORTH

Low-level and High-Level Languages

Beyond the Third Gen

- The Fourth generation of programming languages saw a host of non-procedural programming languages
- They specify what is to be accomplished without specifying how - the first of this was FORTH
- The fifth generation, currently still in its infancy is an outgrowth of artificial intelligence research
- I have got nothing to say about these

Interpreter and Compiler Based Languages

Are we understood (and how)?

Each programming language has to convert human-readable code into Machine code. There are two ways this is achieved - Interpreter-based and Compiler-based.

Interpreter

- Translates program one statement at a time
- No Object code generated
- Takes lesser code analysis time but more execution time
- Easier to debug
- Examples, Python, Ruby, Perl, etc.

Compiler

- Scans the entire program and translates it as a whole into machine code
- Intermediate Object code generated
- Takes more analysis time (compilation) but lesser execution time
- Slightly harder to debug
- Examples, C,C++

The Build Process in C

Take a bow, GCC

Building a C program usually entails the following steps :

1. Writing code
2. Compiling the code (.c files) into object files (.o files) after rectifying any coding errors detected in preliminary attempts at the foresaid
3. Linking the libraries and generating the executable program
4. The executable program can be run by calling its name from an active shell

References

- http://www.programiz.com/article/difference-compiler-interpreter
- http://www.infoplease.com/encyclopedia/science/programming-language-evolution-high-level-languages.html

Outline

Introduction
Programming Language Fundamentals
Baring It All
Interpreter and Compiler based languages
The Compile-Link Build process of C
References
C Programming Fundamentals
Libraries
File Streams
Functions
Miscellaneous Basics
C Programming Not-So-Fundamentals
Pre-Processors
Parsing Command-Line Arguments
Piping In and Piping Out
Processes and Pipes
Working with POSIX Threads
Programming With Class
References
Some Useful Libraries

The Concept of Libraries

You need them more than they need you

- C works as a set of variables, acted upon by functions
- The variables may be declared independently
- Functions, on the other hand, need to be *defined* and *declared* before being called
- Declarations are usually done in ".h" files while definitions are done in ".c" files. Many such ".c" files can be clubbed together into archives as ".a" files. These have to be "linked" at compile-time
- According to various standards (ANSI, K&R, etc), there are a set of basic functions which have become synonymous to the language itself
- The standard libraries (.a and .o files of the definitions) are automatically linked by each compiler - the user just has to include them in the code using

```
#include<_.h>
```

File Streams

And how to domesticate them

- A program interacts with the shell through buffers. These are devices to which output may be written to/read from. The main ones are listed below

Value	Macro	Description
1	stdout	Standard Output Buffer
2	stderr	Standard Error Buffer
3	stdin	Standard Input Buffer

- A file pointer can be declared as,

```
FILE* FVAR;
```

- In order to use a particular buffer, assign it by, `FVAR=stdout;`.

File Streams

File Opening

- To open a new file, call the `fopen` function from the `stdio.h` library as,

  ```
  FVAR = fopen("filename","modestring");
  ```

- The modestring may be,

`r`	read from beg
`r+`	read+write from beg
`w`	write from beg
`w+`	read+write from beg
`a`	append from beg
`a+`	read from beg+write from end

- Upon success, a `FILE` pointer is returned; upon failure, NULL is returned
- `fseek(stream,offset,whence)` may be used to move around in the file. There are macros corresponding to each operation.

File Streams

File Interaction

Stream Operations

- File output (writing into a file), may be done through `fprintf(stream,"string+specifiers",v1,v2,..);`
- File input (scanning from a file), may be done through `fscanf(stream,"string+specifiers",add1,add2,..);`

Write Operations

- File Output, `fwrite(address,blocksize,numblocks,Streams);`
- File Input, `fread(address,blocksize,numblocks,Streams);`
- These functions return the number of blocks written if successful and `NULL` otherwise

File Streams

Shell Interaction - The apple of our eye

- Once a program is run, all the file output calls to stdout may be piped to a separate file by
  `$ ./progname > filename`
- Similarly program input, which the program expects through scanf statements, may be given through
  `$ ./progname < infile`
- The above may be used to string together multiple programs taking outputs as inputs sequentially using the pipe command. The calls would look like
  `$ ./prog1|./prog2|...`
  More later
- It must be noted, unless otherwise piped, the `stderr` outputs are printed to the screen
- This behavior may be modified by adding `2>&1` as a command line argument. We are piping the stderr output (2) to stdout (1)

Functions

And the wizardry

- In order to organize code into more readable forms and avoid unnecessary typing, we have functions, which are constructs that conduct specific operations using specified variables
- A function may be declared as,

```
<return_type> function_name(function arguments);
```

- `<return_type>` specifies what the return value of the function has to be
- The function has got to be defined as,

```
<return_type> function_name(args)
{
    <function_body>
    return <return_variable>;
}
```

- The function may be called with the appropriate arguments to evaluate the statements in the function body

Functions

Passing Arguments

- The function arguments are specified within parantheses "()" as,

  ```
  (data_type var1,data_type var2,...)
  ```

- In the function body, these variables may be called by their names directly
- When an argument is passed in a function call, the passed argument's value is *copied* into the function argument variable. This is call *by value*. *Any changes made to the variable in the function will not alter the variable that was passed in the call*
- In order to call a variable *by reference* we may resort to pass the variable's address into the function as a parameter using the & operator

Functions

An Example - Call by Value

Declaration

```
int retsquare( int );
```

Definition

```
int retsquare( int a );
{
  int ret = a*a;
  return ret;
}
```

Call

```
vsq = retsquare(v);
```

Functions

An Example - Call by Reference

Declaration

```
void makesquare( int* );
```

Definition

```
void makesquare( int* a )
{
  *a = (*a)*(*a);
}
```

Call

```
v = 2;
makesquare(&v);
```

Pointers, Loops, Conditionals, ...

And the foundations

- The programming process in C consists of variables and calculations done on variables
- Toward making it a complete programming language, there are constructs called loops and conditionals
- Conditionals let us evaluate an expression and based on its result conduct operations
- Loops let us do the same operation multiple times over the same or a set of variables (arrays)
- To make use of variables easy we may store a set of them in the memory sequentially (arrays/pointers) or group a set together (structures)
- We may also restrict the kind of values a variable may take to optimize on memory usage (enumeration)
- There is another type of memory optimization which works through memory sharing in structures (unions)

Arrays

Declaration

Static `int A[10];`

Dynamic `int A[] = {1,2,3,4,5,6,7,8,9,10};`

Calls

The members are numbered 0 to 9, signifying the amount of increment to be added to the address of `a[0]`

Misc

An array has to be declared with a constant size only. If a variable sized array is desired, one can use `pointers`

Pointers

Declaration

`malloc` and `calloc` may be used to declare and assign memory to a pointer

Static
```
int *A;
A = malloc(N*sizeof(int));
```

Dynamic
```
int *A = calloc(N,sizeof(int));
```

In addition to doing the identical task, `calloc` assigns value 0 to all the memory locations. In general, `malloc` is faster than `calloc`.

Calls

`*A`, equivalently `A[0]`, access the first member;
`*(A+i)`, equivalently `A[i]`, access the i^{th} member
`free(A);` frees the memory allocated

Misc

Pointers to Pointers may be used for 2D constructs

Structures

Declaration

```
typedef struct{ int id;
                double val; }TYPE;
```

We have used `typedef` so that we may refer to this user-defined data-type with `TYPE`.

Declaration in Stack	**Declaration in Heap**
`TYPE A;`	`TYPE* A;`
`A.id = 1;`	`A = malloc(sizeof(TYPE));`
`A.val = 2.5;`	`A->id = 1;`
(or)	`A->val = 2.5;`
`A = (TYPE){1,2.5};`	(`A[0].id`, `A[0].val` are equivalent)

Unions

- In order to be memory efficient, we may specify that we will use only one variable at a time in a structure
- We can declare a union by replacing `struct` with `union` in the structure declaration statement
- Memory corresponding to the largest variable will be blocked for the structure and the same memory block will be shared between the constituent variables

Enumerations

- We may also be memory efficient by specifying the kind of values a variable of the (user-defined) data-type may take
- We have to declare these data types as enumerations

```
typedef enum{value1=12,value2=24}TYPE;
```

- The above statement declares a type `TYPE` whose variables may be assigned one of two values either by the string literals (`value1` or `value2`) or by their assigned integer values (12 or 24)

Conditionals

Expressions

- In C, all expressions either return a value or the `NULL` macro (usually upon failure)
- 1 is taken as `TRUE` and 0 is taken as `FALSE`
- In any conditional, the contained statements will be evaluated only if the expression is `NOT FALSE`, i.e., does not evaluate to 0. Some operators :

Expression	Name	Unary/Binary	Description
!	`NOT`	1	Boolean `NOT`
&&	`AND`	2	Boolean `AND`
||	`OR`	2	Boolean `OR`
==	Eq	2	0 if inequal
!=	Ineq	2	inverse of equality
<	lesser	2	
<=	less-eq	2	
>	greater	2	
>=	gr-eq	2	

Conditionals

The If Statement

Syntax

```
if ( expression1 ){
    <body_if_expression1_true>
}
else if( expression2 ){
    <body_if_expression2_true>
}
else{
    <body_if_all_false>
}
```

Conditionals

The Switch-Case Statement

This is used when the expression has to be evaluated for more than just 0 and `NOT` 0.

Syntax

```
switch (expression) {
  case val1: <body1>
             break;
  case val2: <body2>
             break;
  default: <body_default>
}
```

The behaviour is very similar to a nest of `if` statements. But note the break statements in each line. All the subsequent statements will be evaluated if not given.

Loops

The for Loop

- Every loop consists of an initialization, a conditional expression, and an increment
- In the `for` loop, all of these are explicitly stated

Syntax

```
for(<initialization>;<condl_expr>;<increment>){
  <Body_of_Loop>
}
```

There may be multiple initialization and increment statements separated by commas (`,`) but the conditional has to be a single expression.
The conditional expression determines when to terminate the loop.

Loops

The while Loops

While

```
while( <expression> ){
    <body_of_loop>
}
```

Do-While

```
do{
    <body_of_loop>
}while(<expression>);
```

- The loop will keep executing as long as `<expression>` returns `TRUE`.
- The increment has to be given inside the body of the loop in such a way that `<expression>` becomes false at some point or it will go on ad `infinitum`. We don't want those!

Outline

Pre-Processor

How can you code without writing code?

- Let's face it - writing code does get monotonous
- Sometimes we might want to alter a small functionality without actually editting parts of the code directly
- We employ Command-Line arguments and Pre-Processor varibles for this
- While pre-processors are more generic, Command-line arguments are convenient by design - but both have separate functionalities
- With **preprocessor variables**, we can decide whether or not to evaluate chunks of code according to our mood **at compile time**
- With **command line arguments**, we can alter values of variables **at run time**

Pre-Processor

How can you code without writing code?

- Preprocessor variables may be defined in the program using

  ```
  #define var val
  ```

- On encountering the above line, the compiler **replaces all occurrences of var with val**. The "\" symbol may be used to string together multiple lines
- To define a variable *at compile time*, we can use the `-D` flag in `gcc`

  ```
  $ gcc Main.c -Da=10
  ```

- Note that a value has also been given to the variable `a`. This is an optional functionality

Pre-Processor

Operations on the Pre-processor variables

- There are some operations which are permitted on the preprocessors. Note that these will be computed only once during the compilation.
  1. The `#define` and `#undef` may be used to define and undefine a variable. A defined preprocessor variable is referred to as a *macro* since it is completely replaced by its value at the end of compilation
  2. The `#ifdef` and `#ifndef` will check if a preprocessor variable is definedd or not. If yes/no (correspondingly), the chunk of code until a `#endif` is encountered is considered for compilation
  3. The `#else` and `#elif` may be seen as homologues to the `else` and `else if` constructs in C
- In addition to the above there are a few others which we will not go into presently

Pre-Processor

Operations on the Pre-processor variables

Main.c - Program with preprocessor dependent compilation

```
#include<stdio.h>
#ifndef a
int function(double var){ return var*var; }
#else
int function(double var){ return var; }
#endif
int main(){ int i;
  fprintf(stdout,"Enter i : ");
  fscanf(stdin,"%d",&i);
  fprintf(stdout,"function(i) = %d\n",function(i));
}
```

In this program there are alternate definitions for the same function `function()` based ontologically on a

Pre-Processor

Operations on the Pre-processor variables

- When the above program is compiled with,

  ```
  $ gcc Main.c -o Main -Da
  ```

  The output is,

  ```
  Enter i : 2
  function(i) = 2
  ```

- When the above program is compiled with,

  ```
  $ gcc Main.c -o Main -Ua
  ```

  The output is [U for `#undef`,

  ```
  Enter i : 2
  function(i) = 4
  ```

Pre-Processor

Using Macro Expansions

- Preprocessors, when used as macros, provide us with the advantage of faster execution (but with larger code size for obvious reasons)
- Consider a function to square it's argument :

```
double square(double a){
  return a*a;
}
```

- With a macro, this may be written as,

```
#define square(a) ((double)a*a)
```

- It is always advisable to explicitly typecast macro expansions and place the whole expression within parantheses, for the compiler merely *replaces* all the occurrences of the macro with its expansion

Parsing Command-Line Arguments

The first step towards building a presentable program

- A big part of what constitutes programming comfort in a UNIX environment is the command-line-argument parsing
- Command line options are options that may be given to a program during run-time to alter the program's functioning. For example, in the command,

  `cp file1 file2`

  `cp` is the program and `file1` and `file2` are its arguments, signifying the source and destination files for a copy operation.
- This gets very convenient since it lets us string together multiple programs together. We can thus reduce the amount of code we need to write per program
- The library `getopt` provides a set of easy to use functions for parsing command line options in a generic fashion

Parsing Command-Line Arguments

Command-Line Arguments in C

In C, the command line arguments are passed into the `main` function as arguments. If the main function is declared as,

```
int main(int argc,char* argv[]);
```

Then the integer `argc` stores the total number of command line arguments, and the string array `argv` stores an array of all the arguments. By default the delimiter for this is space.

Parsing Command-Line Arguments

The getopt class of functions

- In order to use the library and it's functions we have to include `#include<getopt.h>` at the head of the code. No linkers are necessary since the library is part of the standard specification and your compiler links the library files automatically

Parsing Command-Line Arguments

The getopt class of functions

- In order to use the library and it's functions we have to include #include<getopt.h> at the head of the code. No linkers are necessary since the library is part of the standard specification and your compiler links the library files automatically
- There is an inbuilt structure which lets us specify the kind of options and arguments our program expects.¡3-¿

```
struct option {
    const char *name;
    int has_arg;
    int *flag;
    int val;
};
```

An array of this type has to be specified listing out the details of the arguments

Parsing Command-Line Arguments

The getopt class of functions

The "option" Structure

Variable	Values	Description
`name`	string	Long name of the option
`has_arg`	0,1,2	0 - Option does not take an argument
		1 - Option takes an argument
		2 - Option takes an optional argument
`flag`	pointer	Pointer to store value to
`val`	int/char	Single character identifying the option

The `flag` variable may be left `NULL` for all practical purposes.

Parsing Command-Line Arguments

The getopt class of functions

For example, a program which should have 2 variables (a,b) set during run time by command line options must have an object declared as,

```
const struct option op_long[] = {
    {"help", 0, NULL, 'h'},
    {"value_a", 1, NULL, 'a'},
    {"value_b", 1, NULL, 'b'},
    {NULL, 0, NULL, 0}
};
```

- It is customary to have an option for help - in all this makes 3 arguments for our program
- Note the last member of `NULL`s - this is to tell the library that all the options have been specified.

Parsing Command-Line Arguments

The getopt class of functions

- In addition to the above structure, a string has to be declared with only the option value characters. For our examples it will be,

  ```
  const char* const op_short = "ha:b:";
  ```

- All the options that will take an argument are to be suffixed with a colon(:)
- In order to parse the command line options (glean the values) we use either of the functions `getopt` or `getopt_long`. In the case of the former, we need not declare the structure in the previous slide
- The functions return the option value characters if an option is detected and -1 if the command line has been completely parsed

Parsing Command-Line Arguments

Parsing with getopt

- We use a `do-while` loop to parse the command line as follows:

```
do{ next_option = getopt(argc,argv,op_short);
    switch( next_option ){
        case 'h': PrintUsage(); exit(1);
        case 'a': a = atof(optarg); break;
        case 'b': b = atof(optarg); break;
}}while(next_option!=-1);
```

- The argument (if enabled) to the option is stored in a global string, `optarg`
- In the `PrintUsage()` function you may write a brief description of the options to `stderr`
- We have used `atof` from the `stdlib` library to convert a string number into its equivalent floating point representation (`atof("12.6") -> 12.6`)
- To set value 1 to variable a we have to call the program with: `./progname -a 1`

Parsing Command-Line Arguments

Parsing with getopt-long

- Everything is similar to the above implementation except the function call. We use the `getopt_long()` function as follows:

```
next_option =
 getopt_long(argc,argv,op_short,op_long,NULL);
```

- Note that we have made use of the previously declared structure object `op_long` here
- The last argument has to specify the index of the array you want to start parsing with - `NULL` starts from the beginning
- A sample program has been given in a directory enclosed

Piping In and Piping Out

Making the Program Closer to Heart

Consider the following two programs.

Prog1 - Prints a sine wave over a period to `stdout`

```
#include<stdio.h>
#include<math.h>
int main(){double t;
  for(t=0;t<2.*M_PI;t+=0.01)
    fprintf(stdout,"%lf %lf\n",t,sin(t));
}
```

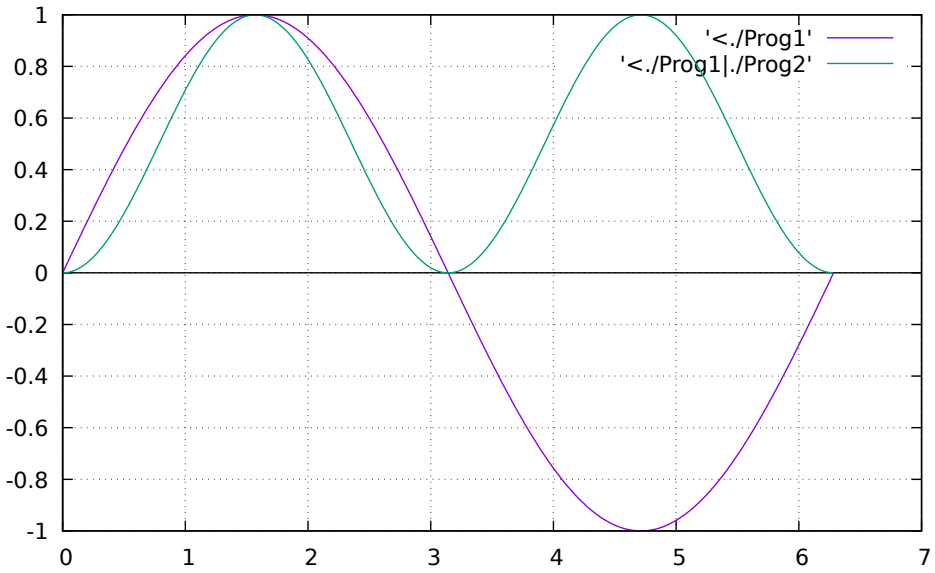
Prog2 - Reads from `stdin` and squares the second column

```
#include<stdio.h>
int main(){double t,v;
  while(fscanf(stdin,"%lf %lf",&t,&v)!=EOF)
    fprintf(stdout,"%lf %lf\n",t,v*v);
}
```

Piping In and Piping Out

Making the Program Closer to Heart

- Running `$ ./Prog1` will output 2 columns, $< t, sin(t) >$
- Piping this output directly to `Prog2` we can obtain $< t, sin^2(t) >$ by Running
  `$ ./Prog1|./Prog2`
- Piping these to `gnuplot` by
  `gp> p '<./Prog1' w l,'<./Prog1|./Prog2' w l`
  we may quickly see the two waveforms,



Processes and Pipes

Empower your Programming Experience

- All programs run as processes, interacting with the RAM for calculations, and file buffers & hard drive memory for outputs and inputs
- Processes are classified as

  Parent Process The process that has called any given process

  Child Process The process that is called by any given process
- Use `ps -ef` to see the currently running processes. The column PID(2nd column) corresponds to the Process ID of the process and the column PPID(3rd column) corresponds to the Parent Process ID
- All programs are cdhild processes of the `bash`

Processes

An Illustration

Output of `ps -ef`

Note that the invoked program `ps`, is a child process of `bash`[3], which is the shell program

```
nidish_+  2796  1201  0 14:23 ?        00:00:18 /usr/lib/gnome-terminal/gn
nidish_+  2803  2796  0 14:23 pts/2    00:00:00 bash
nidish_+  2862  2803  4 14:24 pts/2    00:01:32 emacs CTut_IISTFOSS.tex
nidish_+  2874  2803  0 14:24 pts/2    00:00:06 evince CTut_IISTFOSS.pdf
nidish_+  2880  1201  0 14:24 ?        00:00:00 /usr/lib/evince/evinced
root      2921     2  0 14:29 ?        00:00:00 [kworker/1:1]
root      2964     2  0 14:31 ?        00:00:00 [kworker/0:0]
root      2990     2  0 14:36 ?        00:00:00 [kworker/2:2]
root      3317     2  0 14:42 ?        00:00:00 [kworker/3:1]
root      3338     2  0 14:46 ?        00:00:00 [kworker/0:1]
root      3344     2  0 14:47 ?        00:00:00 [kworker/3:2]
root      3387     2  0 14:50 ?        00:00:00 [kworker/u8:0]
root      3397     2  0 14:51 ?        00:00:00 [kworker/2:0]
root      3405     2  0 14:53 ?        00:00:00 [kworker/1:2]
root      3406     2  0 14:53 ?        00:00:00 [kworker/3:0]
root      3431     2  0 14:55 ?        00:00:00 [kworker/u8:1]
root      3432     2  0 14:56 ?        00:00:00 [kworker/0:2]
root      3454     2  0 14:59 ?        00:00:00 [kworker/1:0]
nidish_+  3456  2803  0 15:00 pts/2    00:00:00 ps -ef
```

[3] Short for Bourne again SHell

The fork() function

Learn to give birth

- The `fork()` function creates an identical instance of the current process
- Once the command is issued, the two processes are completely distinct
- The function has a return value of `pid_t` - fear not! This is just an integer type, disguised by typedef
- In the parent process the fork function returns the pid of the child that is created, and in the child process the function returns 0
- This may be used to distinguish between the two processes to give different instructions to the parent and the child from the same source code

The fork() function

An Example

```
#include<stdio.h>
#include<unistd.h>
int main(){
  pid_t fpid,childpid;
  fpid = getpid();
  childpid = fork();
  if( childpid==0 ){
    fprintf(stdout,"I am a child.\n"
    "My PID is %d.\n"
    "My Parent's PID is %d.\n\n",getpid(),fpid);}
  else{
    fprintf(stdout,"I am a parent.\n"
    "My PID is %d.\n"
    "My Child's PID is %d.\n\n"
            ,getpid(),childpid);}
  return 0;
}
```

The fork() functions

An Example

The Output

```
I am a parent.
My PID is 4672.
My Child's PID is 4673.

I am a child.
My PID is 4673.
My Parent's PID is 4672.
```

- Whether the parent or the child is executed first is a question we can't address
- The `wait()` function may be use to make the parent wait till the child exits and get its exit status - look at the corresponding man pages for more

The execl() class of functions

Same name, different person

- The `execl()` class of functions may be used to make the current process into an instance of another program (same pid, different executable)
- The `execl()` function takes as arguments the name of the program and the command line arguments (appended by `NULL`) that are to be passed to it
- An example, to call `ps` with the argument `-ef` is,

```
execl(``ps'',''ps'',''-ef'',NULL);
```

  Note that the program name itself is the first argument
- Upon failure, the program returns a negative value
- This comes in handy when you have different routines that have to be implemented on the same set of values which can be passed on using command line arguments.

Interprocess Communications

The pipe() Function

- The pipe() function takes an array of integers as argument and writes read and write filedescriptors into them
- Say the argument is declared and passed as,

```
int filedes[2];
pipe(filedes);
```

- `filedes[0]` denotes the read file descriptor and `filedes[1]` denotes the write file descriptor
- Anything written to `filedes[1]` may be read from `filedes[0]`
- It is sufficient if the file descriptors are somehow known to the two processes that are to communicate
- In order to make this work with standard output and standard input, the dup2 function may be used to duplicate file descriptors

Interprocess Communcications

The dup2() Function

- In order to duplicate the read and write file descriptors as `stdin` and `stdout`, the corresponding macros may be used,

```
dup2(filedes[0],STDIN_FILENO);
dup2(filedes[1],STDOUT_FILENO);
```

- Writing in one process and reading from the other is known as half-duplex pipe - it is wise to close the corresponding file descriptors in each process using the `close(filedes[k])` call
- Writing and reading in/from both processes is known as full-duplex pipes - need to be a little careful when dealing with these
- Examples are given in the attached folder

POSIX Threads

Multitask more efficiently

- Honestly, interprocess communication is a little involved - it also involves more overhead
- An easier and more efficient way of processing in parallel is by using subprocesses, or threads
- We introduce the POSIX thread since it is the simplest to cover
- Each thread has a unique *thread ID* stored in the `pthread_t` data type
- A thread is created by assigning to it a function and the argument that is to be passed
- The global variables of the program are shared between all the threads and operations may be done on them parallely
- There are ways to *lock* the global variable to a thread so that none of the others may modify the variable while it is operated upon by the current thread

POSIX Threads

Basic Syntax

- A new thread may be created using the `pthread_create` function. It takes as arguments,
  1. Address of the variable storing the thread id
  2. pthread options (`NULL` works just fine for now)
  3. `void*` Function to be assigned. This has to have a `void*` arg. In C, a `void*` pointer could point to virtually ANYTHING
  4. The `void*` argument for the function
- A sample call would be,

```
pthread_create(&tid,NULL,function,arg);
```

- The main thread can choose to wait for a particular thread to complete execution using the `pthread_join` call which takes as arguments the thread id and a pointer to store the return variable. If you don't expect to do anything with the return variables, just pass `NULL` as below.

```
pthread_join(tid,NULL);
```

POSIX Threads

An Example

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>
void *function(void* a)
{
  int* p = (int*)a;
  fprintf(stdout,"Called Thread"
   " - %d - %d\n",pthread_self(),getpid());
  return 0;
}
int main()
{
  pthread_t tid;
  int i=10;
  pthread_create(&tid,NULL,function,&i);
  fprintf(stdout,"Primary Thread"
   " - %d - %d\n",pthread_self(),getpid());
  pthread_join(tid,NULL);
  return 0;
}
```

OUTPUT

```
Called Thread - 1697670912 - 2851
Primary Thread - 1705953024 - 2851
```

- The program creates a thread with the `pthread_create` call and after printing it's message, waits for the thread to finish with the `pthread_join` call
- We have also demonstrated the use of the `void*` argument under explicit type casting

POSIX Threads

Mutex Locking I

- Since the variables are globally defined, there is always chance for data corruption
- To handle this, we resort to "locking" the variables
- The mutex class of functions provide one such framework to do this
- Under this, each thread has to place a lock before it may perform an operation. Do note that it is upto the programmer to make sure that no thread utilized critical variables without locking them - no error would be thrown since this is just a framework
- A mutex variable may be declared and initialized by,

```
pthread_mutex_t lock;
pthread_mutex_init( &lock,NULL );
```

POSIX Threads

Mutex Locking II

- The variable `lock` only stores the lock status - it is not the shared data
- In a thread, a lock may be placed and removed respectively by,

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

- Suppose a sub-thread has placed a lock and the main thread has attempted to place one, the main thread waits till the sub-thread unlocks
- Remember to make sure your thread unlocks before it goes out of scope - not doing so may result in a `deadlock` where no thread will be able to use the variables

Programming With Class

Swag is for boys, Class is for programmers

- When you program, you have to take readability and presentation of the code very seriously - this helps in working on a code for a prolonged time
- There are some tools which help us do this effectively
- For starters, `GNU make` helps us streamline the compilation and linking process
- Although `make` is beyond the scope of the current presentation, a few tips may be noted:
- Organize your project directory into subdirectories `src/`, `obj/`, `include/`, `bin/`, and `examples/`.
- Store your ".c" files in `src/`, ".o" files in `obj/`, ".h" (and corresponding ".c") files in `include/`, executables in `bin/` and `examples/`
- The `example/` folder may be used for test runs - A `Makefile` in the root will make all this very handy

Programming With Class

Building your first library

library.h

```
#ifndef LIB_DEFD
#define LIB_DEFD

double square(double);
double cube(double);

#endif
```

library.c

```
#include<library.h>
double square(double a)
{  return a*a; }

double cube(double a)
{  return a*a*a; }
```

- Consider the above ".h" file with two functions `square()` and `cube()`
- They are declared in `library.h` and defined in `library.c`

Programming With Class

Building your first library

- In order to make this a library, we first make an object file by running, `$ gcc -c library.c -I.` [`-I.` denotes that `library.h` is in the current directory]
- This creates `library.o`, the corresponding object file
- We add this to an archive file, `libtest.a` by, `$ ar cr libtest.a library.o`
- Any number of object files may be added to an archive file. So you may have one small header file with only declarations and numerous `.c` files where one may find the function definitions
- Linking this to a program is easy, chuck the `lib` and the `.a` parts and give the rest of the name with a `-l` flag preceded by `-L.` specifying where the archive is,

  ```
  $ gcc Main.c -L. -ltest -I. -o Run
  ```

Sample programs may be found in a directory attached herewith

References

Linux `man` pages

Outline

Introduction

Programming Language Fundamentals

Baring It All

Interpreter and Compiler based languages

The Compile-Link Build process of C

References

C Programming Fundamentals

Libraries

File Streams

Functions

Miscellaneous Basics

C Programming Not-So-Fundamentals

Pre-Processors

Parsing Command-Line Arguments

Piping In and Piping Out

Processes and Pipes

Working with POSIX Threads

Programming With Class

References

Some Useful Libraries

Some Useful Libraries

Something worth your time

FFTW (Fastest Fourier Transform in the West) No idea about the name - but comes in really handly for signal processing and other applications

GSL (GNU Scientific Library) Another GNU produce - indispensable for my everyday existence - has got just about anything you might need

GL,GLU,GLUT (OpenGL class of Libraries) A very handy library to be abreast of the basics of - opened my eyes, I should say