More C

50.005 Computer System Engineering

Natalie Agus
INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Learning Objectives

In this intermediate class, we will learn how to:

1. Utilize function pointers
2. Create and utilize functions with `void*` return type
3. Receive command-line arguments
4. Handle File I/Os
5. Error checking and handling
6. Create and manage threads
7. Create and manage processes
8. Create shared memory for IPC

At the end of this class, you may head to e-dimension (Week 3) and answer the questions there to test your understanding. **The test grade is for personal achievement only and not included for computation of grades in 50.005.**

Part 1: Function Pointers

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. By definition, **pointers point to an address in any memory location**, they can also **point** to **at the beginning** of **executable code** as functions in memory.

They are declared in this format: `return_type (*function_name)(argument types)`

For example, create the following header file:

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


int sum(int* array, int size); //declaration of a function

//legal declaration and initialization of pointer to function
int (*sum_function_pointer)(int*, int) = &sum;
```

Here we declare two things:

1. The function `sum`
2. The pointer `sum_function_pointer`, pointing to the address of the function `sum`

You can easily call the function through its pointer as follows:

```c
#include "morec.h"

int sum(int* array, int size){
   int sum_value = 0;
   for (int i = 0; i<size; i++){
       sum_value += array[i];
   }
   return sum_value;
}

int main (){
   int array[10] = {1,2,3,4,5,6,7,8,9,10};

   //call the function using pointer
   int result = sum_function_pointer(array, 10);

   printf("The result is %d \n", result);
   return 0;
}
```

Notice:

1. The function `sum` receives an array in the form of integer pointer, along with its size
2. You can call the function through its pointer using `pointername(args)`

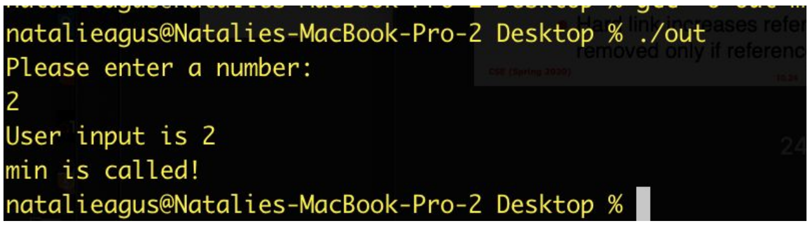
One of the reasons why function pointers are handy is if you have a whole array of them:

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

//declaration of function
int sum(int* array, int size);
int geometric_sum(int* array, int size);
int min(int* array, int size);
int max(int* array, int size);
int stdev(int* array, int size);
int average(int* array, int size);

//legal declaration and initialization of pointer to function
int (*function_pointers[])(int*, int) = {
   &sum, //index 0
   &geometric_sum, //index 1
   &min, //index 2
   &max, //index 3
   &stdev, // index 4
   &average //index 5
};
```

And that you wish to invoke them *by index*:

```
int main (){
   int array[10] = {1,2,3,4,5,6,7,8,9,10};
   char input;
   printf("Please enter a number: \n");
   scanf("%s", &input);

   int user_input = atoi(&input);

   printf("User input is %d \n", user_input);
   // select a function based on user input
   function_pointers[user_input](array, 10);

   return 0;
}
```

Assuming you have implemented the functions, you should be able to invoke a particular function given an index:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out
Please enter a number:
2
User input is 2
min is called!
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

Functions in the same function pointer array must have the same return type and argument types.

For functions that return a pointer, the pointer to that function can be written as:

```
int* test_function(int* array, int size);
int* (*test_function_pointer)(int*, int);
```

Which still follows the same format of `return_type (*function_name)(argument types)`.

Finally, the fun part about having function pointers is that we can **pass a function pointer** as an argument to another function.

For example, declare and initialize the pointer to the `sum` function, and a function that receives such pointer as its argument:

```c
int (*sum_function_pointer)(int*, int) = &sum;
int (*min_function_pointer)(int*, int) = &min;
void func (int (*f)(int*, int), int* array);
```

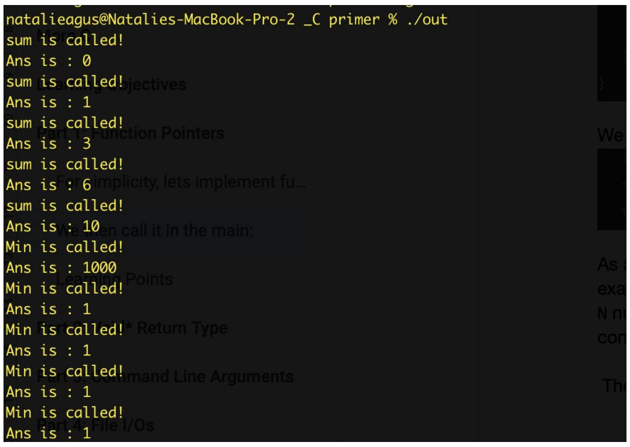
For simplicity, lets implement `func` as follows:

```c
void func(int (*f)(int *, int), int *array)
{
   for (int ctr = 0; ctr < 5; ctr++)
   {
       printf("Sum is : %d \n", (*f)(array, ctr));
   }
}
```

We then call it in the main:

```c
   int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   func(sum_function_pointer, array);
   func(min_function_pointer, array);
```

The output is:



```
natalieagus@Natalies-MacBook-Pro-2 _C primer % ./out
sum is called!
Ans is : 0
sum is called!
Ans is : 1
sum is called!
Ans is : 3
sum is called!
Ans is : 6
sum is called!
Ans is : 10
Min is called!
Ans is : 1000
Min is called!
Ans is : 1
Min is called!
Ans is : 1
Min is called!
Ans is : 1
Min is called!
Ans is : 1
```

In essence, we can ask the same function to run different other functions. In this simple example, of course it can be done by a simple if-else. In practice, this is useful if you have `N` number of functions with `M` different possible arguments. To automate executing all combinations, you'd want to pass function pointers plus its arguments around like this.

Learning Points

1. Create and utilize function pointers (a pointer that points to the start of a function)
2. Understand how to use utilize function pointers in an array
3. Call functions through its pointers
4. Pass function pointers as arguments to other functions.

Part 2: `Void*` Return Type

Void pointers are used during function declarations. We use a `void*` return type permits to return **any type.**

For example, consider this function declared at the header file, and its implementation:

```
void* special_function(int arg);
```

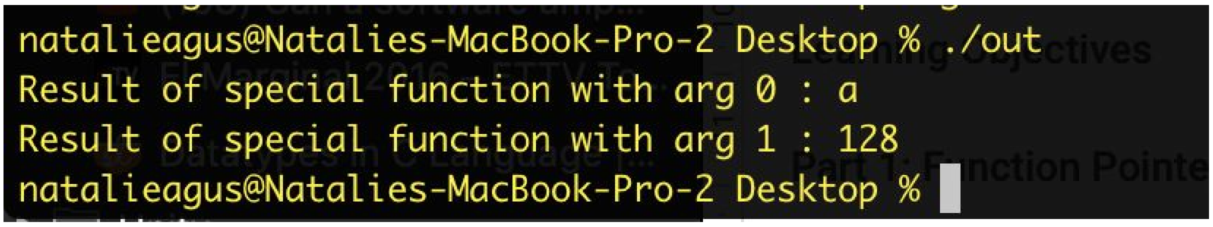
```
void* special_function(int arg){
   if (arg == 0){
       char* c = malloc(sizeof(char));
       c[0] = 'a';
       return c;
   }
   else{
       int* i = malloc(sizeof(int));
       i[0] = 128;
       return i;
   }
}
```

This shows that the function can return either pointer to a char, or pointer to an int, depending on the value of arg. Therefore we declare its return type void * (a generic pointer).

In the main function, we can call the function twice with different arguments:

```
   char c = *((char *)special_function(0));
   int i = *((int *) special_function(1));
   printf("Result of special function with arg 0 : %c \n", c);
   printf("Result of special function with arg 1 : %d \n", i);

   free(c);
   free(i);
```

Which of course as expected, results in different output based on the arg given:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out
Result of special function with arg 0 : a
Result of special function with arg 1 : 128
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

Part 3: Command Line Arguments

Command line arguments can be supplied when you call the executable from your terminal. Each argument is separated by **spaces**. Every process has **at least one argument**, which is itself.

The main function can *optionally* receive argument in this format:

```
int main (int argc, char** argv)
```

Where:

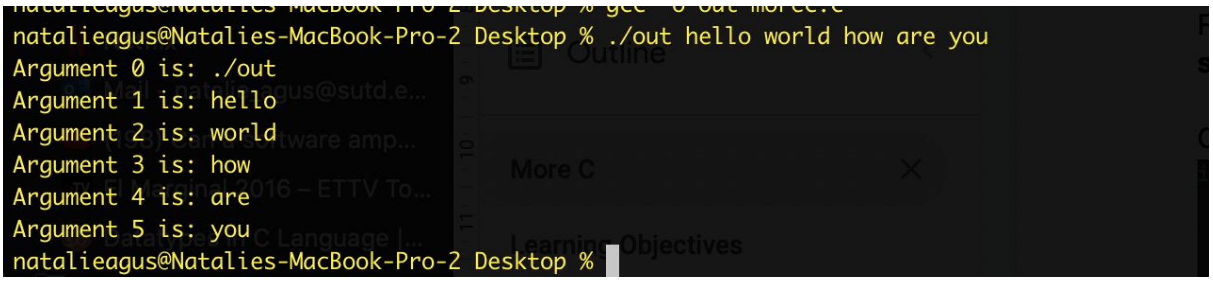
- `argc`: the number of arguments supplied in the command line
- `argv`: a double char pointer, where `argv[i]` points to the **start** of the i^{th} argument

Recall that a **string** itself is a `char*` type, since the pointer **points to the first `char` in the string**. Therefore an *array of strings* is naturally a `char**` type.

Consider this main function:

```
int main (int argc, char** argv){


   if (argc < 2){
       printf("Please supply arguments!\n");
   }


   for (int i = 0; i<argc; i++){
       printf("Argument %d is: %s \n", i, argv[i]);
   }
}
```

The following is the output:



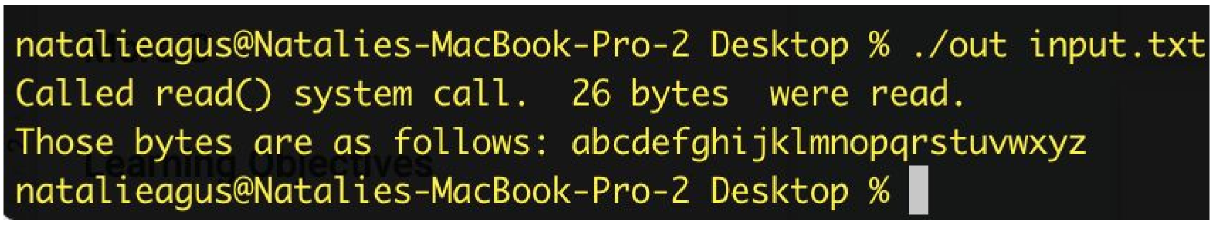
```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out hello world how are you
Argument 0 is: ./out
Argument 1 is: hello
Argument 2 is: world
Argument 3 is: how
Argument 4 is: are
Argument 5 is: you
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

Usually the arguments we supply are filenames, options, etc that are going to be **parsed** by the program (system programs for example allows many options) accordingly to do what we need it to do, e.g: `ls -la` (`-la` is an option to display file info in full format).

Part 4: File I/Os

In this section, we will learn how to open a file, read, and write to it programmatically using C. Before we can **read** or **write** to *any file*, we need to **open** it. After usage, we need to **close** it.

Suppose we want to have a program that receives a filename from the command line argument, as `argv[1]`,



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out input.txt
Called read() system call.  26 bytes  were read.
Those bytes are as follows: abcdefghijklmnopqrstuvwxyz
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

You need three more C standard libraries for this part. Include these in your header file:

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
```

We need to first check that there's at least *two arguments:*

```
   //check arguments
   if (argc < 2)
   {
       printf("Please key in filename\n");
       return 0;
   }
```

Afterwhich, you can use the `open()` system call:

```
   //open the file, with flag of O_RDONLY if you only want to read
   int fd = open(argv[1], O_RDONLY);

   //error checking
   if (fd < 0)
   {
       perror("Failed to open file. \n");
       exit(1);
   }
```

- The `open(filename, mode)` system call returns an **integer**, which is something called a *file descriptor*. You will learn more about this in **week 6.** For now, let's think of it as just an "id" to the file that you have opened.

- If `fd < 0`, it means that the `open()` operation is **not successful**, and you should handle this accordingly. We will talk more about *error handling* in the next part.
- This mode `O_RDONLY` is just to *read the file*. There's other mode: `O_RDWR` to read and write, and `O_CREAT` to create (if file that's being opened doesn't exist).
- You can later on `read()` or `write()` to this file by putting the `fd` as the argument to the functions.
- The filename can be *absolute* or ***relative path*** **(meaning that the path starts at the same current working directory).**
- For now, lets just assume that both the program and the file is in the same folder.

The next step is just to read from the opened file:

```
//initiallize a character array to contain what you will read later
 char char_buffer[128];

 //byte offset
 int byte = 0;

 //read 1 byte by 1 byte, put it into the buffer
 int check_read = read(fd, char_buffer + byte, 1);

 //keep reading 1 byte until nothing else to read
 while (check_read > 0)
 {
     byte++;
     check_read = read(fd, char_buffer + byte, 1);
 }
```

- At first, we need to **allocate** memory (`char_buffer`) to contain the space we need to read the file. As we learned before, this is *a static memory allocation.*
- Statically allocated memory is **automatically released** on the basis of scope, i.e., as soon as the scope of the variable is over, memory allocated get freed.
- *Note : we do not know the size of the file, but for know we know that it is lesser than 128 bytes.*
- We can then read the file using the `read(file_descriptor, array, amount)` system call.
- The system call will return the number of bytes read (the argument amount is basically asking the program to read *up to* `amount` of bytes).

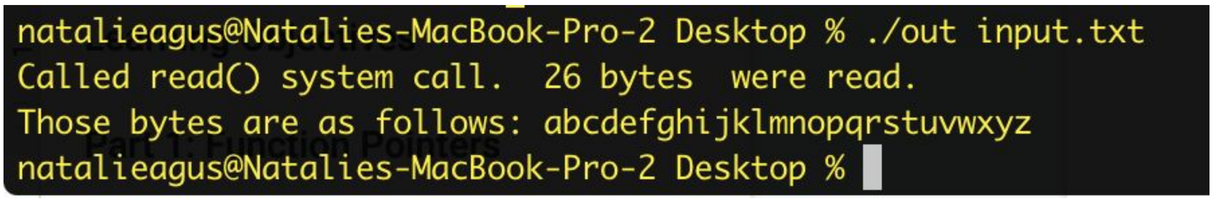
Then, we have the `while` loop there because we are primitively reading the contents of the file **byte by byte** and storing them in the array. That's why we need the `int byte` to take note the **offset address** from the beginning of the `char_buffer`.

If we have reached the end of file (EOF character), `read()` will return 0, and we will break out of the while-loop.

Finally we can print the content and **close the file**:

```
printf("Called read() system call.  %d bytes  were read.\n", byte);

//add terminating character so that you can print it
char_buffer[byte] = '\0';

printf("Those bytes are as follows: %s \n", char_buffer);

//close the file
close(fd);
```

With a sample `input.txt` file, the output of the program is:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out input.txt
Called read() system call.  26 bytes  were read.
Those bytes are as follows: abcdefghijklmnopqrstuvwxyz
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

To **write to file**, a similar protocol is used. Assume we need another argument for the filename of the file *to write to*, so lets modify our argument checking:

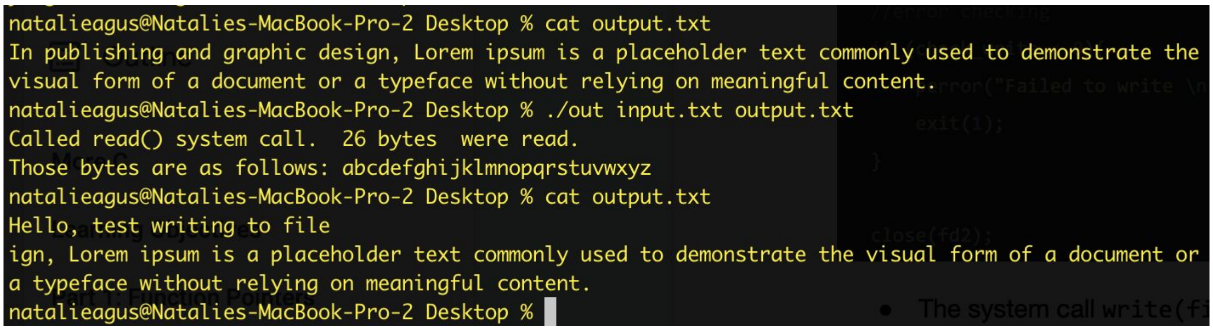
```
//check arguments
if (argc < 3)
{
    printf("Please key in filename\n");
    return 0;
}
```

And the code to write to file:

```
int fd2 = open(argv[2], O_RDWR|O_CREAT, 0666);
char sentence_to_write[128] = "Hello, test writing to file \n";

int check_write = write(fd2, sentence_to_write, strlen(sentence_to_write));

//error checking
if (check_write < 0){
    perror("Failed to write \n");
    exit(1);
}

close(fd2);
```

- The system call `write(file_descriptor, char* sentence, int length)` will write the stated number of length bytes to the file.
- It will return -1 upon *error*, hence it is good to check its status afterwards. More about error checking in the next part.
- Note that this time round, we have the third argument to system call `open()`. These are **file permission**, so that in the case that the file doesn't exist and that we create it, the value `0666` ensures that everybody has permission to read and write (all other users).

Notice that if output.txt **has already existed** with contents longer than the sentence we are intending to write, the **remainder** of the old content will still be there (its not "cleared").



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % cat output.txt
In publishing and graphic design, Lorem ipsum is a placeholder text commonly used to demonstrate the
visual form of a document or a typeface without relying on meaningful content.
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out input.txt output.txt
Called read() system call.  26 bytes  were read.
Those bytes are as follows: abcdefghijklmnopqrstuvwxyz
natalieagus@Natalies-MacBook-Pro-2 Desktop % cat output.txt
Hello, test writing to file
ign, Lorem ipsum is a placeholder text commonly used to demonstrate the visual form of a document or
a typeface without relying on meaningful content.
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

If you would like to *clear* and *overwrite the file*, you need to use O_TRUNC option instead:

```
int fd2 = open(argv[2], O_RDWR|O_CREAT|O_TRUNC, 0666);
```

This **truncates** (set its content to be 0 bytes) the file automatically as you open it.

If you would like to **append** to file:

```
   int fd2 = open(argv[2], O_RDWR|O_CREAT|O_APPEND, 0666);
```

There's also an alternative way to open the file, *which less primitive and more convenient to write:*

```
   FILE *out = fopen(argv[2], "a");
   if (out != NULL){
       fprintf(out, "%s", "Hello Again! \n");
       fclose(out);
   }
   else{
       perror("File failed to be opened\n");
       return 0;
   }
```

- We use fopen(filename, mode) instead. The mode is in terms of string, of which "a" means *append*.
- The system call fopen returns a FILE pointer type.

- You can *write* using `fprintf` instead, this is like *printing to console,* but we do it to the file instead.

Note: do NOT read AND write using the same file descriptor. You will NOT get the behaviour you expected because they share the *same pointer.*

Each time you read, the pointer will be advanced by n number of bytes which you tell it to read, and is *no longer at the beginning of the file.* You can set the absolute byte position (from the beginning of the file) of file pointer using the following before attempting to read() or write() to the file again. You will learn more about it in **week 6.**

```
lseek(file_descriptor, BYTE_absolute_location, SEEK_SET)
```

Learning Points

1. Learn how to **open,** read, and write to file
2. Know the existence of different modes to open a file depending on our purpose
3. Understand the notion of file pointer

Part 5: Error Handling

It is always a good habit to check for *errors* whenever we made a system call or other API calls. You can read the documentation first to check different return types. We have seen one above, for file `open()`:

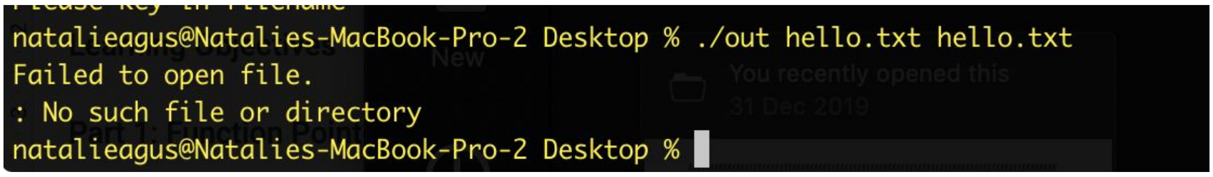
```
//open the file, with flag of O_RDONLY if you only want to read
int fd = open(argv[1], O_RDONLY);

//error checking
if (fd < 0)
{
    perror("Failed to open file. \n");
    exit(1);
}
```

It is very difficult to otherwise debug a C program that's successfully compiled but encounter errors at runtime. There's no default exception handling in C. All you can do is to check if the system call or API call is successful or not from its return value.

How to check error:

- **Always check the return value** of your system call or API call. There's plenty of these library-implemented functions that we have encountered so far regarding file operations and dynamic memory allocations.
- Have your program execute a `perror(),` that will print your custom error message plus the error message that the system has given you.

For example, an attempt to open non-existent file results in this error message:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out hello.txt hello.txt
Failed to open file.
: No such file or directory
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

If you have multiple file `open()`, then you **might want to print more helpful error message** to indicate exactly which `open(filename, mode)` operation causes the error.

Here is another example involving dynamic memory allocation using `malloc()`:

```
int n = 5;
int* ptr = (int*)malloc(n * sizeof(int));
 // Check if the memory has been successfully
// allocated by malloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
```

Upon memory allocation failure, the integer pointer `ptr` will *still* be pointing to a NULL location (only declared and not initialized).

In the next sections from now onwards, you will see such error handlings after each system call or API call. While it may seem tedious and clutter your code, it is very useful to do so to catch bugs during development. You will save a lot of time when debugging.

Learning Points

1. Understand the importance of success-check in API calls
2. Handle errors efficiently without having to print each line to catch the bug
3. Handle errors efficiently without the usual try-catch convenience

Part 6: C Threads

In this part we learn how to create C threads. You can do them using `pthread_create` method, as such:

```
pthread_create(&tid[i], NULL, functionPointerForThread, &point[i]);
```

You can read the manual of what each argument means, but in essence:

1. The first argument of `pthread_create` is the thread id variable.
2. The second argument of `pthread_create` is for creating threads with specific attributes, such as scheduling information, etc. You can leave it as `NULL` to be in default attribute.
3. The third argument is the function you want the thread to execute. The function has to be a type of `void*`, with argument of `void*` (**means it returns a pointer, and accepts a pointer as an argument**. A pointer of *ANY* type, hence `void*`)
4. The fourth argument is the **ADDRESS** of the argument to the function

As an exercise, let's create a thread. Firstly, you need to include the pthread library:

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

And then declare a thread id:

```
   //prepare an array of pthread_t (its thread id)
   pthread_t tid;
```

Then, we need to decide what function must each thread execute. The function must return a generic pointer void*. Let's create this function:

```
void *functionForThread(void* args);
```

Note that this function has:

1. Generic pointer to return
2. Generic pointer as argument
3. Basically these two are just addresses to any data type

If we want to supply more than 1 argument to the function, we need to create a struct that contains all our required arguments and pass the pointer to this struct. Lets for example, use the following struct:

```
/*
A struct creation for example
*/
typedef struct Coordinate{
   int x;
   int y;
   int id;
}Vec2D;
```

Now time to implement the function for demonstration purposes, we want to ask the thread to add the integers in the struct by 10.

```
void *functionForThread(void *args)
{

   //cast the argument into Vec2D type, because we know thats what we fed in as argument in
pthread_create
   Vec2D *myPoint_pointer = (Vec2D *)args;

   //accessing argument data through pointer
   printf("Hello from thread id %d! The coordinate passed is %d, %d \n",
myPoint_pointer->id, myPoint_pointer->x, myPoint_pointer->y);

   //sleep for 2 seconds
   sleep(2);
   //modify the argument
   myPoint_pointer->x = myPoint_pointer->x + 10;
   myPoint_pointer->y = myPoint_pointer->y + 10;

   //cast it back to void* as that's what we are supposed to return
   return (void*) myPoint_pointer;
}
```

Be very careful about the pointer **you pass back**. It is very important for the thread to **only return back its argument, or return a pointer to a memory that's been dynamically allocated using `malloc` or `calloc`**. This is because the **return** will destroy all stack-allocated (local) locations.

If you create a local variable like this:

```
   Vec2D myThreadCopy_Point = *myPoint_pointer;
```

And then return its *POINTER* to this local variable, there's **no guarantee** that the memory location still exists after the function exits, although there's a chance that you're lucky and that modern compiler version can detect and fix it for you automatically. This kind of bug is very hard to catch, especially when the memory location becomes *undefined* and your program still runs, but with the wrong output.

Now, the easy part is to create it in the main function:

```
   pthread_t tid;
   Vec2D point;
   point.x = 1;
   point.y = 2;
   point.id = 0;
   int thread_error_check = pthread_create(&tid, NULL, functionForThread, &point);

   //check error
   if (thread_error_check != 0)
   {
       perror("Failed to create thread. \n");
       exit(1);
   }
```

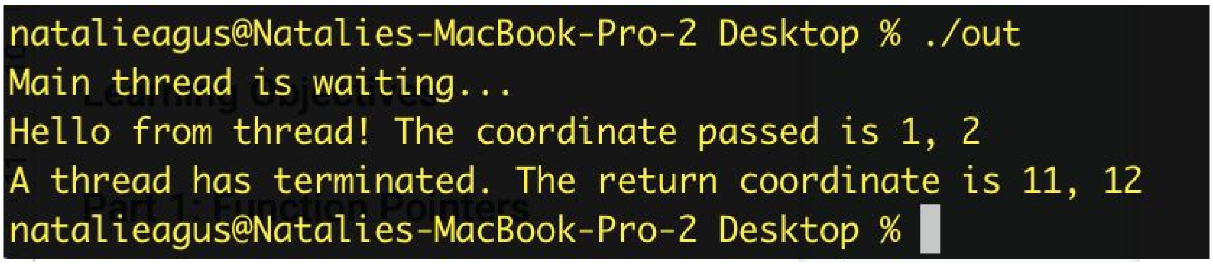
- We initialize the thread id `tid`, the struct `point`, and create the thread using `pthread_create`. You can ignore the second argument to the function, or if you are curious, read it yourself in the manual.
- Notice we pass the ***address*** of the `tid` declared and the **address** of the struct `point` declared.
- We then do the error check properly. At this point, if thread creation is successful, the created thread runs concurrently with the main.

To "reap" return value of the thread, the main thread can use `pthread_join`:

```
   printf("Main thread is waiting...\n");
   void* threadReturn = NULL;
   thread_error_check = pthread_join(tid, &threadReturn);

   if (thread_error_check != 0)
   {
       perror("Failed to join. \n");
       exit(1);
   }

   //cast it to Vec2D pointer type
   Vec2D *threadReturnPointerCasted = (Vec2D *)threadReturn;

   //print its content
   printf("A thread has terminated. The return coordinate is %d, %d \n",
threadReturnPointerCasted->x, threadReturnPointerCasted->y);
```

Recall how the created thread sleeps for 2 seconds up above. `pthread_join` is a blocking operation that allows the main thread to *wait* until the thread with the respective `tid` exits.

The output to the program is:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out
Main thread is waiting...
Hello from thread! The coordinate passed is 1, 2
A thread has terminated. The return coordinate is 11, 12
natalieagus@Natalies-MacBook-Pro-2 Desktop % 
```

Since there's no coordination between the main thread and the created thread, note that sometimes the first and the second sentence order can be the other way around.

Note that threads share:

- **Address space**
- **Heap**
- **Static data**
- **Code segments**
- **File descriptors**
- **Global variables**
- **And many others (signal handlers, etc)**

But threads have their own:

- **Program counter**
- **Registers**
- **Stack**
- **State**

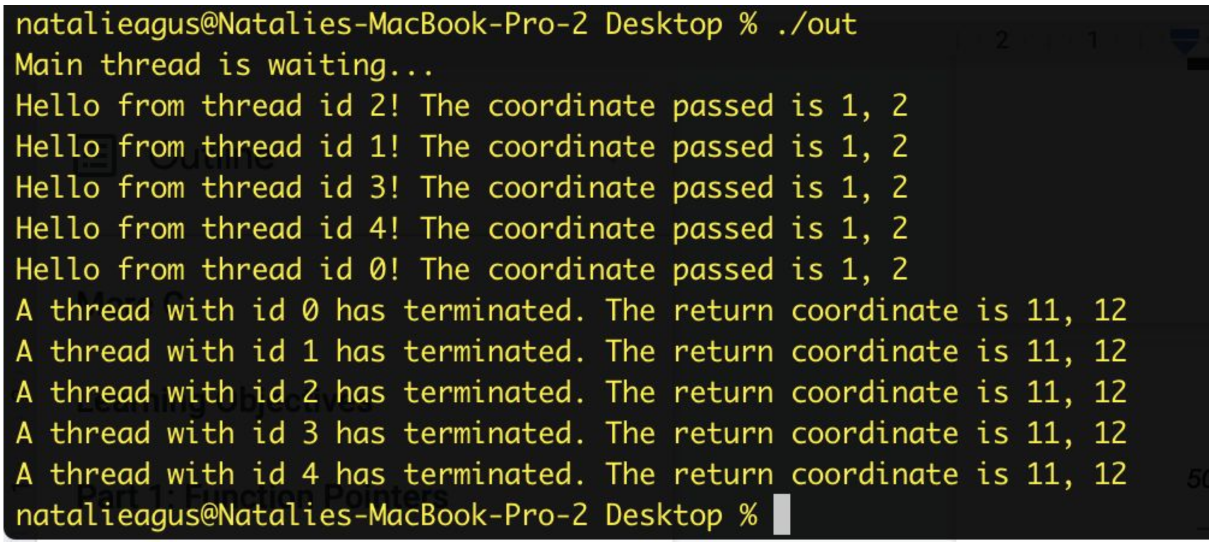
Therefore you are absolutely prone to dangerous race conditions when creating threads and allowing them to modify the same global variables at once. You will learn more about concepts of thread synchronization in class.

You can observe this *race condition* (although not severe, since its just **printing**) by creating several number of threads instead of one. Convert `tid` and `point` into arrays:

```
pthread_t tid[5];

Vec2D point[5];
for (int i = 0; i < 5; i++)
{
    point[i].x = 1;
    point[i].y = 2;
    point[i].id = i;
```

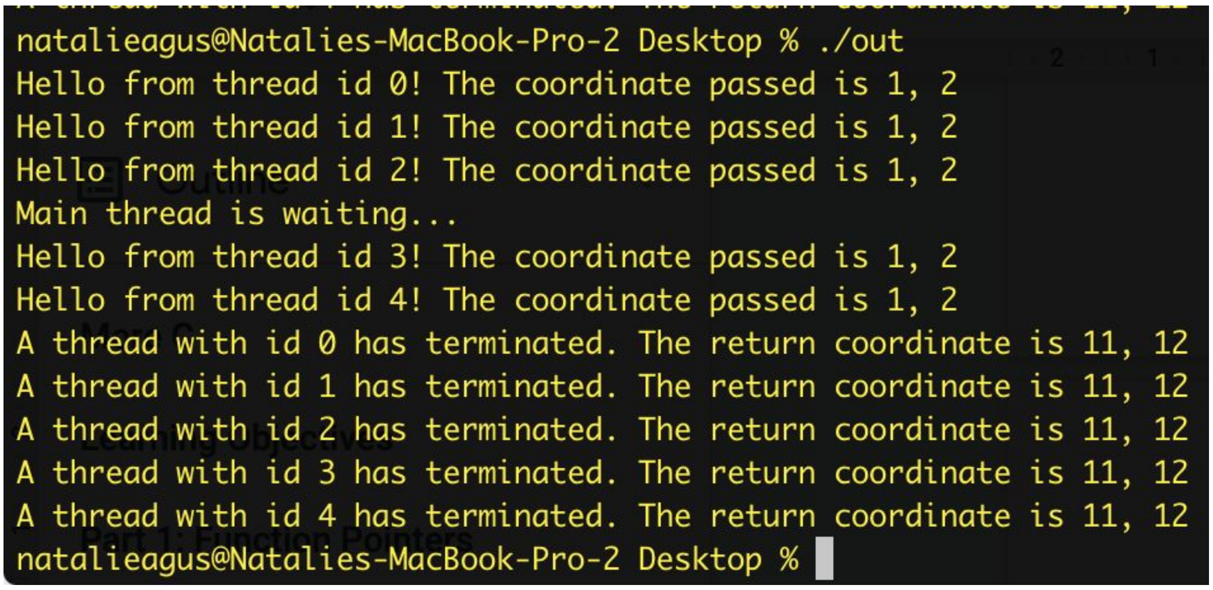
```
        int thread_error_check = pthread_create(&tid[i], NULL, functionForThread, 
&point[i]);
        //check error
        if (thread_error_check != 0)
        {
            perror("Failed to create thread. \n");
            exit(1);
        }
    }

    printf("Main thread is waiting...\n");
    void *threadReturn = NULL;

    for (int i = 0; i < 5; i++)
    {
        int thread_error_check = pthread_join(tid[i], &threadReturn);

        if (thread_error_check != 0)
        {
            perror("Failed to join. \n");
            exit(1);
        }

        //cast it to Vec2D pointer type
        Vec2D *threadReturnPointerCasted = (Vec2D *)threadReturn;

        //print its content
        printf("A thread with id %d has terminated. The return coordinate is %d, %d \n", 
threadReturnPointerCasted->id, threadReturnPointerCasted->x, threadReturnPointerCasted->y);
    }
```

The output will be something like:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out
Main thread is waiting...
Hello from thread id 2! The coordinate passed is 1, 2
Hello from thread id 1! The coordinate passed is 1, 2
Hello from thread id 3! The coordinate passed is 1, 2
Hello from thread id 4! The coordinate passed is 1, 2
Hello from thread id 0! The coordinate passed is 1, 2
A thread with id 0 has terminated. The return coordinate is 11, 12
A thread with id 1 has terminated. The return coordinate is 11, 12
A thread with id 2 has terminated. The return coordinate is 11, 12
A thread with id 3 has terminated. The return coordinate is 11, 12
A thread with id 4 has terminated. The return coordinate is 11, 12
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

Running it again may result in different print order:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out
Hello from thread id 0! The coordinate passed is 1, 2
Hello from thread id 1! The coordinate passed is 1, 2
Hello from thread id 2! The coordinate passed is 1, 2
Main thread is waiting...
Hello from thread id 3! The coordinate passed is 1, 2
Hello from thread id 4! The coordinate passed is 1, 2
A thread with id 0 has terminated. The return coordinate is 11, 12
A thread with id 1 has terminated. The return coordinate is 11, 12
A thread with id 2 has terminated. The return coordinate is 11, 12
A thread with id 3 has terminated. The return coordinate is 11, 12
A thread with id 4 has terminated. The return coordinate is 11, 12
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

Learning Points

1. Create, destroy, and wait for worker threads
2. Pass functions, arguments, and read return values from threads
3. Apply the knowledge of static memory and stack memory
4. Understand the presence of concurrency between threads

Part 7: `fork()`

This is probably the most important part of this handout, as it will need you to incorporate **plenty** of knowledge from the class. The system call fork() allows a process to create another process, which we call a *child process*. The parent and the child process created are **two separate processes**, isolated from one another with different **address space**, completely isolated from each other. Therefore, unlike threads, we cannot have a race condition with two processes that do not share data through any other **special** (interprocess communication) means.

To create a process, one has to call the `fork()` system call. At the point of successful `fork()`, the process **splits into two**, containing the same state of execution up to the point `fork()` is called. Then, you can dictate the child process and the parent process to execute separate instructions by checking the **return value of the `fork()`:**

Consider the following sample code:

```c
int forkReturnValue;

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

pid_t myPID = getpid();
printf("The main process id is %d \n", myPID);

forkReturnValue = fork();

//error checking
if (forkReturnValue < 0){
    perror("Failed to fork. \n");
    exit(1);
}

//child process
if (forkReturnValue == 0){
    //child process will have forkReturnValue of 0
    child_process_function(array, 10,0);
}
else
{
    //parent process will have forkReturnValue of > 0, which is the pid of the child
    //wait for a child (any 1)
    pid_t childPid = wait(NULL);
```

```c
        printf("Child process has finished. Main process exiting\n");
    }

    printf("The address of the array in pid %d starts at %p \n", getpid(), array);
    printf("The value of the array in pid %d is : ", getpid());
    for(int i = 0; i<10 ;i++){
        printf(" %d ", array[i]);
    }
    printf("\n");
```

When the system call fork() returns, the parent will go to the *else* clause, while the child will go to the *if* clause:

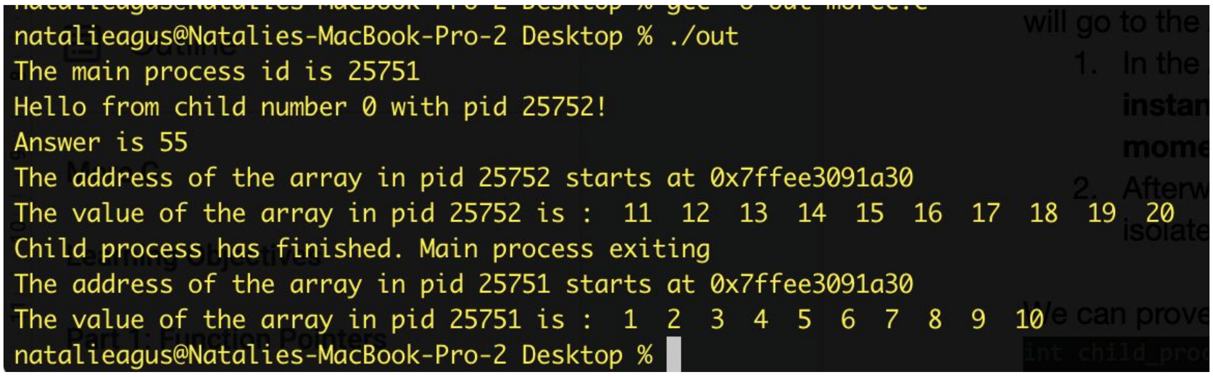
1. In the *if* clause that's executed by the child process, the value array is **already instantiated** since the parent and the child **shares the same state of code the moment `fork()` is called.**
2. Afterwards, the child and the parent process are totally separated processes, isolated from each other, in different **address space,** with different *process id.*
3. A parent process **has to wait for its child to terminate**, otherwise the child process **will become a zombie process** until the parent process terminates as well and kernel does a cleanup. This is done using `wait(NULL)` system call.

We can prove this by making another function for the child process to execute:

```c
int child_process_function(int* array, int size, int id){

    printf("Hello from child number %d with pid %d!\n", id, getpid());
    int answer = 0;
    for (int i = 0; i<size; i++){
        answer += array[i];
        array[i] += 10;
    }

    printf("Answer is %d\n", answer);
    return answer;

}
```

The function accepts three arguments, of which the value inside `array` has been initialized in the parent's process. Since child process inherits the state of the parent's up to `fork()`, then it also gets the instantiated value in `array`.

Below shows the output from executing the above code:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % gcc -o out more.c
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out
The main process id is 25751
Hello from child number 0 with pid 25752!
Answer is 55
The address of the array in pid 25752 starts at 0x7ffee3091a30
The value of the array in pid 25752 is :  11  12  13  14  15  16  17  18  19  20
Child process has finished. Main process exiting
The address of the array in pid 25751 starts at 0x7ffee3091a30
The value of the array in pid 25751 is :  1  2  3  4  5  6  7  8  9  10
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

Things to notice:

- The parent and child has different process id. You can print the current process' id using `getpid()` system call
- The **virtual address** of array in both processes are the same, but we can prove that they are *not a shared array* by printing out its content. In the child's process, each member of `array` is increased by 10 since thats what we asked it to do in child_process_function.
- The parent can only wait for the child to finish executing. However **it does not receive any return value from the child**. The return value of `wait(NULL)` *is the id of the terminated child.*

You can of course create multiple children, **but you have to be very careful and explicitly write instructions for the child process to** *not create more children* unless that's what you intend to do. Otherwise, your child will create more child, and you have exponentially many processes!

Below is a sample code on how to create 5 children and wait for all your children to terminate:

```
int forkReturnValue;


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


pid_t myPID = getpid();
printf("The main process id is %d \n", myPID);


for (int i = 0; i < 5; i++)
{
    forkReturnValue = fork();


    //error checking
    if (forkReturnValue < 0)
    {
        perror("Failed to fork. \n");
        exit(1);
    }

    //child process
    if (forkReturnValue == 0)
    {
        //child process will have forkReturnValue of 0
        child_process_function(array, 10, i);
        break; //dont create more children!
    }
}

//executed by parent process, since the forkReturnValue will retain the pid of the last child created
if (forkReturnValue != 0)
{

    while(wait(NULL) > 0); //wait for all children
    printf("Children processes has all finished. Main process exiting\n");
}

printf("The address of the array in pid %d starts at %p \n", getpid(), array);
printf("The value of the array in pid %d is : ", getpid());
for (int i = 0; i < 10; i++)
{
    printf(" %d ", array[i]);
}
printf("\n");
```

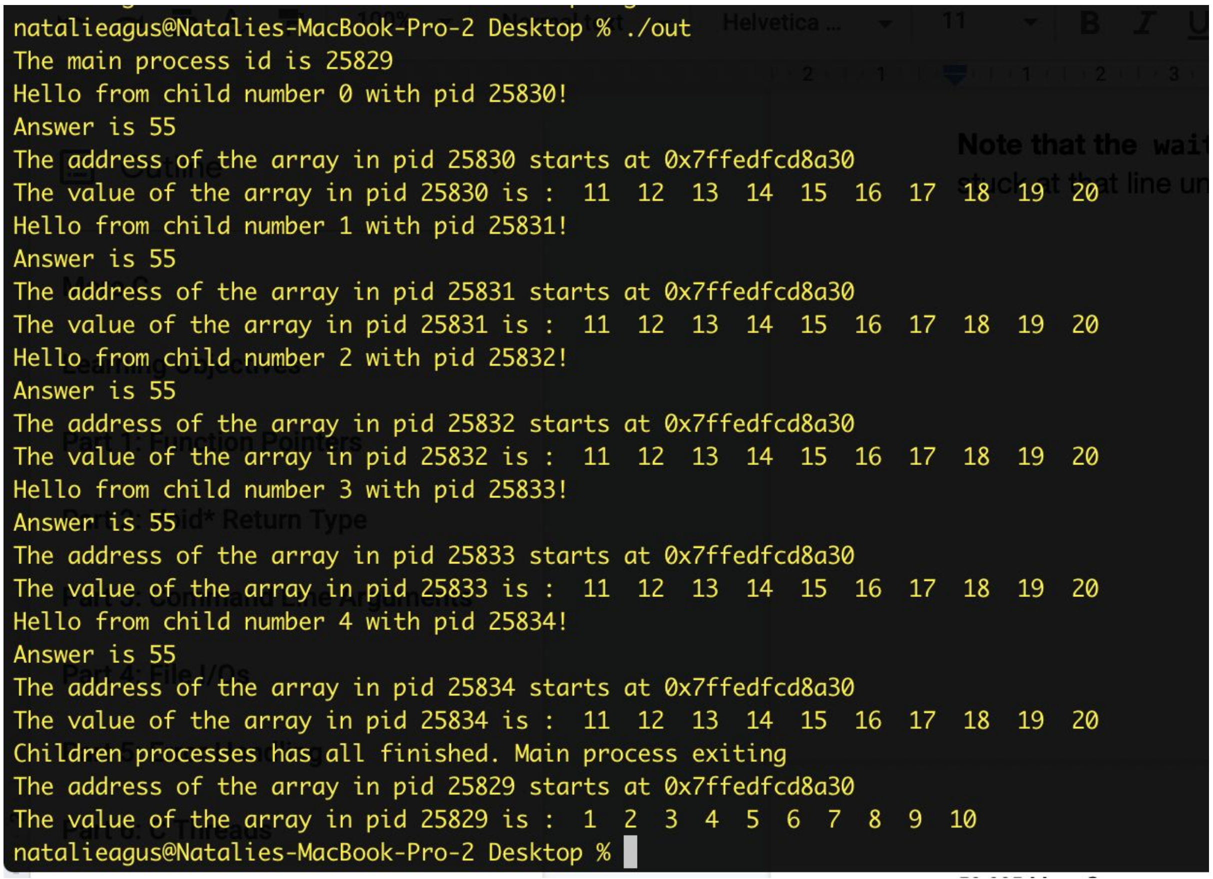
Here, `wait(NULL)` will always **return the pid of the terminated child,** unless there's no more children process, then `wait(NULL)` will return -1. Hence we use a while-loop until it fails.

Note that the `wait(NULL)` system call is *blocking*, meaning that the parent will be stuck at that line until one of its children terminated OR if there's no more live children.

What if there's no break indicated at the *if-clause* when `forkReturnValue` == 0? How many children (and grandchildren processes inclusive) are created?

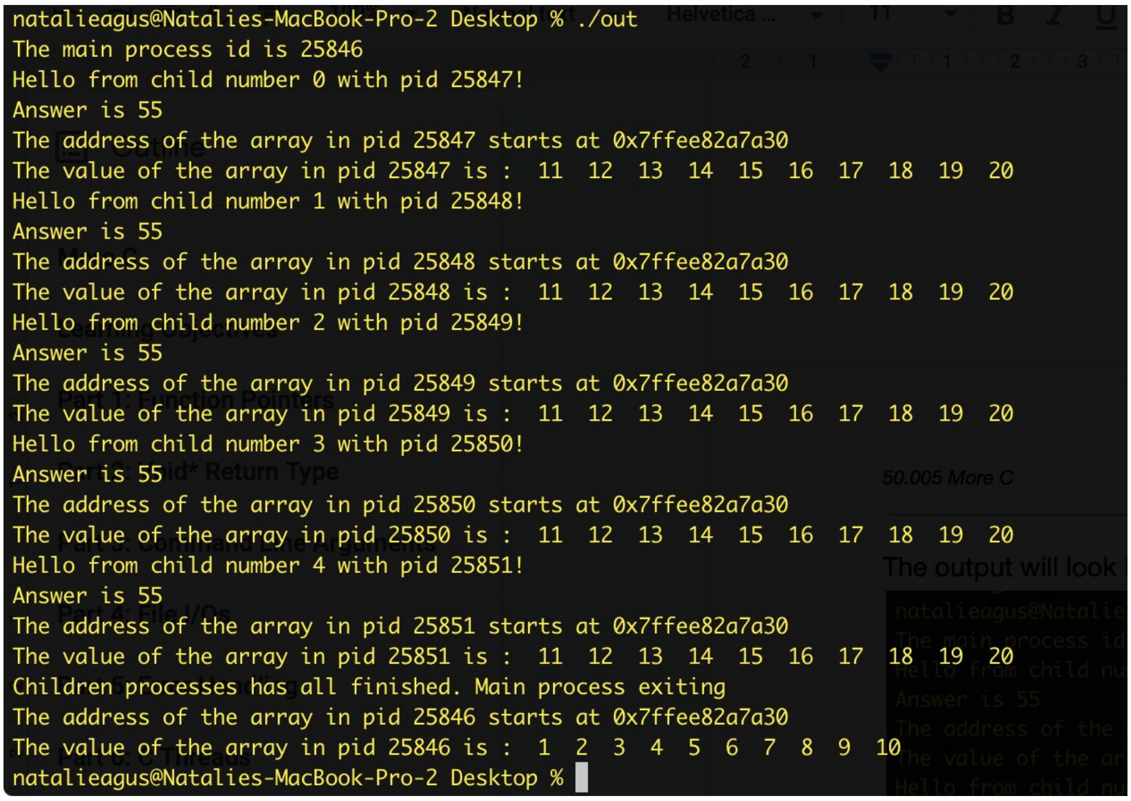
Plenty. The main process will still produce 5 children. Each of the children (lets call this 2nd generation) will produce 4 more children **each.** These 20 (3rd generation) children will produce 3 more children each, and so on. We're sure you can do the math yourself.

The output will look like something like this:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out
The main process id is 25829
Hello from child number 0 with pid 25830!
Answer is 55
The address of the array in pid 25830 starts at 0x7ffedfcd8a30
The value of the array in pid 25830 is :  11  12  13  14  15  16  17  18  19  20
Hello from child number 1 with pid 25831!
Answer is 55
The address of the array in pid 25831 starts at 0x7ffedfcd8a30
The value of the array in pid 25831 is :  11  12  13  14  15  16  17  18  19  20
Hello from child number 2 with pid 25832!
Answer is 55
The address of the array in pid 25832 starts at 0x7ffedfcd8a30
The value of the array in pid 25832 is :  11  12  13  14  15  16  17  18  19  20
Hello from child number 3 with pid 25833!
Answer is 55
The address of the array in pid 25833 starts at 0x7ffedfcd8a30
The value of the array in pid 25833 is :  11  12  13  14  15  16  17  18  19  20
Hello from child number 4 with pid 25834!
Answer is 55
The address of the array in pid 25834 starts at 0x7ffedfcd8a30
The value of the array in pid 25834 is :  11  12  13  14  15  16  17  18  19  20
Children processes has all finished. Main process exiting
The address of the array in pid 25829 starts at 0x7ffedfcd8a30
The value of the array in pid 25829 is :  1  2  3  4  5  6  7  8  9  10
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

Note that these processes are **concurrent**, so the printing by the children isn't always in the order of increasing pid:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out
The main process id is 25846
Hello from child number 0 with pid 25847!
Answer is 55
The address of the array in pid 25847 starts at 0x7ffee82a7a30
The value of the array in pid 25847 is :  11  12  13  14  15  16  17  18  19  20
Hello from child number 1 with pid 25848!
Answer is 55
The address of the array in pid 25848 starts at 0x7ffee82a7a30
The value of the array in pid 25848 is :  11  12  13  14  15  16  17  18  19  20
Hello from child number 2 with pid 25849!
Answer is 55
The address of the array in pid 25849 starts at 0x7ffee82a7a30
The value of the array in pid 25849 is :  11  12  13  14  15  16  17  18  19  20
Hello from child number 3 with pid 25850!
Answer is 55
The address of the array in pid 25850 starts at 0x7ffee82a7a30
The value of the array in pid 25850 is :  11  12  13  14  15  16  17  18  19  20
Hello from child number 4 with pid 25851!
Answer is 55
The address of the array in pid 25851 starts at 0x7ffee82a7a30
The value of the array in pid 25851 is :  11  12  13  14  15  16  17  18  19  20
Children processes has all finished. Main process exiting
The address of the array in pid 25846 starts at 0x7ffee82a7a30
The value of the array in pid 25846 is :  1  2  3  4  5  6  7  8  9  10
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

Learning Points

1. Know the difference between multithreading and multiprocessing
2. Apply the knowledge about process *isolation* thats learned in class
3. Create, and wait for children processes
4. Understand how zombie children are formed
5. Be aware of absence of shared data between processes unless a shared memory / socket is used.
6. Understand and witness the presence of concurrency between multiple processes thats run at the same time.

Part 8: Shared Memory

Recall that parent and child processes are isolated from one another, meaning that there's no way they can share any variable without using special protocols. One of the ways for processes to communicate that we learned in class is by using **shared memory**.

To use the shared memory library, you need to import these four libraries:

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <sys/wait.h>
```

Creating a shared memory is similar to creating dynamically allocated memory using `calloc` or `malloc` that we have learned. Just that there's four parts of it instead:

1. Allocate the shared memory using system call `shmget`
2. Attach the shared memory to the process' address space using `shmat`
3. After done, detach the shared memory from the process' address space using `shmdt`
4. Then deallocate the shared memory using `shmctl` (analogous to `free`)

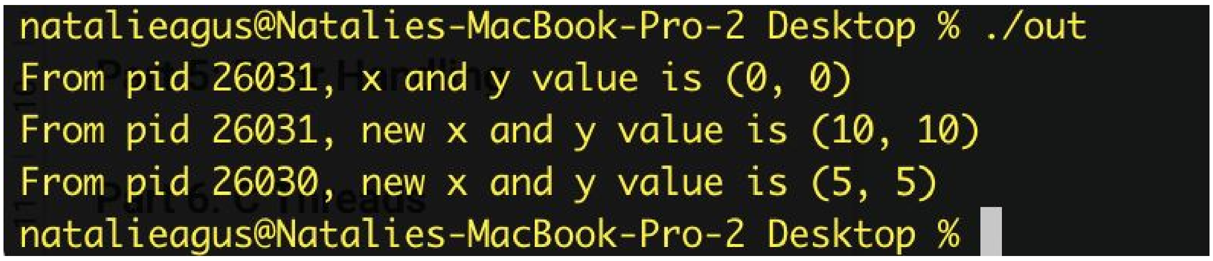
Lets digest this by example:

```
 //1. allocate shared memory, get its id
  int ShmID = shmget(IPC_PRIVATE, sizeof(Vec2D), S_IRUSR | S_IWUSR);
  //2. attach to address space
  Vec2D* ShmPTR = (Vec2D *)shmat(ShmID, NULL, 0);

  //init to zero
  ShmPTR->x = 0;
  ShmPTR->y = 0;

  int pid = fork();
  if (pid < 0)
  {
      printf("*** fork error (server) ***\n");
      exit(1);
  }
  else if (pid == 0)
  {
      printf("From pid %d, x and y value is (%d, %d) \n", getpid(), ShmPTR->x, ShmPTR->y);
      //sleep, hoping parent will finish by then
```

```
        sleep(5);
        printf("From pid %d, new x and y value is (%d, %d) \n", getpid(), ShmPTR->x,
ShmPTR->y);
        //child change the  shared memory value
        ShmPTR->x = 5;
        ShmPTR->y = 5;
        exit(0); //child exits
    }

    //sleep, hoping child will finish printing by then
    sleep(1);
    //parent code
    ShmPTR->x = 10;
    ShmPTR->y = 10;
    wait(NULL); //wait for child
    printf("From pid %d, new x and y value is (%d, %d) \n", getpid(), ShmPTR->x, ShmPTR->y);

    //3. detach shared memory
    shmdt((void *)ShmPTR);
    //4. delete shared memory
    shmctl(ShmID, IPC_RMID, NULL);
```

- Here, in step 1, just like `malloc` or `calloc`, we need to create a shared memory using `shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR)`
- You need not know the detailed meaning of the first and last argument, but if you are interested please read the manual. They have something to do with *permission* to read or write
- Then, in step 2, we need to attach it to the process' address space. This will return the type pointer which you can use normally later on.
- Afterwards, one of the processes must detach and delete it (which is usually the root/parent process)
- Otherwise, you will have **memory leak** until you restart your machine.

The output is as such:



```
natalieagus@Natalies-MacBook-Pro-2 Desktop % ./out
From pid 26031, x and y value is (0, 0)
From pid 26031, new x and y value is (10, 10)
From pid 26030, new x and y value is (5, 5)
natalieagus@Natalies-MacBook-Pro-2 Desktop %
```

As you can see, pid `26031` is the child, and `26030` is the parent. The one printing the lsat message is obviously the parent, because the parent explicitly **waits** for the child to terminate. Using shared memory, they can now *communicate* with each other and print out the values thats set by one another.

Of course these processes execute **concurrently**, but we kind of *synchronize it here* using `sleep(seconds)` function, that is: we are sure that the prints are sequential since the value of seconds in `sleep()` is big enough. However **this is not an ideal way to synchronize between processes as it is:**

1. **Inefficient**
2. **Can get incredibly complicated with large number of processes**
3. **No guarantee that it will avoid race condition**

Next week in class, you are going to learn more details on how to synchronize the execution between processes and threads where necessary.

Learning Points

1. Create, attach, deattach, and destroy shared memory
2. Understand the application of shared memory for interprocess communication
3. Understand the presence of memory leak

Note: you can check if you have properly detach and remove a shared memory using `ipcs` command.

Shared memory **persists** even after your process exits, and it will eat up your resources. To remove the unused shared memory, you can use observe the pid of the process that created the shared memory using `ipcs` command, and then remove it using `ipcrm -m pid.`