**C Basics for kids**

*50.005 Computer System Engineering*

***Materials taken from various resources. Suitable for children aged 4 and above.***

[Learning Objectives](#_grkcv3epkn13)

[Part 1: Function Pointers](#_qhpa1oknj0g6)

[Learning Points](#_4emk14w3fu91)

[Part 2: Void\* Return Type](#_2w9pqc7ub4mz)

[Part 3: Command Line Arguments](#_ri18yh9lrn4i)

[Part 4: File I/Os](#_s5jktf59ubc1)

[Learning Points](#_s8ctrxitahvo)

[Part 5: Error Handling](#_640s97564aiw)

[Learning Points](#_1p2sflytg5a3)

[Part 6: C Threads](#_4q792v30tdkg)

[Learning Points](#_qysiajo1mxz1)

[Part 7: fork()](#_h4ytk5yh056z)

[Learning Points](#_h88063oxivjy)

[Part 8: Shared Memory](#_b06jq7jr3xc0)

[Learning Points](#_9lxc343zyd1x)

[Part 9: Socket](#_9c6ayp2xoglj)

[Learning Points](#_lwr4svrsut6i)

[Summary](#_37nx294qo77h)

# Learning Objectives

**In this class we will learn how to:**

1. Create and manage threads
2. Create and manage processes
3. Create shared memory for IPC
4. Create sockets 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: 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 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 the 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);  } |
| --- |

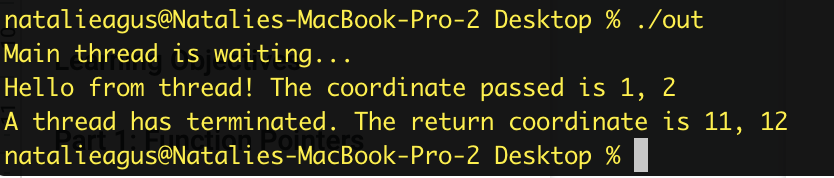
* We initialize the thread id tid, the struct point, and create the thread using pthread\_create. You can ignore the second argument to the function, or if you are curious, read it yourself in the manual.
* Notice we pass the ***address***of the tid declared and the **address** of the struct point declared.
* 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:



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**

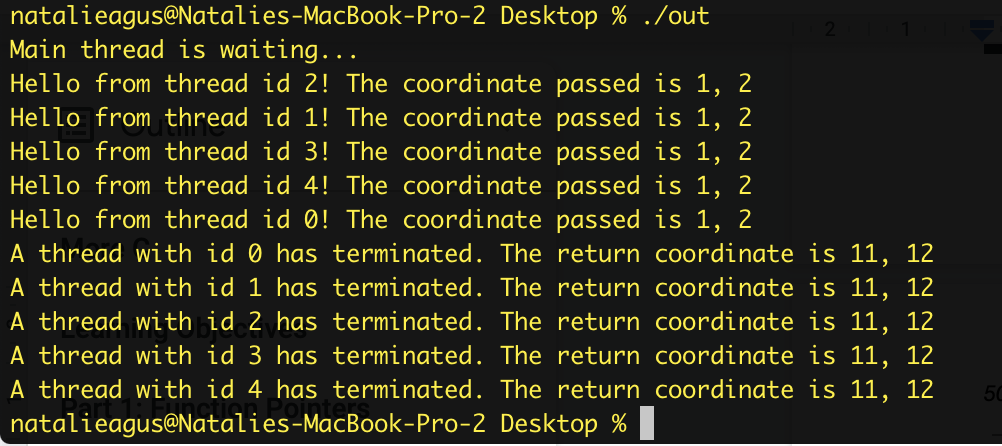
**Therefore you are absolutely prone to dangerous race conditions when creating threads and allowing them to modify the same global variables at once**. You will learn more about concepts of thread synchronization in class.

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

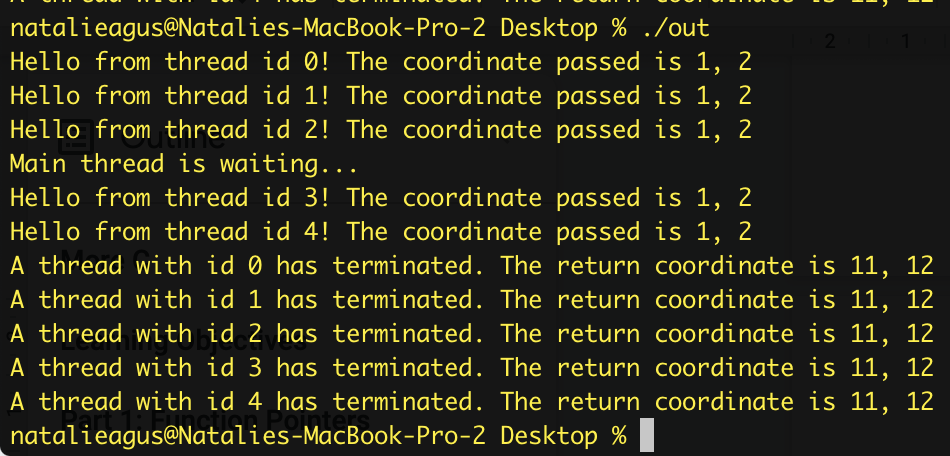
| pthread\_t tid[5];   Vec2D point[5];  for (int i = 0; i < 5; i++)  {  point[i].x = 1;  point[i].y = 2;  point[i].id = i;  int thread\_error\_check = pthread\_create(&tid[i], NULL, functionForThread, &point[i]);  //check error |
| --- |

| 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:



Running it again may result in different print order:



### 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 2: 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:

| 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);  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:

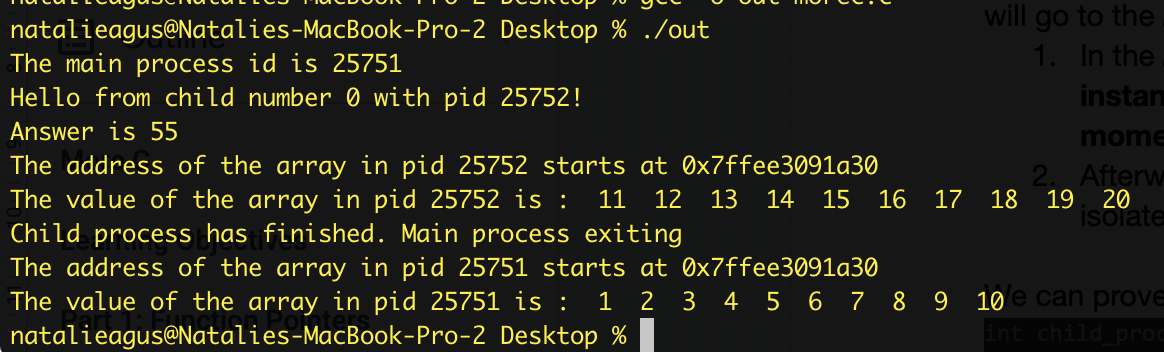
1. In the *if* clause that’s executed by the child process, the value array is **already instantiated** since the parent and the child **share the same state of code the moment fork() is called.**
2. Afterwards, the child and the parent process are totally separated processes, isolated from each other, in different **address spaces,** 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 the 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:

| 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:



Things to notice:

* The parent and child have different process id. You can print the current process’ id using getpid() system call
* The **virtual address** of arrays 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 the array is increased by 10 since that's 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 children, 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"); |
| --- |

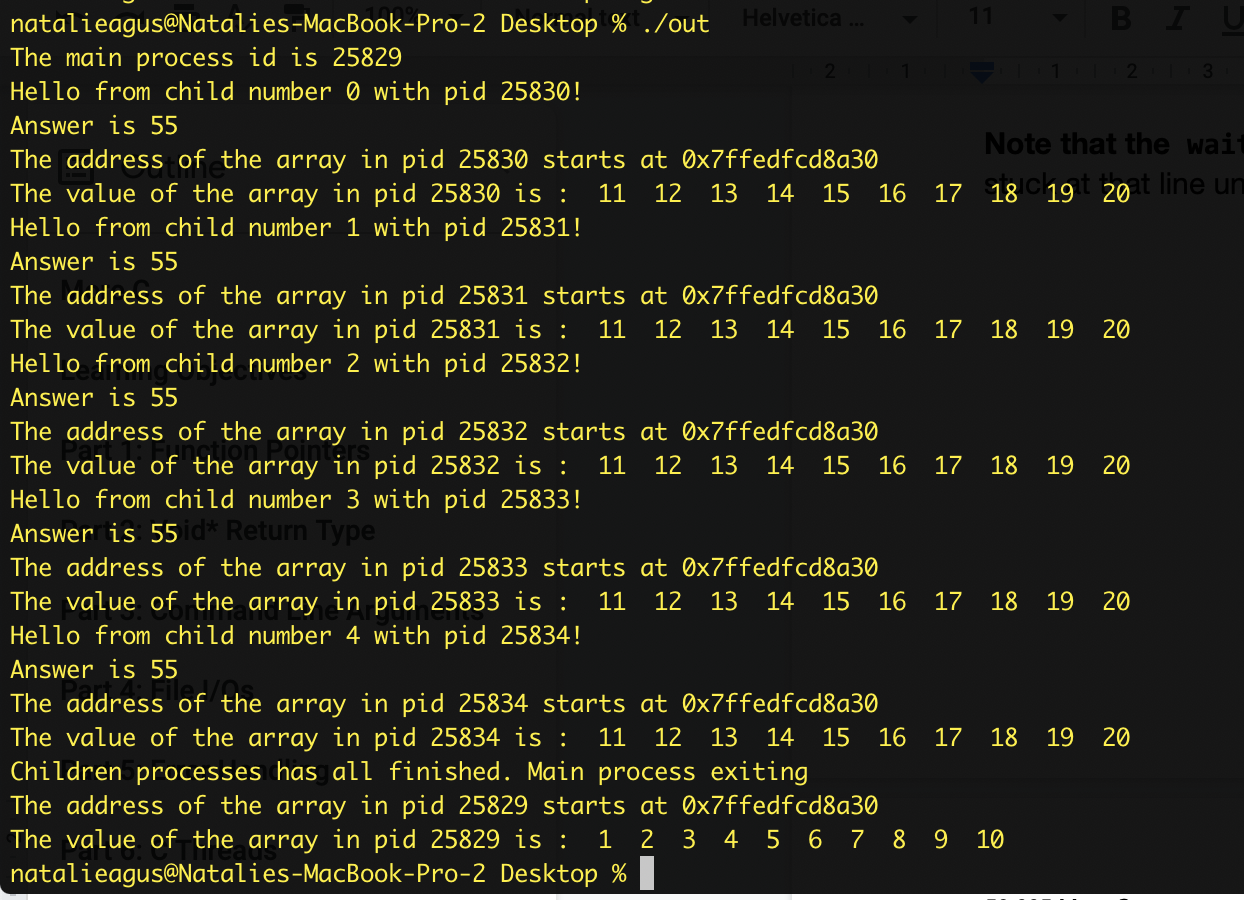
Here, wait(NULL) will always **return the pid of the terminated child,** unless there’s no more children process, then wait(NULL) will return -1. Hence we use a while-loop until it fails.

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

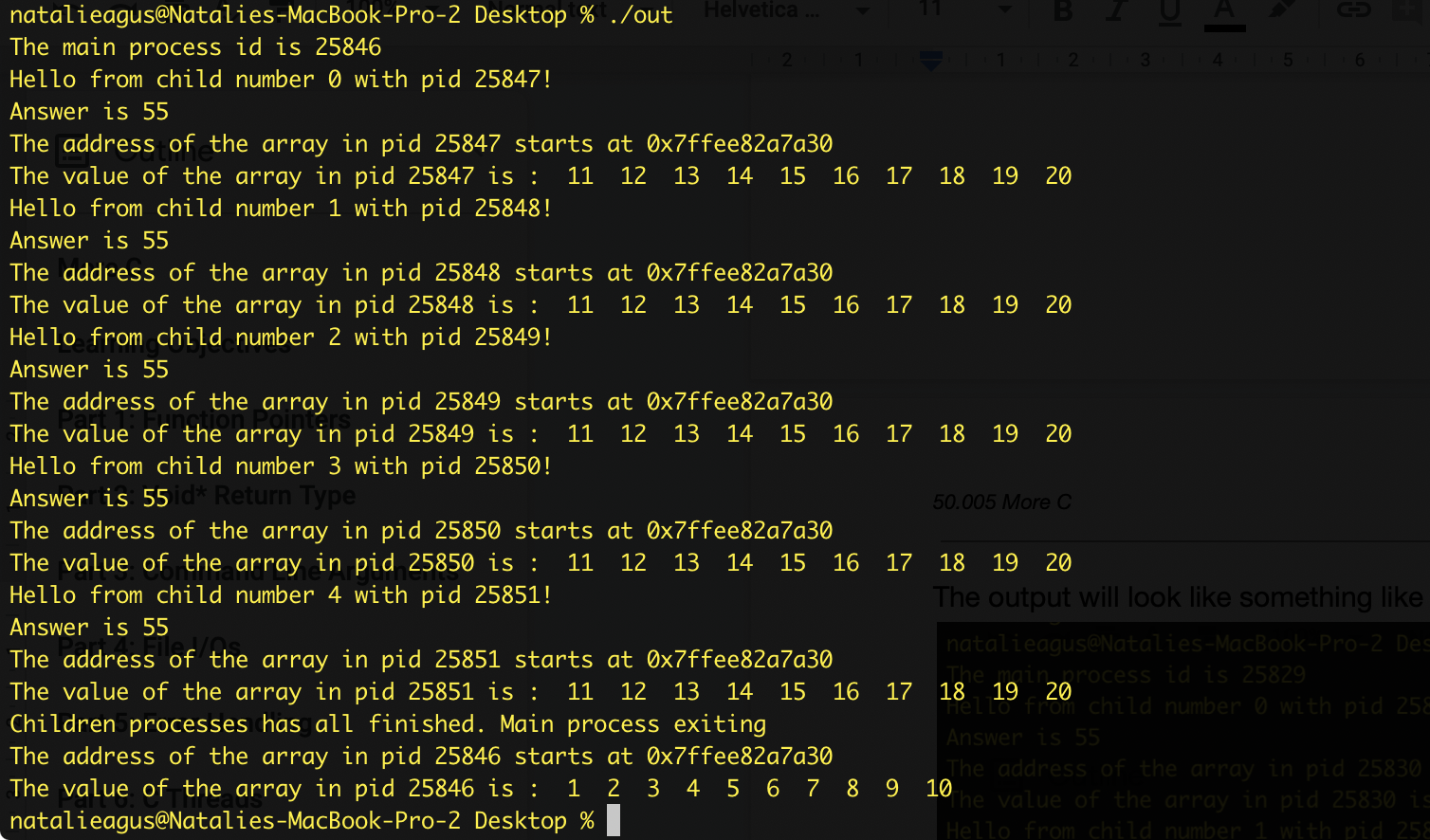
What if there’s no break indicated at the *if-clause* when forkReturnValue == 0? How many children (and grandchildren processes inclusive) are created?

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:



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



### 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 that run at the same time.

# Part 3: 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)

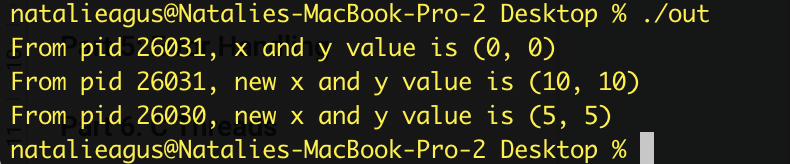
Lets digest this by example:

| //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 a memory **leak** until you restart your machine.

The output is as such:



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 that are 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 detached and removed 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 observe the pid of the process that created the shared memory using ipcs command, and then remove it using ipcrm -m pid.

# Part 4: Socket

Another alternative way for processes to communicate is through **sockets.** To illustrate how socket works, assume we have two processes, call it Server and Client who try to communicate with one another, e.g: pass string of message.

When a socket is created, the program has to specify the *address domain* and the *socket type*. **Two processes can communicate with each other only if their sockets are of the same type and in the same domain.**

There are two widely used address domains,

1. The *unix domain*, in which two processes which share a common file system communicate, and
2. The *Internet domain*, in which two processes running on any two hosts on the Internet communicate.

Each of these has its own address format:

1. The address of a socket in the Unix domain is a character string which is basically an entry in the file system.
2. The address of a socket in the Internet domain consists of the Internet address of the host machine (every computer on the Internet has a unique 32 bit address, often referred to as its IP address) + 16 bits port number

The following example illustrates how to create a socket using the internet address domain.

Create a .c program and name it server.c that contains the following code:

| #include <unistd.h> #include <stdio.h> #include <sys/socket.h> #include <stdlib.h> #include <netinet/in.h> //contains constants and structures needed for internet domain addresses. #include <string.h> #define PORT 8080 |
| --- |

First, import the necessary libraries.

In the main function, prepare necessary variables:

| int main(int argc, char const \*argv[]) {  /\*\*  \* server\_fd and newsockfd are file descriptors, i.e. array subscripts into the file descriptor table . These two variables store the values returned by the socket system call and the accept system call.portno stores the port number on which the server accepts connections.  \* valread is the return value for the read() and write() calls; i.e. it contains the number of characters read or written.  \*\*/  int server\_fd, new\_socket, valread;  /\*\*  \* A sockaddr\_in is a structure containing an internet address. This structure is defined in <netinet/in.h>. Here is the definition:  struct sockaddr\_in {  short sin\_family;  u\_short sin\_port;  struct in\_addr sin\_addr;  char sin\_zero[8];  };  \*/  struct sockaddr\_in address;  int addrlen = sizeof(address);   char buffer[1024] = {0}; //buffer used to read data from socket  char \*hello = "Hello from server"; |
| --- |

Then, you can **create** a new socket using the socket system call:

| // Creating socket file descriptor, using socket system call  if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0)  {  perror("socket failed");  exit(EXIT\_FAILURE);  } |
| --- |

It takes 3 arguments:

1. **First argument:** address domain of the socket. AF\_INET stands for internet domain address. Similarly, if you want to create a Unix domain socket then you can use AF\_UNIX stands for Unix domain address.
2. **Second argument:** the type of socket. You can have either SOCK\_STREAM or SOCK\_DGRAM. You can google about each type, but stream is basically used to send *streams of characters*
3. **Third argument:** protocol, leave it as 0 to let the OS decide the best protocol to do this.

Now, you need to **bind** the socket to the port, by setting up the address struct, and pass it to the bind system call:

| address.sin\_family = AF\_INET;  address.sin\_addr.s\_addr = INADDR\_ANY;  address.sin\_port = htons( PORT );    // Forcefully attaching socket to the port 8080  if (bind(server\_fd, (struct sockaddr \*)&address,   sizeof(address))<0)  {  perror("bind failed");  exit(EXIT\_FAILURE);  } |
| --- |

The bind() system call binds a socket to an address, in this case the address of the **current host** and **port number** on which the server will run.

It takes three arguments:

1. First argument: the socket file descriptor,
2. Second argument: the address to which is bound, and
3. Third argument: the size of the address to which it is bound.

Why this may fail: if socket is already in use on this machine.

After binding, the server listens to the socket:

| if (listen(server\_fd, 3) < 0)  {  perror("listen");  exit(EXIT\_FAILURE);  } |
| --- |

The listen system call allows the process to listen on the socket for connections. There’s two arguments that should be supplied, the first argument being the socket’s file descriptor. The second is the size of the backlog queue. This system call will not fail unless the socket descriptor is invalid.

Afterwards, the server tries to accept for incoming clients. The accept() system call causes the process to block until a client connects to the server:

| if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address,   (socklen\_t\*)&addrlen))<0)  {  perror("accept");  exit(EXIT\_FAILURE);  } |
| --- |

The accept system call wakes up the process when a connection from a client has been successfully established. It returns **a new file descriptor,** and all communication on this connection **should be done using the new file descriptor**.

Meaning of the other two arguments:

1. Second argument: a reference pointer to the address of the client on the other end of the connection,
2. Third argument: the size of this structure given in the second argument

When accept() returns, it means that there’s client who connects to it, and you can read from the socket using read system call, and also send message to the socket using the send system call:

| valread = read( new\_socket , buffer, 1024); //SYSTEM CALL  printf("%s\n",buffer );  send(new\_socket , hello , strlen(hello) , 0 );  printf("Hello message sent\n");  return 0; } |
| --- |

Note also that the read() is also a blocking system call, i.e: will block until there is something for it to read in the socket, i.e. after the client has executed a write().

Now time for the client code. Create a script called client.c containing the following:

| #include <stdio.h> #include <sys/socket.h> #include <arpa/inet.h> #include <unistd.h> #include <string.h> #define PORT 8080   int main(int argc, char const \*argv[]) {  int sock = 0, valread;  struct sockaddr\_in serv\_addr;  char \*hello = "Hello from client";  char buffer[1024] = {0};  if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)  {  printf("\n Socket creation error \n");  return -1;  } |
| --- |

Similarly it creates a socket of the same type. Prepare the address of the server:

| serv\_addr.sin\_family = AF\_INET;  serv\_addr.sin\_port = htons(PORT);    // Convert IPv4 and IPv6 addresses from text to binary form  if(inet\_pton(AF\_INET, "127.0.0.1", &serv\_addr.sin\_addr)<=0)   {  printf("\nInvalid address/ Address not supported \n");  return -1;  } |
| --- |

We are going to connect the server and client run in the same computer, hence we use the address of the *localhost: 127.0.0.1*. In computer networking, localhost is a hostname that means this computer.

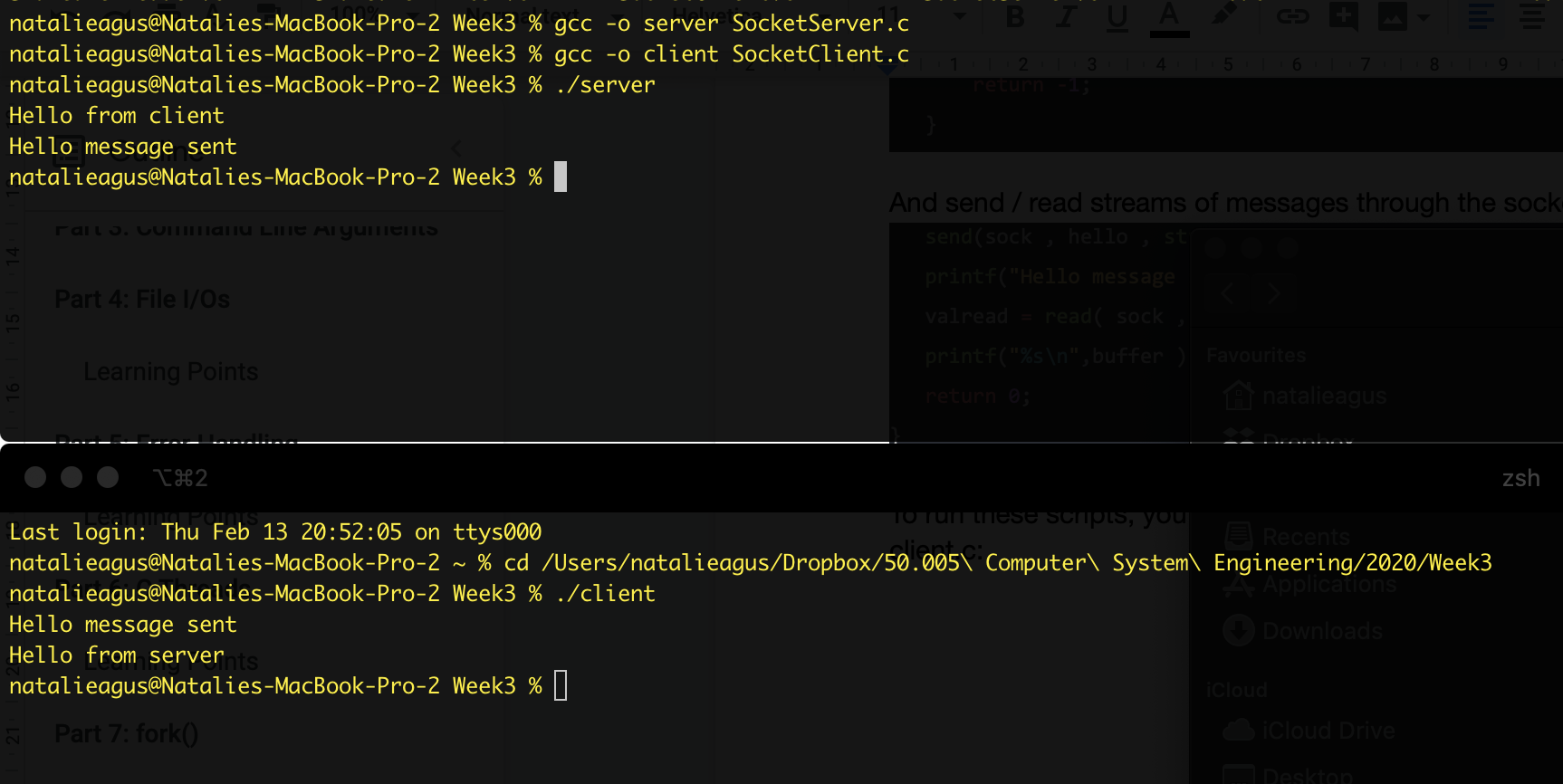
The client can then attempt to connect to the socket:

| if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0)  {  printf("\nConnection Failed \n");  return -1;  } |
| --- |

And send / read streams of messages through the socket:

| send(sock , hello , strlen(hello) , 0 );//SYSTEM CALL  printf("Hello message sent\n");  valread = read( sock , buffer, 1024);  printf("%s\n",buffer );  return 0; } |
| --- |

To run these scripts, you should compile and run server.c first, and then compile and run client.c in separate terminal windows:



### Learning Points

1. Create socket, bind, and connect to it
2. Read and write from socket
3. Know the difference between shared memory and socket as means of IPC

# 

# Summary

Congratulations. You have completed all the training for C in this course. If you’d like to test your knowledge up until now, head to e-dimension and do the quiz (Part 3). The grade is not going to be computed for your overall grade.