

# 北京理工大学

## 操作系统课程设计

实验二、进程控制	Experiment 2, Process Control
----------	-------------------------------

学院：计算机学院  
专业：计算机科学与技术  
学生姓名：夏奇拉  
学号：1820171025  
班级：07111705

# Table of Contents

- Purpose.....3
- Problem Discussion.....3
- Execution [Windows].....5
  - Results and Analysis [Windows].....9
- Execution [Linux].....10
  - Results and Analysis [Linux].....15
- Reference:.....16

# Purpose

Design and implement the "ParentProcess" command for Windows and Linux.

The "ParentProcess" command

- accepts a program to run via command-line arguments,
- creates a separate process to run the program, and
- records the time the program runs.

Implemented under Windows:

- Use `CreateProcess()` to create a process
- Use `WaitForSingleObject()` to synchronize between the "ParentProcess" command and the newly created process
- Call `GetSystemTime()` to get the time

Implemented under Linux:

- Use `fork()/execv()` to create a process runner
- Use `wait()` to wait for the newly created process to end
- Call `gettimeofday()` to get the time

Requirements:

- parent process:
  - Name: `ParentProcess(.exe)`
  - Function: Start `ChildProcess(.exe)`
  - Print the child process start time, end time, time-consuming, accurate to ms
- subprocess:
  - Name: `ChildProcess(.exe)`
  - Function: print: Hi, my name is + your name
  - Delay 3s

Usage of ParentProcess:

```
$ ParentProcess.exe ChildProcess
```

## Problem Discussion

We need to design and implement the "ParentProcess" command for Windows and Linux.

### What is a "parentprocess" command?

The sequence of instructions and data that can be executed a single time, multiple times or concurrently are called programs. A process is the execution of such programs. A parent process is a process that has created one or more child processes. A child process is a process created by another process (the parent process).

For this experiment the parent process would be `parentprocess.exe` and the child process would be `childprocess.exe`. These executable files can be created by compiling a `parentprocess.c` and `childprocess.c` source file respectively.

According to the experiment requirements, the parent process command must accept a program to run via command line arguments. This can be done by adding commands after the name of the program (the executable .exe file) in the command-line shell of the operating system. For example:

IN LINUX	
\$ nano parantprocess.c	//creates the source file
\$ gcc parentprocess.c -o parentprocess	//compiles the source and creates an executable file called parentprocess.exe
\$ ./parentprocess.exe childprocess	//runs the parentprocess program and passes the childprocess command as an argument to it

### Accepting Command Line Arguments in C Programming

In C programming, it is possible to accept command line arguments through the declaration of main().

Main() can accept two arguments.

1. the number of command line arguments.
2. a full list of all the command line arguments.

The full declaration of main looks like this:

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

The Windows specific version looks like this: (The \_T (or \_t) convention is one Microsoft uses to allow building of Unicode and building non-Unicode (usually multi-byte character set) without needing to change source code.)

```
int _tmain (int argc, _TCHAR *argv[])
```

The integer argc is the **argument count**, the number of arguments passed into the program from the command line, including the name of the program.

The array of character pointers is the list of all the arguments. Argv[0] is the name of the program, or an empty string if the name is not available. After that, every element number less than argc is a command line argument.

In the example given above:

```
$ ./parentprocess.exe childprocess //runs the parentprocess program and passes the childprocess command as an argument to it
```

argc would be equal to 2, argv[0] would be ./parentprocess, and argv[1] would be childprocess

## Creating A Separate Process To Run the Program

Creating a child process is different in Windows and Linux. Let's examine how to execute the requirements of this experiment in both Windows and Linux.

## Execution [Windows]

Set up Windows environment for compiling C program by following the steps listed here: [Walkthrough: Compile a C program on the command line | Microsoft Learn](#).

### Step 1: Create the parentprocess.c program

In the Developer Command Prompt for Visual Studio 2022 create a C file using Notepad with the name parentprocess.c by running the following command:

```
> notepad parentprocess.c
```

In the file, write the following code:

```
// parentprocess.c

#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _tmain(int argc, TCHAR *argv[])
{
    SYSTEMTIME st, et; // pointer to a SYSTEMTIME structure to
    receive the current system date and time.

    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    if (argc != 2) /* argc should be 2 for correct execution */
    {
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }

    // Start the child process.
    if (!CreateProcess(NULL, // No module name (use command
line)
                        argv[1], // Command line
                        NULL, // Process handle not inheritable
```

```

        NULL,        // Thread handle not inheritable
        FALSE,       // Set handle inheritance to FALSE
        0,           // No creation flags
        NULL,        // Use parent's environment block
        NULL,        // Use parent's starting directory
        &si,         // Pointer to STARTUPINFO
    structure
        &pi)        // Pointer to PROCESS_INFORMATION
    structure
    )
    {
        printf("CreateProcess failed (%d).\n", GetLastError());
        return;
    }

    GetSystemTime(&st);
    printf("The child process start time is: %02dh:%02dm:%02ds.
%02dms\n", st.wHour, st.wMinute, st.wSecond, st.wMilliseconds);

    // Wait until child process exits.
    WaitForSingleObject(pi.hProcess, INFINITE);

    GetSystemTime(&et);
    printf("The child process end time is: %02dh:%02dm:%02ds.
%02dms\n", et.wHour, et.wMinute, et.wSecond, et.wMilliseconds);

    printf("The child process elapsed time is: %02ds.%02dms\n",
et.wSecond-st.wSecond, et.wMilliseconds-st.wMilliseconds);

    // Close process and thread handles.
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}

```

The header file `windows.h` is the base header file for Win32 programming. It contains declaration of all the functions in the Windows API, all common macros used by Windows programmers for example `HANDLE`, `CreateProcess` and more. The Win32 API can be added to a C programming project by including the `<windows.h>` header file.

Read more about `tchar.h` here: [Generic-Text Mappings in tchar.h | Microsoft Learn](#)

The `CreateProcess()` function creates a new process, which runs independently of the creating process. However, for simplicity, the relationship is referred to as a parent-child relationship.

If `CreateProcess` succeeds, it returns a `PROCESS_INFORMATION` structure containing handles and identifiers for the new process and its primary thread.

The `CreateProcess` function takes a pointer to a `STARTUPINFO` structure as one of its parameters. `STARTUPINFO` structure specifies the window station, desktop, standard handles, and appearance of the main window for a process at creation time.

The members of this structure are used to specify characteristics of the child process's main window. For console processes, the STARTUPINFO structure is used to specify window properties only when creating a new console (either using CreateProcess with CREATE\_NEW\_CONSOLE or with the AllocConsole function). The STARTUPINFO structure can be used to specify the following console window properties:

- The size of the new console window, in character cells.
- The location of the new console window, in screen coordinates.
- The size, in character cells, of the new console's screen buffer.
- The text and background color attributes of the new console's screen buffer.
- The title of the new console's window.

Members of STARTUPINFO: [STARTUPINFOA \(processthreadsapi.h\) - Win32 apps | Microsoft Learn](#)

PROCESS\_INFORMATION structure contains information about a newly created process and its primary thread. Its members include hProcess, hThread, dwProcessId, dwThreadId

Note: A handle is, in essence, an abstract pointer, in the sense that a pointer is usually defined as 'a special variable which points to the memory address of a resource

ZeroMemory() is a macro that fills a block of memory with zeros. It's parameters include *destination[in]*, a pointer to the starting address of the block of memory to fill with zeros and *length[in]* the size of the block of memory to fill with zeros in bytes. This macro has no return value.

The sizeof operator returns the number of bytes occupied by a variable of a given type.

cb is a member of STARTUPINFO and it is the size of the structure in bytes.

### **Use WaitForSingleObject() To Synchronize Between the "Parent-process" Command and the Newly Created Process**

WaitForSingleObject function waits until the specified object is in the signalled state or the time-out interval elapses. It accepts a handle to the object and the time-out interval in milliseconds. If a non-zero value is specified, the function waits until the object is signalled or the interval elapses. If dwMilliseconds is zero, the function does not enter a wait state if the object is not signalled; it always returns immediately. If dwMilliseconds is INFINITE, the function will return only when the object is signalled.

### **Call GetSystemTime() to get the time**

GetSystemTime function retrieves the current system date and time in UTC format. The function accepts a pointer to a SYSTEMTIME structure to receive the current system date and time.

### **Step 2: Create the childprocess.c program**

In the Developer Command Prompt for Visual Studio 2022 create a C file using Notepad with the name childprocess.c by running the following command:

```
> notepad childprocess.c
```

In the file, write the following code:

```
// childprocess.c

#include <stdio.h>
#include <windows.h>

int main()
{
    printf("Hello my name is Xiaqila\n");
    printf("Delay 3s\n");
    Sleep(3000);
    return 0;
}
```

Sleep() function suspends the execution of the current thread until the time-out interval elapses. It accepts the time interval for which execution is to be suspended in milliseconds.

### Step 3: Compile the C program

Use the following command to make executable version of the two programs.

```
> cl parentprocess.c
> cl childprocess.c
```

This will output parentprocess.exe and childprocess.exe

### Step 4: Run the program

Use the following command to run the newly created executable .exe programs.

```
> parentprocess childprocess
```

These parameters are passed to `void _tmain(int argc, TCHAR *argv[])`

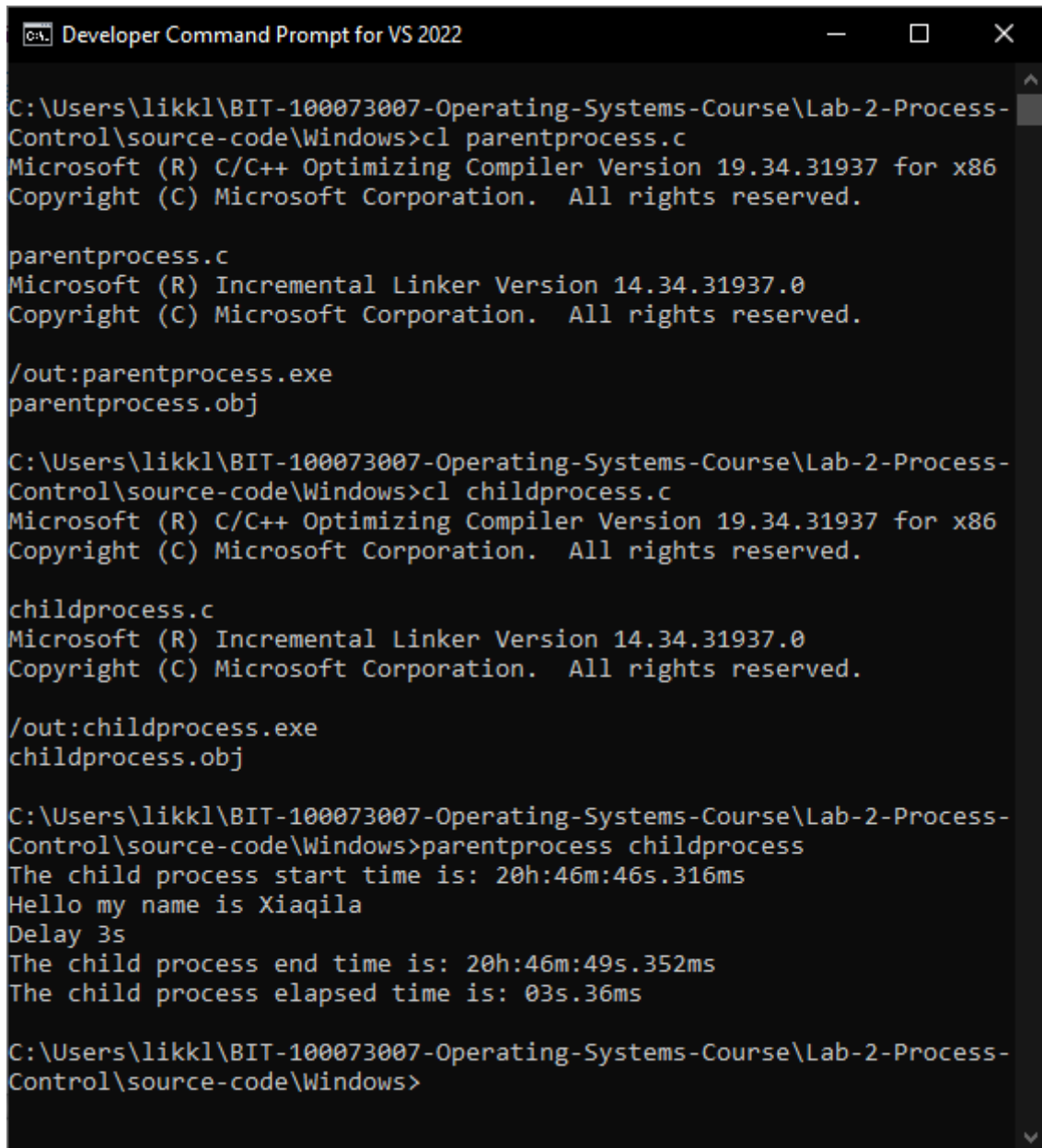
The following output is produced:

```
The child process start time is: 20h:46m:46s.316ms
Hello my name is Xiaqila
Delay 3s
The child process end time is: 20h:46m:49s.352ms
The child process elapsed time is: 03s.36ms
```



## Results and Analysis [Windows]

Running parentprocess childprocess on the command line will start the parent process program. This program uses CreateProcess() to call the child process program which prints "Hello my name is Xiaqila" and sleeps for 3 seconds before returning. The parent process uses WaitForSingleObject() to wait for the child process to end. Once the child process is completed the parent process prints the child process end time and the child process elapsed time.



```
Developer Command Prompt for VS 2022

C:\Users\likkl\BIT-100073007-Operating-Systems-Course\Lab-2-Process-
Control\source-code\Windows>cl parentprocess.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.34.31937 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

parentprocess.c
Microsoft (R) Incremental Linker Version 14.34.31937.0
Copyright (C) Microsoft Corporation. All rights reserved.

/out:parentprocess.exe
parentprocess.obj

C:\Users\likkl\BIT-100073007-Operating-Systems-Course\Lab-2-Process-
Control\source-code\Windows>cl childprocess.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.34.31937 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

childprocess.c
Microsoft (R) Incremental Linker Version 14.34.31937.0
Copyright (C) Microsoft Corporation. All rights reserved.

/out:childprocess.exe
childprocess.obj

C:\Users\likkl\BIT-100073007-Operating-Systems-Course\Lab-2-Process-
Control\source-code\Windows>parentprocess childprocess
The child process start time is: 20h:46m:46s.316ms
Hello my name is Xiaqila
Delay 3s
The child process end time is: 20h:46m:49s.352ms
The child process elapsed time is: 03s.36ms

C:\Users\likkl\BIT-100073007-Operating-Systems-Course\Lab-2-Process-
Control\source-code\Windows>
```

# Execution [Linux]

## fork()

System call `fork()` is used to create processes. It takes no arguments and returns a process ID. The purpose of `fork()` is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the `fork()` system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of `fork()`:

- If `fork()` returns a negative value, the creation of a child process was unsuccessful.
- `fork()` returns a zero to the newly created child process.
- `fork()` returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type `pid_t` defined in `sys/types.h`. Normally, the process ID is an integer. Moreover, a process can use function `getpid()` to retrieve the process ID assigned to this process.

Therefore, after the system call to `fork()`, a simple test can tell which process is the child. Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

## wait()

The system call `wait()` is easy. This function blocks the calling process until one of its child processes exits or a signal is received. For our purpose, we shall ignore signals. `wait()` takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer. One of the main purposes of `wait()` is to wait for completion of child processes.

The execution of `wait()` could have two possible situations.

If there are at least one child processes running when the call to `wait()` is made, the caller will be blocked until one of its child processes exits. At that moment, the caller resumes its execution.

If there is no child process running when the call to `wait()` is made, then this `wait()` has no effect at all. That is, it is as if no `wait()` is there.

## exec()

The created child process does not have to run the same program as the parent process does. The `exec` type system calls allow a process to run any program files, which include a binary executable or a shell script. For this experiment, `execv()` is used. The `execv()` system call requires two arguments:

- The first argument, by convention, should point to the filename associated with the file being executed.
- The `char *const argv[]` argument is an array of pointers to null-terminated strings that represent the argument list available to the new program, which is exactly

identical to the `argv` array used in the main program:

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

The first parameter, `argc` (argument count) is an integer that indicates how many arguments were entered on the command line when the program was started. The second parameter, `argv` (argument vector), is an array of pointers to arrays of character objects. The array objects are null-terminated strings, representing the arguments that were entered on the command line when the program was started.

The first element of the array, `argv[0]`, is a pointer to the character array that contains the program name or invocation name of the program that is being run from the command line. `argv[1]` indicates the first argument passed to the program, `argv[2]` the second argument, and so on.

Note that this argument must be terminated by a zero.

`execv()` returns a negative value if the execution fails (e.g., the request file does not exist).

### **gettimeofday()**

The `gettimeofday()` system function is defined in the `sys/time.h` header. It fills two structures with details about the current time of day:

```
int gettimeofday( struct timeval *, struct tzp *);
```

the *timeval* structure contains two members,

- `tv_sec`: a *time\_t* value, the number of seconds elapsed since January 1, 1970.
- `tv_usec`: a microsecond value, which the computer knows but isn't included with the *time\_t* value

the *tzp* structure contains time zone information.

This will involve writing a C program and building an executable file to run in the terminal as the parent process

#### **Step 1: Create C program**

In the terminal, create a file using the nano editor with the name `parenprocess.c` by running the following command:

```
$ nano parenprocess.c
```

In the file we write the following code:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/time.h>

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

```

    struct timeval start, end, start2;

    pid_t pid;
    pid = fork();

    if(pid==0) {
        int sts_cd = execvp("/home/shak/Documents/BIT-100073007-
Operating-Systems-Course/Lab-2-Process-Control/Source-Code/Linux/
childprocess", argv);
        printf("There is an issues with the running command\n");
        if (sts_cd == -1) {
            printf("execvp error! \n");
            return 1;
        }
    }
    else if ( pid > 0 ) {
        gettimeofday(&start, NULL);
        printf("The child process start time is: %ld seconds %ld
micro seconds\n", start.tv_sec, start.tv_usec);
        int wc = wait(NULL);
        gettimeofday(&end, NULL);
        printf("The child process end time is: %ld seconds %ld
micro seconds\n", end.tv_sec, end.tv_usec);
        printf("The child process elapsed time is: %lds.%ldms\
n", (end.tv_sec - start.tv_sec), (end.tv_usec - start.tv_usec) /
1000);
    }
    else {
        printf("Error while forking \n");
        exit(EXIT_FAILURE);
    }

    return 0;
}

```

### About the code:

In this program, we

- declare structs timeval start end to capture the value returned from gettimeofday().
- declare character cmd as "ls" as the file to be executed when int execvp(const char \*file, char \*const) is run.
- Declare process ID of type pid\_t defined in sys/types.h to capture the process id returned when we run fork().
- Running fork() creates a child process which is the exact copy of the parent process. We conduct a simple test to tell which process@id:ms-vscode.cpptools-extension-pack is the child.
- If fork() returns 0, the child process is executed. It includes the execvp() function which is used to run the childprocess.exe program in the source directory. The childprocess.exe program prints "Hello my name is Xiaqila" and sleeps for 3 seconds before returning.
- If fork() returns a value greater than 0, the parent process continues. We capture

the start time of the process by calling `gettimeofday()`. then the parent process waits for the child process to finish when `wait(NULL)` is called.

- `Wc` captures the process ID of the child process the parent process is waiting for.
- When the child process is complete we use `gettimeofday()` to capture the end time of the child process. The parent process then prints the start time, end time and elapsed time of the child process.

Exit the nano editor by hitting  
CTRL + X  
Y (YES)  
Enter  
CTRL + X

This returns us to the terminal

### Step 3: Create the `childprocess.c` program

In the terminal, create a file using the nano editor with the name `childprocess.c` by running the following command:

```
$ nano childprocess.c
```

In the file we write the following code:

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

int main()
{
    printf("Hello my name is Xiaqila\n");
    printf("Delay 3s\n");
    sleep(3);
    return 0;
}
```

`Sleep()` function suspends the execution of the current thread until the time-out interval elapses. It accepts the time interval for which execution is to be suspended in seconds.

### Step 4: Compile the C program

Use the following command to make executable version of the two programs.

```
$ gcc parentprocess.c -o parentprocess
$ gcc childprocess.c -o childprocess
```

This will output `parentprocess.exe` and `childprocess.exe`

### Step 5: Run the program

Use the following command to run the newly created executable `.exe` programs.

```
$ ./parentprocess childprocess
```

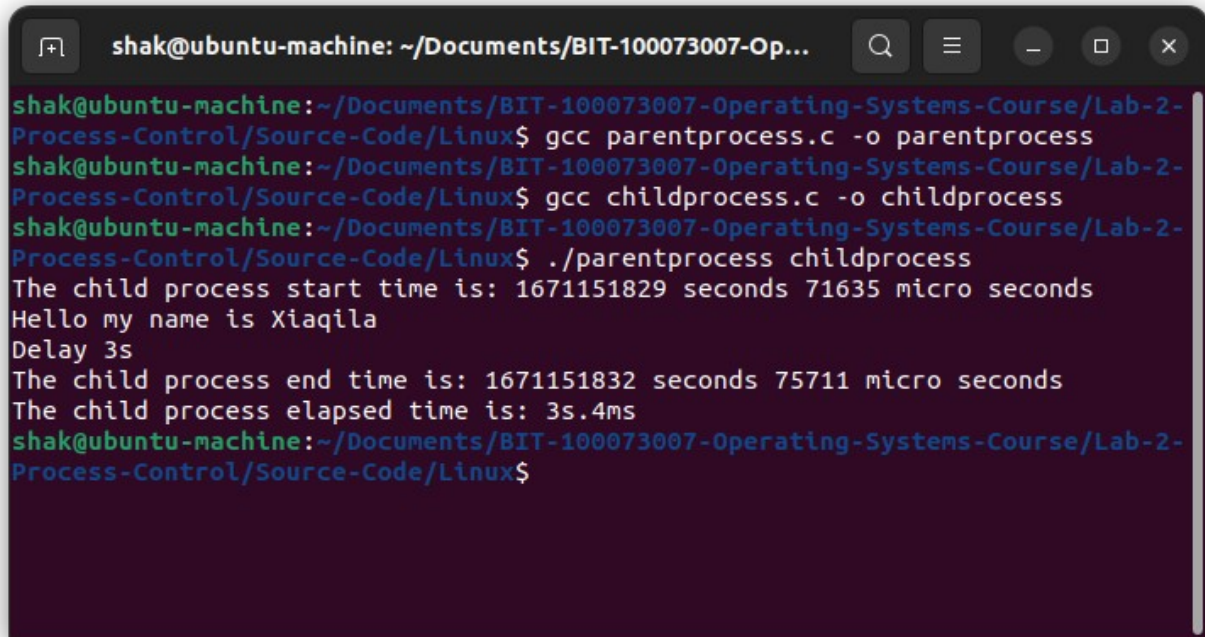
These parameters are passed to `int main (int argc, char *argv[])`

The following output is produced:

```
The child process start time is: 1671151558 seconds 641829 micro  
seconds  
Hello my name is Xiaqila  
Delay 3s  
The child process end time is: 1671151561 seconds 644295 micro  
seconds  
The child process elapsed time is: 3s.2ms
```

## Results and Analysis [Linux]

Running `./parentprocess childprocess` on the command line will start the parent process program. This program uses `fork()` and `execv()` to call the child process program which prints "Hello my name is Xiaqila" and sleeps for 3 seconds before returning. The parent process uses `wait()` to wait for the child process to end. Once the child process is completed the parent process prints the child process end time and the child process elapsed time.



```
shak@ubuntu-machine: ~/Documents/BIT-100073007-Op...
shak@ubuntu-machine:~/Documents/BIT-100073007-Operating-Systems-Course/Lab-2-Process-Control/Source-Code/Linux$ gcc parentprocess.c -o parentprocess
shak@ubuntu-machine:~/Documents/BIT-100073007-Operating-Systems-Course/Lab-2-Process-Control/Source-Code/Linux$ gcc childprocess.c -o childprocess
shak@ubuntu-machine:~/Documents/BIT-100073007-Operating-Systems-Course/Lab-2-Process-Control/Source-Code/Linux$ ./parentprocess childprocess
The child process start time is: 1671151829 seconds 71635 micro seconds
Hello my name is Xiaqila
Delay 3s
The child process end time is: 1671151832 seconds 75711 micro seconds
The child process elapsed time is: 3s.4ms
shak@ubuntu-machine:~/Documents/BIT-100073007-Operating-Systems-Course/Lab-2-Process-Control/Source-Code/Linux$
```

## Reference:

- <https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html>
- <https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/wait.html>
- <http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/exec.html>
- <https://vitux.com/fork-exec-wait-and-exit-system-call-explained-in-linux/>
- <https://nipunbatra.github.io/os2020/labs/>
- <https://www.ibm.com/docs/en/i/7.1?topic=functions-main-function>
- [https://linuxhint.com/gettimeofday\\_c\\_language/](https://linuxhint.com/gettimeofday_c_language/)
- <https://c-for-dummies.com/blog/?p=4236>
- <https://linuxhint.com/execvp-function-c/>
- <https://www.cyberciti.biz/faq/compiling-c-program-and-creating-executable-file/>
- [Command Line Arguments in C - Cprogramming.com](#)
- [Creating Processes - Win32 apps | Microsoft Learn](#)
- [WaitForSingleObject function \(synchapi.h\) - Win32 apps | Microsoft Learn](#)
- [Sleep function \(synchapi.h\) - Win32 apps | Microsoft Learn](#)
- [GetSystemTime function \(sysinfoapi.h\) - Win32 apps | Microsoft Learn](#)
- [ZeroMemory macro \(Windows\) | Microsoft Learn](#)
- [STARTUPINFOA \(processthreadsapi.h\) - Win32 apps | Microsoft Learn](#)
- [PROCESS\\_INFORMATION \(processthreadsapi.h\) - Win32 apps | Microsoft Learn](#)
- <https://man7.org/linux/man-pages/man3/exec.3.html>