FIT2100 Laboratory #4
Process Scheduling and Threads,
Week 8 - Semeseter 2 2024

August 16, 2024

Revision Status:

`Updated by Dr.Charith Jayasekara, Aug 2024.`

Acknowledgements

2016-2018 versions written by Jojo Wong. Content presented in this Laboratory was adapted from the courseware of FIT3042 prepared by Robyn McNamara. Some parts of the content were adapted from the following texts:

- David A. Curry (1996). *UNIX Systems Programming for SVR4*, O'Reilly.
- W. Richard Stevens and Stephen A. Rago (2013). *Advanced Programming in the Unix Environment (Third Edition)*, Addison-Wesley.

Contents

1 Background

In this Laboratory, we will learn about how processes can be created and managed in Unix/Linux using system calls in C. We will also discuss how processes (or jobs) are managed within the command-line shell itself.

Before attending the Laboratory, you should:

- Complete your pre-class readings (Section 2)
- Attempt the Laboratory tasks in Section 3

2 Pre-Laboratory Reading

2.1 Working with Processes

A process can be regarded as an instance of a program that is currently loaded into the memory and being executed. It is possible for multiple instances of the same program to be running at the same time within different processes.

In the Unix/Linux environment, the execution state of each process, with a program running within it, and a lot of other state information are all managed in one of the OS data structures called the *process table*.

Each process has a unique process identifier (*process ID* or simply *PID*) which is a non-negative integer, usually ranging from 0 to about 32,000. The process ID is actually used as an *index* on the process table to obtain the context and other information associated with a particular process.

The process ID of a process can be obtained through a system call with the `getpid()` function defined in the `unistd.h` (Unix standard) library. The special return data type for this function (`pid_t`), is actually just an integer, defined to hold a suitable range of values for process IDs.

Function prototype for the `getpid` function:

```c
#include <unistd.h>
#include <sys/types.h>  /* pid_t is defined in sys/types.h */

pid_t getpid(void);
```

Process Table Some of the information that the OS stores about each process in the process table includes:

Variable	Description
PID	a unique process ID
PPID	the PID of the parent process
UID	the user ID of the owner of the process
TTY	the terminal ID associated with the process (if it is running in a console mode)
STAT	the status of the process

The Unix command `ps` (process status) presents the current state of the process table. To view a list containing all the state information about all the processes currently running on your computer system, use the Unix command `ps` with the options `-e` and `-f`:

```
$ ps -e -f
```

2.2 Creating Processes in C

The system call routine — `fork()` — provided in the `unistd.h` library can be used to create (or *spawn*) a new process from an existing process in Unix/Linux. The new process created as the result of the invocation of `fork()`, starts out as a copy of the parent process.

Function prototype for the `fork` system call:

```
#include <unistd.h>
#include <sys/types.h>

pid_t fork(void);    // Does not take any arguments
```

After executing a `fork()`, *how can we distinguish the parent process from the child process?*

Interestingly, `fork()` is invoked once but *returns twice* — once in the parent process, and once in the newly-created child process. In the parent process, `fork()` returns the process ID (`PID`) of the child process. In the child process, however, `fork()` returns 0. By checking the return value of `fork()`, we can then determine whether execution is continued in the parent process or the child process.

The reason for returning the child's process ID to the parent is that a parent process can indeed spawn many child processes, and there is no function available to retrieve the process IDs of its children.

The reason for returning 0 from `fork()` to the child is that each process can only have one parent, and the child process can always obtain its parent's process ID by invoking the `getppid()` function.

Function prototype for the `getppid` function call:

```
#include <unistd.h>
#include <sys/types.h>

pid_t getppid(void);
```

(**Note:** If `fork()` returns the value of `-1` to the parent process, this indicates that the child process could not be created.)

Parent and Child Processes

After a `fork()`, the parent and child processes have a nearly identical state:

- the child's environment variables are inherited copies of the parent's;
- the child's program code is exactly the same as the parent's;
- the child has its own copy of the same file descriptors (`stdin`, `stdio`, `stderr`) as the parent; however, sharing the same file offset for each descriptor.

There are some differences between the child process and its parent, where:

- the parent and child processes do not share the same memory space (they have their own copies of global variables);
- the child process is issued with its own process ID (`PID`);
- the child process's `PPID` should be the same as the parent's `PID`.

The parent process and the child process are managed separately as far as the OS kernel is concerned.

2.3 Loading and Executing New Programs

Having two copies of the same program in main memory might not be entirely useful.

Often times, we would want the newly created process (i.e. the child process) to load up and execute a different program. This can be accomplished by using one of the several functions from the `exec()` family.

Function prototypes for the exec family of functions:

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ..., const char *argn,
        /* (char*) NULL */);
int execv(const char *path, const char *argv[]);
int execle(const char *path, const char *arg0, ..., const char *argn,
        /* (char*) NULL */, const char *envp[]);
int execve(const char *path, char *const argv[], const char *envp[]);
int execlp(const char *file, const char *arg0, ..., const char *argn,
        /* (char*) NULL */);
int execvp(const char *file, const char *argv[]);
```

The main difference in all these different forms of the `exec()` function, is that the first four functions take a *pathname* argument, and the last two take a *filename* argument. If the *filename* argument specified with a *slash*, it is considered as a pathname for the executable file (or program). Otherwise, the executable file (or program) is searched for in the directories listed in the `PATH` environment variable; and the first such file encountered is then executed.

Note: *Environment variables* are not C variables. They are special string variables maintained by the OS itself. They normally exist before a new process starts, and are inherited from the parent process. If you are interested to see to the current value of `PATH` defined in your system, use the following Unix command.

```
  $ echo $PATH
```

Arguments: One other key difference among these functions is about the passing of the argument list. The functions — `execl`, `execle`, and `execlp` — require each of the command-line arguments to the new program to be passed on as separate arguments.

Note that the end of the argument list is indicated by a `NULL` pointer. If this `NULL` pointer is specified by a constant value of `0`, it should be cast into a `char` pointer (`char *`). For the other three `exec()` functions, they accept the command-line arguments as an array of `char` pointers (`char *argv[]`).

By convention, the argument list (`argv`) should have at least one element — it is essentially the name of the process as displayed by the ps command. The `argv[0]` is usually given as the pathname of the executable file (or program) itself, or the last component of the pathname.

Environment Strings: Another difference that should be noted is the functions `execle` and `execve` allow us to pass on an array of pointers to the environment strings as an argument, which are to be assigned to the environment variables for the new process. For other `exec()` functions that do not accept this additional argument, the new process will inherit the parent process's environment strings.

(**Note:** If your system supports functions such as `setenv` and `putenv`, the environment variables associated with a process can be changed but with some restriction.)

Here is an example of how the `exec()` family of functions can be used. Suppose that we want the new process to run the Unix command "`ls -alR`", this can be accomplished by writing the following C code:

```c
/* exec-ls-alr.c */

#include <unistd.h>
#include <sys/types.h>

int main() {
  pid_t pid;
  char *params[] = {"/bin/ls", "-alR", 0};

  if ((pid = fork()) < 0) {      /* create a new process */
    perror("fork error");
    exit(1);
  } else if (pid == 0) {         /*  in the child process */
    execv("/bin/ls", params);
  } else {                       /* in the parent process */
    sleep(10);
  }
  exit(0);
}
```

How, actually, does a command shell work? By now, you should be able to figure out how the command shells (such as `bash`) work. So long as the user does not exit or logout, the command shell does the following:

1. Display the prompt (i.e. "$")
2. Read the command line from the keyboard
3. Establish that the first "word" in the command line represents an executable program
4. If `fork()` returns 0, run "execv(program, arguments)"
5. Otherwise, wait for the child to finish

2.4 Waiting for Process Termination

The last step in the command shell algorithm is indeed to wait for the child process to terminate and to collect its termination status.

In the above example, we were just assuming that by putting the parent to sleep for 10 seconds, this would be sufficient for the child process to complete its execution. A more robust way to do this is to *block* the operation of the parent process until the child process terminates.

The `wait()` and `waitpid()` functions from the `<sys/wait.h>` library are two of the system calls in C that can be invoked to achieve this. With these functions, we can determine the PID of the child process that has terminated and its status information can be retrieved. Further, we are also able to display useful error messages if a child process did not terminate cleanly.

Function prototypes for the `wait` and `waitpid` functions:

```c
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Some differences between these two functions are that:

- `wait()` can block the calling process (the parent process) until a child process terminates; however `waitpid()` has an option that prevents it from being blocked.
- `waitpid()` does not wait for the child that terminates first; it has a number of options that control which child process it waits for.

Note that `waitpid()` provides greater control over waiting for processes, in which the `pid` argument is assigned with different control values:

Value	Meaning
`pid == -1`	Wait for any child process
`pid > 0`	Wait for the child whose process ID is equal to `pid`
`pid == 0`	Wait for any child process in the same process group
`pid < -1`	Wait for any child process whose process group ID is equal to the absolute value of `pid`

Termination Status The termination status of a child process is stored in the memory location pointed by the `status` argument. If the calling process (i.e the parent process) does not care about the child's termination status and is only interested in waiting for the child to terminate, the `status` argument can be given as a `NULL` pointer.

There is a set of macros (symbolic constants) defined in `sys/wait.h` that can be used to decode the various types of termination status returned by the `wait()` and `waitpid()` functions. Some of the macros are summarised below:

Macro	Description
`WIFEXITED`	Evaluate to `True` if `status` was returned for a child that terminated normally.
`WIFSIGNALED`	Evaluate to `True` if `status` was returned for a child that terminated abnormally due to an uncaught signal.
`WIFSTOPPED`	Evaluate to `True` if `status` was returned for a child that is currently stopped.
`WIFCONTINUED`	Evaluate to `True` if `status` was returned for a child that has been continued from a stopped state.

Zombie Processes After a process has terminated, the operating system keeps track of its status information (in the process table) in case the parent process issues the `wait()` call. If the parent process never does a `wait()`, the information of the child process is not cleared out. The child process is regarded as a "zombie" — i.e. hanging around and occupying an entry in the process table.

Remark: The use of `fork()` and the exec() functions under Unix/Linux is not the same as *multithreading*. To use threads, check out the `pthreads` library.

2.5 Job Control in Unix/Linux

Now, let's look at how processes (or jobs) are managed in the Unix's command shell itself.

By default, a Unix shell (e.g. the `bash` shell) will spawn a child process to run your command and will then wait for that child to terminate before returning to input (i.e. to read in the next command). You can in fact tell the shell not to wait for the child to terminate. To do this, simply putting an ampersand ("&") at the end of the command line (as shown below).

```
$ ps -ef &
```

(**Note:** By ending the command line with "&", you should expect to see the shell prompt is displayed before the ps output — `bash` does not wait for `ps` to complete its execution.)

Stopping Programs You can attempt to stop the running programs by doing the following:

- Press `Ctrl-Z` to make a process stop *temporarily*
- Press `Ctrl-C` to terminate a process *permanently*

Backgrounded and Foregrounded Processes If you stopped a program with `Ctrl-Z`, you can make it run in the background by using the `bg` command. You can then check this by running the `top` command, which shows the processes in the system that are currently taking up the most system resources.

(**Note:** You can also move the backgrounded processes back to the foreground with the `fg` command.)

How do we find out the processes running in the background? By running the `jobs` command, a list of all the backgrounded jobs that the current `bash` session is managing will be displayed. Note that `jobs` is not a program, but is built into `bash`.

Terminating Processes As we may have seen, processes can react to certain keystrokes (such as `Ctrl-C` or `Ctrl-Z`). However, sometimes we might want to stop or terminate a process without foregrounding it or pressing `Ctrl-C`. The solution is to use the `kill` command.

To terminate a process, we first have to find out the process ID (`PID`) or the job ID of the process you want to kill. This can be done by using the `ps` or `jobs` command. Then, use the `kill` command with either the process ID or the job ID.

```
$ kill -level PID
$ kill -level %jobID
```

The `level` option indicates the signal you want to send to the target process that you are attempting to kill. For example, this command kills the process with PID 42: `kill -9 42`.

Below is a summary of the standard Unix signals.

Signal	Name
1	`SIGHUP` (hang up)
2	`SIGINT` (interrupt, Ctrl-C)
3	`SIGQUIT` (quit)
4	`SIGILL` (illegal instruction)
5	`SIGTRAP` (trace trap, used in gdb)
8	`SIGFPE` (floating point exception, e.g. divide by zero)
9	`SIGKILL` (signal cannot be caught or ignored)
11	`SIGSEGV` (segmentation fault)
18	`SIGCONT` (continue if stopped, e.g. using `fg`)
20	`SIGSTP` (terminal stop, Ctrl-Z)

Note that the OS kernel sometimes sends signals to processes. This is indeed how various different kinds of program crashes are implemented — such as segmentation faults and divide-by-zero errors. In fact, pressing `Ctrl-C` sends an interrupt signal to the target process; while pressing `Ctrl-Z` sends a terminal stop signal.

Try looking up the `man` pages for the following Unix commands yourself:

```
ps
top
jobs
uptime
pgrep
pkill
```

Try identifying special processes yourself:

When a new program is executed, a new process is created with the next available process ID. However, there are a few special processes that always have the same process ID, which are usually given the ID value less than 5 — these are called *system processes*. Can you identify which of the two system processes have the process ID of 0 and 1 respectively?

2.6 Understanding Threads

Technically, a *thread* is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.

A thread is a semi-process that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them).

A *thread group* is a set of threads all executing inside the same process (see Figure 1). They all share the same memory, and thus can access the same global variables, the same heap memory, the same set of file descriptors, etc. All these threads execute in *parallel* (i.e. using time slices, or if the system has several processors, then really in parallel).

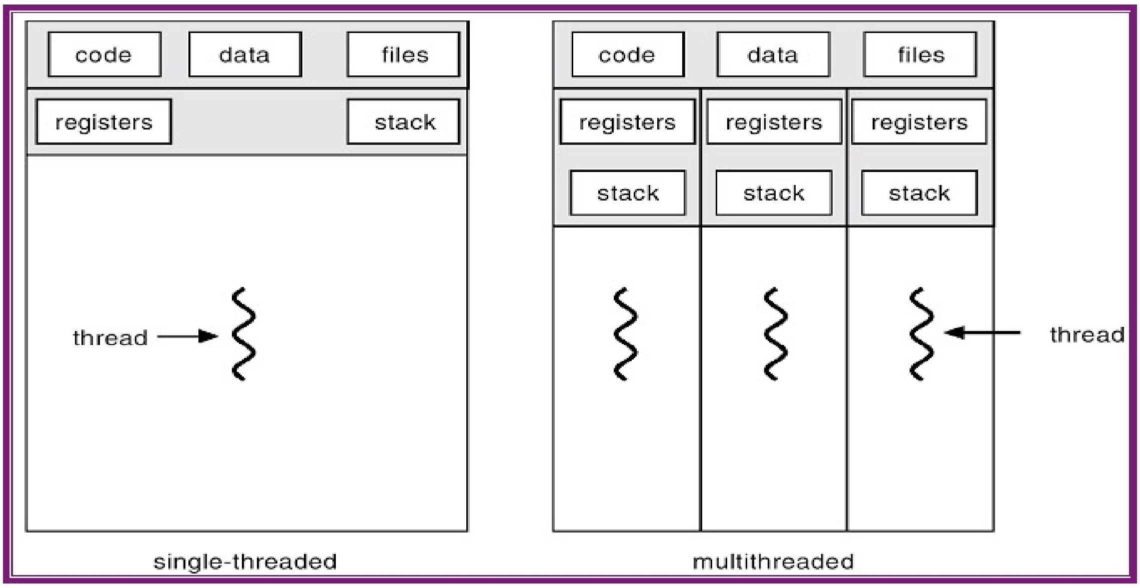

Figure 1: Single-Threaded and Multi-Threaded Programs (Processes)

2.6.1 Pthreads

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other, making it difficult for programmers to develop portable threaded applications.

In order to take full advantage of the capabilities provided by threads, a standardised programming interface was required. For Unix/Linux systems, this interface has been specified by the **IEEE POSIX 1003.1c** standard (1995). Implementations that adhere to this standard are referred to as POSIX threads, or *Pthreads*. Most hardware vendors now offer Pthreads in addition to their proprietary threads.

2.6.2 Are Threads Efficient?

If implemented correctly, threads have some advantages over processes. Compared to the standard `fork()`, threads carry a lot less overhead.

Remember that `fork()` produces a second copy of the calling process. The parent and the child are completely independent, each with its own address space, with its own copies of its variables, which are completely independent of the same variables in the other process.

Threads share a common address space, thereby avoiding a lot of the inefficiencies of multiple processes.

- The kernel does not need to make a new independent copy of the process memory space, file descriptors, etc. This saves a lot of CPU time, making thread creation ten to a hundred times faster than a new process creation. Because of this, you can use a whole bunch of threads and not worry about the CPU and memory overhead incurred. This means you can generally create threads whenever it makes sense in your program.
- Less time to terminate a thread than a process.
- Context switching between threads is much faster than context switching between processes.(Context switching means that the system switches from running one thread or process, to running another thread or process).
- Less communication overheads — communicating between the threads of one process is simple because the threads share the address space. Data produced by one thread is immediately available to all the other threads.

On the other hand, because threads in a group all use the same memory space, if one of them corrupts the contents of its memory, other threads might suffer as well. With processes, the operating system normally protects processes from one another, and thus if one corrupts its own memory space, other processes will not suffer.

2.6.3 Examples using Threads

Example 1: A responsive user interface

One area in which threads can be very helpful is in *user interface* programs. These programs are usually centred around a loop of reading user input, processing it, and showing the results of the processing. The processing part may sometimes take a while to complete, and the user is made to wait during this operation. By placing such long operations in a separate thread, while having another thread to read user input, the program can be more responsive. It may allow the user to cancel the operation in the middle.

Example 2: A Web server

The Web server needs to handle several download requests over a short period. Hence, it is more efficient to create (and destroy) a single thread for each request. Multiple threads can possibly be executing simultaneously on different processors.

Example 3: A graphical interface

In *graphical* programs, the problem is more severe since the application should always be ready for a message from the windowing system telling it to *repaint* part of its window. If it is too busy executing some other tasks, its window will not respond to the user, leading the user to think the program has crashed. In such a case, it is a good idea to have one thread handle the message loop of the windowing system and always ready to get such requests (as well as user input). Whenever this thread sees a need to do an operation that might take a long time to complete (say, more than `0.2` seconds in the worst case), it will delegate the job to a separate thread.

2.6.4 Creating and Destroying Threads

When a *multi-threaded* program starts executing, it has one thread running, which executes the `main()` function of the program. This is already a full-fledged thread, with its own thread ID.

Download the code for the `thread.c` program. While the program itself does not do anything useful, it will help you understand how threads work. Let us take a step-by-step look at what the program does:

- In `main()` we declare a variable called `thread_id`, which is of type `pthread_t`. This is basically an integer used to identify the thread in the system. After declaring `thread_id`, we call the `pthread_create()` function to create a real, living thread.

- The `pthread_create()` function gets four arguments. The first argument is a pointer to `thread_id`, used by `pthread_create()` to supply the program with the thread's identifier. The second argument is used to set some attributes for the new thread. In our case we supplied a `NULL` pointer to tell `pthread_create()` to use the default values.

  Notice that `print_hello()` accepts a `void*` as an argument and also returns a `void*` as a return value. This shows us that it is possible to use a `void*` to pass an arbitrary piece of data to our new thread, and that our new thread can return an arbitrary piece of data when it finishes.

  How do we pass our thread an arbitrary argument? We use the fourth argument in the `pthread_create()` call to pass *a pointer* to the data we intend to provide to our thread. If we do not want to pass any data to the new thread, we can set the fourth argument to `NULL`. The `pthread_create()` returns zero on success and a non-zero value on failure.

- After `pthread_create()` successfully returns, the program will consist of two threads. This is because the main program is also a thread and it executes the code in the `main()` function in parallel to the thread it creates. Think of it this way: if you write a program that does not use POSIX threads at all, the program will be single-threaded (this single thread is called the "main" thread).

- The call to `pthread_exit()` causes the current thread to exit and free any thread specific resources it is taking.

In order to compile a multi-threaded program using `gcc`, **we need to link it with the `pthreads` library.** Assuming you have this library already installed on your system, here is how to compile our first program:

```
$ gcc thread.c -o thread -lpthread
```

Compile the source code and run the `thread` executable. The output should be similar to:

```
Created new thread (208316031)...
Hello from new thread - got 11
```

2.7 Thread Synchronisation

2.7.1 More on Threads

A process as described in the lectures is sometimes called a *heavyweight* process. It contrasts with a thread (or *lightweight* process).

Threads are distinguished from traditional processes in that processes are typically independent, carry considerable state information, and have separate memory address spaces. Multiple threads within a same process, on the other hand, typically share some state information of the process, and share memory and other resources directly.

As mentioned in Section 2.6.2, context switching between threads in the same process is typically faster than context switching between processes.

As an example, when 40 users are logged on to the same Unix server and are running the same text editor such as `vi`, we have 40 threads. In this example, the 40 threads are at least sharing the memory space, which stores the code of the `vi` editor (which is a system program), but each thread has its own data section (i.e. each user is editing his/her own text document).

Every process must have at least one 'thread of control' or it will not do anything. You can create more than one 'thread of control' in one process by invoking the `pthread_create()` function as seen in Section 2.6.4.

Try exploring thread synchronisation yourself:

Download the code for the `thread_sync.c` program. This program creates two threads by calling the `pthread_create()`, and the two threads will run concurrently.

The `main()` just creates two threads of execution and then waits for them to terminate. The functions `pthread1()` and `pthread2()` (for the two threads) are actually of the same code. In this simple example, we just use several nested for loops to take up time on the CPU. You could certainly modify the two functions so that the two threads do some more useful work.

Compile this program using the following command line:

```
$ gcc thread_sync.c -o thread_sync -lpthread
```

Run the `thread` program for 20 times, each time observing the behaviour. Since the threads execute concurrently, you should see both threads start to run immediately (they will both print their start-up messages at the same time). You may find, over a number of runs, that on some occasions Thread 1 terminates first, and on other occasions Thread 2 terminates first, depending on which thread happens to get the CPU first.

2.7.2 "Waiting" for Threads

The `pthread_join()` function for threads is the equivalent of the `wait()` for processes. A call to `pthread_join()` blocks the calling thread until the thread with the identifier equal to the first argument terminates. The first argument to the `pthread_join()` is the identifier of the thread to join. The second argument is a `void` pointer.

Function prototype for the `pthread_join()` function:

```
#include <pthread.h>

int pthread_join(pthread_t tid, void* return_value);
```

If the `return_value` pointer is non-NULL, the `pthread_join()` will place at the memory location pointed to by the `return_value`, the value passed by the thread `tid` through the `pthread_exit()` call.

If we do not care about the return value of the main thread, we will set it to `NULL`.

2.7.3 Joinable and Detached Threads

At any point in time, a thread is either *joinable* or *detached*. (The default state is joinable.)

Joinable threads: Must be reaped or killed by other threads (using the `pthread_join()` function) in order to free memory resources.

Detached threads: Cannot be reaped or killed by other threads, and resources are automatically reaped on termination.

As a thread is joinable by default, we can make a thread detached by calling the `pthread_detach()` function and specifying the *thread ID* to it. A detached thread cannot be made joinable again.

Function prototype for the `pthread_detach()` function:

```c
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

A thread can get its own *thread ID* by calling the `pthread_self()` function, which returns the thread id of type `pthread_t`:

```c
pthread_t tid;
tid = pthread_self();
```

Unless threads need to synchronise among themselves, it is better to call the following instead of the `pthread_join()`:

```c
pthread_detach(pthread_self());
```

Try exploring joinable and detached threads yourself:

Download the code for the `thread_join.c` program. Compile and run it. **Is the output what you expected? Why, or why not?**

2.7.4 Thread Termination

There are several ways for threads to terminate. One way to safely terminate a thread is to call the `pthread_exit()` function, which is the equivalent of `exit()` for processes.

(**Note:** It is not necessary to use the `pthread_exit()` at the end of the main program. Otherwise, when it exits, all running threads will be killed.)

Try terminating threads yourself:

First create a copy of the `thread.c` program you have previously downloaded in (Section 2.6.4) and name the copy as `thread2.c`. Modify the `thread2.c` program as follows.

In the `print_hello()` function, add a line before the `printf` call with `sleep(1);`. This should be the first line of the function. In the `main()` function, comment out the last statement line, which contains the `pthread_exit()` call. Recompile and run the `thread2` executable. **Can you explain what happens? Document your explanation.**

Now, put the `pthread_exit()` call back in the main program, but remove it from the `print_hello()` function. Also add the sleep call to the `main()` function, just before the second `printf` call, and remove it from the `print_hello` function. Recompile and run the `thread2` executable. **Again, can you come up with an explanation of what happens? Likewise, document your explanation. No need to submit with your submission**

3 Assessed Laboratory Tasks

3.1 Task 1 (40%)

Write a C program that creates (or spawns) a child process to execute the Unix command "`date`" with a format string as the command-line argument; while the program itself (the parent process) will execute another Unix command "`cat`" with a filename as the command-line argument.
If you are not sure how the `date` and `cat` commands are meant to behave, experiment with them by running them directly from the command line first.

Instructions:

- The child process should execute the `date` command to display the current date and time in the specified format.
- The parent process should execute the `cat` command to display the contents of the given file.
- If you are not sure how the `date` and `cat` commands are meant to behave, experiment with them by running them directly from the command line first.

3.2 Task 2 (30%)

Begin by downloading the `scheduling.c` file from Moodle. You will find this file in the *downloads* folder provided with the lab materials.

Instructions:

1. First, compile the `scheduling.c` file.
2. Execute the program, specifying `4` when prompted for the number of child processes. Carefully observe and record the process IDs displayed in order.
3. Run the program once more, this time also, specify `4` for the number of child processes. Again, observe and jot down the process IDs.
4. Re-run the program, entering `10` as the number of child processes to create. Record the process IDs.
5. For a final time, run the program with `10` child processes. Note the process IDs.

Analysis:

Based on your observations, did the processes complete in the same order they were initiated? Elaborate on your findings and provide an explanation for the behavior you witnessed. Save your answers as a `.txt` file.

3.3 Task 3 (30%)

In this task, you will write a C program that sums the elements of an array in parallel using multiple threads. The array will be divided into equal-sized chunks, and each thread will be responsible for summing the elements in one chunk. At the end, the main thread will combine the partial sums computed by each thread to get the final sum.

Download `Sum.c` file from Moodle and complete the following:

Requirements:

- Your program should take two command-line arguments: the size of the array N and the number of threads T. The array size N should be evenly divisible by the number of threads T.
- Initialize an array of size N with random integers.
- Create T threads, and assign each thread a unique chunk of the array to sum. For example, if $N = 100$ and $T = 4$, each thread would be responsible for summing 25 elements.
- Each thread should compute the sum of its chunk and store the result in a shared array `partial_sums` of size T.
- The main thread should wait for all threads to finish, then compute the final sum by summing the elements of `partial_sums`.
- Print the final sum.

Instructions:

- Write a function `void *partial_sum(void *args)` that will be executed by each thread. This function should take a pointer to a struct containing the thread's ID, a pointer to the array, the start index of its chunk, and the size of its chunk. It should compute the sum of its chunk and store the result in `partial_sums`.

- In the main function, initialize the array with random integers, create the threads, and wait for them to finish.
- After all threads have finished, compute the final sum by summing the elements of `partial_sums`.
- Print the final sum.
- Free any dynamically allocated memory and destroy any pthread objects before exiting.
- Some of the work is already done in the template.
- Test your code.

3.4 Wrapping Up

Please upload all your work to the Moodle submission link, which should include your .c source files and your responses to any non-coding tasks. Please note that non-code answers should be provided in plain text files. There is no need to include the compiled executable programs. Ensure to upload all the necessary files before the conclusion of your lab class. Any delay in submitting these files to Moodle will result in a late penalty.