Philosophers 01: Threads and Mutexes

By Mia Combeau
In 42 School Projects
July 18, 2022
45 Min read
Add comment

P

The 42 school project philosophers confronts us with the concepts of concurrent programming. The mandatory part of the project introduces us to execution threads as well as the problems that inevitably arise from their shared memory. To rectify these, we will have to put mutexes to good use in order to control access to specific data. We will also need to be wary of our philosopher's sequencing to avoid any deadlocks.

The bonus part of the philosophers project consists of finding another solution to the same problem, but with processes instead of threads and semaphores instead of mutexes. This will be covered in a second article.

This is not a step-by step tutorial and there will be no ready-made solutions. It is a walkthrough, a guide to explore the issues and concepts behind the subject, the possible approaches we might take, and some tips to test our code.

Read the subject [pdf] at the time of this writing.

Table of Contents

The Rules of the Philosophers Project

Formulated in 1965 by Edsger Dijkstra, the dining philosophers problem is a computer science puzzle about concurrent programming. The exercise illustrates the synchronization and shared memory issues among different execution threads. The mandatory part of our philosophers project describes a version of this classic problem which we will study below. It differs slightly from the bonus part version, which will be described in the next article.

The Philosophers' Circumstances

One or more philosophers are having dinner around a circular table. Each philosopher has their own plate and is sitting between two other philosophers, philosopher N -1 and N + 1. Naturally, the last philosopher is sitting near the first. There is a fork between each of their plates and a large dish of spaghetti at the center of the table.

Illustration of the dining philosophers project with 5 philosophers. Clockwise: Sophocles, Plato, Aristotle, Democritus, and Epicurus.

Each philosopher will need to eat before falling asleep. When he wakes up, he will do some thinking before attempting to eat again.

The Philosopher's Dilemma

So far, so good. The problem comes from the fact that **in order to eat, each philosopher must use two forks**. It must be a very special type of spaghetti to require two forks! More intuitively, we could imagine the forks being chopsticks instead. So a philosopher must grab the fork to his right and the one to his left. But there are only as many forks as there are philosophers. This means that in order to eat, a philosopher must borrow his neighbor's fork. His neighbor can't eat at the same time. Only when his meal is done, when he is ready for his nap, will he set his forks back down on the table.

Of course, there are also time constraints. A philosopher can die of starvation if he fails to acquire a fork! Our program will take the following arguments:

- **`number_of_philosophers`**: the number of philosophers around the table,
- **`time_to_die`**: a number representing the time in milliseconds a philosopher has to live after a meal. If a philosopher hasn't started eating `time_to_die` milliseconds after the beginning of his last meal or the beginning of the simulation, he will die.
- **`time_to_eat`**: a number representing the time in milliseconds a philosopher takes to finish his meal. During that time, the philosopher keeps his two forks in hand.
- **`time_to_sleep`**: the time in milliseconds that a philosopher spends sleeping.
- **`number_of_times_each_philosopher_must_eat`**: an optional argument that allows the program to stop if all the philosophers have eaten at least that many times. If this argument is not specified, the simulation carries on unless a philosopher dies.

And on top of everything, our philosophers are also blind, deaf and mute! They **cannot communicate with each other** and therefore cannot warn each other if one of them is about to die.

Saving the Philosophers!

The goal of our program for the philosophers project is therefore to elaborate a sequencing algorithm so that the philosophers can take turns eating, without any of them starving to death. There will be unavoidable casualties in certain cases (for example when there is only one philosopher or when the `time_to_die` is smaller than the `time_to_eat` or the `time_to_sleep`), but we will strive to keep all the philosophers alive as much as possible.

The Philosophers Program Output

Each time a philosopher takes an action, our program must print a message formatted this way:

```
[timestamp_in_ms] [X] has taken a fork
[timestamp_in_ms] [X] is eating
[timestamp_in_ms] [X] is sleeping
[timestamp_in_ms] [X] is thinking
[timestamp_in_ms] [X] died
```

Of course, we must replace `[timestamp_in_ms]` by the actual number of milliseconds elapsed since the start of the simulation, and `[X]` by the ID of the philosopher in question. In the examples of this article, we will use IDs ranging from `1` to `number_of_philosophers` – 1 for clarity. But let's keep in mind that the subject explicitely **requires philosophers to be numbered `1` through `number_of_philosophers`.**

Concurrent Programming

The philosophers project requires us to confront concurrent programming. As opposed to sequential programming, concurrent programming allows a program to perform several tasks simultaneously instead of having to wait for the result of one operation to move onto the next. The operating system itself uses this concept to meet its users expectations. If we had to wait for a song to finish to be able to open our browser, or if we had to restart the computer to kill a program caught in an infinite loop, we'd die of frustration!

There are three ways to implement concurrency in our programs, but we will only concentrate on two of them: processes and threads.

We will take a closer look at implementing concurrency with processes in the bonus part of the philosophers project. However, it will be useful to have some sense of this subject in order to understand threads better. When our program is running, it is known as a process. This process can create child processes that run simultaneously. We've touched on this idea in the pipex project, where we had to create several child processes to execute simultaneous commands. Child processes created this way are clones of their parent. They are independent and have their own memory resources.

This means that one process can't very easily communicate with another one without some kind of inter-process communication mechanism. This flaw is also a strength: a process cannot accidentally overwrite another's virtual memory. However, a concurrent program that uses processes will tend to be slower because of its significant overhead.

So let's take a look at concurrent programming with threads, which also have their own pros and cons.

What is a Thread?

An execution **thread** is a logical sequence of instructions inside a process that is automatically managed by the operating system's kernel. A regular sequential program has a single thread, but modern operating systems allow us to create several threads in our programs, all of which run in parallel.

Each one of a process's threads has its own context: its own ID, its own stack, its own instruction pointer, it's own processor register. But since all of the threads are part of the same process, they share the same virtual memory address space: the same code, the same heap, the same shared libraries and the same open file descriptors.

A thread's context has a smaller footprint in terms of resources than the context of a process. Which means that it is much faster for the system to switch from one thread to the other, compared to switching from one process to another.

Threads don't have the strict parent-child hierarchy that processes do. Rather, they form a group of peers regardless of which thread created which other thread. The only distinction the "main" thread has is being the first one to exist at the beginning of the process. This means that within the same process, any thread can wait for any other thread to complete, or kill any other thread. Additionally, any thread can read and write to the same virtual memory. We will later examine the problems that can arise from that fact.

Using POSIX Pthreads

The standard interface in C to manipulate threads is POSIX with its **<pthread.h>** library. It contains around sixty functions to create, kill and join threads, as well as to manage their shared memory. We will only use a fraction of these in the philosophers project. In order to compile a program using this library, we can't forget to use `gcc`'s `-pthread` option:

```
gcc -pthread main.c
```

Creating a Thread

We can create a new thread from any other thread of the program with the `pthread_create` function. Its prototype is:

```
int pthread_create(pthread_t *restrict thread,
                          const pthread_attr_t *restrict
                          void *(*start_routine)(void *),
                          void *restrict arg);
```

Let's examine each argument we must supply:

- **thread**: a pointer towards a `pthread_t` type variable, to store the ID of the thread we will be creating.
- **attr**: an argument that allows us to change the default attributes of the new thread. This is beyond the scope of our philosophers project, so we will only ever pass `NULL` here.
- **start_routine**: the function where the thread will start its execution. This function will have as its prototype: **`void *function_name(void *arg);`**. When the thread reaches the end of this function, it will be done with its tasks.
- **arg**: a pointer towards an argument to pass to the thread's `start_routine` function. In order to pass several parameters to this function, we will need to give it a pointer to a data structure.

When the `pthread_create` function ends, the thread variable we gave it should contain the newly created thread's ID. The function itself returns 0 if the creation was successful, or another value if an error occurred.

Joining or Detaching Threads

In order to block the execution of a thread until another thread finishes, we can use the `pthread_join` function:

```
int pthread_join(pthread_t thread, void **retval);
```

Its parameters are as follows:

- **thread**: the ID of the thread that this thread should wait for. The specified thread must be joinable (meaning not detached – see below).
- **retval**: a pointer towards a variable that can contain the return value of the thread's routine function (the `start_routine` function we supplied at its creation). Here, we will not need this value: a simple `NULL` will suffice.

The `pthread_join` function returns 0 for success, or another number representing an error code.

Let's note that we can only wait for the termination of a specific thread. There is no way to wait for the first terminated thread without specifying an ID, as the `wait` function for child processes does.

For the philosopher project, we will no doubt want to make our program's main thread wait for each philosopher thread to finish. That way we will be able to free any memory and cleanly and safely terminate the program.

But in some cases, it is possible and preferable to not wait for the end of certain theads at all. In that case, we can detach the thread to tell the operating system that it can reclaim its resources right away when it finishes its execution. For that, we use the `pthread_detach` function (usually right after that thread's creation):

```
int pthread_detach(pthread_t thread);
```

Here, all we have to supply if the thread's ID. We get 0 in return if the operation was a success, or non-zero if there was an error. After detaching the thread, other threads will not be able to kill or wait for this thread with `pthread_join`.

A Practical Example of Threads

Let's write a small, simple program that creates two threads and joins them. The routine of each thread only consists of writing its own ID followed by a roughly-translated philosophic quote from the French novelist Victor Hugo.

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

# define NC     "\e[0m"
# define YELLOW "\e[1;33m"

// thread_routine is the function the thread invokes right after its
// creation. The thread ends at the end of this function.
void    *thread_routine(void *data)
{
        pthread_t tid;

        // Note that the pthread_self() function
        // is NOT allowed in the philosophers project !
        // We are only using it here as an example
        // to display this thread's ID.
        tid = pthread_self();
        printf("%sThread [%ld]: The heaviest burden is to exist without living.%s\n",
                YELLOW, tid, NC);
        return (NULL); // The thread ends here.
}

int     main(void)
{
        pthread_t       tid1;   // First thread's ID
        pthread_t       tid2;   // Second thread's ID

        // Creating the first thread that will go
        // execute its thread_routine function.
        pthread_create(&tid1, NULL, thread_routine, NULL);
        printf("Main: Created first thread [%ld]\n", tid1);
        // Creating the second thread that will also execute thread_routine.
        pthread_create(&tid2, NULL, thread_routine, NULL);
        printf("Main: Created second thread [%ld]\n", tid2);
```

```
	// The main thread waits for the new threads to end
	// with pthread_join.
	pthread_join(tid1, NULL);
	printf("Main: Joining first thread [%ld]\n", tid1);
	pthread_join(tid2, NULL);
	printf("Main: Joining second thread [%ld]\n", tid2);
	return (0);
}
```

When we compile and run this test, we can see that both threads were created and print their IDs correctly. If we run the program several times in a row, we might notice that the threads are always created in order, but sometimes, the main writes its message before the thread and vice versa. This shows that each thread is indeed executing in parallel to the main thread, and not sequentially.

Managing Threads' Shared Memory

One of the greatest qualities of threads is that they all share their process's memory. Each thread does have its own stack, but the other threads can very easily gain access to it with a simple pointer. What's more, the heap and any open file descriptors are totally shared between threads.

This shared memory and the ease with which a thread can access another thread's memory clearly also has its share of danger: it can cause nasty synchronization errors.

Synchronization Errors

Let's go back to our previous example and modify it to see how the shared virtual memory of threads can cause issues. We will create two threads and give each of them a pointer towards a variable in the main containing an unsigned integer, `count`. Each thread will iterate a certain number of times (defined in `TIMES_TO_COUNT`) and increment the count at each iteration. Since there are two threads, we will of course expect the final count to be exactly twice `TIMES_TO_COUNT`.

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

// Each thread will count TIMES_TO_COUNT times
#define TIMES_TO_COUNT 21000

#define NC      "\e[0m"
#define YELLOW  "\e[33m"
#define BYELLOW "\e[1;33m"
#define RED     "\e[31m"
#define GREEN   "\e[32m"

void    *thread_routine(void *data)
{
        // Each thread starts here
        pthread_t       tid;
        unsigned int    *count; // pointer to the variable created in main
        unsigned int    i;

        tid = pthread_self();
        count = (unsigned int *)data;
        // Print the count before this thread starts iterating:
        printf("%sThread [%ld]: Count at thread start = %u.%s\n",
                YELLOW, tid, *count, NC);
        i = 0;
        while (i < TIMES_TO_COUNT)
        {
                // Iterate TIMES_TO_COUNT times
                // Increment the counter at each iteration
                (*count)++;
                i++;
        }
        // Print the final count when this thread
        // finishes its own count
        printf("%sThread [%ld]: Final count = %u.%s\n",
                BYELLOW, tid, *count, NC);
        return (NULL); // Thread ends here.
}

int     main(void)
{
        pthread_t       tid1;
        pthread_t       tid2;
        // Variable to keep track of the threads' counts:
        unsigned int    count;

        count = 0;
        // Since each thread counts TIMES_TO_COUNT times and that
        // we have 2 threads, we expect the final count to be
        // 2 * TIMES_TO_COUNT:
        printf("Main: Expected count is %s%u%s\n", GREEN,
                                    2 * TIMES_TO_COUNT, NC);
        // Thread creation:
        pthread_create(&tid1, NULL, thread_routine, &count);
        printf("Main: Created first thread [%ld]\n", tid1);
        pthread_create(&tid2, NULL, thread_routine, &count);
        printf("Main: Created second thread [%ld]\n", tid2);
        // Thread joining:
        pthread_join(tid1, NULL);
        printf("Main: Joined first thread [%ld]\n", tid1);
        pthread_join(tid2, NULL);
        printf("Main: Joined second thread [%ld]\n", tid2);
        // Final count evaluation:
        if (count != (2 * TIMES_TO_COUNT))
                printf("%sMain: ERROR ! Total count is %u%s\n", RED, count, NC);
        else
                printf("%sMain: OK. Total count is %u%s\n", GREEN, count, NC);
        return (0);
}
```

Output:

Completely by chance, the first time we run the program, the result could be correct. But things aren't always as they appear! The second time we run it, the result is totally incorrect. If we continue to run the program several more times in a row, we'll even come to realize that it is wrong much more often than right… And it isn't even predictably wrong: the final count varies a lot from one run to the next. So what is happening, here?

The Danger of Data Races

If we examine the results closely, we can see that the final count is correct if and only if the first thread finishes counting before the second one starts. The minute their executions overlap, the result is wrong, and always less than the expected result.

So the problem is that both threads often access the same memory area at the same time. Let's say the count is currently 10. Thread 1 reads the value 10. More accurately, it copies that value 10 to its register in order to manipulate it. Then, it adds 1 to get a result of 11. But before it can save the result in the memory area pointed to by the count variable, thread 2 reads the value 10. Thread 2 then increments it to 11 as well. Both threads then save their result and there we have it! Instead of incrementing the count once for each thread, they ended up only incrementing it by one in total… That's why we're loosing counts and our final result is so wrong.

This situation is called a **data race**. It happens when a program is subject to the progression or timing of other uncontrollable events. It is impossible to predict if the operating system will choose the correct sequencing for our threads.

Indeed, if we compile the program with the `-fsanitizer=thread` and `-g` options and then run it, like this:

```
gcc -fsanitize=thread -g threads.c && ./a.out
```

We will get an alert: “WARNING: ThreadSanitizer: data race”.

So is there a way to stop a thread from reading a value while another one modifies it? Yes, thanks to mutexes!

What is a Mutex ?

A mutex (short for “**mut**ual **ex**clusion”) is a synchronization primitive. It is essentially a lock that allows us to regulate access to data and prevent shared resources being used at the same time.

We can think of a mutex as the lock of a bathroom door. One thread locks it to indicate that the bathroom is occupied. The other threads will just have to patiently stand in line until the door is unlocked before they can take their turn in the bathroom.

Declaring a Mutex

Thanks to the `<pthread.h>` header, we can declare a mutex type variable like this:

```
pthread_mutex_t    mutex;
```

Before we can use it, we first need to initialize it with the following function:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *mutexa
```

There are two parameters to supply:

- **mutex**: the pointer to a variable of `pthread_mutex_t` type.
- **mutexattr**: a pointer to specific attributes for the mutex. We will not worry about this parameter in this project, we can just say `NULL`.

The `pthread_mutex_init` function only ever returns 0.

Locking and Unlocking a Mutex

Then, in order to lock and unlock our mutex, we need two other functions. Their prototypes are as follows:

```
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

If the mutex is unlocked, **`pthread_mutex_lock`** locks it and the calling thread becomes its owner. In this case, the function ends immediately. However, if the mutex is already locked by another thread, `pthread_mutex_lock` suspends the execution of the calling thread until the mutex is unlocked.

The **`pthread_mutex_unlock`** function unlocks a mutex. The mutex to be unlocked is assumed to be locked by the calling thread, and the function only sets it to unlocked. Let's be careful to note that this function does not check if the mutex is in fact locked and that the calling thread is actually its owner: a mutex could therefore be unlocked by a thread that did not lock it in the first place. We will need to be careful about arranging `pthread_mutex_lock` and `pthread_mutex_unlock` in our code.

Both of these functions return 0 for success and an error code otherwise.

Destroying a Mutex

When we no longer need a mutex, we should destroy it with the following function:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

This function destroys an *unlocked* mutex, freeing whatever resources it might hold. In the LinuxThreads implementation of POSIX threads, no resources are associated with mutexes. In that case, `pthread_mutex_destroy` doesn't do anything other than check that the mutex isn't locked.

Example Implementation of a Mutex

We can now solve our previous example's problem of incorrect final count by using a mutex. For this, we need to create a small structure that will contain our `count` variable and the mutex that will protect it. We can then pass this structure to our threads' routines.

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

// Each thread will count TIMES_TO_COUNT times
#define TIMES_TO_COUNT 21000

#define NC      "\e[0m"
#define YELLOW  "\e[33m"
#define BYELLOW "\e[1;33m"
```

```c
#define RED     "\e[31m"
#define GREEN   "\e[32m"

// This structure contains the count as well as the mutex
// that will protect the access to the variable.
typedef struct s_counter
{
        pthread_mutex_t count_mutex;
        unsigned int    count;
} t_counter;

void    *thread_routine(void *data)
{
        // Each thread starts here
        pthread_t       tid;
        t_counter       *counter; // pointer to the structure in main
        unsigned int    i;

        tid = pthread_self();
        counter = (t_counter *)data;
        // Print the count before this thread starts iterating.
        // In order to read the value of count, we lock the mutex:
        pthread_mutex_lock(&counter->count_mutex);
        printf("%sThread [%ld]: Count at thread start = %u.%s\n",
                YELLOW, tid, counter->count, NC);
        pthread_mutex_unlock(&counter->count_mutex);
        i = 0;
        while (i < TIMES_TO_COUNT)
        {
                // Iterate TIMES_TO_COUNT times
                // Increment the counter at each iteration
                // Lock the mutex for the duration of the incrementation
                pthread_mutex_lock(&counter->count_mutex);
                counter->count++;
                pthread_mutex_unlock(&counter->count_mutex);
                i++;
        }
        // Print the final count when this thread finishes its
        // own count, without forgetting to lock the mutex:
        pthread_mutex_lock(&counter->count_mutex);
        printf("%sThread [%ld]: Final count = %u.%s\n",
                BYELLOW, tid, counter->count, NC);
        pthread_mutex_unlock(&counter->count_mutex);
        return (NULL); // Thread termine ici.
}

int     main(void)
{
        pthread_t       tid1;
        pthread_t       tid2;
        // Structure containing the threads' total count:
        t_counter       counter;

        // There is only on thread here (main thread), so we can safely
        // initialize count without using the mutex.
        counter.count = 0;
        // Initialize the mutex :
        pthread_mutex_init(&counter.count_mutex, NULL);
        // Since each thread counts TIMES_TO_COUNT times and that
        // we have 2 threads, we expect the final count to be
        // 2 * TIMES_TO_COUNT:
        printf("Main: Expected count is %s%u%s\n", GREEN,
                                2 * TIMES_TO_COUNT, NC);
        // Thread creation:
        pthread_create(&tid1, NULL, thread_routine, &counter);
        printf("Main: Created first thread [%ld]\n", tid1);
        pthread_create(&tid2, NULL, thread_routine, &counter);
        printf("Main: Created second thread [%ld]\n", tid2);
        // Thread joining:
        pthread_join(tid1, NULL);
        printf("Main: Joined first thread [%ld]\n", tid1);
        pthread_join(tid2, NULL);
        printf("Main: Joined second thread [%ld]\n", tid2);
        // Final count evaluation:
        // (Here we can read the count without worrying about
        // the mutex because all threads have been joined and
        // there can be no data race between threads)
        if (counter.count != (2 * TIMES_TO_COUNT))
                printf("%sMain: ERROR ! Total count is %u%s\n",
                                RED, counter.count, NC);
        else
                printf("%sMain: OK. Total count is %u%s\n",
                                GREEN, counter.count, NC);
        // Destroy the mutex at the end of the program:
        pthread_mutex_destroy(&counter.count_mutex);
        return (0);
}
```

Let’s see if our result is still incorrect now:

There! Our result is now always right, every time we run the program, even if the second thread starts counting before the first is finished.

Beware of Deadlocks

However, mutexes can often provoke **deadlocks**. It's a situation in which each thread waits for a resource held by another thread. For example, thread T1 acquired mutex M1 and is waiting for mutex M2. Meanwhile thread T2 acquired mutex M2 and is waiting for mutex M1. In this situation, the program stays pending in perpetuity and must be killed.

A deadlock can also happen when a thread is waiting for a mutex that it already owns!

Let's try to demonstrate a deadlock. In this example, we will have two threads that need to lock two mutexes, `lock_1` and `lock_2` before being able to increment a counter. The routines of the two threads will be slightly different: the first thread will lock `lock_1` first, while thread 2 will start by locking `lock_2`…

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

#define NC      "\e[0m"
#define YELLOW  "\e[33m"
#define BYELLOW "\e[1;33m"
#define RED     "\e[31m"
#define GREEN   "\e[32m"

typedef struct s_locks
{
        pthread_mutex_t lock_1;
        pthread_mutex_t lock_2;
        unsigned int    count;
}       t_locks;

// The first thread invokes this routine:
void    *thread_1_routine(void *data)
{
        pthread_t       tid;
        t_locks         *locks;

        tid = pthread_self();
        locks = (t_locks *)data;
        printf("%sThread [%ld]: wants lock 1%s\n", YELLOW, tid, NC);
        pthread_mutex_lock(&locks->lock_1);
        printf("%sThread [%ld]: owns lock 1%s\n", BYELLOW, tid, NC);
```

```c
	printf("%sThread [%ld]: wants lock 2%s\n", YELLOW, tid, NC);
	pthread_mutex_lock(&locks->lock_2);
	printf("%sThread [%ld]: owns lock 2%s\n", BYELLOW, tid, NC);
	locks->count += 1;
	printf("%sThread [%ld]: unlocking lock 2%s\n", BYELLOW, tid, NC);
	pthread_mutex_unlock(&locks->lock_2);
	printf("%sThread [%ld]: unlocking lock 1%s\n", BYELLOW, tid, NC);
	pthread_mutex_unlock(&locks->lock_1);
	printf("%sThread [%ld]: finished%s\n", YELLOW, tid, NC);
	return (NULL); // The thread ends here.
}

// The second thread invokes this routine:
void	*thread_2_routine(void *data)
{
	pthread_t	tid;
	t_locks		*locks;

	tid = pthread_self();
	locks = (t_locks *)data;
	printf("%sThread [%ld]: wants lock 2%s\n", YELLOW, tid, NC);
	pthread_mutex_lock(&locks->lock_2);
	printf("%sThread [%ld]: owns lock 2%s\n", BYELLOW, tid, NC);
	printf("%sThread [%ld]: wants lock 1%s\n", YELLOW, tid, NC);
	pthread_mutex_lock(&locks->lock_1);
	printf("%sThread [%ld]: owns lock 1%s\n", BYELLOW, tid, NC);
	locks->count += 1;
	printf("%sThread [%ld]: unlocking lock 1%s\n", BYELLOW, tid, NC);
	pthread_mutex_unlock(&locks->lock_1);
	printf("%sThread [%ld]: unlocking lock 2%s\n", BYELLOW, tid, NC);
	pthread_mutex_unlock(&locks->lock_2);
	printf("%sThread [%ld]: finished.%s\n", YELLOW, tid, NC);
	return (NULL); // The thread ends here.
}

int	main(void)
{
	pthread_t	tid1;	// ID of the first thread
	pthread_t	tid2;	// ID of the second thread
	t_locks		locks;	// Structure containing 2 mutexes

	locks.count = 0;
	// Initialize both mutexes :
	pthread_mutex_init(&locks.lock_1, NULL);
	pthread_mutex_init(&locks.lock_2, NULL);
	// Thread creation:
	pthread_create(&tid1, NULL, thread_1_routine, &locks);
	printf("Main: Created first thread [%ld]\n", tid1);
	pthread_create(&tid2, NULL, thread_2_routine, &locks);
	printf("Main: Created second thread [%ld]\n", tid2);
	// Thread joining:
	pthread_join(tid1, NULL);
	printf("Main: Joined first thread [%ld]\n", tid1);
	pthread_join(tid2, NULL);
	printf("Main: Joined second thread [%ld]\n", tid2);
	// Final count evaluation:
	if (locks.count == 2)
		printf("%sMain: OK. Total count is %d\n", GREEN, locks.count);
	else
		printf("%sMain: ERROR ! Total count is %u\n", RED, locks.count);
	// Mutex destruction:
	pthread_mutex_destroy(&locks.lock_1);
	pthread_mutex_destroy(&locks.lock_2);
	return (0);
}
```

As we can see in the following output, most of the time, there is no problem with this configuration because the first thread has a small head start on the second one. But sometimes, both threads lock their first mutexes exactly at the same time, in which case the program stays blocked because the threads are caught in a deadlock.

Studying this result, we can clearly see that the first thread locked `lock_1` and the second locked `lock_2`. The first thread now wants to lock `lock_2` and the second wants `lock_1`, but neither have any way of getting those mutexes. They are deadlocked.

Dealing with Deadlocks

There are several ways to deal with deadlocks like these. Among other things, we can:

- ignore them, but only if we can prove that they will never happen. For example when the time intervals between resource requests are very long and far between.
- correct them when they happen by killing a thread or by redistributing resources, for example.
- prevent and correct them before they happen.
- avoid them by imposing a strict order for resource acquisition. This is the solution to our previous example: the threads should both ask for `lock_1` first.
- avoid them by forcing a thread to release a resource before asking for new ones, or before renewing its request.

We will see later what kind of deadlocks we should watch out for in our philosophers project and some ways to manage them.

Managing Time in the Philosophers Project

The concept of time is very important in the philosophers project. Philosophers have a certain amount of time to eat and sleep, and die if they don't manage to get a meal within a certain time frame.

The Gettimeofday Function

First of all, we need to be able to get the operating system's temporal values. For that, we need the system library `<sys/time.h>`. It contains the **`gettimeofday`** function, that has the following prototype:

```
int gettimeofday(struct timeval *tv, struct timezone *tz)
```

The parameters we must supply are the following:

- **tv**: a `timeval` type data structure that contains the number of seconds (`tv_sec`) and the number of microseconds (`tv_usec`) that have passed since the 1st of January 1970, a date chosen for technical reasons by the creators of the Unix operating system.
- **tz**: a `timezone` type structure. Since the use of this structure is now obsolete and that we don't need it for the philosophers project in any case, we will simply put `NULL` here.

When we call this function, it will fill the `timeval` structure and return 0 if the call was successful, or -1 if there was an error.

In order to calculate the time in milliseconds as required by the subject of the philosophers project, we need to apply the following formula after collecting the data into a `tv` structure with `gettimeofday`:

```
(tv.tv_sec * 1000) + (tv.tv_usec / 1000)
```

Indeed, the equivalence between seconds, milliseconds and microseconds is as follows:

Seconds (s)	Milliseconds (ms)	Microseconds (µs)
1	1 000	1 000 000
0.001	1	1 000
0.000001	0.001	1
1^{0} seconds	1^{-3} seconds	1^{-9} seconds

The Usleep Function

The **`usleep`** function in the `<unistd.h>` library allows us to stop the execution of the calling thread for a certain number of microseconds. Its prototype is:

```
int usleep(useconds_t usec);
```

Let's take note that this function guarantees the suspension of execution for *at least* **usec** microseconds. This means that the suspension might be longer depending on the operating system's activity and its granularity.

The `usleep` function returns 0 for success and -1 for failure.

Example of Time Display in Milliseconds

Let's create a little program to test the display of time in milliseconds. It will first save the start time in milliseconds at the start of the program. Every 100 ms, it will print the time since the start of the program. After 2000 ms, it stops.

```c
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>

// Function that returns the current time since the
// 1st of January 1970 in milliseconds
time_t  get_time_in_ms(void)
{
        struct timeval  tv;

        gettimeofday(&tv, NULL);
        return ((tv.tv_sec * 1000) + (tv.tv_usec / 1000));
}

int     main(void)
{
        time_t  start_time;
        time_t  end_time;
        time_t  time;

        start_time = get_time_in_ms();
        end_time = start_time + 2000;
        time = get_time_in_ms();
        printf("Time in ms = %ld. Time since start = %ld\n", time, time - start_time);
        while (get_time_in_ms() < end_time)
        {
                usleep(100 * 1000); // Suspend for 100 milliseconds
                time = get_time_in_ms();
                printf("Time in ms = %ld. Time since start = %ld\n",
                                                time, time - start_time);
        }
        return (0);
}
```

Output:

We can see here that a small delay sets in and increases over time because of the `usleep` function. We will have to take this into account and try to rectify this problem in our philosophers project so that our timestamps are as precise as possible.

Strategies for the Philosophers Project

For the mandatory part of the philosophers project, we get two clues concerning the structure of our program:

- Each philosopher must be a thread.
- Each fork must be protected by a mutex.

On top of that, we know that we will no doubt need at least one structure to manage the variables unique to each philosopher and another to manage the general variables of the program. Among those, we will probably need at least one mutex to protect the message-printing by philosophers so that their messages don't get scrambled, and an array of mutexes for the forks.

That's already a good start for the project: we can start parsing the arguments and initializing our structures. Next, we can create our philosophers' basic eat-sleep-think routines. Then, we need to devise some strategies to detect the death of a philosopher and to handle the forks in an optimal way to keep our philosophers alive…

Detecting Death in the Philosophers Project

Each philosopher must be a thread, but not all threads must be philosophers! When we create the philosopher threads, nothing is stopping us from creating an extra thread. That supervisor thread will have only one goal: check the status of each philosopher in a loop. If one of the philosophers is about to die, the supervisor can then stop the simulation. Likewise if all the philosophers have eaten the number of times indicated at the start of the program.

In order to indicate the status of the simulation, we could create a boolean variable protected by a mutex. Only the supervisor thread could change this variable. But the philosopher threads would be able to read it to check the status of the simulation before and during each of their actions, and before printing a status update message. If this variable is equal to 1 for example, the threads will see that they can safely continue their instruction sequence. However, if it is equal to 0, they must immediately stop what they are doing because the simulation must end as quickly as possible.

One important thing to note: it's a little counter-intuitive, but **a philosopher can starve while he is eating**! The subject indicates that a philosopher dies if he has not eaten `time_to_die` since the start of the simulation or since *the start* of his last meal. This implies that he can die while eating, for example in the case where the `time_to_eat` is greater than the `time_to_die`.

If someone dies, we must print the message "X died" in less than 10 ms after the actual death. No other message can follow this announcement and the simulation must end immediately.

The Usleep Problem in the Philosophers Project

The `usleep` function enables us to suspend the execution of a philosopher thread for the required time to eat, sleep or think. The first thing to remember is that `usleep` measures time in microseconds. In order to indicate a time in the order of milliseconds, we will have to multiply it by 1000:

```
// To suspend the thread for time_to_eat
// we need to multiply by 1000 to get the
// time in microseconds for usleep:
usleep(time_to_sleep * 1000);
```

But the simulation of our philosophers program might end in the middle of a meal or a nap. So we will probably want to check if the simulation has stopped or not during this suspension time. If the simulation has stopped, we need to be able to wake the thread up so that it can end as quickly as possible, especially if the `time_to_sleep` or `time_to_eat` is very long. The problem is that there is no way to interrupt `usleep`.

So we'll probably want to create our own function to suspend a thread, one that can periodically check the status of the simulation. It might be constructed something like this:

```
// Parameters of philo_sleep:
//     table = pointer to the variables of the "table", i.e. the program variables
//             like the flag indicating the end of the simulation
//     sleep_time = the time in milliseconds that the thread must be suspended for,
//                  typically time_to_sleep or time_to_eat
void    philo_sleep(t_table *table, time_t sleep_time)
{
        // Variable to measure when the philosopher thread must resume:
        time_t  wake_up;

        wake_up = get_time_in_ms() + sleep_time;
        // Loop as long as it's not wake up time:
        while (get_time_in_ms() < wake_up)
        {
                // Check if the simulation has stopped. If it has,
                // stop the loop (and this function) immediately
                // to continue thread execution:
                if (has_simulation_stopped(table))
                        break ;
                // If the simulation is still going, usleep for a short time:
                usleep(100);
        }
}
```

Handling Forks in the Philosophers Project

The greatest challenge in the philosophers project is managing the forks. As we've seen previously, there is one fork between each philosopher and a philosopher needs two forks to be able to eat. If there is only one philosopher, there will only be one fork on the table, which means that he will inevitably starve to death.

The philosophers must never duplicate forks when they grab them. This is partly why the subject of the philosophers project requires that we "protect the forks state with a mutex for each of them". This can mean one of two things :

- the forks are themselves mutexes.
- the forks are variables protected by mutexes.

This choice is up to each student's interpretation since the subject does not elaborate on this point.

Since the forks are placed between each philosopher, philosophers must take the forks on either side of their own plates, not the ones on the other side of the table.

An Immediate Deadlock

The subject of the philosophers project does not allow us to use any of the pthread functions that would allow us to test whether a mutex is in use or not. So a philosopher cannot check in advance which one of his two forks is free to inform his choice of which fork to take first. This means that if his first fork is unavailable, the philosopher won't be able to try to take his second fork instead. He will stay suspended until his first fork is firmly in hand before attempting to take his second fork.

If every philosopher pounces on his right hand fork-mutex at once for example, we will have an immediate deadlock, as illustrated here.

Immediate deadlock! No one can eat...

We've seen that it is impossible to predict the order in which the operating system will execute each thread. This immediate deadlock will not happen every time we run the program, but it will happen often enough to cause problems.

We need to find a solution to eliminate the possibility of this deadlock. And that solution might well be a combination of the ones described below.

Setting the Fork Back on the Table

The obvious solution to a deadlock like this is to make a philosopher set his fork back down if he can't get his second fork. But how can we interrupt a mutex's thread suspension?

Of course, we can't. What we can do however, is create a small structure for our forks, which will contain a boolean variable indicating the status of the fork (in use or not) and a mutex to protect it. Something like this, perhaps:

```
typedef struct s_fork
{
        pthread_mutex_t fork_lock;
        bool            in_use;
}       t_fork;
```

A philosopher taking a fork will have to lock the mutex, change the value of the `in_use` variable to 1 and unlock the mutex again. When attempting to take his second fork, he can lock its mutex, check the variable to see if it is available. If not, he'll unlock the mutex and release his first fork by setting the first fork's variable back to 0. Then, he will wait a little before trying to take both forks again. In the meantime, his neighbor can use that first fork to eat if need be. This system avoids most deadlock possibilities.

Of course, to avoid spamming messages like "X has taken a fork", they should only be printed when the philosopher actually has both forks in hand.

The main objection to this method is the fact that the subject of the philosopher project explicitly forbids communication between philosophers. From a technical perspective, this solution doesn't require any communication between a philosopher and his neighbor, only between the philosopher and the forks. But would a philosopher that didn't know anything about what is happening around him really have the instinct to put his fork back down on the table if he couldn't get his second fork? That's a question that will have to be debated during the evaluation if this method is used.

Right-Handed and Left-Handed Philosophers

The problem with the immediate deadlock is that the philosophers all grab their right fork first. Yet a philosopher's right fork is also his neighbor's left fork. If we designate one out of two philosophers as left-handed, that should solve the immediate deadlock issue.

One of every two philosophers is left-handed. This avoids an immediate deadlock.

In this illustration, philosophers that have an even-numbered ID are right-handed and will therefore try to take their right fork first. The opposite is true for the odd-numbered philosophers, who are left-handed and try to pick up their left fork first. Here, philosopher 0 is right handed and takes his right fork, f0. Philosopher 1, being left-handed, first takes his left fork, f2. This forces right-handed philosopher 2 to wait since his right fork, f2, is in use. And left-handed philosopher 3 grabs his left fork f4 before right handed philosopher 4, who must now wait.

Thanks to this distinction between right and left-handed philosophers, there is no possibility of immediate deadlock. Philosophers 0 and 3 will probably eat first, then philosophers 4 and 1, followed by philosopher 2. It's also possible philosophers 1 and 3 will eat first, followed by 0 and 2, and finally 4. In any case, deadlocks are avoided.

In the case of an ever number of philosophers, this solution is very efficient and produces reliable results. However, a problem surfaces quickly if there is an odd number of philosophers around the table…

Wrongful Deaths with an Uneven Number of Philosophers

Just about once every three or four times, a philosopher dies when he shouldn't. Another philosopher takes his fork, and so his meal, before he gets a chance to. These unreliable results are simply due to the thread execution order that the operating system chose, which, of course, has no concept of which philosopher needs to eat before another.

See a wrongful death output example

Here is an example of bad output with only the right and left-handed philosopher method. With these arguments (5 philosophers, 1200 ms as time_to_die, 300 ms to eat, 300 ms to sleep, and 2 mals to eat), every philosopher should eat at least two meals and no one should die. Here though, someone did die…

The slide below is a direct transcription of the erroneous result above. Arrows to the left and right of the image allow us to move forwards and backwards in time. Below each image are notes to help us examine each step of this simulation's evolution.

In this example, there are five philosophers. The time_to_die is 12, the time_to_eat and the time_to_sleep is 3. Even-numbered philosophers are right-handed and odd-numbered philosophers are left-handed.

Philosophers 0 and 2 are right-handed. Therefore, they take their right forks, f0 and f2 respectively, while philosopher 3 is left-handed and takes fork f4. Since philosopher 1's left fork and philosopher 4's right fork are busy, those two will wait to be able to eat.

Philosopher 1 starts eating, as well as philosopher 3, who managed to grab fork f3 before philosopher 2, who will now have to wait his turn.

Times advances. Philosophers 0 and 3 are eating, while philosophers 1, 2 and 4 wait for their forks.

Times advances. Philosophers 0 and 3 are still eating, while philosophers 1, 2 and 4 wait patiently.

Times ticks up. Philosophers 0 and 3 are done eating and set their forks down before taking their nap. Philosopher 2 is still holding his right fork.

Philosophers 2 and 4 grab their napping neighbors' forks. Philosopher 1 still hasn't eaten and must wait some more for his left fork, presently in use by philosopher 2.

Time ticks by. Philosophers 2 and 4 are eating while philosophers 0 and 3 sleep. Philosopher 1 waits patiently for his fork.

Time advances. Philosophers 2 and 4 are still devouring their meal while philosophers 0 and 3 nap. Philosopher 1 is still waiting for his fork.

Time ticks by. Philosophers 2 and 4 are done with their meal and set their forks down again before falling asleep. Philosophers 0 and 3 are awake. Philosopher 1 readies himself to pounce on the fork he's been longing for.

Philosopher 1 finally got his left fork but didn't have time to grab his right fork: philosopher 0 was quicker. He will now have to wait a little longer while philosophers 0 and 3 eat. Philosophers 2 and 4 are presently sleeping.
This moment is not going to be fatal to philosopher 1, but we can observe the same issue reappear over and over in this program, and it is sometimes fatal.

Time ticks by. Philosophers 0 and 3 enjoy their meal while philosopher 1's stomach growls. Philosophers 2 and 4 are sleeping peacefully.

Time ticks by. Philosophers 0 and 3 are still eating while philosopher 1 is starving. Philosophers 2 and 4 are still sound asleep.

Time advances. Philosophers 0 and 3 are full and fall asleep right away while philosophers 2 and 4 are waking up. Desperate philosopher 1 is ready to kill for his right fork.

Well, it wasn't as hard as philosopher 1 anticipated to grab his right fork since his neighbor was sleeping. He is now happily eating in the company of philosopher 4. Philosophers 0 and 3 are sleeping. Philosopher 2 is getting hungry, but philosopher 1 was holding his right fork firmly in hand when he woke up. He just needs a little patience.

Time ticks by. Philosophers 1 and 4 eat in peace while their neighbors 0 and 3 sleep. Philosopher 2 is waiting for his fork.

Time ticks by. Philosophers 1 and 4 are still masticating while their neighbors 0 and 3 dream. Philosopher 2 is now starving…

Time passes. Philosophers 1 and 3 fall asleep after their meal, while philosophers 0 and 3 are waking up. Philosopher 2 is absolutely starving and must grab both of his forks and eat now if he is to survive.

Philosopher 2 takes his right fork but philosopher 3 beats him to it! Oblivious, philosopher 3 just signed philosopher 2's death warrant. He begins his meal in the company of philosopher 0 while philosophers 1 and 4 are sleeping. This moment, where a philosopher takes a fork too early at the expense of another, happened earlier with philosopher 1. It was not fatal then but it is this time.

Time ticks by. Philosopher 2 will starve soon. Oblivious, philosophers 0 and 3 eat while philosophers 1 and 4 sleep.

Time advances. Philosopher 2 is on his death bed. Still oblivious, philosophers 0 and 3 keep eating while philosophers 1 and 4 keep sleeping.

Philosophers 0 and 3 finish their meals and fall asleep. Philosophers 1 and 4 wake up to find the lifeless corpse of their colleague, philosopher 2, who starved to death. In a parallel universe, this death might have been prevented!

If we with to use this right and left-handed method for philosophers, we will need to find a solution to the problem of the uneven number of philosophers. We could perhaps think of a way to better stagger each philosopher’s meals.

Delaying Certain Philosopher Threads

Another solution to avoid an immediate deadlock is to create a small delay between threads at the start. This gives a small head start to the philosophers that go first, enough to grab their forks before the others.

A Delay Between Each Thread's Creation

For example, we could add a small `usleep` after creating each thread to give the first philosopher a head start on the second one and so on.

The thing is, our philosopher program must be able to simulate up to 200 philosophers without creating too much of a delay in the timestamps it prints out. If we add a 1 millisecond delay at the launch of each philosopher, the 200th philosopher will start at the 200th millisecond, which could bring about a premature death. Plus, if we have a supervisor thread to detect a philosopher's death, it might consider a not-yet-created philosopher thread as dead! And we can't apply the same 200 ms delay to the supervisor thread without running the risk of missing some deaths. For example, if there are 200 philosophers and that the `time_to_die` is 1 ms, we need to be able to detect the death immediately, not 200 ms later.

A Delay at the Beginning of Odd-Numbered Philosophers' Routines

So what if we added a small `usleep` only for the philosopher threads that have an odd-numbered ID? That would allow even-numbered philosophers to eat first and odd-numbered philosophers would have to wait their turn. This also wouldn't create a large time lag.

This solution is very effective in a great majority of cases. However, we must still choose an appropriate time delay. We can't make the odd-numbered philosophers wait for `time_to_eat` because if `time_to_eat` happens to be 0, meaning philosophers eat instantaneously, we will end up with the same initial deadlock we are trying to avoid, especially if `time_to_sleep` is also 0! We'd better define a small value like 1 or 2 milliseconds. That would be enough time to let the first batch of philosophers take both of their forks.

We still need to be careful with this solution if there is an odd number of philosophers. Just like the right and left-handed philosopher solution described above, one might die prematurely because his neighbors repeatedly steal his meals…

Giving the Philosophers Time to Think

Our philosophers program does not take an argument corresponding to the length of time each philosopher spends thinking. The thinking routine could very well consist of nothing more than printing the status “X is thinking” and moving on directly to fork-hunting in order to eat.

But the subject does not explicitly forbid us from calculating a small period of reflection for each philosopher. Shouldn’t a philosopher have time to think about deeper things than the availability of his forks ?

Why Calculate a Time to Think?

Calculating this time to think is very useful to avoid a philosopher eating too often. If he does, he might deprive his neighbor of several meals in a row, which ends up killing him. A time to think allows time for the hungriest philosophers to pick up forks, and doesn’t require any actual communication between philosophers. Indeed, the calculation for this time to think is strictly personal, requiring no information from any other philosopher. It is simply the calculation of when this philosopher will get hungry again.

How to Calculate a Time to Think

```
time_to_think = (time_to_die - (get_time_in_ms() - philo->last_meal) - time_to_eat) / 2
```

Here, we are calculating the time a philosopher has before he absolutely needs to eat without starving at `time_to_die`. In order to give him enough of a margin to find his forks, we subtract the actual `time_to_eat` and divide the result by two. Let’s say that this philosopher’s last meal was at 200 ms and he slept for another 200 ms. We are now at 600 ms. The `time_to_die` is 1000 ms and the `time_to_eat` is 200 ms:

```
time_to_think = (time_to_die - (get_time_in_ms() - philo->last_meal) - time_to_eat) / 2
              = (   1000     - (       600       -       200       ) -     200    ) / 2
              = ( 1000 - 400 - 200 ) / 2
              = ( 600 - 200 ) / 2
              = 400 / 2
              = 200
```

We’ve determined here that the philosopher will die if he doesn’t eat in the next 600 ms. We give him a margin of `time_to_eat`, meaning we consider that he needs to eat within the next 400 ms maximum. Then, we divide this time by two to give him a safe 200 ms time to think. This way, he will have enough time to find his forks and eat, all the while giving his neighbors a chance to eat first.

If this result is negative, the philosopher will of course need to try to eat right away, so his time to think should be set back to 0. We’ll also want to test the case where the time to think is very long. If `time_to_die` is 10,000 ms and `time_to_eat` is 10 ms, we won’t want to wait 4,900 milliseconds or so before trying to find forks. That would be much too long of a delay before printing a status message to the screen if all the philosophers were thinking at the same time. To avoid this, we can simply limit the time to think to 500 or 600 ms maximum.

The Case of the Lone Philosopher

A lone philosopher must take his one and only fork and die at `time_to_die` milliseconds because he has no second fork to be able to eat. This particular case might very well create a self-deadlock if we tell the philosopher that his second fork is the same one as his first. The thread will stay suspended indefinitely, waiting for a mutex that it already owns.

The simplest solution to this problem is probably to modify the philosopher thread routine if it is the only one present at the table. We can write a simple function for it that only picks up one fork, waits `time_to_die` ms and immediately dies.

Of course, if we are using the “setting the fork back on the table” method described earlier, the philosopher thread will not encounter a deadlock since he will be juggling with his fork until the supervisor thread notices he has not eaten by `time_to_die`.

Tips for Testing the Philosophers Project

The most important thing to remember when testing our philosophers project or any other program that makes use of threads, is to test the same thing many times in a row. Oftentimes, synchronization errors won’t be apparent on the first or second or even third run. It depends on the order the operating system chooses for the execution of each thread. By running the same test over and over, we may see a large variation in the results. For example:

- if a philosopher dies one time out of six when running the program with the exact same arguments, there must be a problem either with the order in which philosophers take the forks, or in the system that detects a philosopher's death.
- if, one time out of ten with the same arguments. our program stays stuck indefinitely, then there is a possible deadlock that did not occur in the previous tests.

Our goal, on top of keeping philosophers alive, is of course to avoid these types of synchronization errors. We need to rigorously test our code, with the same arguments over and over, to ensure we get coherent results and don't miss hidden errors.

Tools to Test the Philosophers Project

There are some tools we can use to help us detect thread-related errors like possible data races, deadlocks and lock order violations:

- The `-fsanitize=thread -g` flag we can add when compiling with gcc. The `-g` option displays the specific files and line numbers involved.
- The thread error detection tool Helgrind that we can run our program with, like this: `valgrind --tool=helgrind ./philo <args>`.
- DRD, another thread error detection tool that we can also run our program with, like this: `valgrind --tool=drd ./philo <args>`.

Of course, these tools, in particular those of Valgrind, slow our simulation down and often provoke premature deaths among the philosophers…

And as always, we can't forget to check for memory leaks with `-fsanitize=address` and `valgrind`!

Tests and Expected Results for the Philosophers Project

The expected results of the following tests are only presented here as an aid to the development of the philosopher project. They don't come from the official evaluation grid, which means they are open to interpretation. Other ways of reasoning may be accepted during evaluation if they are well thought out and explained.

Another important aspect of the project is that **the death of a philosopher must be displayed within 10 ms** of the event. This means that when the expected result of a test below says "a philosopher dies at 200 ms", displaying the death message at 203 ms is acceptable, whereas displaying it at 211 ms is not.

The mandatory arguments of philosophers are shown in bold in the following table for clarity's sake. As a reminder, the arguments of the program are as follows:

```
./philo <arg1> <arg2> <arg3> <arg4> [arg5]
        - arg1 = number_of_philosophers
        - arg2 = time_to_die
        - arg3 = time_to_eat
        - arg4 = time_to_sleep
        - arg5 = (optional) number_of_times_each_philosop
```

Test	Expected Result

`./philo`
`./philo` **`1`**
`./philo` **`1 2`**
`./philo` **`1 2 3`**
	Invalid argument / usage message.

`./philo` **`4 500 abc 200`**	Invalid argument.
`./philo` **`4 500 200 2.9`**	Invalid argument.
`./philo` **`4 -500 200 200`**	Invalid argument.
`./philo` **`4 2147483648 200 200`**	Invalid argument.
`./philo` **`0 800 200 200`**	Invalid argument.
`./philo` **`500 100 200 200`**	2 defensible solutions: – Invalid argument. (ex. Max 200 philosophers) – A philosopher dies at 100 ms.
`./philo` **`4 2147483647 200 200`**	No one dies.
`./philo` **`4 200 2147483647 200`**	A philosopher dies at 200 ms.
`./philo` **`4 800 200 2147483647`**	A philosopher dies at 800 ms.
`./philo` **`2 800`**	No one dies.

`200 200`	
`./philo 5 800 200 200`	No one dies.
`./philo 5 0 200 200`	A philosopher dies at 0 ms.
`./philo 5 800 0 200`	No one dies.
`./philo 5 800 200 0`	No one dies.
`./philo 5 800 0 0`	No one dies.
`./philo 5 800 200 200 0`	2 defensible solutions: – Invalid argument. – Simulation stops immediately because everyone ate 0 times.
`./philo 4 410 200 200`	No one dies.
`./philo 1 200 200 200`	Philosopher 1 takes a fork and dies at 200 ms.
`./philo 4 2147483647 0 0`	No one dies.
`./philo 4 200 210 200`	A philosopher dies at 200 ms.
`./philo 2 600 200 800`	A philosopher dies at 600 ms.
`./philo 4 310 200 200`	A philosopher dies at 310 ms.
`./philo 3 400 100 100 3`	No one dies, each philosopher eats at least 3 times.
`./philo 200 800 200 200 9`	No one dies, each philosopher eats at least 9 times.
`./philo 200 410 200 200`	A philosopher dies at 410 ms.

Completed Code for the Philosophers Project

My complete philosophers code is available over here, on GitHub. If you are a 42 student, I encourage you to use the elements above to build your own code in your own way, before glancing at mine.

Please leave a comment if you've found another solution or other ways to test philosophers!

Sources and Further Reading

- Bryant, R., O'Hallaron, D., 2016, *Computer Systems: a Programmer's Perspective*, Chapter 12: Concurrent Programming, p. 1007 – 1076
- Arpaci-Dusseau R., Arpaci-Dusseau, A., 2018, *Operating Systems: Three Easy Pieces*, Part II: Concurrency [OSTEP]
- Wikipedia, *Dining philosophers problem* [Wikipedia]
- Wikipedia, *Concurrent computing* [Wikipedia]
- Wikipedia, *Mutual exclusion* [Wikipedia]
- Stackoverflow, *Why is 1/1/1970 the "epoch time" ?* [Stackoverflow]
- The Linux Programmer Manual:
  - *pthread_create(3)* [man]
  - *pthread_join(3)* [man]
  - *pthread_detach(3)* [man]
  - *pthread_mutex_init/lock/unlock(3)* [man]
  - *gettimeofday(2)* [man]

- Valgrind User Manual, *Helgrind: a thread error detector* [cs.swan.ac.uk]
- Valgrind User Manual, *DRD: a thread error detector* [valgrind.org]
- Spaghetti vector graphic by Vecteezy.

42 cursus C concurrency data race deadlock mutex thread

ABOUT THE AUTHOR

Mia Combeau

Student at 42Paris, digital world explorer. I code to the 42 school norm, which means for loops, switches, ternary operators and all kinds of other things are out of reach… for now!

VIEW ALL POSTS

ADD COMMENT

Comment

Name *

Email *

Website

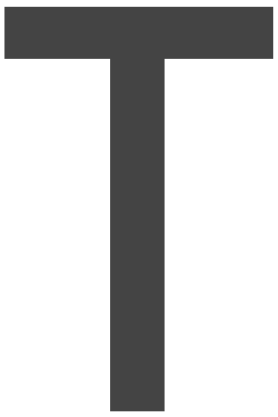
▢ Save my name, email, and website in this browser for the next time I comment.

SUBMIT COMMENT

READ MORE

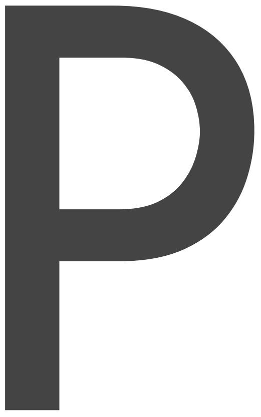
The Difference Between a Terminal, a Console and a Shell

June 26, 2022



Push_Swap: an Efficient Positional Sorting Algorithm

June 24, 2022



Why a Blog is a Great Developer Tool

June 20, 2022

Local, Global and Static Variables in C

June 17, 2022

By Mia Combeau July 18, 2022
To Top
To Top
Push_Swap: an Efficient Positional Sorting Algorithm

MENU

- Legal Notice
- Contact

CATEGORIES

- 42 School Projects
- C Programming
- Computer Science
- Methodology
- Programming Tools