# Philosopher Project Documentation

## Introduction

The Philosopher project aims to teach the basics of threading and mutexes in C programming. This documentation will cover each aspect of the project step-by-step, starting with an introduction to threads and mutexes.

## 1. Understanding Threads and Mutexes

### 1.1 What is a Thread?

A thread is a single sequence stream within a process. Because threads have some of the properties of processes. Threads are often called lightweight processes because they share the same memory space and resources of the main process but execute independently.

**What are the differences between process and thread?**

Threads are not independent from each other unlike processes. As a result, threads shares with other threads their code section, data section and OS resources like open files and signals. But, like processes, a thread has its own program counter (PC), a register set, and a stack space.

**Why Multithreading?**

Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

Threads operate faster than processes due to following reasons:

1) Thread creation is much faster.

2) Context switching between threads is much faster.

3) Threads can be terminated easily.

4) Communication between threads is faster.

**Can we write multithreading programs in C?**

The standard interface in C to manipulate threads is POSIX with its <pthread.h> library. It contains **around sixty functions to create and join threads,** as well as to manage their shared memory.

[**POSIX Threads (or Pthreads)**](http://en.wikipedia.org/wiki/POSIX_Threads)

Is a POSIX standard for threads. Implementation of pthread is available with gcc compiler. You can visit sites below, holding CTRL and click on the site:

[https://www.personal.kent.edu/%7Ermuhamma/OpSystems/Myos/threads.htm](https://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/threads.htm)

<https://www.codequoi.com/en/threads-mutexes-and-concurrent-programming-in-c/>

### 1.2 What is a Mutex?

A mutex (short for mutual exclusion) is a synchronization tool used to manage access to shared resources in a multi-threaded environment. It ensures that only one thread can access a critical section of code at a time, preventing data races and ensuring data integrity.

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.

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

pthread\_mutex\_t mutex;

#### 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);

When we no longer need a mutex, we should destroy it with the following **pthread\_mutex\_destroy function:**

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

## 2. The Philosopher Project

The Philosopher project uses threads to simulate each philosopher and mutexes to manage access to shared resources (i.e., forks). The following sections will detail the implementation and logic of the project step by step.

The Philosopher project is a challenge from the 42 curriculum aimed at simulating the classic synchronization problem known as the Dining Philosophers Problem using threads and mutexes in C. The goal is to create a multi-threaded program where philosophers alternate between eating, sleeping, and thinking, while properly managing access to shared resources (forks) using mutexes.

In this project, each philosopher is represented as a separate thread.

Here, mutexes are used to manage access to the forks, ensuring that no two philosophers can use the same fork simultaneously.

### Dining Philosophers Problem

The problem involves a number of philosophers sitting at a round table, each needing two forks to eat. The challenge is to design a protocol where philosophers can eat without causing a deadlock or starvation.

## 📂 Project Structure

philosopher/

│── Makefile

│── philosopher.h

│── philosopher.c

│── main.c

│── init.c

│── arguments.c

│── cleanup.c

│── monitor.c

│── threads.c

│── utils.c

### **1️⃣ Makefile**

* Compiles all source files and generates the philo executable.
* Supports the following rules:
  + make → compiles the program
  + make clean → removes object files
  + make fclean → removes object files and executable
  + make re → recompiles everything from scratch

### **2️⃣ Header File: philosopher.h**

* Contains structure definitions and function prototypes.
* Defines the philosopher structure and simulation data structure.

### **Detailed Explanation of Data Structures in** philosopher.h

In philosopher.h, we have two key structures:

1.1 t\_philosopher → Represents each philosopher.

typedef struct s\_philosopher

{

int id;

pthread\_t thread;

int eating;

int times\_eaten;

long last\_meal\_time;

size\_t start\_time;

pthread\_mutex\_t \*left\_fork;

pthread\_mutex\_t \*right\_fork;

struct s\_data \*data;

} t\_philosopher;

What does each member represent?

|  |  |  |
| --- | --- | --- |
| **Type** | **Variable** | **Description** |
| int | id | Philosopher’s ID (from 1 to num\_philosophers). |
| pthread\_t | thread | Each philosopher runs in a separate thread. |
| int | eating | 1 if the philosopher is eating, 0 otherwise. |
| int | times\_eaten | Number of times the philosopher has eaten. |
| long | last\_meal\_time | The timestamp of the last meal in milliseconds. |
| size\_t | start\_time | The moment the philosopher was created. |
| pthread\_mutex\_t \* | left\_fork | Mutex for the left fork. |
| pthread\_mutex\_t \* | right\_fork | Mutex for the right fork. |
| struct s\_data \* | data | Pointer to the global t\_data structure. |

✅ How does it work?

* Each philosopher has two mutexes (for the left and right forks).
* Each philosopher tracks their last meal time to check if they have died.
* Each philosopher runs in a separate thread, and synchronization is handled with mutex.

✅ 2. Explanation of t\_data (Global simulation structure)

2.1 t\_data → Stores all the information about the simulation.

typedef struct s\_data

{

int num\_philosophers;

int time\_to\_die;

int time\_to\_eat;

int time\_to\_sleep;

int num\_times\_to\_eat;

t\_philosopher \*philosophers;

pthread\_mutex\_t \*forks;

long start\_time;

int dead\_flag;

pthread\_mutex\_t dead\_lock;

pthread\_mutex\_t meal\_lock;

pthread\_mutex\_t write\_lock;

pthread\_t monitor\_thread;

} t\_data;

What does each member represent?

|  |  |  |
| --- | --- | --- |
| **Type** | **Variable** | **Description** |
| int | num\_philosophers | Number of philosophers. |
| int | time\_to\_die | Maximum time a philosopher can survive without eating. |
| int | time\_to\_eat | Time taken by a philosopher to eat. |
| int | time\_to\_sleep | Time a philosopher spends sleeping. |
| int | num\_times\_to\_eat | If set, the simulation stops after each philosopher eats this many times. |
| t\_philosopher \* | philosophers | Array of t\_philosopher structures, containing all philosophers. |
| pthread\_mutex\_t \* | forks | Array of mutexes, representing forks. |
| long | start\_time | The simulation start time in milliseconds. |
| int | dead\_flag | 1 if a philosopher has died, 0 if the simulation continues. |
| pthread\_mutex\_t | dead\_lock | Protects the dead\_flag variable to avoid data races. |
| pthread\_mutex\_t | meal\_lock | Protects updates to last\_meal\_time. |
| pthread\_mutex\_t | write\_lock | Ensures synchronized terminal output (printf()). |
| pthread\_t | monitor\_thread | A separate thread that monitors philosophers for death. |

✅ How does it work?

* t\_data holds all the information needed for the simulation.
* philosophers is an array → each philosopher is a thread.
* forks is an array of mutexes → each mutex represents a fork.
* dead\_flag is used to stop the simulation if a philosopher dies.
* pthread\_mutex\_t is used to prevent data races in shared variables (forks, dead\_lock, meal\_lock, write\_lock).

### **3️⃣ main.c**

* The main.c file is the entry point of the program. It initializes the philosopher simulation, creates the necessary threads, and manages cleanup at the end.

int start\_threads(t\_data \*data)

{

int i;

if (pthread\_create(&data->monitor\_thread, NULL,

(void \*)monitor, data) != 0)

return (1);

i = 0;

while (i < data->num\_philosophers)

{

if (pthread\_create(&data->philosophers[i].thread, NULL,

(void \*)philosopher\_routine, &data->philosophers[i]) != 0)

return (1);

i++;

}

return (0);

}

🔎 What does it do?

* Creates the monitor thread → This thread continuously checks if any philosopher has died.
* Creates a thread for each philosopher → Each philosopher runs independently in its own thread.

✅ Why is it important?

* The monitor thread ensures that the simulation stops immediately if a philosopher dies.
* Each philosopher thread executes the philosopher\_routine(), allowing philosophers to eat, sleep, and think simultaneously.

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

{

t\_data data;

if (parse\_arguments(argc, argv, &data))

return (1);

if (initialize\_philosophers(&data))

return (1);

if (create\_threads(&data))

return (1);

cleanup(&data);

return (0);

}

**🔎** What does it do?

| **Step** | **Function** | **Description** |
| --- | --- | --- |
| 1. Parse input arguments | parse\_arguments() | Validates and extracts simulation parameters from the command line. |
| 2. Initialize philosophers and mutexes | initialize\_philosophers() | Sets up philosopher structures and forks (mutexes). |
| 3. Create monitor and philosopher threads | create\_threads() | Starts all philosopher threads and the monitor thread. |
| 4. Cleanup and free memory | cleanup() | Cleans up memory, destroys mutexes, and ensures a proper program exit. |

✅ Why is it important?

* It manages the entire execution flow of the program, from initialization to cleanup.
* If any step fails (e.g., invalid arguments, memory allocation failure, thread creation failure), the program exits gracefully.

### **4️⃣ init.c**

The init.c file is responsible for allocating memory, initializing mutexes, and setting up the philosophers' structures before the simulation starts.

✅ 1. allocate\_memory() Function

static int allocate\_memory(t\_data \*data)

{

data->philosophers = malloc(sizeof(t\_philosopher) \* data->num\_philosophers);

data->forks = malloc(sizeof(pthread\_mutex\_t) \* data->num\_philosophers);

if (!data->philosophers || !data->forks)

{

free(data->philosophers);

free(data->forks);

return (1);

}

return (0);

}

**🔎** What does it do?

* Allocates memory for an array of philosophers (t\_philosopher).
* Allocates memory for an array of forks (pthread\_mutex\_t).
* If allocation fails, it frees any allocated memory and returns 1 (failure).

✅ Why is it important?

* Ensures that there is enough memory to store philosopher structures and fork mutexes.
* Prevents memory leaks by freeing memory in case of allocation failure.

**✅ 2. init\_mutexes() Function**

**static int init\_mutexes(t\_data \*data)**

**{**

**int i;**

**i = 0;**

**while (i < data->num\_philosophers)**

**{**

**if (pthread\_mutex\_init(&data->forks[i], NULL) != 0)**

**{**

**return (1);**

**}**

**i++;**

**}**

**if (pthread\_mutex\_init(&data->write\_lock, NULL) != 0)**

**{**

**return (1);**

**}**

**if (pthread\_mutex\_init(&data->dead\_lock, NULL) != 0)**

**{**

**return (1);**

**}**

**if (pthread\_mutex\_init(&data->meal\_lock, NULL) != 0)**

**{**

**return (1);**

**}**

**return (0);**

**}**

**🔎** What does it do?

* Initializes a mutex for each fork → pthread\_mutex\_init(&data->forks[i], NULL).
* Initializes global mutexes for synchronization:
  + write\_lock → Protects console output (printf()).
  + dead\_lock → Protects dead\_flag (prevents data races).
  + meal\_lock → Protects last\_meal\_time (ensures proper meal tracking).
* If any mutex initialization fails, the function returns 1.

✅ Why is it important?

* Ensures thread safety → Prevents simultaneous access to shared resources (forks, console output, death checks).
* Mutexes are essential for synchronization between philosopher threads.

✅ 3. initialize\_each\_philosopher() **Function**

**static void initialize\_each\_philosopher(t\_data \*data)**

**{**

**int i;**

**i = 0;**

**while (i < data->num\_philosophers)**

**{**

**data->philosophers[i].id = i + 1;**

**data->philosophers[i].times\_eaten = 0;**

**data->philosophers[i].eating = 0;**

**data->philosophers[i].start\_time = get\_timestamp();**

**data->philosophers[i].last\_meal\_time = get\_timestamp();**

**data->philosophers[i].left\_fork = &data->forks[i];**

**if (i == 0)**

**data->philosophers[i].right\_fork**

**= &data->forks[data->num\_philosophers - 1];**

**else**

**data->philosophers[i].right\_fork = &data->forks[i - 1];**

**data->philosophers[i].data = data;**

**i++;**

**}**

**}**

**🔎** What does it do?

* Assigns an ID to each philosopher (i + 1).
* Initializes tracking variables:
  + times\_eaten = 0 → Philosopher has not eaten yet.
  + eating = 0 → Philosopher is initially not eating.
  + start\_time and last\_meal\_time → Set using get\_timestamp().
* Assigns forks to each philosopher:
  + Each philosopher's left fork is the fork at their index.
  + Each philosopher's right fork is the previous philosopher's fork (except for the first philosopher, who takes the last fork).

✅ Why is it important?

* Ensures that philosophers can grab the correct forks.
* Tracks each philosopher’s eating and timing information.

✅ 4. initialize\_philosophers() **Function**

**int initialize\_philosophers(t\_data \*data)**

**{**

**if (allocate\_memory(data))**

**{**

**return (1);**

**}**

**if (init\_mutexes(data))**

**{**

**return (1);**

**}**

**data->start\_time = get\_timestamp();**

**data->dead\_flag = 0;**

**initialize\_each\_philosopher(data);**

**return (0);**

**}**

**🔎** What does it do?

| **Step** | **Function** | **Description** |
| --- | --- | --- |
| 1. Allocate memory | allocate\_memory(data) | Allocates memory for philosophers and forks. |
| 2. Initialize mutexes | init\_mutexes(data) | Sets up mutexes for forks and global synchronization. |
| 3. Set simulation start time | data->start\_time = get\_timestamp(); | Marks the start of the simulation. |
| 4. Initialize each philosopher | initialize\_each\_philosopher(data); | Sets IDs, meal tracking, and fork assignments. |

✅ Why is it important?

* Prepares the simulation environment before creating philosopher threads.
* Ensures proper memory allocation and synchronization.

**5️⃣ arguments.c**

The arguments.c file is responsible for **parsing, validating, and converting command-line arguments** before the simulation starts.

**✅ 1. ft\_atoi() Function**

### **int ft\_atoi (const char \*str)**

**{  
 long result;  
 int sign;  
 int i;**

**result = 0;  
 sign = 1;  
 i = 0;  
 while (str[i] == ' ' || str[i] == '\t' || str[i] == '\n'**

**|| str[i] == '\v' || str[i] == '\f' || str[i] == '\r')**

**i++;  
 if (str[i] == '-' || str[i] == '+')  
 {  
 if (str[i] == '-')  
 sign = -1;  
 i++;  
 }  
 while (str[i] >= '0' && str[i] <= '9')**

**{**

**result = result \* 10 + (str[i] – '0');  
 if (result > INT\_MAX || result < INT\_MIN)**

**return (-1);**

**i++;**

**}**

**return ((int)(sign \* result));**

**}**

**🔎** What does it do?

* Converts a string into an integer, similar to atoi().
* Ignores whitespace at the beginning.
* Handles positive and negative signs.
* Validates input:
  + If the number exceeds INT\_MAX or INT\_MIN, it returns -1 (error).
  + If the input contains non-numeric characters, it stops conversion.

✅ Why is it important?

* Ensures that only valid numbers are converted from the command-line arguments.
* Prevents integer overflow.

**✅ 2. check\_arguments() Function**

**static int check\_arguments(int argc, char \*\*argv)  
{  
 int i;  
 if (argc < 5 || argc > 6)  
 {  
 printf("Error: Incorrect number of arguments.\n");  
 return (1);**

**}  
 i = 1;  
 while (i < argc)  
 {  
 if (ft\_atoi(argv[i]) <= 0)  
 {  
 printf("Error: Arguments must be positive numbers.\n");  
 return (1);  
 }**

**i++;}  
 return (0);  
}**

**🔎** What does it do?

* Checks if the correct number of arguments is provided (5 or 6).
* Ensures that all arguments are positive numbers using ft\_atoi().
* If any argument is negative or zero, it prints an error and returns 1 (failure).

✅ Why is it important?

* Prevents invalid input from causing undefined behavior in the simulation.

**✅ 3. validate\_constraints() Function**

**static int validate\_constraints(t\_data \*data, int argc)**

**{**

**if (data->num\_philosophers > 300 || data->num\_philosophers <= 0)**

**{**

**printf("Error: Invalid number of philosophers (1-300).\n");**

**return (1);**

**}**

**if (data->time\_to\_die < 60 || data->time\_to\_eat < 60**

**|| data->time\_to\_sleep < 60)**

**{**

**printf("Error: Timings must be at least 60ms.\n");**

**return (1);**

**}**

**if (argc == 6 && data->num\_times\_to\_eat <= 0)**

**{**

**printf("Error: num\_times\_to\_eat must be > 0.\n");**

**return (1);**

**}**

**return (0);**

**}**

**🔎** What does it do?

* Limits the number of philosophers to a range of 1-300 to ensure performance.
* Ensures timing constraints:
  + time\_to\_die, time\_to\_eat, and time\_to\_sleep must be at least 60ms (prevents unrealistic simulations).
* Validates the optional num\_times\_to\_eat (must be greater than 0 if provided).

✅ Why is it important?

* Ensures realistic and manageable simulations.
* Prevents instabilities caused by too many philosophers.

✅ 4. parse\_arguments() **Function**

### **int parse\_arguments(int argc, char \*\*argv, t\_data \*data)**

**{**

**if (check\_arguments(argc, argv))**

**return (1);**

**data->num\_philosophers = ft\_atoi(argv[1]);**

**data->time\_to\_die = ft\_atoi(argv[2]);**

**data->time\_to\_eat = ft\_atoi(argv[3]);**

**data->time\_to\_sleep = ft\_atoi(argv[4]);**

**if (argc == 6)**

**data->num\_times\_to\_eat = ft\_atoi(argv[5]);**

**else**

**data->num\_times\_to\_eat = -1;**

**return (validate\_constraints(data, argc));**

**}**

**🔎** What does it do?

| **Step** | **Function** | **Description** |
| --- | --- | --- |
| 1. Check if arguments are valid | check\_arguments(argc, argv) | Ensures valid numbers and correct count. |
| 2. Convert arguments to integers | ft\_atoi(argv[i]) | Converts values and stores them in t\_data. |
| 3. Validate constraints | validate\_constraints(data, argc) | Checks philosopher limits and timing rules. |

✅ Why is it important?

* Ensures that all command-line arguments are correct before starting the simulation.
* Prevents the simulation from running with invalid values.

### **6️⃣ philosopher.c**

The philosopher.c file contains the core routines that define the behavior of each philosopher. It includes functions for checking if the simulation is active, locking forks, eating, sleeping, and thinking.

✅ 1. is\_simulation\_active() **Function**

**int is\_simulation\_active(t\_data \*data)**

**{  
 pthread\_mutex\_lock(&data→dead\_lock);  
 if (data→dead\_flag)  
 {  
 pthread\_mutex\_unlock(&data→dead\_lock);  
 return (0);  
 }  
 pthread\_mutex\_unlock(&data→dead\_lock);  
 return (1);  
}**

**🔎** What does it do?

* Locks the dead\_flag mutex to check if a philosopher has died.
* Returns 1 if the simulation is still running, otherwise returns 0.

✅ Why is it important?

* Ensures that all threads stop immediately if the simulation ends (a philosopher dies).
* Prevents race conditions by locking access to dead\_flag.

**✅ 2.** lock\_forks**() Function  
  
int lock\_forks(t\_philosopher \*philo, t\_data \*data)**

**{**

**if (!is\_simulation\_active(data))**

**return (0);**

**printf("Philosopher %d waiting for left fork\n", philo->id);**

**pthread\_mutex\_lock(philo->left\_fork);**

**printf("Philosopher %d took left fork\n", philo->id);**

**print\_status(data, philo->id, "has taken a fork");**

**printf("Philosopher %d waiting for right fork\n", philo->id);**

**pthread\_mutex\_lock(philo->right\_fork);**

**printf("Philosopher %d took right fork\n", philo->id);**

**print\_status(data, philo->id, "has taken a fork");**

**return (1);**

**}**

**🔎** What does it do?

* Checks if the simulation is still active before attempting to take forks.
* Locks the left fork first, then locks the right fork.
* Prints status messages when a fork is taken.

✅ Why is it important?

* Ensures that philosophers do not try to take forks after the simulation has ended.
* Prevents deadlocks by ensuring a structured order for picking up forks.

**✅ 3. eat() Function**

**int eat(t\_philosopher \*philo, t\_data \*data)**

**{**

**pthread\_mutex\_lock(philo→left\_fork);  
 print\_status(data, philo->id, "has taken a fork");  
 if (data->num\_philosophers == 1)  
 {  
 my\_usleep(data→time\_to\_die);  
 pthread\_mutex\_unlock(philo→left\_fork);  
 return (0);  
 }  
 pthread\_mutex\_lock(philo→right\_fork);  
 print\_status(data, philo->id, "has taken a fork");  
 print\_status(data, philo->id, "is eating");  
 pthread\_mutex\_lock(&data→meal\_lock);  
 philo->last\_meal\_time = get\_timestamp();  
 philo→times\_eaten++;  
 pthread\_mutex\_unlock(&data→meal\_lock);  
 my\_usleep(data→time\_to\_eat);  
 pthread\_mutex\_unlock(philo→right\_fork);  
 pthread\_mutex\_unlock(philo->left\_fork);**

**return (1);**

**}**

**🔎** What does it do?

* Locks both forks before eating.
* Handles the case where there is only one philosopher (forces starvation).
* Updates last\_meal\_time and times\_eaten.
* Sleeps for time\_to\_eat before releasing forks.

✅ Why is it important?

* Ensures proper synchronization when eating.
* Tracks when a philosopher last ate, which is crucial for death detection.

**✅ 4. sleep\_and\_think() Function**

**int sleep\_and\_think(t\_philosopher \*philo, t\_data \*data)**

**{**

**if (!is\_simulation\_active(data))**

**return (0);**

**print\_status(data, philo->id, "is sleeping");  
 my\_usleep(data→time\_to\_sleep);  
 if (!is\_simulation\_active(data))  
 return (0);  
 print\_status(data, philo->id, "is thinking");  
 return (1);  
}**

**🔎** What does it do?

* Checks if the simulation is still active before sleeping or thinking.
* Sleeps for time\_to\_sleep.
* Thinks after sleeping before attempting to eat again.

✅ Why is it important?

* Ensures that the philosopher follows the correct sequence: eat -> sleep -> think -> repeat.
* Prevents unnecessary computation by stopping execution if the simulation ends.

### **7️⃣ threads.c**

**✅ 1. create\_monitor\_thread() Function**

**int create\_monitor\_thread(t\_data \*data, pthread\_t \*observer)**

**{  
 if (pthread\_create(observer, NULL, &monitor, data) != 0)  
 return (1);  
 return (0);  
}**

**🔎** What does it do?

* Creates the monitor thread using pthread\_create().
* If thread creation fails, it returns 1.

✅ Why is it important?

* The monitor thread constantly checks if a philosopher has died.
* If a philosopher dies, the simulation stops immediately.

**✅ 2. create\_philosopher\_threads() Function**

**int create\_philosopher\_threads(t\_data \*data)  
{  
int i;**

**i = 0;  
while (i < data→num\_philosophers)  
 {  
 if (pthread\_create(&data->philosophers[i].thread, NULL,  
 philosopher\_routine, &data->philosophers[i]) != 0)  
 return (1);  
 i++;  
 }  
 return (0);  
}**

**🔎** What does it do?

* Creates a thread for each philosopher using pthread\_create().
* If any thread fails to be created, it returns 1.

✅ Why is it important?

* Each philosopher must run in a separate thread to eat, sleep, and think independently.

**✅ 3. join\_threads() Function**

**int join\_threads(t\_data \*data, pthread\_t observer)**

**{  
 int i;  
 if (pthread\_join(observer, NULL) != 0)**

**return (1);  
 i = 0;  
 while (i < data→num\_philosophers)  
 {  
 if (pthread\_join(data->philosophers[i].thread, NULL) != 0)  
 return (1);  
 i++;  
 }  
 return (0);  
}**

**🔎** What does it do?

* Waits for the monitor thread to finish (pthread\_join(observer, NULL)).
* Waits for each philosopher thread to finish execution.

✅ Why is it important?

* Prevents unfinished execution → ensures all threads terminate properly before the program exits.
* Without pthread\_join(), the main thread could terminate while other threads are still running.

**✅ 4. create\_threads() Function**

**int create\_threads(t\_data \*data)**

**{  
 pthread\_t observer;  
   
 if (create\_monitor\_thread(data, &observer))**

**return (1);  
 if (create\_philosopher\_threads(data))  
 return (1);  
 if (join\_threads(data, observer))  
 return (1);  
 return (0);  
}**

**🔎** What does it do?

| **Step** | **Function** | **Description** |
| --- | --- | --- |
| 1. Create the monitor thread | create\_monitor\_thread() | Checks for philosopher deaths. |
| 2. Create philosopher threads | create\_philosopher\_threads() | Starts philosopher execution. |
| 3. Join all threads | join\_threads() | Ensures proper cleanup. |

✅ Why is it important?

* Manages all thread creation and execution in one place.
* Ensures all philosophers start before the simulation begins.

**5. philosopher\_routine() Function**

### **void \*philosopher\_routine(void \*pointer)**

**{  
 t\_philosopher \*philo;**

**philo = (t\_philosopher \*)pointer;  
 if (philo->id % 2 == 0)  
 my\_usleep(1);  
 while (!should\_stop(philo))  
 {  
 if (!eat(philo, philo→data))  
 break ;  
 if (!sleep\_and\_think(philo, philo→data))  
 break ;  
 }  
 return (NULL);  
}**

**🔎** What does it do?

* Delays execution for even-numbered philosophers (my\_usleep(1)) to prevent immediate deadlocks.
* Runs in a loop until the simulation stops:
  + Eat
  + Sleep
  + Think
* If a philosopher cannot eat or sleep, the loop breaks.

✅ Why is it important?

* This function defines how each philosopher behaves.
* The loop stops execution when the simulation ends.

### **8️⃣ utils.c**

The utils.c file contains helper functions used throughout the simulation. These functions handle printing status messages, measuring time, sleeping accurately, and checking whether the simulation should continue or stop.

✅ 1. print\_status() **Function**

**void print\_status(t\_data \*data, int id, const char \*status)**

**{  
 long timestamp;**

**if (!data || !status)  
 return ;  
 pthread\_mutex\_lock(&data→write\_lock);  
 if (data->dead\_flag == 0)  
 {**

**timestamp = get\_timestamp() - data→start\_time;  
 printf("%ld %d %s\n", timestamp, id, status);  
 fflush(stdout);  
 }  
 pthread\_mutex\_unlock(&data→write\_lock);  
}**

🔎 What does it do?

* Locks the write\_lock mutex to ensure safe console output.
* Checks if the simulation is still running before printing.
* Calculates the timestamp (relative to the simulation start time).
* Prints the philosopher’s status message in the format:

[timestamp] [philosopher\_id] [status]  
 Flushes stdout to ensure immediate printing.

✅ Why is it important?  
 Prevents race conditions when multiple philosophers print at the same time.   
 Ensures synchronized and structured output.

**✅ 2. get\_timestamp() Function  
   
long get\_timestamp(void)  
{  
 struct timeval tv;  
   
 gettimeofday(&tv, NULL);  
 return ((tv.tv\_sec \* 1000) + (tv.tv\_usec / 1000));**

**}**

🔎 What does it do?

* Uses gettimeofday() to get the current time.
* Converts seconds + microseconds into milliseconds.

✅ Why is it important?  
Provides accurate time measurement for checking last meal times and status updates.

**✅ 3. my\_usleep() Function**

**void my\_usleep(long time)  
{  
 struct timeval start;  
 struct timeval current;  
 long elapsed;**

**gettimeofday(&start, NULL);  
 while (1)  
 {  
 gettimeofday(&current, NULL);  
 elapsed = (current.tv\_sec - start.tv\_sec) \* 1000  
 + (current.tv\_usec - start.tv\_usec) / 1000;  
 if (elapsed >= time)  
 break ;  
 usleep(100);  
 }  
}**

**🔎** What does it do?

* Records the current time using gettimeofday().
* Uses a loop to wait until the required time (in milliseconds) has passed.
* Calls usleep(100) in small intervals to reduce CPU usage.

✅ Why is it important?

* Provides precise timing for sleeping and eating without busy-waiting.
* Prevents CPU overload, unlike a direct usleep(time \* 1000).

**✅ 4. should\_continue() Function**

**int should\_continue(t\_philosopher \*philo, t\_data \*data)  
{  
 (void)philo;  
 pthread\_mutex\_lock(&data→dead\_lock);  
 if (data→dead\_flag)  
 {  
 pthread\_mutex\_unlock(&data→dead\_lock);  
 return (0);  
 }  
 pthread\_mutex\_unlock(&data→dead\_lock);  
 return (1);  
}**

**🔎** What does it do?

* Locks the dead\_lock mutex to safely read dead\_flag.
* Returns 1 if the simulation is still running, otherwise returns 0.

✅ Why is it important?

* Prevents philosophers from continuing execution after a philosopher has died.

**✅ 5. should\_stop() Function**

**int should\_stop(t\_philosopher \*philo)  
{  
 pthread\_mutex\_lock(&philo→data→dead\_lock);  
 if (philo→data→dead\_flag)  
 {  
 pthread\_mutex\_unlock(&philo→data→dead\_lock);  
 return (1);  
 }  
 pthread\_mutex\_unlock(&philo→data→dead\_lock);  
 return (0);  
}**

🔎 What does it do?

* Checks if the simulation should stop by reading dead\_flag safely.
* Locks and unlocks the dead\_lock mutex to avoid race conditions.

✅ Why is it important?

* Used inside the philosopher’s loop to determine when execution should stop.

### **9️⃣** monitor.c

The monitor.c file is responsible for tracking philosopher deaths and determining when the simulation should stop. It runs in a separate thread and constantly checks if any philosopher has exceeded their time\_to\_die or if all philosophers have eaten the required number of meals.

**✅ 1. monitor() Function**

## void \*monitor(void \*arg) {

## t\_data \*data;

## data = (t\_data \*)arg; while (1) {

## if (check\_simulation\_end(data)) { pthread\_mutex\_lock(&data→dead\_lock); data->dead\_flag = 1; pthread\_mutex\_unlock(&data→dead\_lock); break ; } usleep(500); } pthread\_exit(NULL); }

🔎 What does it do?

* Runs in a separate thread.
* Calls check\_simulation\_end() to determine if:
  + A philosopher has died.
  + All philosophers have eaten enough.
* If the simulation should stop, it sets dead\_flag = 1.

✅ Why is it important?

* Ensures that the simulation stops immediately when necessary.
* Prevents race conditions by locking dead\_lock before modifying dead\_flag.

**✅ 2. check\_if\_all\_ate() Function**

**int check\_if\_all\_ate(t\_data \*data)**

**{**

**int i;**

**int finished\_eating;**

**i = 0;**

**finished\_eating = 0;  
 if (data->num\_times\_to\_eat == -1)**

**return (0);**

**while (i < data->num\_philosophers)**

**{**

**pthread\_mutex\_lock(&data→meal\_lock);  
 if (data->philosophers[i].times\_eaten >= data->num\_times\_to\_eat) finished\_eating++;  
 pthread\_mutex\_unlock(&data→meal\_lock);  
 i++;  
 }  
 if (finished\_eating == data→num\_philosophers)  
 return (1);  
 return (0);  
}**

**🔎** What does it do?

* Checks if all philosophers have eaten num\_times\_to\_eat meals.
* Uses meal\_lock to safely access times\_eaten.
* Returns 1 if all philosophers have eaten enough, otherwise returns 0.

✅ Why is it important?

* If all philosophers have eaten enough, the simulation should end.
* Prevents incorrect termination due to race conditions.

**✅ 3. check\_simulation\_end() Function**

**int check\_simulation\_end(t\_data \*data)**

**{**

**int i;**

**i = 0;  
 while (i < data→num\_philosophers)  
 {  
 if (check\_philosopher\_dead(data, I))  
 {  
 pthread\_mutex\_lock(&data→dead\_lock);  
 data->dead\_flag = 1;  
 pthread\_mutex\_unlock(&data→dead\_lock);  
 return (1);  
 }  
 i++;  
 }  
 if (data->num\_times\_to\_eat > 0 && check\_if\_all\_ate(data))  
 {**

**pthread\_mutex\_lock(&data→dead\_lock);  
 data->dead\_flag = 1;  
 pthread\_mutex\_unlock(&data→dead\_lock);  
 return (1);  
 }  
 return (0);  
}**

**🔎** What does it do?

* Loops through all philosophers:
  + Calls check\_philosopher\_dead() to see if any philosopher has died.
* If any philosopher has died, it sets dead\_flag = 1.
* If all philosophers have eaten enough, it also stops the simulation.

✅ Why is it important?

* Ensures that the simulation stops immediately when a philosopher dies.
* Prevents unnecessary execution when the required number of meals is reached.

**✅ 4. check\_philosopher\_dead() Function**

**int check\_philosopher\_dead(t\_data \*data, int I)  
{  
long current\_time;**

**pthread\_mutex\_lock(&data→meal\_lock);  
current\_time = get\_timestamp();  
if ((current\_time – data→philosophers[i].last\_meal\_time)  
 >= data→time\_to\_die)  
 {  
 pthread\_mutex\_unlock(&data→meal\_lock);  
 pthread\_mutex\_lock(&data→dead\_lock);  
 if (data->dead\_flag == 0)  
 {   
 data->dead\_flag = 1;  
 printf("%ld %d died\n", current\_time – data→start\_time,  
 data→philosophers[i].id);  
 }  
 pthread\_mutex\_unlock(&data→dead\_lock);  
 return (1);  
 }  
 pthread\_mutex\_unlock(&data→meal\_lock);  
 return (0);  
}**

**🔎** What does it do?

* Checks if a philosopher has exceeded time\_to\_die.
* Uses meal\_lock to safely access last\_meal\_time.
* If the philosopher has died:
  + Sets dead\_flag = 1.
  + Prints the death message:

[timestamp] [philosopher\_id] died

* Ensures thread safety by using pthread\_mutex\_lock().

✅ Why is it important?

* Prevents race conditions when checking if a philosopher is dead.
* Ensures dead philosophers stop execution immediately.

### **🔟** cleanup.c

The cleanup.c file is responsible for deallocating memory and destroying mutexes after the philosopher simulation ends, ensuring proper cleanup of system resources.

✅ 1. cleanup() Function

void cleanup(t\_data \*data)

{  
 int i;

i = 0;  
 while (i < data→num\_philosophers)  
 {  
 pthread\_mutex\_destroy(&data→forks[i]);  
 i++;  
 }  
 pthread\_mutex\_destroy(&data→meal\_lock);  
 pthread\_mutex\_destroy(&data→dead\_lock);  
 pthread\_mutex\_destroy(&data→write\_lock);  
 free(data→forks);  
 free(data→philosophers);  
}

**🔎** What does it do?

| **Step** | **Action** | **Description** |
| --- | --- | --- |
| 1. Destroy fork mutexes | pthread\_mutex\_destroy(&data->forks[i]) | Ensures that all forks (mutexes) are properly released. |
| 2. Destroy global mutexes | pthread\_mutex\_destroy(&data->meal\_lock)  pthread\_mutex\_destroy(&data->dead\_lock)  pthread\_mutex\_destroy(&data->write\_lock) | Prevents memory leaks and undefined behavior. |
| 3. Free allocated memory | free(data->forks)  free(data->philosophers) | Releases dynamically allocated structures. |

✅ Why is it important?

* Prevents memory leaks by properly freeing allocated structures.
* Ensures all mutexes are destroyed before program termination.
* Essential for proper resource management in a multithreaded program.

**✅ 2. ft\_strlen() Function**

**int ft\_strlen(char \*str)**

**{**

**int i;**

**if (str == NULL)  
 return (0);  
 i = 0;  
 while (str[i] != '\0')  
 i++;  
 return (i);  
}**

🔎 What does it do?

* Returns the length of a string.
* Handles NULL input safely by returning 0.

✅ Why is it important?

* Used internally when handling string-based operations (though minimal in this project).

## 🚀 **How to Compile and Run**

### **1. Compile the program**

make

### **2. Run the program**

./philo number\_of\_philosophers time\_to\_die time\_to\_eat time\_to\_sleep [number\_of\_times\_each\_philosopher\_must\_eat]

Example:

./philo 5 800 200 200

* 5 philosophers
* 800 ms time\_to\_die
* 200 ms time\_to\_eat
* 200 ms time\_to\_sleep

## 🛠️ **Testing the Program**

* Run multiple test cases to check for deadlocks, correct execution, and memory leaks.

./philo 1 800 200 200  
 (The single philosopher takes a fork but cannot eat because there is no second fork.

* The philosopher waits until time\_to\_die expires and dies.
* The program exits cleanly after displaying).
* ./philo 5 800 200 200 (Five philosophers should start eating, sleeping, and thinking.)
* ./philo 4 310 100 100   
   (time\_to\_die (310ms) is only slightly longer than time\_to\_eat + time\_to\_sleep (100ms + 100ms = 200ms), so philosophers must eat fast to survive.)  
  ./philo 4 410 200 200 5 (The four philosophers eat exactly 5 times each.)
* valgrind --leak-check=full --show-leak-kinds=all ./philo 5 800 200 200  
   The simulation should run without memory leaks.