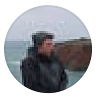
# Philosophers 42 Guide— “The Dining Philosophers Problem”

The famous “Dining Philosophers Problem” explained/walkthrough — 42 cursus project


Dean Ruina · Follow
11 min read · Aug 21, 2023

Listen Share

Here is my philosophers repository for refrence. I’ll explain here the general idea of my solution. If you’re interested in seeing my code you can check it here:

**GitHub - DeRuina/philosophers: Philosophers 42 Explained/Walkthrough. The Dining Philosophers...**

Philosophers 42 Explained/Walkthrough. The Dining Philosophers Problem guide - GitHub - DeRuina/philosophers...

github.com

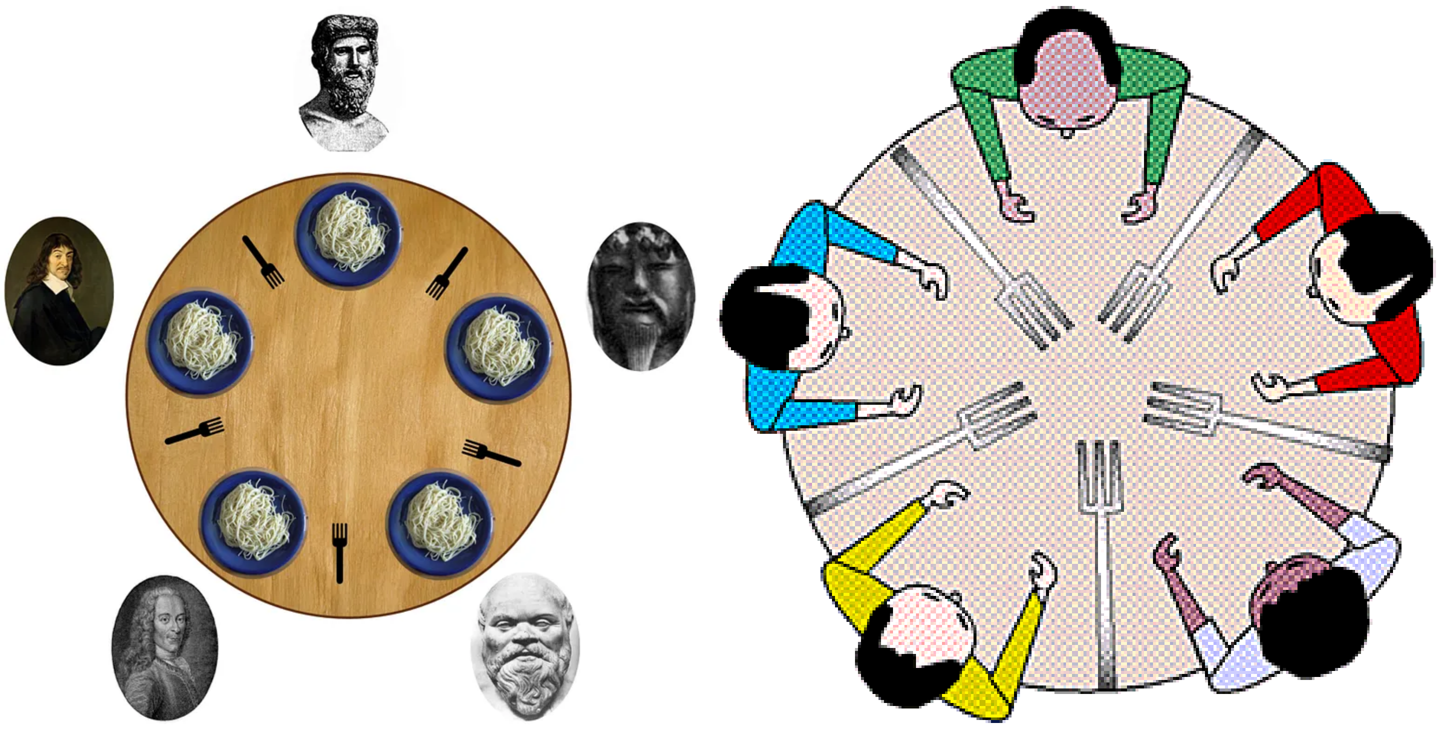
## The Dining Philosophers Problem

The dining philosophers problem is a famous problem in computer science used to illustrate common issues in concurrent programming. The problem was originally formulated in 1965 by Edsger Dijkstra, and is stated as follows:

> *X amount of philosophers sit at a round table with bowls of food.*
> *Forks are placed in front of each philosopher.*
> *There are as many forks as philosophers.*
> *All day the philosophers take turns eating, sleeping, and thinking.*
> *A philosopher must have two forks in order to eat, and each fork*
> *may only be used by one philosopher at a time. At any time a*
> *philosopher can pick up or set down a fork,*
> *but cannot start eating until picking up both forks.*

> *The philosophers alternatively eat, sleep, or think.*
> *While they are eating, they are not thinking nor sleeping,*
> *while thinking, they are not eating nor sleeping,*
> *and, of course, while sleeping, they are not eating nor thinking.*

Let me start by explaining the general idea. First of all, we have to imagine a round table, X num of philosophers sitting around it and each of them brings a fork and places it in front of them. At this point we know that a philosopher can do three things: eat, sleep, or think, but in order to eat he has to pick two forks (the one in front of him and another one to his right or to his left, in my solution he picks the one to his right, both work — different implementation). Let’s use a picture to have a more concrete idea of what we are talking about:



The number of forks is equal to the number of philosophers

Let’s say there are 5 philosophers sitting at the table. Philosopher 1 wants to eat, so he picks the fork in front of him and the one to his right (the one in front of philosopher 5), at this point, we notice that philosopher 2 can’t eat nor does philospher 5, since philosopher 1 picked the fork in front of him and in front of philisopher 5. this might seem a little obvious but keep in mind this situation because the main problem of this project is how to organize the eating action of the philosophers. Probably the first solution that came to your mind is to simply make the odd and even philos eat separately, well we are not going to do that, it’s too hard coded and we would lose the meaning of the project, philos have to organize by themselves. We will be using threads and implement a multithreading solution. I’m attaching the project subject as well so you could understand completely what is needed.

In order to understand the solution you’ll needs to understand the concept of threads first, here are some good videos I recommend you watch:

- General introduction to threads
- Introduction to threads with code examples
- Short introduction to threads (pthreads) **
- * Code Vault covers all the knowledge you need for this project/problem in this playlist

## Data Races (Race Conditions) — What Are They?

Data races are a common problem in multithreaded programming. Data races occur when multiple tasks or threads access a shared resource without sufficient protections, leading to undefined or unpredictable behavior.

- two or more threads concurrently accessing a location of memory
- one of them is a write
- one of them is unsynchronized

In simpler words a race condition can happen when 2 or more threads are trying to access and modify the same variable at the same time, it can lead to an error in the final value of the variable, it doesn’t mean it will for sure happen though. For an example let’s think of a function that deposits the amount you insert to your bank account, If we use multithreading and use 2 threads and want to deposit 300 using the first thread and 200 using the second you will think our bank account will have a total of 500, but that’s not particularly the case, let’s see it in code:

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

// the initial balance is 0
```

```
int balance = 0;

// write the new balance (after as simulated 1/4 second delay)
void write_balance(int new_balance)
{
  usleep(250000);
  balance = new_balance;
}

// returns the balance (after a simulated 1/4 seond delay)
int read_balance()
{
  usleep(250000);
  return balance;
}

// carry out a deposit
void* deposit(void *amount)
{
  // retrieve the bank balance
  int account_balance = read_balance();

  // make the update locally
  account_balance += *((int *) amount);

  // write the new bank balance
  write_balance(account_balance);

  return NULL;
}

int main()
{
  // output the balance before the deposits
  int before = read_balance();
  printf("Before: %d\n", before);

  // we'll create two threads to conduct a deposit using the deposit function
  pthread_t thread1;
  pthread_t thread2;

  // the deposit amounts... the correct total afterwards should be 500
  int deposit1 = 300;
  int deposit2 = 200;

  // create threads to run the deposit function with these deposit amounts
  pthread_create(&thread1, NULL, deposit, (void*) &deposit1);
  pthread_create(&thread2, NULL, deposit, (void*) &deposit2);

  // join the threads
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
```

```
    // output the balance after the deposits
    int after = read_balance();
    printf("After: %d\n", after);

    return 0;
  }
```

You would think if you run this code that the balance will be 500 but the output we get is actually 200, now, why is that? Here is a visualization of the above program’s execution:

```
  Thread #1              Thread #2               Bank Balance

  Read Balance  <---------------------------------- 0
  balance = 0
                         Read Balance  <------------- 0
                         balance = 0


  Deposit +300
  balance = 300
                         Deposit +200
                         balance = 200
```

When the two deposit functions run at the same time, they both read the balance which is 0, they both deposit the amount inserted and change the balance variable locally, but when they write it to the variable itself, they overwrite each other. thread #1 writes 300 first but thread #2 writes as well and changes the variable to 200. How can we solve this? easily we just need to attach a lock, let me introduce you to mutex.

- What are Race Conditions? — Great explanation down to the Assembly level
- Race Conditions Explained With An Example

## Mutex

Now that we know what is a race condition let’s see what is the solution. Imagine a lock that protects a block of code and it can be only executed by the lock owner until he unlocks the lock. Taking the previous example we can avoid the overwrite by adding a lock in the deposit function. if thread #1 reaches the lock thread #2 will just have to wait until thread #1 is done executing the code and reaches the unlock, only then thread #2 will enter and execute himself.

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

// the initial balance is 0
int balance = 0;

// write the new balance (after as simulated 1/4 second delay)
void write_balance(int new_balance)
{
  usleep(250000);
  balance = new_balance;
}

// returns the balance (after a simulated 1/4 seond delay)
int read_balance()
{
  usleep(250000);
  return balance;
}

// carry out a deposit
void* deposit(void *amount)
{
  // lock the mutex
  pthread_mutex_lock(&mutex);

  // retrieve the bank balance
  int account_balance = read_balance();

  // make the update locally
  account_balance += *((int *) amount);

  // write the new bank balance
  write_balance(account_balance);

  // unlock to make the critical section available to other threads
  pthread_mutex_unlock(&mutex);

  return NULL;
}
```

```c
int main()
{
  // mutex variable
  pthread_mutex_t mutex;

  // output the balance before the deposits
  int before = read_balance();
  printf("Before: %d\n", before);

  // we'll create two threads to conduct a deposit using the deposit function
  pthread_t thread1;
  pthread_t thread2;

  // initialize the mutex
  pthread_mutex_init(&mutex, NULL);

  // the deposit amounts... the correct total afterwards should be 500
  int deposit1 = 300;
  int deposit2 = 200;

  // create threads to run the deposit function with these deposit amounts
  pthread_create(&thread1, NULL, deposit, (void*) &deposit1);
  pthread_create(&thread2, NULL, deposit, (void*) &deposit2);

  // join the threads
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);

   // destroy the mutex
  pthread_mutex_destroy(&mutex);

  // output the balance after the deposits
  int after = read_balance();
  printf("After: %d\n", after);

  return 0;
}
```

You have surely noticed that we initialize and destroy the mutex, and you have to do that every time you want to use a mutex (destroy it after you finished using it) otherwise it won’t work.

Here is another visualization with the locks:

```
Thread #1              Thread #2              Bank Balance

                        **  LOCK  **

  WAIT @ LOCK           Read Balance  <------------- 0
      |                 balance = 0
      |
      |                 Deposit +200
      |                 balance  = 200
      |
      |                 Write Balance  ------------> 200
      |                 balance = 200
      |
  LOCK FREE             ** UNLOCK **

  **  LOCK  **

  Read Balance  <------------------------------------ 200
  balance = 0

  Deposit +300
  balance = 500

  Write Balance  -----------------------------------> 500
  balance = 500

  ** UNLOCK **
```

- What is a mutex in C? (pthread_mutex)
- Mutex Introduction (pthreads)

## Step By Step Guide/Walkthrough

### First Step: Checking Valid Input

The first thing we need to do before we even start initializing anything is to check the program input. The program will receive 4 or 5 arguments so the first thing should be to throw an error if we receive more or less. Let’s analyze the input we will receive: 5 800 200 200 7

- 5 — The number of philosophers
- 800 — The time a philosopher will die if he doesn’t eat
- 200 — The time it takes a philosopher to eat

- 200 — The time it takes a philosopher to sleep
- 7 — Number of times all the philosophers need to eat before terminating the program **

** optional argument

Basically, all we need to do is to check that the input contains only numbers, they should all be bigger than 0 except the number of meals each philo needs to eat (edge case). In the evaluation form, it says we should not test with more than 200 philos so you can set the limit not to be more than 200.

**Second Step: Structures**

In order for you to understand the way I approached and solved this project I’ll share with you the structures I made. Because each philosopher needs to be a thread and all the data needs to pass to the routine functions, structures are the best option. I created 2 structures, The program structure which holds all of the philosophers (in an array), 3 mutex, and one dead_flag, and the philo structure where we have all of the general data, 3 mutex pointers that point to the mutex in the program structure, 2 mutex pointers for the forks, and on dead pointer which points to the dead flag in the program structure.

```
typedef struct s_philo
{
        pthread_t                       thread;
        int                                     id;
        int                                     eating;
        int                                     meals_eaten;
        size_t                          last_meal;
        size_t                          time_to_die;
        size_t                          time_to_eat;
        size_t                          time_to_sleep;
        size_t                          start_time;
        int                                     num_of_philos;
        int                                     num_times_to_eat;
        int                                     *dead;
        pthread_mutex_t *r_fork;
        pthread_mutex_t *l_fork;
        pthread_mutex_t *write_lock;
        pthread_mutex_t *dead_lock;
        pthread_mutex_t *meal_lock;
}                                                       t_philo;

typedef struct s_program
```

```
{
        int                             dead_flag;
        pthread_mutex_t  dead_lock;
        pthread_mutex_t  meal_lock;
        pthread_mutex_t  write_lock;
        t_philo                 *philos;
}                               t_program;
```

**Third Step: Initialization**

Because we know the maximum amount of philosophers our program can be tested with (200) and I wanted to avoid dealing with leaks, freeing, and allocating, and mainly because I wanted the performance to be faster I decided to keep all the memory on the stack and not on the heap by initializing a philo structure array, a mutex array for the forks and the program structure all in the main. From there I initialize the program variables, initialize all the mutexes for the mutex fork array, and lastly the philosophers — input variables and point the pointers to all the mutexes and the dead_flag.

**Fourth Step: Thread Creation, Philo Routine, And Monitor**

Now we need to create the threads and join them. We will create as many threads as philosophers we have, each philo needs to be a thread and we will create an extra thread (I called it observer) which will monitor everything. Each philo thread will run the philo routine function and the observer will run the monitor function.

## Philo Routine()

The routine will be the function executed over and over by the philos, Basically ,I created a loop that will break as soon as the dead flag is 1, in other words as soon as a philo is dead. Remember:

> *The philosophers alternatively eat, sleep, or think. While they are eating, they are not thinking nor sleeping, while thinking, they are not eating nor sleeping, and, of course, while sleeping, they are not eating nor thinking.*

So in our loop, they will eat, sleep and think. Let’s start with the easiest one when they think we just need to print a message “X is thinking” (X is the philo number), When they sleep we need to make them sleep the length of the input inserted by the user using our ft_usleep (described in the bottom of this page) and then print the message “X is sleeping”. Now to the eating part, We will lock the right fork first using pthread_mutex_lock and print the message, and do the same with the left fork. Then he will eat using ft_usleep again and only then he will drop the forks by unlocking the locks, before that we change some variables that give our monitor indications but that's the general idea.

**Monitor()**

This thread will be running and monitoring the whole program, it has 2 checks in it that run infinitely until a philo dies or they all ate the number of meals they need to (last input argument). Basically, we will check that the time a philo needs to die didn't surpass the last meal he had and that he is not concurrently eating. If he indeed died we change the dead flag to 1 and that will break the loop in all of the threads. The other check is to see if all the philos finished eating the amount of meals they need to, and if they did we will again change the dead flag to one and break the threads loop.

### Fifth Step: Destroying All The Mutexes

The last step is to Destroy all the mutexes you initialized, otherwise, they won't work. In this step, we will free all the data we allocated if we chose to allocate it(we didn't).

## Utils Functions — Improtant

### Sleep Function Delay

Different machines perform the sleep function with different accuracy. You can check your machine by running the script in my repo.

**GitHub - DeRuina/philosophers: Philosophers 42 Explained/Walkthrough. The Dining Philosophers...**

Philosophers 42 Explained/Walkthrough. The Dining Philosophers Problem guide - GitHub - DeRuina/philosophers...

github.com

This can help make sure other stuff running on the computer doesn't interfere with Philosopher's timings. For this reason, as well we created the ft_usleep function. Run the script by downloading it and:

```
python3 delay_o_meter.py
```

or by running my Makefile with the command:

```
make delay
```

**ft_usleep**

```
// Improved version of sleep function
int     ft_usleep(size_t milliseconds)
{
        size_t  start;

        start = get_current_time();
        while ((get_current_time() - start) < milliseconds)
                usleep(500);
        return (0);
}
```

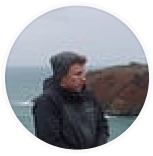
**get_current_time**

```
// Gets the current time in milliseconds

size_t  get_current_time(void)
{
        struct timeval  time;

        if (gettimeofday(&time, NULL) == -1)
                write(2, "gettimeofday() error\n", 22);
        return (time.tv_sec * 1000 + time.tv_usec / 1000);
}
```

That’s the general idea and explanation, my repo is linked in the top if you want to see how I did it by code.

Dining Philosophers Multithreading 42 Coding School 42 Network

C Programming



Follow

## Written by Dean Ruina

4 Followers

## More from Dean Ruina