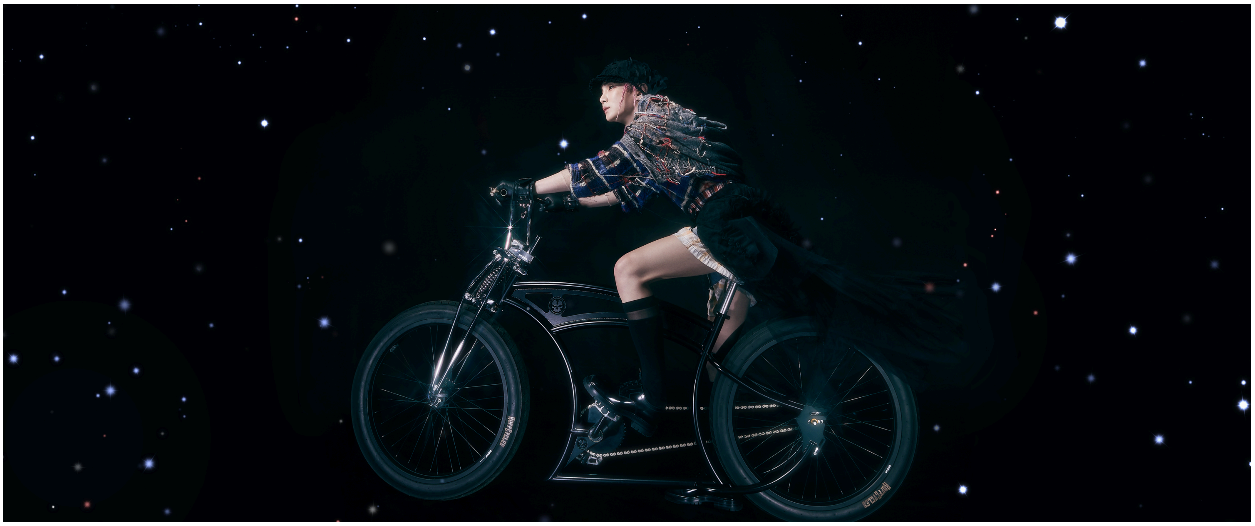

The stupid philosophers....................

- **wtf r threads? an explanation for stupid people...**
  - **anal ogy**
    - If a **process** = your apartment, a **thread** would be like *(trigger warning)* your *flatmate.....*
    - All flatmates **(threads)** in the same apartment **(process)** share the same utilities **(memory)** - fridge, bathroom, TV, even your *leftovers* in the fridge if you don't put a note on it **(lock a mutex).**
    - They like to do their own things *independently*: One is *cooking*, another one is *showering*, one is *watching TV* in the living room. But sometimes they *clash*:
      - You're trying to wash your underwear, but the washing machine is really cheap and shitty *(as they tend to be in shared houses)*, so it doesn't automatically *lock the door*.
      - Your flatmate on the other hand, is extremely stupid *(as they tend to be in shared houses)* and opens the machine in the middle of the

wash - *causing the entire room to flood!* This is called a **race condition**. *And this is where we need a **mutex**....*

▼ and what r mu..... mutexes?

- A **mutex** = a toilet stall lock.... you lock it while you use the stuff thats inside... and unlock when you're done.
- Only one **thread** can **lock** and access the **protected data** at a time - *we don't want to poop with our friends right next to us trying to pee in the same toilet...*
- Prevents **race conditions** → two buddies **(threads)** trying to sit & shit in the toilet **(edit data)** at the same time and breaking it.

🚽 Mutexes in Philosophers

🔧 Part	🔐 Why It Needs a Mutex
`fork->mutex`	Ensures only one philosopher can use a fork at a time.
`write_mutex`	Avoids messy output by letting only one message print at a time.
`philo->mutex`	Protects each philo's meal info *(shared with monitor)*
`table->mutex`	Protects shared flags like `end_simulation`.

▼ a cringe SHINee analogy....

- SHINee members = **philosophers (that run threads)**
- Microphones = **forks (with mutexes)**
- They must grab **two mics** to rehearse 🎶
- They all live in the same dorm (shared memory), so coordination is fast
- Manager = **Monitor (another thread)**
- They all have a meal log **(philo→last_meal_time & philo→meals_eaten)**
- The manager **(monitor)** checks those diaries **(data)** to make sure they don't miss lunch & hit their protein goals.

- To avoid smudging **(race conditions)**, he only holds the diary to read while no one is writing **(mutex)**

- aka… we must put a safety lock **(mutex)** to make sure *no one can open the washing machine* **(access the shared data)** while its being used **(edited)**. Or, if your flatmates are trustworthy enough, a warning sign…

▼ so?

- Philosophers **(*structs* *that are used to run* *threads*)** try to **lock** two forks **(mutexes)**.
- If both forks are free, they **eat** for a set amount of time **(argv[3])**; otherwise they wait **(think).**
  - **→** ***time_to_think*** = (*time_to_die* - *time_to_eat* - *time_to_sleep*) / 2.
    - ./philo 5 800 200 200 → ***time_to_think*** = (800 - 200 - 200) / 2 = **200.**
- After each meal, they need to sleep for a set amount of time **(argv[4])**.
- The **monitor thread** watches their `last_meal_time` using **mutexes** to avoid reading while it's being changed.

▼ thread functions

▼ *pthread_create*

▼ *int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);*

Starts a new thread. duh. what else did you think.

▼ how to use

1. **int** - returns an error code if ≠ 0
2. ***thread** - a pointer to where to store the *thread ID*
3. ***attr** (thread attributes) - a pointer to pthread_attr_t object or NULL for default (enough for philosophers)

4. ***(*start_routine)(void *)** - the function that the thread will run. it must take a void* argument and return a void *.
5. **void *arg** - the argument passed to the thread function ^

- ▼ *pthread_join*
  - ▼ *int pthread_join(pthread_t thread, void **retval);*

    waits for a thread to finish whatever its doing. its just like, to make sure its cleaned up before the the main program exits i guess.

    - ▼ how2use vv
      1. **int** - error code if ≠ 0
      2. **thread** - the thread you want to wait for (e.g. philo[i].thread_id)
      3. ****retval** (return value) - a pointer to capture the value that the function passed to **pthread_create** returns (optional)
         - **pthread_join(philo[i].thread, NULL);** → “i dont give a FUCK what this stupid B-word wants to say to me LOL”
         - ▼ **pthread_join(philo[i].thread, &result);** → “ily honey tell me everything in detail…”
           - **Status code or result**
             - Example: `return (void *)EXIT_SUCCESS;` or `return (void *)&error_code;`
           - **Result data**
             - Example: a pointer to a struct with processed data or computation results.
           - **Resource handles or objects**
             - Example: a pointer to a buffer filled by the thread.
           - **Signal completion**
             - Threads can return a simple flag or pointer to indicate they finished successfully.

- **Example:** `return (NULL);`
- **Error info or messages**
  - A thread can return a pointer to an error message string or detailed error info.

- pthread_detach
  - int pthread_detach(pthread_t thread);

    Makes a thread run independently. Detaches the thread, so it cleans up its own resources when done. Use when you don't need to wait for a thread's result and want it to free its resources automatically.

- void pthread_exit
  - void pthread_exit(void *retval);

    terminates the current thread (we dont need to call it if our thread has default attributes, our thread already does it! so i wont elaborate for now)

- pthread_mutex_init
  - int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);

    Initializes a mutex to protect shared data from simultaneous access of course.

- pthread_mutex_lock
  - int pthread_mutex_lock(pthread_mutex_t *mutex);

    locks the mutex, Protects a critical section by allowing only one thread at a time.

- pthread_mutex_unlock
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);

    Unlocks the mutex. Allows others to access the protected resource.

- pthread_mutex_destroy

- int pthread_mutex_destroy(pthread_mutex_t *mutex);

  Cleans up the mutex, releases resources used by the mutex after you’re done.

the code

library

```
#ifndef PHILO_H
# define PHILO_H

// external libraries         what we use them for:
# include <stdio.h>          // printf
# include <stdlib.h>         // malloc, free
# include <unistd.h>         // usleep
# include <sys/time.h>       // gettimeofday & struct timeval
# include <pthread.h>        // pthread_create/join, pthread_mutex_init/lock/u
# include <stdbool.h>        // true/false
# include <limits.h>         // INT_MAX
# include <errno.h>          // EINVAL, ENOMEM, EAGAIN, EPERM, ESRCH, EI

/* declaring variable types */
typedef   pthread_mutex_t t_mutex;
typedef struct s_table t_table; // bc we want philo in table & table in philo!

/* declaring possible operations to pass into pthread/mutex functions */
typedef enum e_opcode
{
    LOCK,
    UNLOCK,
    INIT,
    DESTROY,
    CREATE,
    JOIN,
    DETACH
```

```
}   t_opcode;

/* declaring time units to pass into gettime function */
typedef   enum   e_time
{
    SECOND,
    MILLISECOND,
    MICROSECOND
}   t_time;

/* declaring possible actions/ states of philosophers to pass into write func
typedef enum e_status
{
    EAT,
    SLEEP,
    THINK,
    RIGHT_FORK,
    LEFT_FORK,
    DIE,
    FULL
}   t_status;

/* each fork contains: */
typedef   struct s_fork
{
    t_mutex   mutex;            // can only be accessed by 1 philo at a time
    int    id;                  // each fork is shared by 2 philos
}   t_fork;

/* each philosopher contains: */
typedef struct s_philo
{
    int    id;                  // table→philos[i + 1]
    long meals_eaten;           // increases after eating
    bool full;                  // if (meals_eaten == table→max_meals)
    long last_meal_time;        // time passed from last meal
```

```
    t_fork   *left_fork;         // each philo needs 2 forks to eat
    t_fork   *right_fork;
    pthread_t    thread_id;        // each philo runs a thread
    t_table *table;             // pointer to table data
    t_mutex   mutex;              // only one thread edits data at the same time
}   t_philo;

/* the table contains : */
typedef struct    s_table
{
    long philo_nbr;            // total number of philos (argv[1])
    long time_to_die;          // how long each philo can go without eating
    long time_to_eat;          // how long it takes to eat
    long time_to_sleep;        // how long it takes to sleep
    long meals_nbr;            // argv[5] | FLAG if -1
    long start_simulation;     // start time of the simulation
    bool end_simulation;       // true when a philo dies or all are full
    bool all_threads_ready;    // true when all philo threads initialized & creat
    long threads_running;      // how many threads running
    pthread_t    monitor;          // checks each philos meals_eaten and last_m
    t_mutex   mutex;              // multiple threads have access so we need to l
    t_mutex   write_mutex;        // used to synchronize writing output
    t_fork   *forks;             // pointer to all forks
    t_philo *philos;            // pointer to all philos
}   t_table;

/* wrapper.c */
void *handle_malloc(size_t bytes);
void handle_mutex(t_mutex *mutex, t_opcode opcode);
void handle_threads(pthread_t *thread, void *(*foo)(void *), void *data, t_o
/* error.c */
void error(const char *msg);
void thread_error(int status, t_opcode opcode);
void mutex_error(int status, t_opcode opcode);
/* parse.c */
void    parse_input(t_table *table, char **argv);
```

```
/* init.c */
void data_init(t_table *table);
/* dinner.c */
void start_dinner(t_table *table);
void think(t_philo *philo, bool pre_sim);
/* utils.c */
void clean(t_table *table);
long	gettime(t_time time);
void	precise_usleep(long usec, t_table *table);
/* set_get.c */
void set_bool(t_mutex *mutex, bool *dst, bool value);
void set_long(t_mutex *mutex, long *dst, long value);
long get_long(t_mutex *mutex, long *src);
bool get_bool(t_mutex *mutex, bool *src);
bool end_simulation(t_table *table);
/* sync.c */
void wait_threads(t_table *table);
bool	all_threads_running(t_mutex *mutex, long *threads, long philo_nbr);
void	increase_long(t_mutex *mutex, long *value);
void	desync_philos(t_philo *philo);
/* write.c */
void	write_status(t_status status, t_philo *philo);
/* monitor.c */
void	*monitor_dinner(void *data);
bool sim_fin(t_table *table);

#endif
```

▼ MAIN

```
#include "philo.h"

/* USAGE: ./philo 5(philo_nbr) 800(time_to_die) 200(time_to_eat) 200(time
int	main(int params, char	**argv)
{
	t_table table;
```

```
    if (params == 5 || params == 6) // correct input
    {
        parse_input(&table, argv);    // we check if its valid
        data_init(&table);            // we assign data to table, philo & fork
        philo_dinner(&table);         // here's where the magic happens
        clean(&table);                // we destroy mutexes & free memory
    }
    else
    {
        error("Error: invalid number of arguments.\nUSAGE: ./philo number_o
    }
    return (0);
} // the rest is self explanatory, right?
```

▼ Checking the input

$(parse_input)$

```
#include "philo.h"

/* functions to check for spaces & digits */
static   bool is_space(char c)
{
    return ((c >= 9 && c <= 13) || c == 32);
}

static   bool is_digit(char c)
{
    return (c >= '0' && c <= '9');
}

/* function to check if the input is a positive value */
static const char    *valid_input(const char *str)
{
    int    len;
```

```
    const char  *num;

    len = 0;
    while (is_space(*str))
        ++str;
    if (*str == '+')
        ++str;
    else if (*str == '-')
        error("Input Invalid: values must be positive.");
    if (!is_digit(*str))
        error("Input Invalid: value must be a digit");
    num = str;
    while(is_digit(*str++))
        ++len;
    if (len > 10)
        error("Input Invalid: value must not be larger than INT_MAX");
    return (num);
}

/* atol to protect against overflow */
static long  ft_atol(const char *str)
{
    long num;

    num = 0;
    str = valid_input(str);
    while (is_digit(*str))
        num = (num * 10) + (*str++ - '0');
    if (num > INT_MAX)
        error("Input Invalid: value must not be larger than INT_MAX");
    return (num);
}

/* function to check user input & write it into table struct */
void parse_input(t_table *table, char **argv)
{
```

```
    table->philo_nbr = ft_atol(argv[1]);
    table->time_to_die = ft_atol(argv[2]) * 1000; // time in microseconds (
    table->time_to_eat = ft_atol(argv[3]) * 1000;
    table->time_to_sleep = ft_atol(argv[4]) *1000;
    if (argv[5])
        table->max_meals = ft_atol(argv[5]);
    else
        table->max_meals = -1;
    if (table->time_to_die < 60000 || table->time_to_sleep < 60000 || tabl
        printf("Warning: 🌙 Heads up! Tiny time values (<60ms) can make
}
```

Why do we convert to microseconds (* 1e3/ * 1000) ? - Your whole philosopher program checks time using microseconds (because that's how `gettimeofday()` or `usleep()` works). So you want **everything to be in the same unit**, like everyone dancing to the same beat 🕺

▼ Preparing table & guests $(data_init)$

```
#include "philo.h"

/* function that fills the fork structs */
static void   assign_forks(t_philo *philo, t_fork *forks, int pos)
{
    int    nbr;

    nbr = philo->table->philo_nbr;
    philo->right_fork = &forks[(pos + 1) % nbr]; //wraps back to 1st fork if
    philo->left_fork = &forks[pos];
    if (philo->id % 2 == 0) // if ID is even...
    {
        philo->right_fork = &forks[pos]; //opposite assignment
        philo->left_fork = &forks[(pos + 1) % nbr];
    } // to prevent deadlock-> make philos grap 'other' fork first
}
```

```
/* function that fills the philo struct with data */
static void philo_init(t_table *table)
{
    int     i;
    t_philo *philo; // to make code more readable

    i = 0;
    while (i < table→philo_nbr)
    {
        philo = table→philos + i; //assigning ptr to ptr of 'i'th philo of table
        philo→id = i + 1; //bc we dont want the first philo to be called '0'
        philo→full = false; // same as table→philos[i].full etc etc
        philo→meals_eaten = 0;
        philo→table = table; // assign table ptr so we can access table fror
        handle_mutex(&philo→mutex, INIT); //bc handle_mutex expects po
        assign_forks(philo, table→forks, i);
        i++;
    }
}

/* function that fills the table struct with data before starting the simulat
void data_init(t_table *table)
{
    int     i;

    i = 0;
    table→end_simulation = false;
    table→all_threads_ready = false;
    table→threads_running = 0;
    table→philos = handle_malloc(sizeof(t_philo) * table→philo_nbr);
    handle_mutex(&table→mutex, INIT);
    handle_mutex(&table→write_mutex, INIT);
    table→forks = handle_malloc(sizeof(t_fork) * table→philo_nbr);
    while (i < table→philo_nbr)
    {
```

```
        handle_mutex(&table→forks[i].mutex, INIT);
        table→forks[i].id = i;
        i++;
    } // create all forks before assigning them to philos
    philo_init(table);
}
```

why do we ask every 2nd philosopher to grap the 'other fork' first? If everyone grabs the same side first, no one will start eating, because everyone will have one fork in their hand. That would be a deadlock.

different ways to pass pointers:

handle_mutex(&table→forks[i].mutex, INIT) is the same as

t_fork *fork;

fork = table→fork + i;

handle_mutex(&fork→mutex, INIT);

▼ The actual dinner $(philo_dinner)$

```
void philo_dinner(t_table *table)
{
    int    i;
    i = 0
// 1. if no meals are to be eaten, skip everything
    if (table→meals_nbr == 0)
        return ;
// 2. edge case: only 1 philo
    else if (table→philo_nbr == 1)
    {
        handle_threads(&table→philos[0].thread_id, lonely_philo, &table→p
    }
// 3. create philo_nbr of threads
    else
    {
```

```
        while (i < table->philo_nbr)
        {
            handle_threads(&table->philos[i].thread_id, dinner_sim, &table->
            i++;
        }
    }
// 4. create monitor thread to check if philos are dead or full
    handle_threads(&table->monitor, monitor_dinner, table, CREATE);
    // store simulation start time
    table->start_simulation = gettime(MILLISECOND);
    // signal that threads may begin
    set_bool(&table->mutex, &table->all_threads_ready, true);
    i = 0;
    while (i < table->philo_nbr) // wait for philo threads to finish
        handle_threads(&table->philos[i++].thread_id, NULL, NULL, JOIN);
    set_bool(&table->mutex, &table->end_simulation, true); // all joined? s
    handle_threads(&table->monitor, NULL, NULL, JOIN); // join monitor t
}
```

▼ or **philo_dinner: step by step** or

1. Early return if no meals needed (eeaaaasy)

2. Special case for 1 philo

    ▼ **lonely_philo** br

    run_dinner calls handle_threads, handle_threads calls pthread_create: `pthread_create(&table->philos[0].thread_id, NULL, lonely_philo, &table->philos[0]);`

    telling the thread of philo[0] to run:

```
static void   *lonely_philo(void *arg) //void *arg is &table->ph
{
    t_philo *philo;

// 1. cast void* arg to convert it back to usable philo struct
    philo = (t_philo *)arg;
```

```
// 2. wait until all_threads_ready = true
    wait_threads(philo->table);
// 3. setting last_meal_time to now, simulation starts with a fu
    set_long(&philo->mutex, &philo->last_meal_time, gettime(
// 4. increase number of threads currently running for monit
    increase_long(&philo->table->mutex, &philo->table->threa
// 5. simulate philo to pick up the only fork :,(
    write_status(RIGHT_FORK, philo);
// 6. wait until he dies :(
    while (!sim_fin(philo->table))
        usleep(philo->table->time_to_die);
// 7. When the simulation has ended, announce the philo's d
    write_status(DIE, philo);
// 8. End the thread function by returning NULL (standard fo
    return(NULL);
}
```

3. Create thread for each philo

▼ **dinner_sim** cu

run_dinner calls *handle_threads*, *handle_threads* calls *pthread_create*:

`pthread_create(&table->philos[i].thread_id, NULL, dinner_sim, &table->philos[i]);` telling philo[i] to run:

```
static void   *dinner_sim(void *arg) // arg is &table->philo[i]
{
    t_philo *philo;
// 1. converting void * back to a philo pointer for the thread t
    philo = (t_philo *)arg;
// 2. wait until all_threads_ready == true
    wait_threads(philo->table);
// 3. set time of last meal to now, full bellies for a fair start
    set_long(&philo->mutex, &philo->last_meal_time, gettime(
// 4. let monitor know thread is running
```

```
    increase_long(&philo->table->mutex, &philo->table->threa
// 5. desync one before starting so that no deadloks happen
    desync_philos(philo);
// 6. run simulation loop until its over
    while (!sim_fin(philo->table))
    {
        if (philo->full)
            break ; // a. stop if full
        eat(philo); // b. take forks, set time, update meals, write
        write_status(SLEEP, philo); // c. write sleep status
        precise_usleep(philo->table->time_to_sleep, philo->tabl
        think(philo, false); // e. write think status and think if the
    }
// 7. exit with return NULL (standard for pthreads)
    return (NULL);
}
```

▼ **eat**

```
static void   eat(t_philo *philo)
{
    handle_mutex(&philo->right_fork->mutex, LOCK); // lo
    write_status(RIGHT_FORK, philo); // announce philo p
    handle_mutex(&philo->left_fork->mutex, LOCK); // lock
    write_status(LEFT_FORK, philo); // announce philo pic
    // set last_meal_time to current time
    set_long(&philo->mutex, &philo->last_meal_time, getti
    philo->meals_eaten++; // increase meal counter
    write_status(EAT, philo); // announce philo eating
    precise_usleep(philo->table->time_to_eat, philo->table
    if (philo->table->max_meals > 0 && philo->meals_eate
    {
        set_bool(&philo->mutex, &philo->full, true);
        write_status(FULL,philo); // announce that philo is f
    }
```

```
    handle_mutex(&philo->right_fork->mutex, UNLOCK); /
    handle_mutex(&philo->left_fork->mutex, UNLOCK); //
}
```

- **think**

```
void think(t_philo *philo, bool pre_sim)
{
    long t_eat;
    long t_sleep;
    long t_think;

// we only announce the philo thinking if the simulation s
    if (!pre_sim)
        write_status(THINK, philo);
// if philo_nbr is even, we dont need thinking delays
    if (philo->table->philo_nbr % 2 == 0)
        return ;
    t_eat = philo->table->time_to_eat;
    t_sleep = philo->table->time_to_sleep;
// calculate how long the philosopher should think
    t_think = t_eat * 2 - t_sleep;
    if (t_think < 0)
        t_think = 0; // negative thinking time = no thinking ti
// think for 42% of thinking time, leave the rest for other
    precise_usleep(t_think * 0.42, philo->table);
}
```

- What about the other 58% of the thinking time?

  The rest of the thinking time happens **naturally** as the philosopher progresses through the rest of the loop (or simulation). Here's how:

  - The philosopher **sleeps** for 42% of the thinking time here, giving some initial stagger or pause.

- The **remaining time** is spent in other actions like trying to pick forks, eating, or in the main loop itself, which also takes some time.

So, the 42% sleep here is a way to **partially delay** the philosopher, just enough to desynchronize them without making them waste *all* their thinking time just waiting.

▼ Why don't philosophers think in all cases?

When there's an even number of philosophers, they already take turns naturally. Half will pick up forks first, the other half will wait — no need for extra delay - Desynchronization happens automatically due to the way threads are created.

▼ **helper functions**

wait_threads → simple spinlock function to check bool status every 500 microseconds.

```
void	wait_threads(t_table *table)
{
	while (!get_bool(&table->mutex, &table->all_threads_ready))
		usleep(500);
}
```

increase_long → locks mutex before changing a given value, unlocks when done.

```
void	increase_long(t_mutex *mutex, long *value)
{
	handle_mutex(mutex, LOCK);
	(*value)++;
	handle_mutex(mutex, UNLOCK);
}
```

desync_philos → gives certain philosophers waiting time or thinking breaks based on philo_nbr

```
void    desync_philos(t_philo *philo)
{
// case 1: even number of philos
        if (philo->table->philo_nbr % 2 == 0)
        {
        // philo's id is even? wait until odd id philos grab the
                if (philo->id % 2 == 0)
                        precise_usleep(30000, philo->table);
        }
// case 2: odd number of philos
        else
        {
            // philo's id is odd? sleep while even id philos gra
                if (philo->id % 2)
                        think(philo, true);
        }
}
```

`desync_philos` helps **stagger the start** of philosophers a little bit, so they **don't all try to grab forks at the exact same time** (which could cause chaos and deadlocks).

▼ but wait.. what's **precise_usleep**?

`usleep()` is **not precise** enough for sensitive timing — it often **oversleeps** because:

- the OS may delay the wakeup,
- it's not real-time accurate (especially for small delays like 1–10 ms).

In the **Philosophers project**, we need **tight control** over timing — like waking up **as soon as** a philosopher has died or finished eating. That's why we implement a **more accurate sleep** manually.

```
void precise_usleep(long time_to_wait, t_table *table
{
    long  target_time;
    long time_left;

    // Save the target time in microseconds
  target_time = gettime(MICROSECOND) + time_to_w

    // Loop until the total desired time has passed
    while (gettime(MICROSECOND) < target_time)
    {
        // Stop early if simulation is finished
        if (sim_fin(table))
            break ;

        // How much more time do we still need to wait
        time_left = target_time - (gettime(MICROSECON

        // If there's more than 1000 microseconds (1 mi
        if (time_left > 1000)
            usleep(time_left / 2); // sleep for half of the re
        else
        {
            // If under 1 ms left, wait by checking the time
            while (gettime(MICROSECOND) < target_time
                ; // do nothing, just wait
        }
    }
}
```

4. Create monitor thread

▼ **monitor_dinner** dr **(to-do)**

philo_dinner calls handle_threads, handle_threads calls pthread_create:

`pthread_create(&table->monitor, NULL, monitor_dinner, table);`

telling the monitor thread to run:

```
void *monitor_dinner(void *arg) // arg is table
{
	t_table *table;
	int i;
// 1. Convert back void * into struct ptr using a cast
	table = (t_table *)arg;
// 2. Wait until all threads are fully running
	while (!all_threads_running(&table->mutex, &table->thre
		usleep(500);
// 3. Keep monitoring until the simulation ends
	while (!sim_fin(table))
	{
		i = 0;
	// a. loop through each philo while the simulation is run
		while (i < table->philo_nbr && !sim_fin(table))
		{
		 // b. if philo died, announce death & signal end of
			if (philo_died(&table->philos[i]))
			{
				write_status(DIE, &table->philos[i]);
				set_bool(&table->mutex, &table->end_sin
				break ;
			}
			i++;
		}
		usleep(100); // small sleep to reduce CPU stress
	}
 // 4. End the thread function by returning NULL
	return (NULL);
}
```

▼ **philo_died**

```
static bool philo_died(t_philo *philo)
{
	long	since_last_meal;
	long	ms_to_die;

// If philo is full, he's alive
	if (get_bool(&philo→mutex, &philo→full))
		return (false);
// Calculate how much time passed since last meal
	since_last_meal = gettime(MILLISECOND) - get_lon
// Convert to milliseconds
	ms_to_die = philo→table→time_to_die / 1000;
// if time_to_die has passed without meals, philo died :(
	if (since_last_meal > ms_to_die)
		return (true);
// else he's alive :)
	return (false);
}
```

▼ Why not just use microseconds everywhere?

- Using **milliseconds** reduces the size of numbers and can be easier to work with for human-scale timing like seconds and milliseconds.
- Milliseconds are often precise enough for timing philosophers' actions.
- It keeps calculations simpler and more readable.

5. Start the simulation time and signal that threads may begin
   &table→all_threads_ready = true
6. Wait for all philo threads to finish
   pthread_join(table→philos[i].thread_id, NULL)
7. Mark the simulation as done
   &table→end_simulation = true

8. Join the monitor thread last

- ▼ **join_thread explained in detail** vi he
  - ▼ 🧵 What does it mean to "join" a thread?

    Think of each philosopher like a SHINee member doing a solo stage 🎤✨.

    When the show ends, the manager (your start_dinner function) waits behind the scenes and **doesn't leave** until each member finishes their performance.

    Joining a thread means:

    "Dear thread… I'll wait for you patiently until you're completely done before I move on."

    In C, `pthread_join(thread_id, NULL)` means:

    > "Dear thread… I'll wait for you patiently until you're completely done before I move on."

    - Pause the main thread.
    - Wait for the thread with that `thread_id` to finish.
    - Don't go to the next line until it's really done.
  - ▼ 🌼 What happens when we do this:

    ```
    while (i < table→philo_nbr)
       handle_threads(&table→philos[i++].thread_id, NULL, NULL, JOIN);
    ```

    This means:

    1. You're going through each philosopher's thread, one by one.
    2. You tell the main thread: "Hey, please wait here until **this** philosopher is finished with their simulation (they're full or died)."

3. Once a thread is done, you move to the next philosopher.
4. Only after **all philosopher threads** are done, the loop finishes.

▼ 🧠 Why do we do it?

- 💡 If you don't join threads, your program could **end too early** while philosopher threads are still running.
- 💥 It might crash or cause weird behavior like memory leaks or lost output.
- ✅ Joining makes sure everything is **clean**, **synchronized**, and **finished** before you move on.

▼ 🐾 Visual example

Let's say we have 3 philosophers, they each eat in their own thread.

```
pthread_join(philo1_thread);
pthread_join(philo2_thread);
pthread_join(philo3_thread);
```

It's like you're gently checking:

- 🧍 "Philo 1, are you done?"
- (waits)
- ✅ "Okay, thank you. Philo 2, are you done?"
- (waits)
- ✅ "Great job. Philo 3, are you done?"
- (waits)
- ✅ "Everyone is done. Now we can clean up and say goodbye."

▼ 🌈 And after joining?

After we’ve joined all philosopher threads, we can safely:

1. Signal the simulation is done (table→all_threads_ready = true);
2. Join the monitor thread (pthread_join(*thread, NULL));
3. Free memory, destroy mutexes (pthread_mutex_destroy(mutex));

   ▼ clean(&table);

```
void	clean (t_table *table)
{
	t_philo *philo;
	int	i;

	i = 0;
	while (i < table→philo_nbr)
	{
		philo = table→philos + i;
		handle_mutex(&philo→mutex, DESTRO
		i++
	}
	handle_mutex(&table→write_mutex, DESTR
	handle_mutex(&table→mutex, DESTROY);
	free(table→forks);
	free(table→philos);
}
```

4. Exit the program peacefully 💭💤 (return (0));

▼ cleaning up

$(clean)$

no further explanation needed.

```
void clean (t_table *table)
{
 int  i;

  i = 0;
// destroy each philosopher's mutex
  while (i < table→philo_nbr)
  {
    handle_mutex(&table→philos[i].mutex, DESTROY);
    handle_mutex(&table→forks[i].mutex, DESTROY);
    i++;
  }
// destroy the mutex used for writing output
  handle_mutex(&table→write_mutex, DESTROY);
// destroy the table mutex
  handle_mutex(&table→mutex, DESTROY);
// free all allocated memory
  free(table→forks);
  free(table→philos);
}
```