**StageRace – thread theory**

i did yesterday a stage race exercise in preparation for the philosophers project in42 and im gonna need for you to clarify some parts. attacehd you will find the code. My first question is for you to explain to me the pthread\_create function and its arguments, in particular the part of the function pointer it takes as an argument

Pensó durante 4 segundos

La función **pthread\_create** se usa para lanzar un hilo nuevo. Su firma es:

int pthread\_create(pthread\_t \*thread,

const pthread\_attr\_t \*attr,

void \*(\*start\_routine)(void \*),

void \*arg);

* \**pthread\_t thread*: dirección donde se guarda el identificador del hilo creado.
* \**const pthread\_attr\_t attr*: atributos del hilo (pasar NULL para valores por defecto).
* **void \*(\*start\_routine)(void \*)**: puntero a la función que ejecutará el hilo. Debe tener esta forma:
  + Recibe un parámetro de tipo void \* (por ejemplo, una estructura con datos para el hilo).
  + Devuelve void \* (puede usarse para devolver información al pthread\_join).
* \**void arg*: argumento que se le pasa a start\_routine (en tu código, &runners\_array[i]).

En tu ejemplo:

thread\_creation\_return = pthread\_create(

&threads[i], // guarda el ID del hilo

NULL, // atributos por defecto

runner\_routine, // puntero a la función void \*runner\_routine(void \*)

&runners\_array[i] // argumento para runner\_routine

);

Aquí runner\_routine debe coincidir con void \*func(void \*). Cuando el hilo arranca, recibe &runners\_array[i] como arg, y su retorno (NULL) se ignora en el pthread\_join.

**Tú dijiste:**

always speak in English, even if i switch to Spanish inadvertently. Please explain succintly what a thread is in this context and also the syntax of the many \* of the function argument

Pensó durante un par de segundos

A **thread** here is a separate flow of execution within your program—like a “lightweight” process that runs concurrently but shares the same memory space.

The pthread\_create argument

void \*(\*start\_routine)(void \*)

breaks down as follows:

* void \* (before the parentheses) is the function’s return type (a pointer to anything).
* (\*start\_routine) means “start\_routine is a pointer to a function.”
* (void \*) after it means that this function takes one argument of type void \* (a pointer to anything).

So overall, start\_routine is a pointer to a function that takes a void \* argument and returns a void \*.

**Tú dijiste:**

so when u say pthread\_create is a function to launch a new thread, its a bit incomplete, dont u think? the actual purpose of the function is to execute a function within a thread, not just launch a thread. It's purpose is double-folded, at least, no?

You’re right—pthread\_create does two things at once:

1. **Allocates and starts a new thread** in the process’s address space.
2. **Immediately begins executing** the given start\_routine function inside that thread.

In other words, it doesn’t just “spawn” a thread idly; it both creates the thread and hands it control so it can run your function (void \*start\_routine(void \*)) right away.

**Tú dijiste:**

and this makes sense in this exercise because all the runners run at the same time, right? how does that relate to the philosophers project? they dont just do one action, like in this program? they eat, think and sleep, runners just run

Yes—threads aren’t limited to a single action. In the race exercise, each thread’s start routine simply has one “run” call and then exits. In the philosophers project, each philosopher thread’s start routine is a loop that repeatedly thinks, picks up forks (eats), then sleeps, then repeats—so pthread\_create still spawns a thread and begins running its function, but that function contains multiple actions in a loop rather than one.

**Tú dijiste:**

why do we multiply the sleep\_ms by 1000

Because usleep expects its argument in microseconds and sleep\_ms is in milliseconds, you multiply by 1000 to convert milliseconds → microseconds.

**Tú dijiste:**

since pthread\_create is executed in thread\_launcher in a loop for each runner, it's not really executed concurrently, right? i mean, there's certain advantage for runner[0], no?

Pensó durante un par de segundos

The calls to pthread\_create do happen one after the other, so thread 0 is created (and begins running) slightly before thread 1, thread 2, etc. However, as soon as each pthread\_create returns, that new thread is free to run in parallel with any threads already started. In practice:

* Thread 0 might get a few microseconds of “head start,” but once thread 1 is created, both can execute concurrently.
* The OS scheduler then decides how to share CPU time among all threads, so any tiny advantage for runner 0 quickly disappears.

In short, creation is serial, but execution is concurrent—any real advantage for runner 0 is negligible.

**Tú dijiste:**

execution is concurrent? doesnt execution for runner[0] start right away the moment its thread is launched, which means before the other runners threads?

Pensó durante un par de segundos

Yes—when you call pthread\_create(&threads[0], …), the OS makes thread 0 runnable immediately, so it **could** begin executing before you even call pthread\_create for thread 1. In practice, though, a few things happen:

1. **Thread startup latency:** The new thread is simply marked runnable; it doesn’t necessarily start executing user code the instant you return from pthread\_create. The scheduler might delay it while you’re still in the loop setting up thread 1.
2. **Scheduler interleaving:** As soon as you create thread 1 (or any other), the OS scheduler can preempt thread 0 and switch to thread 1 (or vice versa) almost instantly. That tiny “head start” for runner 0 is usually just microseconds and vanishes as soon as multiple threads are active.

So yes, runner 0 is eligible to run first, but real execution quickly becomes concurrent under the scheduler. Any practical advantage for runner 0 is negligible.

**Tú dijiste:**

explain the first argument of pthread create

The first argument to pthread\_create is a pointer to a pthread\_t variable:

pthread\_t thread\_id;

pthread\_create(&thread\_id, …);

* You pass &thread\_id so the function can fill in that pthread\_t value.
* After the call succeeds, thread\_id uniquely identifies the new thread (for joins, cancellations, etc.).

**Tú dijiste:**

is it usual that the first argument is a single thread and not a thread of a thread array?

The first argument always needs to be the address of a single pthread\_t where the new thread’s ID will be stored.

* If you’re only creating one thread, you can do:

pthread\_t tid;

pthread\_create(&tid, …);

* If you’re creating several threads in a loop, you usually declare an array:

pthread\_t threads[N];

for (int i = 0; i < N; i++) {

pthread\_create(&threads[i], …);

}

So it’s not that you can’t use a standalone variable—it’s just common to use an array element (&threads[i]) when you need multiple threads.

**Tú dijiste:**

i just mean in general, pthread\_create is used to create several threads, not just one, right?

Yes. You can call pthread\_create any number of times—once if you only need a single thread, or in a loop to spawn many. Each call’s first argument must be the address of a distinct pthread\_t variable (or an element of a pthread\_t array) so you can track that thread’s ID.

**Tú dijiste:**

you are not answering my question, what im asking is if usually pthread\_create is used to create several threads, not just one

Yes. In most multithreaded programs, you call pthread\_create multiple times (often in a loop) to spawn several threads. You can use it just once if you only need a single thread, but its common use is to create many.

**Tú dijiste:**

why is pthread\_join used afterwards? wahts its purpose?

pthread\_join blocks the calling thread until the specified thread terminates. Its purposes are:

1. **Synchronization**: It ensures you don’t proceed (e.g., exit main) before the child thread finishes.
2. **Resource cleanup**: Joining reclaims system resources used by the thread.
3. **Return value**: It can retrieve the value returned by the thread’s start routine (if you pass a non-NULL second argument).

**Tú dijiste:**

child thread?

“Child thread” simply refers to any thread that was created by another thread (the “parent”). When you call pthread\_create from the main thread (or any thread), the new thread is often called its child. It’s just terminology—both threads share the same process memory.

**Tú dijiste:**

so all threads in this program are child threads, no?

Not exactly—only the threads you create with pthread\_create are “child” threads. The original thread (often called the main thread) isn’t a child. In your program, the main thread spawns each runner thread; those runner threads are its children.

**Tú dijiste:**

i thought the main was a process, not a thread

Every C program starts as a single thread (the “main thread”) running inside one process. In other words:

* **Process**: An OS‐level container with its own memory space.
* **Thread**: A path of execution within that process.

When your program begins, the process has one thread (the main thread). Calling pthread\_create spawns additional threads (child threads). So “main” is a thread inside the process, not a separate process.

**Tú dijiste:**

you said "pthread\_join blocks the calling thread until the specified thread terminates. Its purposes are: Synchronization: It ensures you don’t proceed (e.g., exit main) before the child thread finishes. Resource cleanup: Joining reclaims system resources used by the thread. Return value: It can retrieve the value returned by the thread’s start routine (if you pass a non-NULL second argument)." but in this case, the program runs several threads, one per runner, so your grammar is incorrect, no? also, explain "Return value: It can retrieve the value returned by the thread’s start routine (if you pass a non-NULL second argument)."

You’re right—when you have multiple runners, it should say that pthread\_join waits for **each** child thread to finish (not “the child thread”).

Regarding the return value: each thread’s start routine has the signature

void \*start\_routine(void \*arg);

When that function returns (or calls pthread\_exit(ptr)), it hands back a void \* result. If you call

void \*retval;

pthread\_join(threads[i], &retval);

then after pthread\_join returns, retval holds whatever your start\_routine returned. If you pass NULL instead of &retval, you ignore that return value.

**Tú dijiste:**

do i have that retval ni my code?

No. In your code you do:

pthread\_join(threads[i], NULL);

That second argument is NULL, so you’re not capturing any return value. If you wanted a retval, you’d write something like:

void \*retval;

pthread\_join(threads[i], &retval);

**Tú dijiste:**

and what info does that retval hold? im seeing its a void pointer, why void?

The retval you’d get from pthread\_join is exactly whatever your thread’s start routine returned (e.g. via return ptr; or pthread\_exit(ptr);). It’s a void \* because C needs a generic pointer type—this way you can return a pointer to any data structure (or even encode an integer by casting). If your routine just does return NULL;, the retval is NULL.

**Tú dijiste:**

thread start routine? you mean thread\_launch?

The “thread start routine” is the function you pass as the third argument to pthread\_create—in your code, that’s runner\_routine.

By contrast, thread\_launcher is just a helper that calls pthread\_create in a loop. Each new thread begins executing runner\_routine, not thread\_launcher.

**Tú dijiste:**

so what do you mean by me thread start routine

The “thread start routine” is simply the function that each new thread begins executing. In your code, that’s runner\_routine. When you call

pthread\_create(&threads[i], NULL, runner\_routine, &runners\_array[i]);

you’re telling the OS: “Create a new thread and immediately run runner\_routine(&runners\_array[i]) inside it.” That function (with signature void \*runner\_routine(void \*)) is what we call the start routine.

**Tú dijiste:**

ok, remind me the prototype for pthread\_create

int pthread\_create(pthread\_t \*thread,

const pthread\_attr\_t \*attr,

void \*(\*start\_routine)(void \*),

void \*arg);

**Tú dijiste:**

so the third argument is the function and the 4th is the argument passed to that function, right?

Exactly. In

pthread\_create(pthread\_t \*thread,

const pthread\_attr\_t \*attr,

void \*(\*start\_routine)(void \*),

void \*arg);

* The **3rd argument** (start\_routine) is the function pointer
* The **4th argument** (arg) is the void \* passed into that function when the new thread starts.

**Tú dijiste:**

im confused, to pthread\_create we pass the runner\_data struct of every runner, but then in the runner\_routine function, we do "runner = (t\_runner\_data \*)arg;", i think im close to understand it, we need to cast it to the struct \* type, so we can access the different fields of the struct for that runner in particular

Yes. pthread\_create’s fourth parameter is a void \*, so you pass it the address of that runner’s t\_runner\_data (e.g. &runners\_array[i]). Inside runner\_routine(void \*arg), you do:

t\_runner\_data \*runner = (t\_runner\_data \*)arg;

to cast the generic void \* back to the correct struct pointer. That lets you read runner->some\_field for that specific runner.

**Tú dijiste:**

is it really necessary that casting? i think i read somewhere that casting a void pointer to a real data type was unnecessary because the casting could happen implicitly

Pensó durante unos pocos segundos

No. In plain C, you can assign a void \* to any object pointer (like t\_runner\_data \*) without a cast—so this would also compile:

t\_runner\_data \*runner = arg;

(You only need the explicit cast in C++.)

**Tú dijiste:**

we do "race = runner->race;" because the race data is shared by every thread and we want to get the specific race data of this specific runner? isnt all race data the same for every runner?

You’re right that every runner points to the same race struct. The line

race = runner->race;

just copies that shared pointer into a local variable for convenience. You’re not getting a “different” race per runner—each runner->race is the same address—but having it in a local race variable makes the code clearer inside runner\_routine.

**Tú dijiste:**

what could have been another way to retrieve the race data without getting it from this specific runner?

Pensó durante unos pocos segundos

You could have made the race pointer a global (or file‐static) variable instead of storing it inside each t\_runner\_data. Then in runner\_routine you’d just do:

extern t\_race\_data \*race; // declared at top of file

// …

void \*runner\_routine(void \*arg) {

// no need for runner->race; just use the global race directly

// …

}

Or, if you still want to pass something into the thread, you could pass a pointer to race itself (rather than the whole runner struct) and, if needed, look up the runner data by index inside the routine.

**Tú dijiste:**

cant use global variables

You have only two ways to give each thread access to race without globals:

1. **Bundle it into a separate “args” struct**  
   Instead of relying on runner->race, create a tiny struct that holds both pointers:

typedef struct {

t\_runner\_data \*runner;

t\_race\_data \*race;

} thread\_args;

Then, in your launcher loop:

thread\_args args[N];

for (int i = 0; i < N; i++) {

args[i].runner = &runners\_array[i];

args[i].race = shared\_race\_ptr;

pthread\_create(&threads[i], NULL, runner\_routine, &args[i]);

}

Inside runner\_routine(void \*arg), cast back:

thread\_args \*a = (thread\_args \*)arg;

t\_runner\_data \*runner = a->runner;

t\_race\_data \*race = a->race;

This way you still avoid globals, but you’re not “pulling” race out of the runner struct itself.

1. **Use thread‐specific storage (TSD)**  
   Create a pthread\_key\_t before spawning any threads:

pthread\_key\_t race\_key;

pthread\_key\_create(&race\_key, NULL);

Then, as soon as each thread starts (first lines of runner\_routine), do:

pthread\_setspecific(race\_key, shared\_race\_ptr);

Later in that same thread, whenever you need race, call:

t\_race\_data \*race = pthread\_getspecific(race\_key);

That also avoids globals, because each thread reads race out of its own “thread‐local” slot rather than from a shared global variable or from inside runner\_data.

**Tú dijiste:**

if all threads are running concurrently, how can pthread\_mutex\_lock(&race->print\_mutex); work?

When a thread calls pthread\_mutex\_lock(&race->print\_mutex), it asks the OS to “own” that mutex before proceeding. If no other thread holds it, the lock succeeds immediately and the thread continues. If another thread already locked it, the caller blocks (gets suspended) until the mutex is unlocked. Even though all threads are runnable “concurrently,” the mutex ensures that only one at a time can pass that lock call and enter the protected section. In other words, threads still run in parallel, but any thread that reaches a locked mutex will be paused by the scheduler until the mutex becomes available.

**Tú dijiste:**

so if 3 threads reach the code line where the lock is executed and a 4th thread had already locked it, does the rest of the code for those 3 threads stop until the mutex is unlocked?

Yes. When those three threads each call pthread\_mutex\_lock(&race->print\_mutex) and a fourth thread already holds the lock, all three will block right at that call. They won’t execute any further until the mutex is unlocked. Once the fourth thread calls pthread\_mutex\_unlock, one of the waiting threads acquires the lock and continues; the others keep waiting.

**Tú dijiste:**

so what does exactly the pthread\_mutex\_t is and what is print\_mutex used for? does it lock only the print\_mutex variable? the whole data struct? the thread? the function executed by the thread?

A pthread\_mutex\_t is just a mutex object—a lock you can acquire and release. In your code, print\_mutex is a member of the shared race structure used to ensure only one thread at a time enters the “print” section (so their console output doesn’t interleave).

When a thread does pthread\_mutex\_lock(&race->print\_mutex), it doesn’t “lock” the variable itself, the whole struct, or the thread. Instead, it gains exclusive permission to proceed past that line. Any other thread that reaches that same lock call will block until the first thread calls pthread\_mutex\_unlock(&race->print\_mutex). In other words, the mutex serializes access to whatever code (or data) you wrap between lock and unlock—in this case, the printing routine—but it does not freeze the entire struct or stop threads from running elsewhere.

**Mutex guidelines:**

1. **Keep critical sections minimal**

* Only protect the shared data that must be accessed atomically.
* Do any computation (e.g., timestamp capture) before acquiring the lock.
* This reduces **contention** (when multiple threads compete and wait for the same mutex).

2. **Avoid blocking operations inside locks**

* Don’t perform I/O, memory allocation, sleeps, or long loops while holding a mutex.
* If you need to do slow work, copy out under lock, release it, then proceed.

3. **Use fine-grained locking when possible**

* Give each independent data piece its own mutex instead of one big lock.
* This lowers contention, since threads aren’t all waiting on a single mutex.

4. **Maintain a consistent lock order**

* If you must lock more than one mutex, always acquire them in the same global sequence.
* That prevents **deadlock** (when two or more threads each hold one lock and wait indefinitely for the other’s lock).

5. **Prefer trylock or timed lock sparingly**

* pthread\_mutex\_trylock attempts to lock without blocking; if it’s already held, it immediately returns an error.
* A **timed lock** (e.g., pthread\_mutex\_timedlock) waits up to a specified timeout before giving up.
* Use these only when you need to avoid blocking forever—for example, to fail fast if a resource isn’t available.

**Explain pthread\_join and pthread\_detach**

**pthread\_join** means “I’ll wait here until that thread finishes.” It lets you

* block until the thread ends
* retrieve its return value
* clean up its resources.

**pthread\_detach** means “I don’t need to wait or join this thread.” It lets the thread

* run and finish on its own
* automatically free its resources when done
* but you cannot join it or get its return value.

**Tú dijiste:**

but in my code its not just doing it on a single thread, but on all of the threads of the array, so im wondering, is it joining all the child threads together, in a way similar to the way that strjoin joins strings?

No. Calling pthread\_join in a loop just waits for each thread to finish, one by one. It doesn’t merge or “concatenate” their results like strjoin does with strings. It simply blocks until that specific thread ends and then moves to the next.

**Tú dijiste:**

whats the point in calling it join if its not joining anything?

The term “join” in pthread\_join means “wait for and synchronize with” a specific thread, not “merge” data. When you call pthread\_join(thread, …), you’re telling the calling thread to block until thread finishes, then reclaim its resources (and optionally grab its return value). In other words, you’re joining your execution flow to that thread’s completion.

can a thread not finish? how would we know? does join return error if the thread does not end

A thread can fail to ever terminate if it gets stuck (e.g., infinite loop or deadlock). In that case, a plain pthread\_join will simply block forever waiting for it. It does **not** return an error just because the thread hasn’t ended; it only returns an error if you misuse it (for example, if you pass an invalid thread ID, try to join a detached thread, or a thread tries to join itself). If you need to detect or avoid waiting indefinitely, you’d have to use non‐blocking variants (like your platform’s pthread\_tryjoin\_np or a timed‐join extension) or implement your own timeout.

**Tú dijiste:**

so whats the point in join if the thread is going to finish anyway

Even though a thread will eventually exit, you use pthread\_join so that:

1. **You wait for it to finish before moving on.**  
   If you don’t join (and your main thread reaches return or exits), the whole process can end while its threads are still running—so they’d be killed prematurely.
2. **Its resources get cleaned up.**  
   By default threads are “joinable,” meaning the OS holds onto their exit status and stack until someone calls pthread\_join. If you never join, those resources leak.
3. **You can retrieve its return value.**  
   If your thread’s start routine returns something (a pointer), pthread\_join is how you grab that.

In short, even though a thread “will finish,” pthread\_join makes sure you pause until it does, clean up properly, and optionally collect its result.

**Tú dijiste:**

what was my thread start routine in this program? runner\_routine?

Yes. In your code, the thread start routine is the function

void \*runner\_routine(void \*arg);

You pass runner\_routine as the third argument to pthread\_create, so each new thread begins by executing that function.